Executing Microservice Applications on Serverless, Correctly

KONSTANTINOS KALLAS*, University of Pennsylvania, USA
HAORAN ZHANG*, University of Pennsylvania, USA
RAJEEV ALUR, University of Pennsylvania, USA
SEBASTIAN ANGEL, University of Pennsylvania & Microsoft Research, USA
VINCENT LIU, University of Pennsylvania, USA

While serverless platforms substantially simplify the provisioning, configuration, and management of cloud applications, implementing correct services on top of these platforms can present significant challenges to programmers. For example, serverless infrastructures introduce a host of failure modes that are not present in traditional deployments. Individual serverless instances can fail while others continue to make progress, correct but slow instances can be killed by the cloud provider as part of resource management, and providers will often respond to such failures by re-executing requests. For functions with side-effects, these scenarios can create behaviors that are not observable in serverful deployments.

In this paper, we propose µ2sls, a framework for implementing microservice applications on serverless using standard Python code with two extra primitives: transactions and asynchronous calls. Our framework orchestrates user-written services to address several challenges, such as failures and re-executions, and provides formal guarantees that the generated serverless implementations are correct. To that end, we present a novel service specification abstraction and formalization of serverless implementations that facilitate reasoning about the correctness of a given application's serverless implementation. This formalization forms the basis of the µ2sls prototype, which we then use to develop a few real-world microservice applications and show that the performance of the generated serverless implementations achieves significant scalability (3-5× the throughput of a sequential implementation) while providing correctness guarantees in the context of faults, re-execution, and concurrency.

CCS Concepts: • **Software and its engineering** → *Correctness*; **Distributed programming languages**; *Semantics*; • **Information systems** → *Web services*; • **Computer systems organization** → *Cloud computing*.

Additional Key Words and Phrases: microservices, stateful serverless, transactions

ACM Reference Format:
Konstantinos Kallas, Haoran Zhang, Rajeev Alur, Sebastian Angel, and Vincent Liu. 2023. Executing Microservice Applications on Serverless, Correctly. *Proc. ACM Program. Lang.* 7, POPL, Article 13 (January 2023), 29 pages. https://doi.org/10.1145/3571206

1 INTRODUCTION

For today's internet and web services, a dominant architecture is that of *microservices*. Consider a service like Facebook or a similar social network. While each user's initial request may get routed to a single frontend web server, an army of other machines and services—typically on the order of hundreds—are also necessary to assemble the newsfeed, execute real-time advertisement auctions, query a friend graph, and eventually compose the final response [Nazir et al. 2008]. Other modern services (e.g., Google, Uber, etc.) exhibit similar patterns. Structuring applications as microservices promotes modular design, quick iteration, and simple reuse of well-defined application pieces.

*The two marked authors contributed equally to the paper.

2475-1421/2023/1-ART13
https://doi.org/10.1145/3571206

While developers might provision and manage a fixed number of machines (servers) to run each of the above tasks (listening for incoming RPC requests in a loop, accepting them, processing them, fetching any relevant state from a backend database, and returning a result), microservice tasks are increasingly deployed using so-called *serverless infrastructure*. This style of service architecture—enabled by cloud platforms like AWS Lambda, Azure Functions, and Google Cloud Functions—allows programmers to break down their application into small units of functionality that cloud providers then automatically distribute over the machines in their data centers. This frees programmers from worrying about the prosaic but difficult tasks associated with provisioning, scaling, and maintaining the underlying computational, storage, and network resources of the system; they, instead, only need to focus on the logic executed on each request. This arrangement is also particularly beneficial for customers from a business perspective because it outsources all liability to the service provider: the cost of provisioning or load balancing mistakes is shouldered entirely by the cloud provider—the customer only pays for the resources used when the functions execute.

While the above operational and cost benefits make serverless architectures an attractive alternative to traditional server-oriented deployment, implementing services on the new paradigm can be challenging due to the fundamental differences in the architectures particularly when manipulating state and recursion. For example, serverless instances are ephemeral, and therefore all persistent data of an application needs to be stored and managed in external storage. Furthermore, serverless infrastructures introduce a host of failure modes not present in traditional microservice deployments. Individual serverless instances can fail while others continue to make progress, correct but slow instances can be killed by the cloud provider as part of resource management, and providers will often respond to such failures by re-executing requests. In functions with side-effects, these scenarios can create behaviors that are not observable in serverful deployments.

Indeed, many of the early successes in serverless computing were relatively simple applications that could execute without the aid of either persistent state or supporting services and, thus, could sidestep the above issues. However, recent academic work on stateful serverless frameworks [Jia and Witchel 2021a,b; Sreekanti et al. 2020b; Zhang et al. 2020a,b] suggests that serverless computing has broader applicability than those early use cases and that better programmer support is valuable. This is not just an academic exercise: industry is investing significant resources in stateful serverless as well [Amazon 2020; Bonér 2020; Burckhardt et al. 2021; Ray 2022; Temporal 2022].

