Transactions across serverless functions leveraging stateful dataflows

de Heus, Martijn ; Psarakis, Kyriakos; Fragkoulis, Marios; Katsifodimos, Asterios

DOI
10.1016/j.is.2022.102015

Publication date
2022

Document Version
Final published version

Published in
Information Systems

Information Systems 108 (2022) 102015

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/is

Transactions across serverless functions leveraging stateful dataflows

Martijn de Heus [a], Kyriakos Psarakis [a], Marios Fragkoulis [b,1], Asterios Katsifodimos [a,*]

[a] Delft University of Technology, Netherlands
[b] Delivery Hero SE, Germany

ARTICLE INFO

Article history:
Received 17 October 2021
Received in revised form 12 February 2022
Accepted 23 February 2022
Available online 4 March 2022
Recommended by Gottfried Vossen

Keywords:
Serverless
Transactions
FaaS
Two-phase commit
Sagas
Streaming dataflows

ABSTRACT

Serverless computing is currently the fastest-growing cloud services segment. The most prominent serverless offering is Function-as-a-Service (FaaS), where users write functions and the cloud automates deployment, maintenance, and scalability. Although FaaS is a good fit for executing stateless functions, it does not adequately support stateful constructs like microservices and scalable, low-latency cloud applications. Recently, there have been multiple attempts to add first-class support for state in FaaS systems, such as Microsoft Orleans, Azure Durable Functions, or Beldi. These approaches execute business code inside stateless functions, handing over their state to an external database. In contrast, approaches such as Apache Flink's StateFun follow a different design: a dataflow system such as Apache Flink handles all state management, messaging, and checkpointing by executing a stateful dataflow graph providing exactly-once state processing guarantees. This design relieves programmers from having to "pollute" their business logic with distributed systems error checking, management, and mitigation. Although programmers can easily develop applications without worrying about messaging and state management, executing transactions across stateful functions remains an open problem.

In this paper, we introduce a programming model and implementation for transaction orchestration of stateful serverless functions. Our programming model supports serializable distributed transactions with two-phase commit, as well as eventually consistent workflows with Sagas. We design and implement our programming model on Apache Flink StateFun to leverage Flink's exactly-once processing and state management guarantees. Our experiments show that the approach of building transactional systems on top of dataflow graphs can achieve very high throughput, but with latency overhead due to checkpointing mechanism guaranteeing the exactly-once processing. We compare our approach to Beldi that implements two-phase commit on AWS lambda functions backed by DynamoDB for state management, as well as an implementation of a system that makes use of CockroachDB as its backend.

1. Introduction

The idea of democratizing distributed systems programming is not new. Approaches like Distributed ML [1], and Erlang [2] aimed at simplifying the programming and deployment of distributed applications. Erlang [2] first introduced the actor model, which Akka [3] implemented later in Scala, offering a very low-level programming model. Virtual Actors [4,5], inspired by Pat Helland's *entities* [6], try to abstract away the execution primitives from programmers.

Serverless computing [7] is a cloud computing execution model promising to simplify the programming, deployment, and operation of scalable cloud applications. In the serverless model, developer teams upload their code written in a high-level API, and the cloud platform takes care of the application's deployment and operations. Serverless computing aims to substantially increase cloud adoption by remedying the status quo in the cloud landscape, where developer teams need to possess skills in distributed systems, data management, and cloud execution model internals to use the cloud effectively.

Function-as-a-service & messaging. The most prominent serverless offering is Function-as-a-Service (FaaS), where users write functions, and the cloud providers automate deployment and operation. However, FaaS offerings lack the state management support and the ability to perform transactional workflows across multiple functions and state backends [8,9], which are needed by general-purpose cloud applications. In addition, none of the current FaaS approaches offers message-delivery guarantees, failing to support *exactly-once* processing: the ability of a function to mutate a function's state exactly one time per incoming message.

* Corresponding author.
E-mail addresses: m.deheus@tudelft.nl (M. de Heus), k.psarakis@tudelft.nl (K. Psarakis), marios.fragkoulis@deliveryhero.com (M. Fragkoulis), a.katsifodimos@tudelft.nl (A. Katsifodimos).
[1] Work done while at TU Delft.

https://doi.org/10.1016/j.is.2022.102015

When a system does not guarantee exactly-once processing, the burden of debugging and handling of systems errors (e.g., machines failing, network partitions, or stragglers) falls on the shoulders of programmers [10]. These programmers then have to "pollute" their business logic with extra consistency checks, state rollbacks, timeouts, or recovery mechanisms, for example. [11]. The result is that the majority of the application code is not comprised of business logic but error checking, management, and mitigation [12]. Sooner or later, programming distributed systems logic at the application level leads to problems of state consistency, bugs, and eventually significant service outages.

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 [13]. However, in the closed world of dataflow systems, exactly-once processing is possible [14–16] as in stateful dataflows, the system has *full control over both messaging and state management*. Apache Flink's StateFun[2] is, to the best of our knowledge, the first approach to build a FaaS execution engine on top of a streaming dataflow system offering exactly-once processing guarantees even under complex failure scenarios. However, StateFun's approach can also be implemented on top of other dataflow systems [14,17–20].

Such dataflow systems can execute stateful functions as follows: incoming events represent function execution requests routed to continuous stateful operators that execute the corresponding functions. With proper, consistent fault tolerance mechanisms [15,16], state of the art stream processing systems operate at high-throughput and low-latency. At the same time, they guarantee the correctness of execution even in the presence of failures. As we show in this paper, this set of properties can serve as a means of supporting *transactions* with minimal involvement from the application developers.

Transactional SFaaS. Although there is ongoing work in supporting stateful FaaS (SFaaS) applications, mutating state transactionaly, *across* functions remains an open problem. The only approach addressing distributed transactions in an SFaaS setting is Beldi [21], which provides fault-tolerant ACID transactions on stateful workflows across functions by logging the functions' operations to a serverless cloud database. Cloudburst [22] with HydroCache [23] provides causal consistency on function workflows forming a DAG by leveraging Anna [24], a key–value store with conflict resolution policies in place. Cloudburst does not provide isolation between DAG workflows.

In contrast with the aforementioned approaches, developer teams in the microservices and cloud applications landscape go to extreme lengths when they need to implement transactional workflows across the boundaries of a single service or function. The most common approach adopted is the Saga pattern [25]. The Saga pattern separates a transaction into sub-transactions that proceed independently with the benefit of improved performance but at the risk of having to undo or compensate the changes of successful sub-transactions when at least one of the involved sub-transactions fails. In addition, compensating actions can be challenging when concurrent changes are applied to the state because Sagas do not require any means of isolation. For this reason, state consistency needs to be dealt with at the application level. On the other hand, applications that prioritize consistency over performance implement distributed transactions using the two-phase commit protocol. Two-phase commit (TPC) [26] offers ACID, serializable transactions but imposes blocking operations across functions participating in a transaction, which penalizes performance in return for strict atomicity. Apparently, distributed transactions and the Saga pattern serve opposing goals.

In this work, we draw inspiration from best practices in developing microservices and cloud applications and offer developers a programming model that supports both Sagas and distributed transactions with two-phase commit. Our implementation for authoring workflows across stateful functions in FaaS with transactional guarantees is publicly available on GitHub.[3] We implement the two approaches on an open-source stateful FaaS system, Apache Flink's [14] StateFun.[4]

In summary, our work makes the following contributions:

- We argue for implementing transactional workflows on a stateful dataflow engine and present the advantages of this approach
- We propose a programming model for transactional workflows across stateful serverless functions
- We implement the two main approaches used by cloud application practitioners to achieve transactional guarantees: two-phase commit and Saga workflows
- We evaluate two transactional schemes using an extended version of the YCSB benchmark on a cloud infrastructure
- We compare against the state-of-the-art academic SFaaS proposal that supports serializable transactions and one of the most popular transactional distributed database systems.

This manuscript is an extended version of a paper that received the best paper award at the 15th ACM DEBS conference [27]. This version has four novel aspects: (*i*) experiments against two state of the art approaches, (*ii*) a new section on the relation of our work with deterministic databases, (*iii*) additional technical description of our work, and (*iv*) a discussion on transactional workflows for web and cloud applications.

Paper structure. Section 2 gives the motivation of this work and explains the benefits of running transactions on dataflow graphs, while Section 3 presents the background. Next, Section 4 introduces the concept of coordinator functions, and Section 5 details their implementation and the introduced programming model. The experimental setup is presented in Section 7, while the performance of coordinator functions is evaluated in Section 8. Section 9 presents the related work. Finally, Section 11 summarizes the work and discusses interesting areas for further research.

2. Transactions on streaming dataflows

Serverless platforms come in different flavors. One breed of SFaaS systems (e.g., Apache Flink StateFun and [9]) is built on top of a stateful streaming dataflow engine. This architecture bears important implications for supporting transactions because of how distribution, state management, and fault tolerance work.

Network communication between distributed components in a typical streaming dataflow engine is implemented via FIFO network channels that guarantee exactly-once processing and preserve delivery order. In a serverless FaaS system, this characteristic obviates the need for handling lost messages and implementing retry logic concerning function invocations in transactional workflows. Messaging errors and retries are a significant source of friction and development effort at the application level, and those are offered by the underlying dataflow system.

State management in state-of-the-art streaming systems achieves exactly-once processing guarantees by taking consistent snapshots of the system's distributed state periodically [15]. The snapshots capture a globally consistent state of the system at a

[2] http://statefun.io.

[3] https://github.com/delftdata/flink-statefun-transactions.

[4] https://statefun.io.

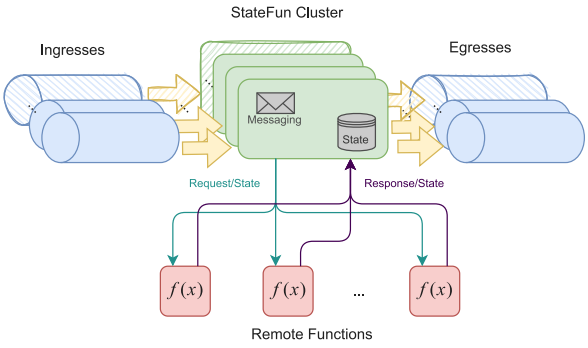


Fig. 1. StateFun Architecture.