To address the above challenges, we propose μ2sls, a framework for correctly implementing microservice applications on serverless. Our framework allows developers to specify microservice applications by writing simple Python code with two added primitives: transactions and asynchronous service invocations. It then orchestrates this Python code with a runtime to correctly run on Knative [Knative 2022], an open-source serverless platform, handling persistent state management and providing transactional execution guarantees despite faults and concurrency in the serverless platform. Our work is the first to support asynchronous calls inside transactions on serverless. Prior work either: (1) only supports transactions that do not involve calls to other services [Jangda et al. 2019], which prevents the expression of many applications; or (2) is restricted to synchronous calls inside transactions [Zhang et al. 2020a], which exposes fewer opportunities for concurrency.

To ensure that our framework generates correct serverless implementations of the users' microservice applications, we formalize the source and target semantics and prove that any implementations generated by our framework are correct with respect to their specifications. First, we give formal semantics to service specifications, which correspond to the Python code that μ2sls supports and can be used to develop microservice applications. We give formal semantics to microservice applications defined using service specifications using a labeled transition system that keeps track of requests, responses, and local and persistent state for each service; these semantics do not model failures or low-level state management. We then give formal semantics to serverless (SLS) implementations of service applications by defining a transition system that exposes serverless implementation details such as faults, re-executions, and the decoupling of storage and compute.

To bridge the gap between service specifications and serverless implementations, μ2sls defines a formal relationship between the two, describing (i) when an implementation is correct with respect to a specification and (ii) how to generate correct implementations from a given specification. We define correctness as observational refinement, i.e., any observable behavior of the implementation can also be produced by the specification, and we prove that our framework generates correct implementations using a simulation relation. To facilitate the proof of correctness, we introduce an intermediate representation that captures state management concerns but not faults and re-executions. Our formalization effort guided μ2sls's implementation and uncovered two subtle correctness issues, one affecting the atomicity of persistent updates in the context of transactions and another that leads to half-committed results when combining asynchronous calls with transactions. The implementation of μ2sls also identifies a few performance overheads and introduces corresponding optimizations, such as creating a custom wrapper of Python builtin dictionaries to allow for higher concurrency, improving the performance of the formalized translation. Our prototype implementation of μ2sls is open source and available for download: github.com/eniac/mu2sls.

In summary, our work makes the following contributions:

- A formalized service specification abstraction for stateful microservice applications that supports asynchronous calls and transactions (Section 4) and a formalization of serverless implementations of service applications (Section 6).
- A formal relationship that describes when a serverless implementation is correct with respect to a service specification, as well as a proof that μ2sls generates correct serverless implementations from service specifications (Sections 5 and 6)
- A prototype implementation of μ2sls that orchestrates applications defined using service specifications to generate correct serverless implementations (Section 7). We used the prototype to develop the microservice applications from the Deathstar Benchmark Suite [Gan et al. 2019] and evaluated the latency and throughput characteristics of the generated implementations (Section 8).

2 BACKGROUND

Applications today often comprise various services, each of which handles an incoming request, performs some small task, and returns a response. This *microservice* paradigm has many benefits over prior (monolithic) architectures in which all functionality existed within a single component. For example, microservices are modular, so they can be implemented in any language with any features as long as they expose an appropriate API (e.g., REST). This also allows teams to design, develop, scale, and optimize each microservice independently.

A common set of features used by microservices is:

- *State.* Microservices can be stateful and reuse state across requests; they may even persist this state locally or remotely.
- *Asynchronous calls.* Due to the fact that each service implements limited functionality, services need to call other services. They typically leverage asynchronous calls to contact multiple services in parallel.
- *Concurrency Control.* Microservices commonly use multi-threading to ensure high throughput, and they use standard techniques (e.g., locking, transactions) to ensure correct state handling. They also sometimes use distributed transactions to perform atomic operations across services. For example, a travel site might want to use a transaction when the customer books both a

hotel and a flight (each type handled by a different microservice), ensuring that either both reservations succeed or neither does.

Microservices can be deployed on VMs in the cloud, but they are increasingly being deployed on top of *serverless* platforms like AWS Lambda, Azure Functions, and Google's Cloud Functions.

2.1 Serverless platforms

Serverless is a new paradigm that simplifies the development of applications by outsourcing most of the resource management complexity. In serverless, the developer does not need to worry about managing machines and VMs, routing and load balancing requests, scaling up or down resources, updating the OS or runtime, etc. Instead, the developer simply uploads a function or "lambda" (e.g., Python code that implements some functionality) that is invoked on-demand via *triggers* such as an HTTP request to a particular URL or a timer. Autoscaling happens automatically: when invocations occur, the provider spawns the function on any of its available machines; when it completes, the provider may tear down the function to free up resources. This makes functions *ephemeral*, which is a primary reason why serverless functions store their state in an external database and query it explicitly each time they need to access it. The delegation of scaling to the provider also enables adaptive pricing; serverless is usually billed based on how long a function runs regardless of the compute instances that have been spawned in the underlying infrastructure.