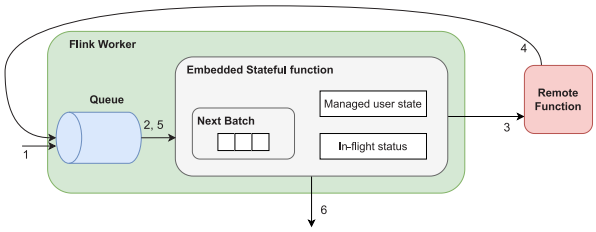


Fig. 2. Original communication flow for remote functions.

specific point in time and are used to recover the system's state upon failure. Exactly-once means that the changes brought by each function execution instance are recorded in the system's state exactly once, even in the face of failures. For transactions, this capability is essential because fault recovery of transactions can piggyback on the underlying fault tolerance mechanism with zero effort and knowledge by the application. Given that a big part of code and effort is spent on failure handling, fault tolerance, and virtual resiliency [28] provided at the system level can play a significant role.

Furthermore, unlike traditional streaming queries where the computations are fully encapsulated within the system's operators, it is common to have nondeterministic side effects (typically calls to external services or remote key–value stores) in microservices and cloud applications. However, the traditional fault tolerance mechanisms of streaming dataflow systems were not designed to support non-determinism prevalent in general-purpose applications. Thus, the consistency of applications and the integrity of transactions are at risk when transactions involve nondeterministic operations. Extending the fault tolerance approach of streaming dataflow systems to support nondeterministic computations [16] is an important step towards opening their adoption for executing general-purpose applications. Recent work [22,29] also recognizes the dataflow model as a key enabler for the SFaaS systems of the future.

In short, we believe that stateful streaming dataflows and the associated research that has been proposed so far [30–32] can alleviate the burden of building rich stateful and transactional applications on top of streaming dataflows. This paper presents a step in this direction.

3. Preliminaries

In this section we first present our transactional model (Section 3.1). Then, in Section 3.2 we describe the functionality and internals of Apache Flink StateFun, which forms the backbone of our proposed solution. Lastly, in Section 3.3 we list the requirements that an SFaaS system should satisfy in order to be considered as a backend for our work.

3.1. Transaction model

In the context of this work, a transaction is an atomic execution of a set of stateful function invocations. More specifically, the transactional model introduced in this paper considers transactions defined up-front. This is referred to as *single-shot* [33] or *one-shot,one-shot-h-store transactions* in prior works. We follow the definition of H-Store's [34] *one-shot transactions*, which states that the output of a function (query) cannot be used as input to subsequent functions (queries) in the same transaction. Since the output of functions is not used by subsequent ones, the execution of functions involved in a transactional workflow is independent of one another. This simplifies coordination of the transaction across the system while still providing a practical model for transactions. Widely used database services, such as Amazon's DynamoDB [35], support one-shot transactions [33]. In an SFaaS system, one-shot transactions provide a significant advantage: functions can implement arbitrary business logic in a general-purpose programming language such as Java or Python instead of being limited to the API supported by a specific database, such as DynamoDB. Thus, this advantage translates to considerable flexibility in the programming model.

3.2. Apache Flink StateFun

Apache Flink StateFun[5] offers an abstraction and runtime for users to implement stateful cloud functions. A stateful function implemented by user code is referred to as a function type and describes the state held by this function type. Multiple instances based on the same function type can exist in parallel and are identified by an id. Each of these function instances encapsulates its own state and can be uniquely addressed by the combination of its type and id. Function instances can be invoked from other function instances or through ingresses such as Kafka. Function instances can have four different controlled side effects; (1) state updates, (2) function invocations, (3) delayed function invocations, (4) egress messages (for example, Kafka). StateFun supports end-to-end exactly-once guarantees from ingress to egress, including any state updates.

Architecture. In Fig. 1 we present the general system architecture of Apache Flink StateFun. The interface with the system is based on the ingress/egress pattern (e.g., ingest/produce Kafka messages). The Apache Flink StateFun cluster lies in the system's core, consisting of multiple workers that manage both messaging and the partitioned state, leading to stateless remote functions. However, this means that the state needs to be transferred along with the request to each specific function for processing. After processing, both the response and the new state are returned to the StateFun cluster. This architecture's major benefit is that since StateFun manages both messaging and state exactly-once semantics is easier to achieve than other architectures.

Embedded vs. remote functions. Functions can be deployed both inside the StateFun workers (referred to as embedded functions) and outside the StateFun cluster (co-located and remote functions). Embedded functions are simply an abstraction on top of stateful streaming operators in Flink, therefore providing exactly-once and fault-tolerance guarantees. StateFun allows dynamic communication between these streaming operators by introducing a cycle in the streaming graph. The co-located and remote functions are entirely stateless because the state is persisted

[5] https://flink.apache.org/stateful-functions.html.

within StateFun. This paper focuses on remote functions as these can leverage existing FaaS services such as AWS Lambda to autoscale the compute layer. Fig. 2 shows how remote functions work. Each function instance is represented by an embedded stateful function in the StateFun cluster. This standardized embedded function is responsible for managing the state of the function instance and the communication with the remote function that may be deployed anywhere. The persisted data in the embedded stateful function with the communication pattern for remote functions are shown in Fig. 2.

```
def serializable_transfer(context, message: Transfer):
    subtract_credit = SubtractCreditMessage(
                          amount=message.amount)

    context.tpc_invocation("account_function",
                           message.debtor,
                           subtract_credit)
    add_credit = AddCreditMessage(
                           amount=message.amount)
    context.tpc_invocation("account_function",
                           message.creditor,
                           add_credit)
```

Listing 1: Two-phase commit coordinator function.

Function invocations as dataflow messages. Invocations that are sent to a function instance arrive in a queue, as shown in step 1 of Fig. 2. If the embedded stateful function is ready to process the next invocation, it pulls a message (invocation parameters) from the queue (step 2). When no invocation is being executed at the remote function, the remote function is called. However, if the remote function is busy with a previous function call, the current invocation message is appended to the next batch. Batching is used as an optimization in order to avoid multiple remote calls to external functions at the expense of latency (see Section 8.1). Batches are also used to preserve the invocation order and order of state access (the batch has to wait until the state updates caused by the previous batch have been performed), thus ensuring linearizability at the function instance level.

In step 3, the stateless remote function is called through a Protobuf interface that contains both the (keyed) state required for the remote function to operate and the invocation parameters of the function. The stateless remote function can execute the (batch of) invocations and will be ready to return the updated state back to the Flink worker that made the call. In step 4, the response of the stateless remote function is appended to the queue of incoming messages to the function. The response includes any side effects caused by the invocation(s), including updates to the user-defined state.

When the response from the stateless function is processed (step 5), the side effects caused by the invocation(s) are applied to the state of the embedded function, updating the managed state in the embedded stateful function. If any invocations are batched, the next batch of invocations is sent to the remote stateless function, and the batch is truncated. When there are no batched invocations, the in-flight status is cleared. Finally, any outgoing function invocations are sent to the queues of their respective function instances, and egress messages are sent to their respective egresses (step 6).

3.3. Assumptions & requirements

As we describe in the next section, our coordinator functions rely on an underlying SFaaS system for bookkeeping the state of ongoing transactions and reliable messaging. To allow this, the underlying system should satisfy two requirements.

Exactly-once processing guarantees. Firstly, it is required for all communication to be reliable and executed with exactly-once processing guarantees. Thus, we require that the underlying system is fault-tolerant [14] to ensure atomicity in case of a failure in the middle of a transaction. This also means that the state is durable within a snapshot/checkpoint, even in the event of failures. If we can rely on exactly-once processing guarantees, message replay, and error handling, a significant part of transaction coordination can be greatly simplified. Flink StateFun does guarantee exactly-once processing.

Linearizable operations. The second requirement is that the operations for any specific function instance should be linearizable, which means that there is a given order that operations are performed on the function instance, and the state encapsulated in this instance. Accordingly, a function invocation will always have the correct state of the function instance in order to implement transactions. Since Flink StateFun's function instances use a single replica of the state per function instance and a single process executes function invocations for that function instance in a sequential FIFO manner, this ensures linearizable operations per function instance.

```
def sagas_transfer(context, message: Transfer):
    subtract_credit = SubtractCreditMessage(
                        amount=message.amount)
    add_credit = AddCreditMessage(
                 amount=message.amount)
    context.saga_invocation_pair("account_function",
                                 message.debtor,
                                 subtract_credit,
                                 add_credit)
    context.saga_invocation_pair("account_function",
                                 message.creditor,
                                 add_credit,
                                 subtract_credit)
```

Listing 2: Saga coordinator function.

4. Approach overview

In this section, we introduce the concept of stateful coordinator functions and provide an overview of our approach. Our approach is based on the simple observation that since an underlying SFaaS system provides exactly-once processing and message delivery guarantees, conceptually, it would be much simpler to implement a transaction coordinator as a regular, stateful function. With this in mind, we opted for implementing a transaction API on top of stateful functions, which we present in Table 1. Notably, further work is required to raise the transaction abstractions at an even higher level [9,31] as syntactic sugar.

4.1. API

A stateful coordinator function is a stateful function that preserves state about the execution of a given transaction. Coordinator functions have the ability to force other function instances to abort or compensate for the changes they applied.

API overview. Our coordinator function implements two transaction coordination patterns: two-phase commit and Sagas [25]. A complete example of a coordinator function for two-phase commit and Saga is shown in Listings 1 and 2 respectively. In short, to coordinate a two-phase commit transaction, the user needs to invoke function instances via `tpc_invocation`, while for a Saga, an invocation *pair* is expected, which consists of the normal transaction invocation and the corresponding compensation invocation to be sent to the same function instance. A Saga invocation pair can be called with `saga_invocation_pair`. An important difference between the behavior of the two schemes is that a failure in a Saga workflow will incur a compensating function call.

Table 1
Coordinator functions' Python API.

Function	Description
Shared coordinator function methods	
`send_on_success`(*type*, *id*, *message*)	Sends a message to another function instance if the transaction is successful
`send_after_on_success`(*delay*, *type*, *id*, *message*)	Sends a delayed message if the transaction is successful
`send_egress_on_success`(*type*, *egress_message*)	Sends a message to an egress if the transaction is successful
`send_on_failure`(*type*, *id*, *message*)	Sends a message to another function instance if the transaction failed
`send_after_on_failure`(*delay*, *type*, *id*, *message*)	Sends a delayed message if the transaction failed
`send_egress_on_failure`(*type*, *egress_message*)	Sends a message to an egress if the transaction failed
Two-phase commit function methods	
`tpc_invocation`(*type*, *id*, *message*)	Add a function invocation to the transaction
`send_on_retryable`(*type*, *id*, *message*)	Sends a message if the transaction aborted because of a deadlock
`send_after_on_retryable`(*delay*, *type*, *id*, *message*)	Sends a delayed message if the transaction aborted because of a deadlock
`send_egress_on_retryable`(*type*, *egress_message*)	Sends a message to an egress if the transaction aborted because of a deadlock
Sagas function methods	
`saga_invocation_pair`(*type*, *id*, *message*, *compensating_message*)	Add a pair of a message and a compensating message to the transaction
Ordinary functions	
`FunctionInvocationException`	Raised to fail the function invocation

4.2. Two-phase commit

The `serializable_transfer` function of Listing 1, receives a context (the underlying context of StateFun as we have extended it to support transactions) and a message. The message is of type `Transfer`, and it contains three fields: the amount of money transfer, a creditor, and a debtor. The amount mentioned in the message must be subtracted from the debtor and transferred to the creditor. To this end, assuming that there is a function type registered in the system as `account_function`, as per the original StateFun API, we need to construct an object containing the parameters for the `account_function` and push that message to the transaction coordinator. This is done in lines 5–7: we give the TPC coordinator the function type to invoke, alongside the id of the debtor to form the address of the function instance and the `SubtractCreditMessage` which is going to be given to that function as a parameter. Subsequently, we do the same for the creditor: we construct an `AddCreditMessage` and we pass it over to the function type `account_function`. In short, the transaction coordinator function instance will make sure that the two function instances are invoked with serializable guarantees. It does this by coordinating a two-phase commit protocol across the function instances with locking to ensure isolation. More details on these aspects are given in Section 5.