While early serverless applications were primarily stateless, the benefits of serverless (elasticity, the delegation of operational concerns, and adaptive pricing) are fundamental and extend beyond purely stateless behavior [Jonas et al. 2019]. There is strong evidence for the use of serverless for stateful workloads in recent academic work on stateful serverless frameworks [Jia and Witchel 2021a,b; Klimovic et al. 2018; Sreekanti et al. 2020b; Zhang et al. 2020a,b], as well as recent industry offerings [Amazon 2020; Bonér 2020; Burckhardt et al. 2021; Ray 2022; Temporal 2022].

Building microservice applications on serverless: Building a microservice application on serverless requires addressing a variety of challenges. These include:

- *State management.* Developers need to pack all of the application's state into data structures that can be stored in a key-value (KV) store or relational database. This is error-prone and requires effort. Since the state is stored in external services, users must deal with failed calls, managing the connection to the service, authentication tokens, etc.
- *At-least-once execution semantics.* Cloud providers use retries to guarantee that a serverless function is executed at least once. This leads to non-trivial complexity for stateful applications, since developers must ensure that re-execution of a function does not corrupt the application's state (e.g., decrements a user's account balance twice).
- *Isolation and atomicity.* When a cloud provider receives multiple requests for a serverless function, the provider spawns multiple instances concurrently. If the function is stateful, this will lead to concurrent accesses to the application's state. It is the developer's job to ensure that such accesses are safe, whether through locks, concurrent data structures, or by designing the application to work with weaker consistency semantics.
- *Reasoning and testing.* Debugging a stateful application that runs on a serverless platform is difficult. Unit tests can give some idea of the basic functionality, while end-to-end tests require a full deployment with various databases, serverless schedulers, etc. There is currently no way to test for functional properties in a more controlled and scalable manner.

To summarize, the key challenge that developers face when running microservice applications on existing serverless runtimes is finding a way to ensure that their application *correctly guarantees exactly-once execution semantics.* That is, that their application behaves as if it were running on a single fault-free server that processes requests exactly once despite the fact that the functions are

```
class Frontend(object):
    async def request(self, user_id,
                      flight_id, hotel_id):
        BeginTx()

        ## Try to reserve a hotel and flight
        hotel_f = AsyncInv('Hotel', 'res_hotel',
                           hotel_id, user_id)
        flight_f = AsyncInv('Flight', 'res_flight',
                            flight_id, user_id)
        h_ret, f_ret = await WaitAll(hotel_f, flight_f)

        ## If any of the two failed abort the txn
        if not (h_ret and f_ret):
            AbortTx()
            return (False, "Order Failed")
        else:
            CommitTx()

        ## Place the order
        SyncInvoke('Order', 'place_order',
                   user_id, flight_id, hotel_id)
        return (True, "Order Successful")
```

```
class Hotel(object):
    def __init__(self):
        self.hotels = {} # type: Persistent[dict]

    async def res_hotel(self, hotel_id, user_id):
        ## Retrieve the hotel from the database
        BeginTx()
        hotel = self.hotels[hotel_id]

        ## If there is capacity,
        ##   add the user in the hotel clients
        if hotel.capacity > 0:
            hotel.clients.append(user_id)
            hotel.capacity -= 1
            self.hotels[hotel_id] = hotel
            ret = True
        else:
            ret = False
        CommitTx()
        return ret
```

Fig. 1. µ2sls code for two services of the travel application: the frontend (left) and the hotel service (right). µ2sls' library provides transaction APIs (BeginTx/CommitTx/AbortTx) and invocation APIs (SyncInv/AsyncInvoke/WaitAll) that work with Python's built-in async/await syntax.

ephemeral, requests can be re-executed at any time (even after the original request completes), and there is no way for the developer to control the number of concurrent instances of a function.

3 OVERVIEW

Our framework addresses the aforementioned challenges by allowing developers to write applications using *service specifications*, i.e., simple Python code with the addition of two primitives: transactions and asynchronous service invocations. Our framework then orchestrates this code to run correctly in a serverless environment, providing transactional and exactly-once execution guarantees despite faults and concurrency in the underlying platform.