4.3. Sagas

Similarly to two-phase commit, our API offers the ability to specify Sagas: as seen in Listing 2, the `saga_invocation_pair` function in line 6 will receive the target function name, the ID of the debtor as well as two messages: the `subtract_credit` and its compensating action `add_credit`. If there is a failure during the execution of `subtract_credit` our Sagas transaction coordinator will execute the compensating action `add_credit` which will put back the original credit to the debtor's account. The details on how Sagas are executed are given in Section 5.

4.4. Extensions to regular functions

To allow the execution of a transaction by the two types of coordinator functions across any arbitrary function instances, some extensions to regular functions are required.

First, functions that can partake in a coordinated transaction need to be able to fail explicitly. After a failure is communicated to a coordinator function, it results in a transaction rollback. Currently, there is no notion of failing an invocation in Flink StateFun; the function invocation may simply perform no side effects. To allow explicit failure, a field containing these details is added to the protocol between StateFun and the remotely deployed functions. From the API perspective, a function failure can be triggered by throwing an exception. The failure of a function can be roughly compared to integrity constraint violations based on the state encapsulated in a function instance in traditional database terms.

Second, any batching mechanism needs to be changed. TPC coordinator functions ensure isolated transactions. This means that any function invocation that is part of such a transaction may not be batched between other function invocations. Third, appropriate locking should be implemented on the level of function instances to ensure the isolation of serializable transactions based on two-phase commit coordinator functions.

Finally, the function instances should transparently communicate with the coordinator functions to not burden developers with this task.

5. Transactional workflows

In this section, we present our Python API in more detail, and we present the implementation for transactional workflows across stateful serverless functions on Apache Flink StateFun. Our implementation consists of coordinator functions that enforce either a distributed serializable transaction with a two-phase commit or a Saga workflow as a transaction without isolation.

5.1. Coordinator functions

Coordinator functions instrument transactional workflows across ordinary Stateful functions. To achieve this, coordinator functions encapsulate the state of active transactional workflows that they are in charge of but hold no state of the participating function executions or custom user-defined state. A coordinator function can be invoked simply by its name (uniquely identified by a type internally) and an ID generated randomly at initialization time. Then an input message will arrive at the coordinator's input queue. If the coordinator function is involved in an ongoing transaction, the message will be queued until the workflow that is executing completes. The coordinator functions' Python API is listed in Table 1.

Fig. 3 shows the common communication flow between a coordinator function and regular function instances. Specializations of this communication for two-phase commit and Saga workflows are described in Section 5.2 and Section 5.3 respectively. Messages that are not always sent in both cases are annotated with

a *. Fig. 3 shows the enriched internal structure for regular function instances compared to Fig. 2. These are the extensions that we implement for regular functions so that they can participate in transactional workflows.

5.2. Saga coordination

The programming model of the Saga coordinator function is shown in Listing 2 through an example. Table 1 presents the API. In Sagas, the developer is responsible for defining pairs of function invocations so that the invocation of the second function compensates the one of the first function [25]. Besides this, the Saga coordinator function can also define side effects (e.g., outgoing egress messages) based on different completion scenarios of the transaction (success or failure). The function invocations composing a Saga are executed in parallel in the current implementation.[6] In the following, we describe the messages specifically for Sagas seen in Fig. 3.

Initialization & remote coordinator function call. First, a message is sent to the coordinator function to initialize a transaction (step 1). The message is taken from the queue to initialize the transaction (step 2). Then, the remote Saga coordinator function is called with the incoming message (step 3). The remote function returns the definition of the Saga workflow to its embedded counterpart (step 4). This includes the function invocations involved in the transaction and their compensating invocations, as well as the side effects to perform on success or failure.

Processing the remote coordinator function's result. When the embedded function processes the result of the remote function (step 5), a random transaction ID is generated, and a map is created holding the addresses of function instances and the result of their execution (at this stage, those are initialized as `null` values). It follows that only one invocation per function instance can be involved in a particular workflow. If multiple invocations of a single function instance are required, this can be solved at the application level by allowing a single message, which combines multiple function invocations to be sent to the function instance.

Invoking regular functions. In step 6, each of the participating regular (non-coordinator) function instances receives a function invocation in its input queue. All the invocations are sent simultaneously, and the function instances can do the work in parallel. These function invocations are distinguishable as function invocations that belong to a Saga workflow. Each Saga function invocation is fetched from the queue, and it is either directly sent to the remote function or batched with other invocations for efficiency (step 7). Because Sagas do not require isolation, a function invocation can be batched with other invocations. Then it is sent to the regular remote function (step 8). After processing it, the function's response is added to the queue of its stateful embedded representation in StateFun (step 9). When the response of the stateless remote function is processed in the embedded stateful function at step 10, the indices in the in-flight function invocation metadata and new list added to the Protobuf interface, i.e., the regular function extensions, are used to identify the result status of the Saga function invocations and the corresponding coordinator's addresses. If the function invocation fails, no side effects of the function are performed. After this, this function can continue processing other function invocations.

[6] We plan to expose a configuration for the intended behavior in the API, in order to optionally make these sequential.

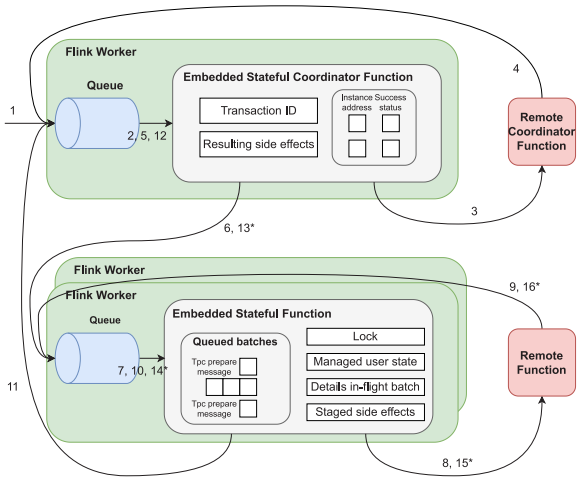


Fig. 3. Communication flow for transactions.

Saga success vs. compensation. Based on the success status of the Saga function invocation, a success or failure message is sent to the coordinator function (step 11). When the embedded coordinator function processes the success status of each function invocation, the map is updated with either a success or failure status (step 12). If a function instance fails, any function instances that successfully executed their function invocation are messaged with their respective compensating actions (step 13), and the side effects in case of a failure are performed (steps 14, 15, 16). The coordinator function has to wait until the result of all function invocations is received before it is done. In case any of the function invocations fails, the coordinator function sends the compensating messages to all function instances that successfully processed their invocation. Note that there is no need to send compensating invocations to function instances that failed since those function instances have applied no side effects. The compensating messages are processed as regular messages and are only required when any of the function invocations fail. This means that the performance of a Saga workflow will be worse if it is likely to fail as this will require extra messaging and processing, up to double. As a matter of fact, this is the trade-off offered by optimistic transaction approaches like Sagas.

5.3. Two-phase commit coordination

In Listing 1 we presented the programming model for a two-phase commit coordinator function; Table 1 shows the available functions of the two-phase commit API. Similar to Saga coordinator functions, two-phase commit coordinator functions can also define side effects to execute for any completion scenario. Beyond successful and failed completion, two-phase commit transactions can also be complete as "retryable". This occurs when the transaction is aborted due to a deadlock. In the following, we describe the workflow of the two-phase commit as seen in Fig. 3. Note that the initialization of the workflow, i.e., steps 1–5, is the same as in Sagas. Thus, we do not detail it here.

PREPARE & two-phase locking growing phase. Each involved function instance is messaged with its respective function invocation in step 6. This message is identifiable as a `PREPARE` message of the two-phase commit protocol. When a two-phase commit function invocation arrives at the embedded stateful regular function and a batch of invocations for this function is currently in-flight, this two-phase commit function invocation is not batched with

other invocations. Instead, the two-phase commit function invocations split up batches and are sent to the remote function in isolation as seen in Fig. 3. This practice increases the complexity of the batching mechanism, as it now requires a queue of batches rather than an append-only batch as shown in Fig. 2.

Invoking regular remote functions. When the message (and current state) is processed and sent to the remote function in steps 7 and 8, the transaction ID and the address of the two-phase commit coordinator function are stored in the details of the in-flight batch of invocations. The lock on the function instance is also set at this point. The response of the stateless remote function includes the result status of the function invocation and any side effects (step 9). Suppose a `FunctionInvocationException` is thrown at the stateless remote function. In that case, the response of the remote function is discarded, a response to the coordinator function instance is sent to notify it that the invocation failed, and the regular function instance's lock is removed as it knows the transaction will be aborted. If the function invocation is successful, the lock is kept, and a success response is sent to the coordinator function instance. The state effects are then stored as staged side effects in the function instance (step 10). Any other messages that arrive while the function instance is locked are put in the queued batches.

`ABORT` & Two-phase locking shrinking phase upon Failure. The message at step 11 notifies the two-phase commit coordinator function instance whether the function invocation succeeded. If the two-phase commit function instance receives the message that a function invocation failed (step 12), it immediately sends an `ABORT` message to all other function instances and performs the appropriate side effects (step 13), and calls the two-phase lock shrinking phase. After this, the two-phase commit function is done.

`COMMIT` & Two-phase locking shrinking phase. If the two-phase commit function instance receives the message that a function invocation was successful, it updates the map it keeps of all involved function instances. If all function instances succeed, it sends `COMMIT` messages to all involved function instances and publishes the appropriate side effects (i.e., applies the changes to the embedded function state).

`COMMIT/ABORT` & Two-phase locking shrinking phase. When a function instance receives a `COMMIT` message (step 14), it executes its staged side effects, releases the lock and continues processing the next request. When a function instance receives an `ABORT` message, it discards its staged changes, releases the lock, and continues processing. Note that a function could also receive the `ABORT` message while the `PREPARE` message is still in the queue or in-flight. In this case, the `PREPARE` message is discarded. Messages 15 and 16 are never sent for two-phase commit transactions.

Deadlock detection. Due to the use of locks, the two-phase commit protocol is susceptible to deadlocks. A deadlock can happen when two or more different two-phase commit transactions wait on the locks on function instances that are held by other transactions. To deal with deadlocks, we have implemented a deadlock detection mechanism, which we describe below. All participants in the two-phase commit transaction can be partitioned across different machines, and the state of active transactions is encapsulated in different coordinator function instances. Thus, we do not want transactions to rely on any centralized component for handling deadlocks. We implemented the Chandy–Misra–Haas algorithm [36] that provides a simple way to detect deadlocks in a distributed manner, without dependence on a single global coordinator. Whenever a deadlock is detected in a transaction, it immediately completes as a retry-able transaction and sends `abort` messages to all involved function instances. Upon receiving a retry-able result status, a two-phase commit regular function may send itself a delayed invocation with the same initial message (and possibly a counter attached) to perform a retry. This is left to the developer so that the system remains flexible across various use cases.

6. Towards strict serializability

Our approach offers serializable transactions by virtue of using the two-phase locking protocol. Under certain transactional scenarios, which we discuss in this section, our approach can achieve *strict* serializability, where the processing of transactions happens in the same order that the transactions have reached the system. In order to achieve strict serializability, our approach would require extensions. In the following, we explain various design decisions or changes that need to occur in our system to support different flavors of serializability.

Single-partition transactions. A single-threaded operator instance executes every operation to the state of a given partition. Thus, single-partition transactions are guaranteed to be processed in a serial manner. This also follows that single-partition transactions will be guaranteed strict serializability even when executed in a distributed fashion. Moreover, transactions that operate on different partitions are going to scale horizontally.

Multi-partition transactions. In the general case, a transaction in our approach can access multiple functions, mutate multiple state partitions, or both. Since two-phase locking is used, the system can enforce serializability across multiple functions and data partitions of the same function. In addition, our approach does not guard against changes in the order of transaction executions. For example, induced by transaction aborts due to a deadlock, or system failures, transactions may be re-submitted for execution.

Strict serializability. Our approach features three core advantages that provide important foundations for achieving strict serializability. First, since we support one-shot transactions, the system is aware of the keys that will be touched from a transaction prior to its execution. Furthermore, these one-shot transactions can be arranged prior to their execution in a specific serial order — that order can be set to be the order of arrival, thus guaranteeing strict serializability. Second, Apache Flink, which executes our transactions, recovers from a failure by falling back to the latest completed checkpoint and re-processes input requests following the checkpoint. This strategy allows us to reconstruct the exact same state as prior to the failure under the assumption of deterministic computations. Finally, data-parallel processing in disjoint state partitions allows us to execute concurrent transactions in a parallel manner and without the need for concurrency control.

Relation to deterministic databases. Interestingly, the three aforementioned characteristics of our approach resemble design choices opted by deterministic databases [37–39], which achieve strict serializability: the concurrent processing of a specific set of transactions across a distributed system is guaranteed to result in one, single runtime state.

Furthermore, one could draw inspiration from deterministic databases for advancing its transactional model in two ways. First, transactions on dataflow systems would benefit from an input transaction log for pre-determining the order of transactions in a way that would not introduce aborts during execution, essentially implementing a protocol like Calvin [40]. Second, one could leverage a determinism service [16] to wrap nondeterministic computations, which would cause its state to diverge when recovering from a failure. Essentially, pre-ordering a batch of transactions and ensuring deterministic transaction processing would help dataflow-based transactional FaaS systems guarantee strict serializability.

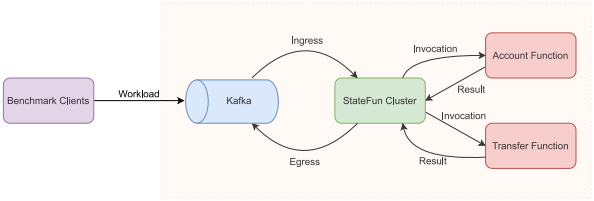


Fig. 4. StateFun benchmark application architecture.

7. Experimental setup

In this section, we describe in detail our experimental evaluation methodology. For the lack of a benchmark aimed at SFaaS, we opted for an extension of the *Yahoo! Cloud Serving Benchmark* (YCSB) [41] benchmark.

7.1. Benchmark workload

In YCSB, the first step is to insert records into the system with a unique ID and several task-specific fields. After the data insertion stage, the benchmark performs operations on the initialized state. YCSB defines `read` and `write` operations as part of their core workloads. Because this work's main contribution is distributed transactions across stateful function instances, we added a new operation based on an extension introduced in [42]. This operation is called a `transfer`, and it atomically subtracts `balance` from one account and adds this to another, meaning that records also include a numeric `balance` field. These additions mean that the workloads can consist of the following three operations:

`read` Reads the state associated with a single key and outputs it to the egress.

`write` Updates a field associated with a single key and outputs a *success* message to the egress.

`transfer` Requires two keys and a specified amount, subtracts the amount from the balance of one key, and adds it to the other. Depending on the transaction result, the output is either a success or failure message to the egress.

Across experiments, we vary the proportion of each operation in the resulting workloads. In YCSB, the user selects the probability distribution of the operations' record IDs. In this work, we assume uniform key distributions. The added benefit is that the number of requests for a single key can be increased transparently by decreasing the system's total number of records. Finally, YCSB allows variations in the number of fields and the size of the values associated with each field. In this evaluation process, all records have ten fields containing a single random string of 128 bits and a single integer field. A StateFun application is implemented with the following two functions to support the operations defined in Section 7.1:

– *Account function.* This is a regular function containing the record state for each key. It processes messages to read the state, updates the fields, and subtracts or adds balance as part of a transaction. It throws an exception and rolls back the transaction if the key does not exist, or there is not enough balance to subtract the transaction amount.

– *Transfer function.* The transfer function is a transactional/coordinator function that takes a message consisting of two different keys and an amount. That message represents a transaction consisting of two function invocations, one to each of the function keys. This function is implemented with both the two-phase commit and the Saga API.

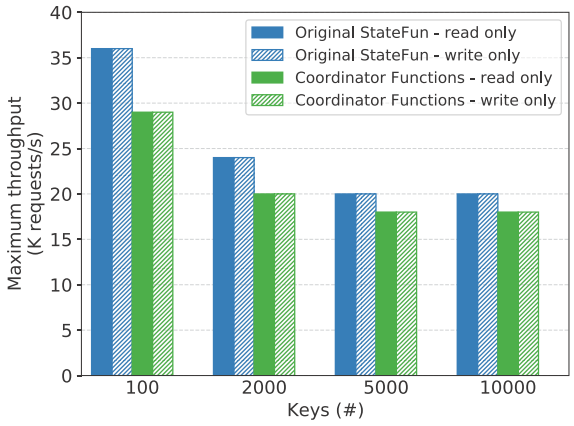


Fig. 5. Maximum throughput for the original StateFun vs. StateFun with coordinator functions.

Fig. 4 depicts the architecture of the system under test. The benchmark publishes the workload to a Kafka cluster. StateFun reads from Kafka as ingress, invokes the appropriate functions, and then publishes the result to a Kafka topic as an egress. For CockroachDB (v21.1.7), Kafka clients read from the relevant topics and submit queries to the not geo-replicated database.

Although CockroachDB and Kafka can provide exactly-once semantics individually, since the state (CockroachDB) and messaging (Kafka clients) are not managed by a single entity and under the same checkpointing mechanism, this deployment offers at-least-once semantics. More specifically, the clients that consume Kafka queues that deliver the transaction-initiating events need to pull an event from a Kafka topic, submit a query to CockroachDB and acknowledge the execution of the respective transaction back to Kafka. However, in the event of a client (or database) failure, the transaction may be executed, but the message to the queue may never be acknowledged. Not having returned the acknowledgment to Kafka, the client will re-execute the same transaction after recovery. In general, unless the transactions come with application-specific idempotence keys, the system by itself cannot enforce exactly-once processing guarantees, falling back to at-least-once guarantees.

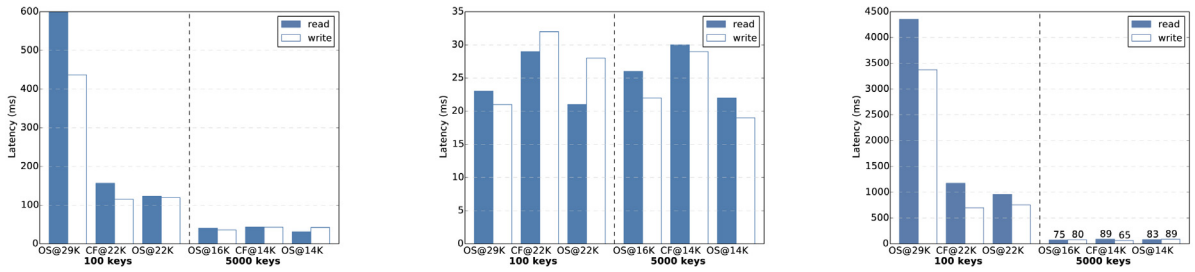
Our StateFun-based implementation and the CockroachDB deployment are deployed on SurfSara[7], an HPC cloud with instances with up to 80-vCPUs. For our experiments, we used a two-VM Kubernetes cluster to simplify deployment and management of the system's separate components with enough vCPUs to support the system's configuration under test. Beldi was deployed on AWS. All components shown in Fig. 4 can be horizontally scaled as necessary. Additionally, we give the Kafka cluster and the clients enough resources to ensure that they can handle the load: when a bottleneck appears, it can be attributed to the system performing the application logic, i.e., the StateFun cluster, CockroachDB, or Beldi's API.

7.2. Evaluation metrics

We evaluate the systems based on two metrics. First, the throughput is either at max or stable (80%), showing the number of workload operations the system can handle per second, and the latency, showing the time it takes to process an operation.

The maximum throughput of each workload and system configuration is found by steadily increasing the input throughput

[7] https://userinfo.surfsara.nl/systems/hpc-cloud.



(a) Mean (b) Median (c) 95th percentile

Fig. 6. Graphs comparing latencies of original StateFun (OS) and StateFun with coordinator functions (CF) at different throughputs for read-only and write-only workloads.

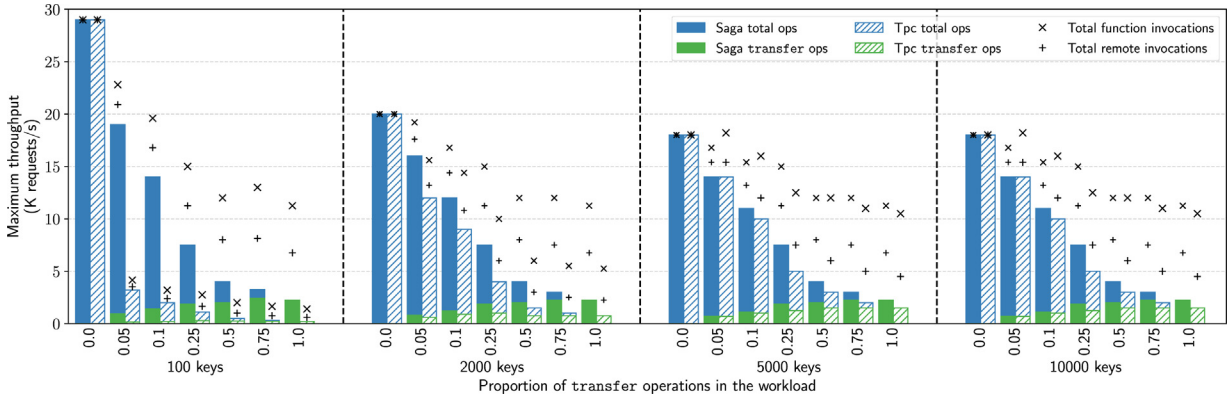


Fig. 7. Maximum throughput for workloads with increasing proportions of `transfer` operations in the workload.

created by the benchmark clients in Kafka until the StateFun cluster/CockroachDB can no longer consistently handle the load, as measured by the system's output throughput in Kafka. At some point, the output throughput starts fluctuating, and we define this value as the maximum throughput for the configuration. In the comparison with Beldi, we could not measure it this way since it will always rescale to accommodate the new load. So our approach in this matter is to take the 80% throughput of the StateFun configuration and run Beldi with the same input throughput.

We use the Kafka event time for the ingress and egress events of operations to measure their end-to-end latency. Since latency is always dependent on the throughput, in our experiments, we set the throughput to 80% of the maximum throughput to allow consistent operation of the system under test and measure the latency accurately. When comparing latencies, the different throughput rates at which the latency is measured should be considered.

8. Experiment results

In this section, we go through the experimental evaluation of our system that is split into six experiments with the following goals.

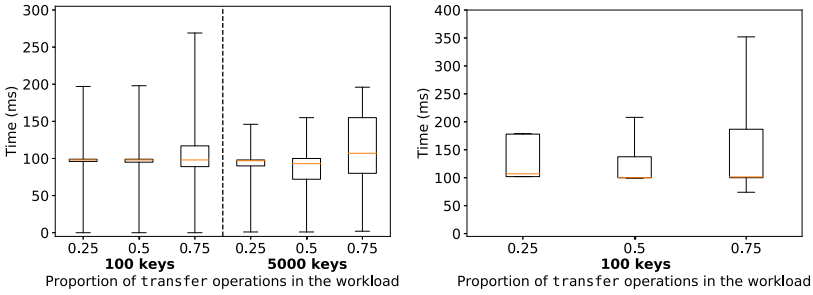
(i) Determine the overhead that function coordination introduced to StateFun (Section 8.1).
(ii) Compare between the two transaction protocols with/out rollback operations (Section 8.2).
(iii) Evaluate the system's scalability (Section 8.3).
(iv) Perform a microbenchmark with a fixed number of machines and a variable number of keys and proportions of `transfer` operations (Section 8.4).
(v) Compare against the CockroachDB with Kafka clients deployment (Section 8.5).
(vi) Compare against Beldi (Section 8.6).

In terms of resources used, for (i, ii, iv, v), we used three 4-CPU StateFun workers/CockroachDB nodes, and for (iii), each worker had 2 CPUs. In (v), we kept the default settings meaning that CockroachDB replicates data three times for fault tolerance and high availability. For (vi), we allowed AWS and DynamoDB to autoscale while measuring the maximum concurrency reached by AWS Lambda.

8.1. Coordination overhead

In the first experiment, the performance of StateFun with coordinator functions is compared against the original on non-transactional workloads to see how much computational overhead the coordination logic has added. In Fig. 5 we show the maximum throughput achieved by the two systems for a varying number of keys. While in Fig. 6 we show the different latencies for the systems across `read` and `write` workloads at different throughputs and numbers of keys.

Throughput. The first observation we can make is that there is a 20% decrease in throughput in the case of 100 keys that plateaus to 10% as the number of keys increases. The decreased performance is because of the batching mechanism being more complex than the original append-only approach by enforcing isolated function invocations as part of a two-phase commit transaction. In addition, coordinator functions keep track of transaction progress, which incurs some overhead. Another observation is that there is no noticeable throughput difference between workloads with only `read` or `write` operations. The reason behind this behavior is that, in StateFun, both operations need to access the remote function, making the communication layer the bottleneck.



Keys	Transfer proportion	Deadlocks / transfer ops
100	0.25	9/12014 (0.07%)
	0.5	27/24107 (0.11%)
	0.75	82/35875 (0.22%)
5000	0.25	0/60121
	0.5	0/120089
	0.75	0/179794

(a) Time regular locks are held (b) Time to detect deadlocks (c) Frequency of deadlocks

Fig. 8. Details of locking behavior for a workload for 100 and 5000 keys with various proportions of transfers without rollbacks. The boxplots show the 5th and 95th percentiles.

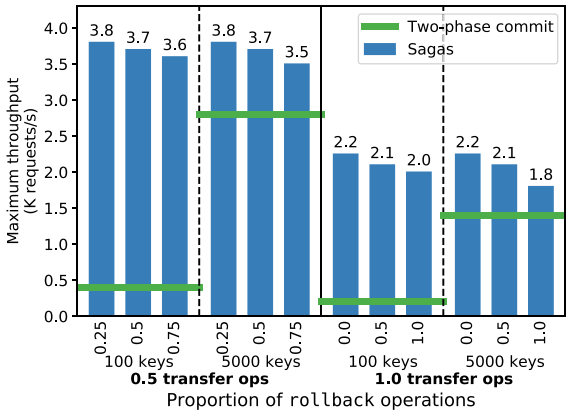


Fig. 9. Throughput with different proportions of rolled back `transfer` operations for workloads with 50% and 100% `transfer` operations.

Latency. The latencies in Fig. 6 are approximately 20% higher for our version of StateFun for `read` workloads. However, as the number of keys increases, the difference becomes smaller, towards 7%. This decrease in performance is due to the additional logic required for function coordination. Another interesting observation is the indifference in performance for `write` workloads. The reason is that StateFun batches every read operation before serialization, adding up over time for larger batches. In contrast, only the last version needs to be serialized for writes. Additionally, serialization happens at the remote function for both types of operations, explaining why it does not affect throughput, but it does affect latency.

Finally, we consider the introduced overhead as a reasonably low price to pay for having full-fledged transaction execution primitives added to the system.

8.2. Sagas vs two-phase commit

The second experiment shows a performance comparison between the two implemented transaction protocols, their impact on the maximum throughput in perfect conditions (Fig. 7), and with failures, measuring the impact of locking for the two-phase commit (Fig. 8) and of rollbacks (Fig. 9) for the Saga protocols. In these experiments, we set a certain proportion of the workload to be `transfer` operations and the remaining proportion is equally shared between `read` and `write` operations. In our case, each `transfer` operation causes three remote function invocations (coordinator function and one function per account holder taking part in the transfer). When evaluating two-phase commit functions, we do not include messages sent to detect deadlocks in the total number of invocations. Therefore, the indicator should be considered a lower bound on the actual number of messages. Finally, we used a uniform key access distribution for these experiments. At the same time, in some real-world scenarios, this can be skewed (e.g., lots of transactions on very active accounts vs. a long tail of inactive ones).

Fig. 7 plots the achieved throughput against the absolute number of `transfer` operations in the workload with a varying number of keys given that the benchmark provided the accounts enough balance to ensure all transactions succeeded. It also displays indicators for the absolute amount of total internal function invocations, considering additional internal invocations required for transactions and the absolute amount of total remote function invocations. We observe that Sagas perform much better than two-phase commit for a few keys (100 and 2000). This happens for two reasons: i) Sagas can still benefit from the batching mechanism of StateFun since they do not require isolation, and (ii) the locking in two-phase commit severely limits the throughput. However, it is also interesting that two-phase commit performs comparably to Sagas for a higher number of keys (5000–10000) even though it provides much stronger guarantees. This is because there is less contention on a single function, decreasing the effect of locking, while batching provides no benefits, as also shown in Fig. 5. A second observation from Fig. 7 is that the total function invocations still drop when the proportion of transactions increases. This is because the total function invocations account for the additional messaging required to coordinate transactions, leading to the overall throughput of workloads with a high proportion of `transfer` operations being relatively low.

Locking overhead. In Fig. 8 we measure the behavior of locking and deadlocks that accompany the two-phase commit protocol. The lock duration is measured between the point in time where the function instance sends the response to the `prepare` message and when it either receives a `commit` or `abort` message, sending the next batch to the remote function. In Fig. 8(a), we see little to no difference in the median across the different workloads but, when the proportion of `transfer` operations is higher, the higher percentiles increase significantly. Next, we want to measure the deadlock frequency, and Fig. 8(b) shows the number of deadlocks against the total number of `transfer` operations in the workload. As expected, there are no deadlocks for workloads on 5000 keys since the contention is low. For 100 keys, we observe an increasing number of deadlocks while the proportion of `transfer` operations increase. However, the percentage of deadlocks across all `transfer` operations is still small. Finally, Fig. 8(c) shows the time it takes to detect a deadlock, i.e., perform the Chandy–Misra–Haas algorithm. We observe that the median of the time this takes is similar across all workloads, and it also shows that as the amount of `transfer` operations increases, so do the higher percentile times.

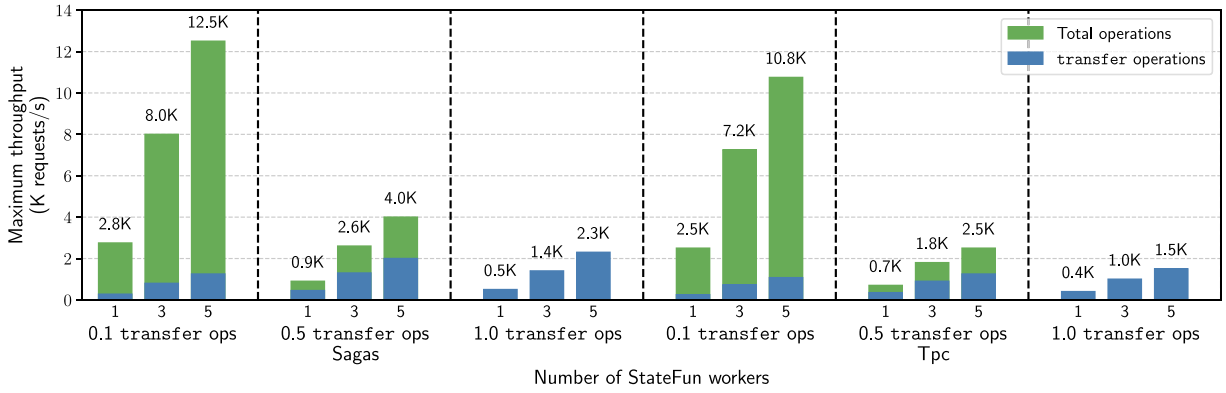
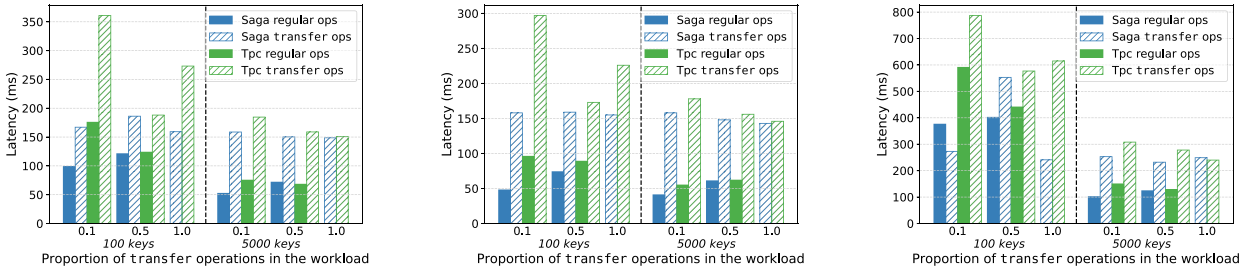


Fig. 10. Maximum throughput for the system with 5000 keys for different numbers of StateFun workers for workloads with different proportions of `transfer` operations.



(a) Mean (b) Median (c) 95th percentile

100 keys						5000 keys					
Sagas			Tpc			Sagas			Tpc		
0.1	0.5	1.0	0.1	0.5	1.0	0.1	0.5	1.0	0.1	0.5	1.0
11K	3K	2K	1.5K	0.38K	0.16K	9K	3K	2K	8K	2K	1.2K

(d) Throughputs at which latency was measured

Fig. 11. Graph comparing latencies for Sagas and two-phase commit coordinator function for different keys and transaction proportions in the workload at 80% of the respective maximum throughputs.

Rollback overhead. Fig. 9 shows the maximum throughput for workloads with 50% and 100% `transfer` operations where different proportions of `transfer` operations fail for Sagas and two-phase commit coordinator functions. As expected, when using two-phase commit, a rollback does not increase the load in the system because the coordinator function needs to send a second message either way. Again, nothing out of the ordinary happened as the proportion of `transfer` operations to be rolled back increased. The throughput decreased as the protocol required additional compensating messages to be sent in the system. However, with 5000 keys, the difference is small at 50% `transfer` operations: 8% when going from 25 to 75% rollbacks and increasing to 18% with 100% `transfer` operations. This is larger than the 100 keys case that can still leverage the batching mechanism of StateFun and limit the performance drop to 10% in the worst case. Still, no matter the decrease in performance due to the compensating actions of the Saga protocol, it remains 20% faster than two-phase commit in the worst-case scenario of 5000 keys and 75% rollbacks.

8.3. Scalability comparison

In the last experiment, we evaluated the scalability of the proposed system with coordinator functions. In Fig. 10 we display the maximum throughput for both two-phase commit and Sagas at different amounts of StateFun workers and different transaction proportions in the workload. For Sagas, the scalability from 1 to 5 workers is close to 90% throughout for all workloads. For two-phase commit, the scalability from 1 to 5 workers starts at 87% at 10% `transfer` operations and drops to 75% at 100% `transfer` operations.

The reason for the low decrease in scalability on both protocols is that as workers increase, more traffic needs to go over the network. In the Sagas' case, the efficiency does not decrease across all workloads for the same reasons as expressed in Section 8.2. Namely, the system can still utilize batching, no locking is required, and the number of messages is two times lower than the two-phase commit protocol when all transactions succeed. On the other hand, the 8% decrease in scalability in two-phase commit from 10% to 100% `transfer` operations is due to the protocol's requirements for locks, more messages, and the inability to use batching. Considering all the impeding factors, it still achieves decent efficiency with strong consistency guarantees in fully transactional workloads.

8.4. Micro benchmark

As a final experiment, we conduct a microbenchmark on the system. At first, we keep the number of resources fixed, and then for every `transfer` proportion and number of keys, we measure the throughput at 80% load and the corresponding latency. By the results presented in Fig. 11 we can see that for a use case

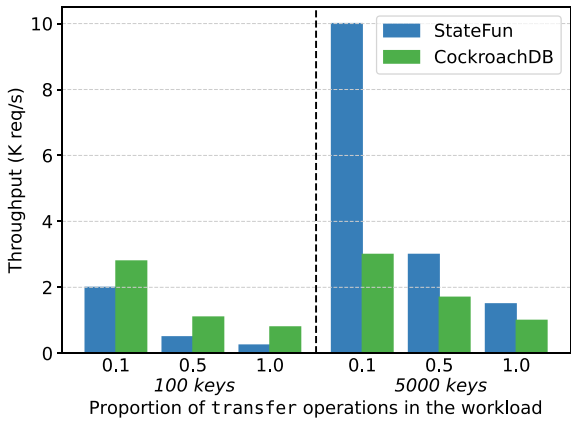


Fig. 12. Comparing the maximum throughput of CockroachDB and Flink StateFun for workloads with different proportions of `transfer` operations (the remaining operations are `read` or `update` operations with equal probability). Both systems are deployed with 3 instances, each with 4 CPUs.

Table 2
Latency compared for StateFun and CockroachDB, each system was ran at 80% of the maximum throughput measured as shown in Fig. 12.

100 keys				
Operations	**StateFun**		**CockroachDB**	
	Median	95th %tile	Median	95th %tile
`Transfer` (0.1)	297	787	48	86
`Non-transfer`	96	591	47	79
`Transfer` (0.5)	173	577	22	72
`Non-transfer`	89	441	17	68
`Transfer` (1.0)	226	615	41	114
5000 keys				
Operations	**StateFun**		**CockroachDB**	
	Median	95th %tile	Median	95th %tile
`Transfer` (0.1)	178	308	36	113
`Non-transfer`	55	150	21	107
`Transfer` (0.5)	156	278	21	64
`Non-transfer`	62	129	11	62
`Transfer` (1.0)	146	240	43	78

with a low number of keys, the Sagas beat by a large margin the two-phase commit protocol in both throughput, with more than a 650% increase in performance, and latency that is at least two times lower. The contention becomes less of a problem for a larger number of keys. We observe a smaller difference between the two protocols at around 40% on average for throughput and a stable difference in latency around 20%. To conclude, Sagas seems to be the obvious choice for a few keys or high contention; if the business logic permits it. In any other case, the choice is mainly about the consistency guarantee requirements since the difference is not that significant.

8.5. Comparison against CockroachDB

We compare the performance of StateFun against a production-grade distributed database, CockroachDB, in terms of throughput and latency. Due to the fundamental differences between the two systems, this is merely a reference comparison. In this experimental setting, the input requests consist of a varying proportion of transactional and non-transactional requests. We signify transactional requests as transfer operations and non-transactional requests as non-transfer operations.

As Fig. 12 shows, CockroachDB outperforms StateFun in terms of throughput by a constant factor when transactions are evoked on a small number of unique keys. In addition, this experiment configuration examines the performance of the two systems when there is high lock contention since subsequent transactions on the same key have to wait for previous ones to complete. Notably, the performance difference in terms of throughput remains the same while the proportion of transactions in the input request set increases from 0.1 to 0.5 to 1. When there are many keys, e.g., 5000, StateFun outperforms CockroachDB. In fact, for a small proportion of transactions (0.1), StateFun achieves four times more throughput. As the number of transactions grows, the performance difference shrinks. These results can be explained by a more sophisticated or aggressive batching mechanism that allows StateFun to batch non-transactional requests efficiently. When there are many non-transactional requests, the effect of batching provides a significant performance advantage, which is shrinking as the number of non-transactional requests becomes smaller.

On the other hand, CockroachDB is superior in terms of latency performance as Table 2 depicts. Both median latency and latency at the 95th percentile are roughly six times better on average than StateFun's in all configurations. This result can be explained because CockroachDB is run with default settings, and there is no batching implemented at the application level. This contributes to lower throughput, but it also favors lower latency. On the other hand, StateFun can inherently apply batching at several points in the system, such as when (i) it sends a request to a remote function, (ii) fetches requests from Kafka, and (iii) produces responses to Kafka.

In summary, CockroachDB seems more suitable for handling skewed transactional workloads, although the performance improvement against StateFun is constant in terms of throughput. Thus, a potential superiority based on the locking mechanism of CockroachDB is capped and does not result in a scalable advantage. Furthermore, CockroachDB replicates data three times, leading to additional overhead but providing the capacity to serve requests even in the case of node failures. On the other hand, StateFun provides no replication and needs to recover from a checkpoint following a node failure. On the other hand, StateFun can leverage its sophisticated batching mechanism to drive significantly better throughput for workloads containing a modest number of transactions. Notably, while CockroachDB supports full transactional SQL and StateFun supports only one-shot functions, due to the simplicity of the workload, the feature set should not have a significant impact on performance. In addition, the executed workloads allow for less locking and more batching. Finally, CockroachDB demonstrates reliably low latency in all configurations, roughly six times lower than StateFun.

8.6. Comparison against Beldi

We also compare StateFun with a stateful function as a service library and runtime, Beldi, which runs on AWS Lambda and uses DynamoDB as backend storage for transactions. Because of the intricacies of the serverless environment and the restricted way it can be configured, we limit our comparison to latency performance given a fixed amount of throughput requests since we have limited visibility to the number of resources used by Beldi. AWS only exposes the concurrency level of the Lambda functions and allows restricting that to a max number. In Table 3 max concurrency refers to the max concurrency utilized by AWS Lambdas. Max concurrency was fairly stable throughout each experiment. Notably, there is no information regarding the specification of the underlying hardware that is used.

Furthermore, even latency performance does not provide a fair comparison because Beldi only measures latency from when

Table 3
Comparison between latencies of Beldi and StateFun.

100 keys							
Operations	**Throughput**	**StateFun**			**Beldi**		
		CPU	Median	95th %tile	Max. concurrency	Median	95th %tile
`Transfer` (0.1)	1.5K	80	298	723	128	49	739
`Non-transfer`			99	572		83	693
`Transfer` (1.0)	0.16K	80	223	532	182	91	724
5000 keys							
Operations	**Throughput**	**StateFun**			**Beldi**		
		CPU	Median	95th %tile	Max. concurrency	Median	95th %tile
`Transfer` (0.1)	8K	80	184	287	1000[a]	123	174
`Non-transfer`			52	184		50	77
`Transfer` (1.0)	1.2K	80	146	273	902	114	847

[a]Experiment is throttled and runs at a lower throughput ≈4K, i.e., experiment lasted longer.

a request's execution starts until the time it completes without considering the amount of time spent for routing and waiting in an input queue before the request's execution begins. Consequently, a performance throttle in Beldi due to excess load will not show in the measured latency. We try to compensate for this by measuring the experiment's completion time and estimating Beldi's actual throughput. On the other hand, we run StateFun in an IaaS cloud infrastructure where we provide it with a specific amount of computational resources and measure latency end-to-end. The disparity between the two infrastructures and experimental settings limits the insights that can be extracted.

Table 3 shows the experimental results, from which we draw two notable observations regarding latency. For non-transfer operations, regardless of the number of keys, StateFun and Beldi achieve the same level of low-latency performance. Beldi demonstrates 2–3 times superior performance in terms of median latency for transfer operations, while tail latency at the 95th percentile suggests no important differences between the two systems. In Beldi, latency only captures delays that are internal to the system, which may be owed to lock contention inside Beldi, communication stalls between Lambda functions and DynamoDB, as well as queuing in DynamoDB. Unfortunately, it is impossible to pinpoint the exact factors and their merit in the observed tail latency.

Lastly, an important factor in the experiments is that Beldi is let free to auto-scale up to 1000 concurrently executing functions. This aggressive availability of resources far exceeds the 80 CPUs given to the remote functions executing on StateFun. Interestingly, when the number of unique keys is large, meaning that lock contention is low, this level of concurrency is not adequate to accommodate the input throughput of 8K requests per second. In this case, AWS Lambdas used all the available concurrency, and the execution of requests was throttled, waiting for CPUs to become available. Given the experiment's duration, we approximated the level of throughput achieved by Beldi at 4K requests per second. Note that Beldi's latency remains unaffected since it does not account for external delays, such as queuing. On the other hand, we observe that when the number of unique keys is small, meaning that lock contention is high, Beldi is quite efficient. It used more concurrency than what was available to StateFun, but at the same overall level of magnitude. Beldi's efficiency is probably owed to juggling between requests that can execute immediately and others that should be put to sleep until they can get hold of the lock they require to proceed.

Finally, the observed performance of Beldi does not account for garbage collection. Beldi features a garbage collector to shrink its transaction log periodically, but the garbage collector does not need to run during the presented experiments because their duration is too short. In general, however, the garbage collector is expected to add overhead not represented in our set of experiments.

9. Related work

SFaaS systems. SFaaS has been a very active area in both research and the open-source community. From the research community, the most relevant work is Beldi [21] which, like AFT [43], builds on top of Amazon's AWS Lambda to add fault tolerance and transaction support allowing for more complex state management. Their principal difference is that Beldi's execution environment is completely serverless, while AFT relies on external servers for transaction support. To make that happen, Beldi uses atomic logging, extending Olive [44], to ensure fault tolerance for read and write operations, with garbage collection to manage the logs' growth. Regarding transactions, Beldi supports a variant of the two-phase commit protocol enforcing strong consistency guarantees with wait-die deadlock prevention. Cloudburst with Hydrocache [23] provides causal consistency guarantees within the same DAG workflow backed by Anna [24], a key–value state backend. Another promising SFaaS system, FAASM [45], supports direct memory access between functions while maintaining isolation and speeds up initialization times compared to containers. At the time of writing, FAASM does not provide transactional support. Finally, the two most prominent open-source SFaaS projects are Cloudstate[8], based on stateful actors, and Apache Flink StateFun, which is presented in detail in Section 3.2. In Cloudstate, communication is allowed between different actors within the same cluster and between user-defined functions over gRPC with at-least-once processing guarantees.

Transactional programming model. The most notable difference among these systems in terms of programming model is state access. Both StateFun and Cloudstate encapsulate state within a specific function instance. In contrast, Cloudburst and Beldi allow any function access to any state stored in Anna or DynamoDB, respectively. Regarding transactions, only Beldi offers a programming model where the programmer writes two markers (begin/end_tx), and every function invocation in between will execute as part of a transaction. Our contribution is a programming model that supports transactions on StateFun with the choice of strong or relaxed consistency guarantees as shown in Section 4.1.

Stream processing transactions. Furthermore, transactions on top of stream processing systems have received some attention in the literature. In [46] the authors introduce a transactional model over both data streams and traditional tabular data. Following a similar model, in [47] the authors add guarantees for snapshot isolation and consistency across partitioned state. Then TSpoon [48], an extension of FlowDB [32]), proposes a data

[8] https://cloudstate.io/.

management system built on top of a stream processor that supports transactions, giving the option of both strong and weak transactional guarantees and queryable state. Our work focuses on transactional workflows between generic stateful functions executed on a serverless dataflow system.

Distributed databases. The mentioned stream processing systems share the same main goal as distributed databases [49–54], that is, how to scale to multiple machines while providing serializable transactional guarantees. This is an old problem in database research. The R* system [49] was one of the first to try the two-phase commit protocol with distributed deadlock detection. Then more recent approaches like H-store [50] showed that distributed database solutions could provide both very high performance and transactional guarantees when transactions touch a single partition. Currently, research in distributed databases revolves around globally distributed databases with Spanner [51] introducing serializable transactions using a timestamp mechanism across all locations/machines based on atomic clocks. Furthermore, approaches like Carousel [52] and SLOG [53] improve globally distributed database transactions. Carousel enhances transaction execution by minimizing network usage, while SLOG offers a fine-grained transaction protocol based on the proximity between the data and the client. Finally, CockroachDB [54] provides serializable globally distributed transactions without a complicated time mechanism.

Benchmarks. The large variety of use cases and systems makes them difficult to compare using a standardized benchmark. The related benchmarks that could be used to evaluate SFaaS systems are the *Yahoo! Cloud Serving Benchmark* (YCSB) [41] and the DeathStarBench [55]. Given that StateFun is based on Flink, which is a stream processing system, a stream processing benchmark [56] would be another alternative. However, its workloads are not representative of those executed by an SFaaS system. In addition, we did not consider TPC-C [57] because it was created to test relational database management systems, including transactions, and requiring many additional features not present in SFaaS. We ultimately chose to develop and use an extension of YCSB [42] that introduced explainable transactional workloads, allowing for an easier interpretation of the results.

10. Discussion & open problems

Programming models for the cloud. Although the stateful dataflow model has been very successful as an execution model, it has not been leveraged thus far as an intermediate representation. Historically, MapReduce/Hadoop [58] and Dryad [59] were first proposed as a means of authoring and executing distributed data-parallel applications using high-level language constructs, such as Java functions and LINQ [60] respectively. Many systems have followed that execution model subsequently, including streaming dataflow systems such as Apache Storm [61], Flink [14], Naiad [62]. However, none of these systems could execute general-purpose cloud applications; their programming model focuses on distributed collection processing and adopts a functional programming API.‘

Dataflows for cloud applications. We believe that abstractions such as stateful functions can play the role of a high-level programming model for dataflow engines and have a high impact on cloud programming. The current approach to program in the cloud is to either use domain-specific languages (DSLs) such as Bloom [63], Hilda [64] and Erlang [2], or as libraries in within mainstream languages like Akka [3], Spring Boot ()). The main observation here is that the programmer either has to learn a new domain-specific language, or they have to use libraries that leak implementation details to the business logic. Very close to the spirit of this work are virtual actors, and Orleans [4,5] from which Apache Flink's StateFun drew inspiration. In turn, Orleans and virtual actors draw their inspiration from Pat Helland's *entities* [6]. However, Orleans requires a specialized runtime and does not offer exactly-once function execution. As we show in this paper, implementing very complex protocols (with lots of corner cases) can be simpler since we benefit from the state management and exactly-once guarantees of modern dataflow systems. Since dataflow systems nowadays are well understood, scalable and consistent, we believe that they will play a critical role in the future of execution engines for the cloud.

Future dataflow systems. However promising they can be, dataflow engines still suffer from several issues. Stream processors such as Apache Flink [14], or Jet [20] have been designed for continuous operation on high-throughput streams. However, stateful functions have very different workload characteristics. For instance, lots of cloud applications may have to call external services – a source of non-determinism [16], and functions calling other functions, expecting return values, introduce cycles in the dataflow graph. Current dataflow systems either do not support cycles or support a few special cases of cycles. This is because cycles can cause deadlocks and various other issues [15,65] that need to be dealt with. Finally, in this paper, we introduced transactions at the function level without having to touch the core of Apache Flink's dataflow engine. However, proper implementation of transactions would require the dataflow system itself to be aware of transaction boundaries (e.g., commit, prepare) and incorporate transaction processing into its fault-tolerance protocol. We think that more research needs to be performed to get dataflow systems fully capable of leveraging their potential.

11. Conclusions

In this paper, we tackle the problem of supporting transactional workflows across cloud applications on a serverless platform. This problem is notorious in the microservices and cloud applications landscape. In addition to that, we introduced a programming model and corresponding implementation for authoring workflows across stateful serverless functions with configurable transactional guarantees. Developers can opt for a distributed transaction across functions with strict atomicity and consistency guarantees or a Saga workflow that provides eventual atomicity and consistency. These complementary alternatives faithfully represent the requirements of real-world use cases. We described our implementation on top of Apache Flink StateFun, an open-source production-grade serverless sFaaS platform, and evaluated our implementation on an extended version of the YCSB benchmark that we developed in terms of (a) throughput and latency overhead against the original StateFun, (b) performance efficiency between distributed transactions and Saga workflows, and (c) scalability. We found that our transactional workflows add affordable overhead to the system around 10%, Sagas significantly outperform distributed transactions on a scale of 15%–34% depending on the amount of ongoing transactional workflows in the system, and scalability manifests a factor of 90% for Sagas compared to 75%–87% for two-phase commit. Furthermore, our comparison against a serverless SFaaS runtime showed that our work could achieve higher throughput, but it also incurs higher latency. Finally, we compared against a popular distributed database, CockroachDB, which achieved better performance in high contention scenarios and in terms of latency. Notably, our work achieved better results in sparse key distributions, while it provides exactly-once processing semantics compared to our deployment of Kafka with a CockroachDB backend at-least-once semantics.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work has been partially funded by the H2020 project OpertusMundi No. 870228 and the AI for Fintech ICAI lab. Experiments were carried out on the Dutch national e-infrastructure with the support of SURF Cooperative.

References

[1] C.D. Krumvieda, Distributed ML: Abstracts for Efficient and Fault-Tolerant Programming, Cornell University, 1993.
[2] J. Armstrong, Programming Erlang: Software for a Concurrent World, Pragmatic Bookshelf, 2013.
[3] D. Wyatt, Akka Concurrency, Artima Incorporation, 2013.
[4] S. Bykov, A. Geller, G. Kliot, J.R. Larus, R. Pandya, J. Thelin, Orleans: cloud computing for everyone, in: SoCC, 2011.
[5] P. Bernstein, S. Bykov, A. Geller, G. Kliot, J. Thelin, Orleans: Distributed Virtual Actors for Programmability and Scalability, MSR-TR, 2014.
[6] P. Helland, Life beyond distributed transactions: an apostate's opinion, in: ACMQueue, 2016.
[7] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, et al., Cloud programming simplified: A berkeley view on serverless computing, 2019, arXiv.
[8] J.M. Hellerstein, J.M. Faleiro, J.E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, C. Wu, Serverless computing: One step forward, two steps back, 2018, CoRR.
[9] A. Akhter, M. Fragkoulis, A. Katsifodimos, Stateful functions as a service in action, in: VLDB, 2019.
[10] S.S. de Toledo, A. Martini, A. Przybyszewska, D.I.K. Sjøberg, Architectural technical debt in microservices: A case study in a large company, in: Proceedings of the Second International Conference on Technical Debt, in: TechDebt '19, IEEE Press, 2019, pp. 78–87.
[11] T. Killalea, The hidden dividends of microservices: Microservices aren't for every company, and the journey isn't easy, ACM Queue 14 (3) (2016) 25–34.
[12] R. Laigner, Y. Zhou, M.A.V. Salles, Y. Liu, M. Kalinowski, Data management in microservices: State of the practice, challenges, and research directions, Proc. VLDB Endow. 14 (13) (2021) 3348–3361.
[13] L. Lamport, R. Shostak, M. Pease, The Byzantine generals problem, ACM Trans. Program. Lang. Syst. 4 (3) (1982) 382–401.
[14] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, K. Tzoumas, Apache flink TM : Stream and batch processing in a single engine, IEEE Data Eng. Bull. (2015).
[15] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, K. Tzoumas, State management in apache flink: Consistent stateful distributed stream processing, in: VLDB, 2017.
[16] P. Silvestre, M. Fragkoulis, D. Spinellis, A. Katsifodimos, Clonos: Consistent causal recovery for highly-available streaming dataflows, in: SIGMOD, 2021.
[17] P. Carbone, M. Fragkoulis, V. Kalavri, A. Katsifodimos, Beyond analytics: The evolution of stream processing systems, in: SIGMOD, 2020.
[18] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, M. Zaharia, Structured streaming: A declarative API for real-time applications in apache spark, in: SIGMOD, 2018.
[19] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, S. Whittle, Millwheel: fault-tolerant stream processing at internet scale, in: VLDB, 2013.
[20] C. Gencer, M. Topolnik, V. Ďurina, E. Demirci, E.B. Kahveci, A.G.O. Lukáš, J. Bartók, G. Gierlach, F. Hartman, U. Yılmaz, M. Doğan, M. Mandouh, M. Fragkoulis, A. Katsifodimos, Hazelcast jet: Low-latency stream processing at the 99.99th percentile, in: VLDB, 2021.
[21] H. Zhang, A. Cardoza, P.B. Chen, S. Angel, V. Liu, Fault-tolerant and transactional stateful serverless workflows, in: OSDI, 2020.
[22] V. Sreekanti, C. Wu, X.C. Lin, J. Schleier-Smith, J.E. Gonzalez, J.M. Hellerstein, A. Tumanov, Cloudburst, in: VLDB, 2020.
[23] C. Wu, V. Sreekanti, J.M. Hellerstein, Transactional causal consistency for serverless computing, in: SIGMOD, 2020.
[24] C. Wu, J. Faleiro, Y. Lin, J. Hellerstein, Anna: A KVS for any scale, in: ICDE, 2018.
[25] H. Garcia-Molina, K. Salem, Sagas, in: ACM Sigmod Record, 1987.
[26] J.N. Gray, Notes on data base operating systems, in: R. Bayer, R.M. Graham, G. Seegmüller (Eds.), Operating Systems: An Advanced Course, Springer Berlin Heidelberg, 1978.
[27] M. de Heus, K. Psarakis, M. Fragkoulis, A. Katsifodimos, Distributed transactions on serverless stateful functions, in: DEBS, 2021.
[28] J. Goldstein, A. Abdelhamid, M. Barnett, S. Burckhardt, B. Chandramouli, D. Gehring, N. Lebeck, C. Meiklejohn, U.F. Minhas, R. Newton, et al., Ambrosia: Providing performant virtual resiliency for distributed applications, in: VLDB, 2020.
[29] A. Cheung, N. Crooks, J.M. Hellerstein, M. Milano, New directions in cloud programming, in: CIDR, 2021.
[30] R.C. Fernandez, M. Migliavacca, E. Kalyvianaki, P. Pietzuch, Making state explicit for imperative big data processing, in: USENIX ATC, 2014.
[31] A. Katsifodimos, M. Fragkoulis, Operational stream processing: Towards scalable and consistent event-driven applications, in: EDBT, 2019.
[32] L. Affetti, A. Margara, G. Cugola, Flowdb: Integrating stream processing and consistent state management, in: DEBS, 2017.
[33] D. Terry, Transactions and Scalability in Cloud Databases—Can't We Have Both? USENIX Association, 2019.
[34] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E.P.C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, D.J. Abadi, H-store: A high-performance, distributed main memory transaction processing system, in: VLDB, 2008.
[35] S. Sivasubramanian, Amazon dynamodb: a seamlessly scalable non-relational database service, in: SIGMOD, 2012.
[36] K.M. Chandy, J. Misra, L.M. Haas, Distributed deadlock detection, in: ACM Trans. Comput. Syst., 1983.
[37] D.J. Abadi, J.M. Faleiro, An overview of deterministic database systems, Commun. ACM (2018).
[38] A. Thomson, D.J. Abadi, The case for determinism in database systems, in: VLDB, 2010.
[39] K. Ren, A. Thomson, D.J. Abadi, An evaluation of the advantages and disadvantages of deterministic database systems, in: VLDB, 2014.
[40] A. Thomson, D.J. Abadi, The case for determinism in database systems, in: VLDB, 2010.
[41] B.F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears, Benchmarking cloud serving systems with YCSB, in: SOCC, 2010.
[42] A. Dey, A. Fekete, R. Nambiar, U. Röhm, YCSB+T: Benchmarking web-scale transactional databases, in: ICDE Workshops, 2014.
[43] V. Sreekanti, C. Wu, S. Chhatrapati, J.E. Gonzalez, J.M. Hellerstein, J.M. Faleiro, A fault-tolerance shim for serverless computing, in: EuroSys, 2020.
[44] S. Setty, C. Su, J.R. Lorch, L. Zhou, H. Chen, P. Patel, J. Ren, Realizing the fault-tolerance promise of cloud storage using locks with intent, in: OSDI, 2016.
[45] S. Shillaker, P. Pietzuch, Faasm: Lightweight isolation for efficient stateful serverless computing, in: 2020 USENIX Annual Technical Conference, USENIX ATC 20, 2020.
[46] I. Botan, P.M. Fischer, D. Kossmann, N. Tatbul, Transactional stream processing, in: EDBT, 2012.
[47] P. Götze, K.-U. Sattler, Snapshot isolation for transactional stream processing., in: EDBT, 2019.
[48] L. Affetti, A. Margara, G. Cugola, TSpoon: Transactions on a stream processor, J. Parallel Distrib. Comput. (2020).
[49] C. Mohan, B. Lindsay, R. Obermarck, Transaction management in the r* distributed database management system, in: TODS, 1986.
[50] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E.P. Jones, S. Madden, M. Stonebraker, Y. Zhang, et al., H-store: a high-performance, distributed main memory transaction processing system, in: VLDB, 2008.
[51] J.C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J.J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al., Spanner: Google's globally distributed database, in: TOCS, 2013.
[52] X. Yan, L. Yang, H. Zhang, X.C. Lin, B. Wong, K. Salem, T. Brecht, Carousel: Low-latency transaction processing for globally-distributed data, in: SIGMOD, 2018.
[53] K. Ren, D. Li, D.J. Abadi, Slog: Serializable, low-latency, geo-replicated transactions, in: VLDB, 2019.
[54] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, et al., Cockroachdb: The resilient geo-distributed sql database, in: SIGMOD, 2020.
[55] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, et al., An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems, in: ASPLOS, 2019.
[56] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, V. Markl, Benchmarking distributed stream data processing systems, in: ICDE, 2018.
[57] F. Raab, TPC-C - The standard benchmark for online transaction processing (OLTP), in: J. Gray (Ed.), The Benchmark Handbook, Morgan Kaufmann, 1993.
[58] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Commun. ACM (2008).

[59] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. Gunda, J. Currey, DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language, in: OSDI, 2008.
[60] E. Meijer, B. Beckman, G. Bierman, Linq: reconciling object, relations and xml in the. net framework, in: SIGMOD, 2006.
[61] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J.M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al., Storm@ twitter, in: SIGMOD, 2014.
[62] D.G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, M. Abadi, Naiad: a timely dataflow system, in: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, 2013.
[63] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J.M. Hellerstein, R. Sears, Boom analytics: exploring data-centric, declarative programming for the cloud, in: Proceedings of the 5th European Conference on Computer Systems, 2010.
[64] F. Yang, J. Shanmugasundaram, M. Riedewald, J. Gehrke, Hilda: A high-level language for data-drivenweb applications, in: 22nd International Conference on Data Engineering, ICDE'06, IEEE, 2006.
[65] A. Lattuada, F. McSherry, Z. Chothia, Faucet: a user-level, modular technique for flow control in dataflow engines, in: SIGMOD BeyondMR Workshop, ACM, 2016.