3.1 Example: Travel Reservation Application

Figure 1 shows the simplified code of two services from a travel reservation application (Cf. Expedia) in our programming model (service specifications). The service on the left is the frontend that handles incoming reservation requests, which could have been generated by a web app client. The service on the right is a hotel reservation service that checks whether a hotel has adequate capacity and then reserves a room for the user. This application was originally developed as a serverful microservice application as part of the Deathstar Benchmark Suite [Gan et al. 2019], and we reimplemented it in our setting using service specifications.

The application can be used to book both hotels and flights, and it ensures that a booking will either succeed or fail for both, making sure that the user doesn't book a hotel without securing a flight to their destination. This is achieved by wrapping the flight and hotel bookings in a transaction in the frontend; if both reservations succeed, the system can commit the transaction and send a final invocation to an order service that places the final order and completes the reservation. Otherwise, the transaction aborts, and a failed reservation message is returned to the user.

Service Specifications: Service specifications are structured as Python classes that support several requests (their methods) and contain persistent state components (fields marked with a Persistent type). We chose type comments as non-invasive lightweight annotations to indicate the persistent fields so that the same code can be easily used in other contexts without requiring

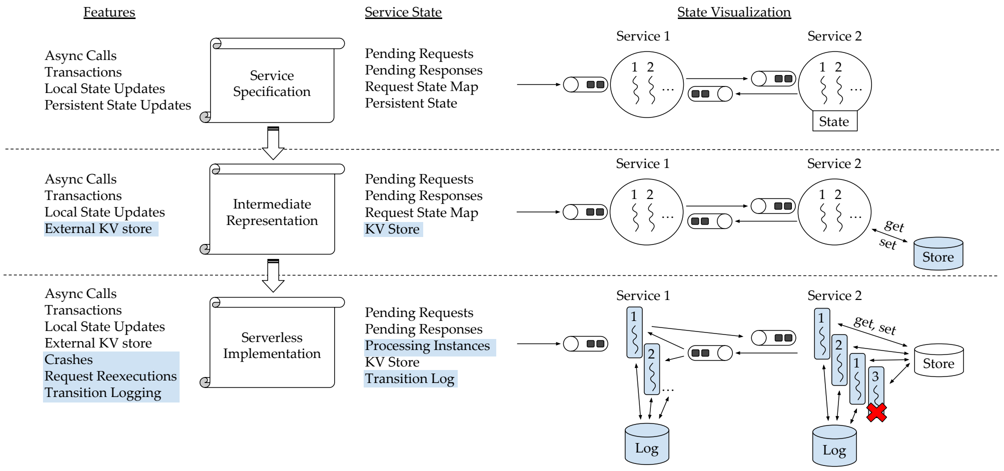


Fig. 2. **μ2sls overview:** The three abstraction layers that are defined in the framework. The components highlighted in light blue are the ones that differ from the layer above.

changes. The persistent fields are standard Python objects that support methods, e.g., the dictionary supports a `.keys()` method that returns all dictionary keys and a `.__get_item__()` method, which is called when accessing its values using `dict[i]`. Developers do not need to worry about how to implement their data structures in a remote persistent store or how to access them in the context of faults and re-executions. Informally, the execution semantics guarantee that different requests execute exactly-once[1] but concurrently, only sharing their persistent fields and guaranteeing isolation and atomicity using transactions. All method calls to persistent objects are guaranteed to be linearizable [Herlihy and Wing 1990] (and thus execute atomically). They are used as the concurrent computation quantum with which users interact with the shared persistent state. Finally, note that the calls to both the hotel and flight reservation services are asynchronous to enable concurrent processing since the result of the calls do not depend on each other; the frontend then waits for both calls to complete without regarding their completion order. Our work is the first to support this in the context of serverless, as prior work either only supports single-service transactions (i.e., transactions that do not include invocations to other services) [Jangda et al. 2019] or is restricted to synchronous calls inside a transaction [Zhang et al. 2020a].

3.2 Framework Overview

Our framework, μ2sls, orchestrates the user-written service specifications with a library of runtime components to correctly execute on serverless, taking care of state management and exactly-once execution in the context of crashes and re-executions. Our runtime and orchestration build on Beldi [Zhang et al. 2020a], a system for the exactly-once execution of stateful serverless workflows, and aim to generate implementations similar to their hand-tuned serverless implementations of microservice applications. Similarly to Beldi, our framework: (i) generates unique identifiers to track invocations, (ii) logs nondeterministic system transitions, and (iii) implements transactions using a two-phase locking protocol. In contrast to Beldi, our framework: (i) supports asynchronous calls in the context of cross-service transactions (e.g., Frontend service in Figure 1) by tracking pending

[1] Note that what is informally called "exactly-once" execution in this and prior work [Burckhardt et al. 2021; Jangda et al. 2019] does not capture arbitrary effects, e.g., sending an email can never be guaranteed to execute exactly once; but instead refers to updates on the persistent state and calls to other services in the application. The notion of correctness that we use and prove is made precise in Sections 5 and 6.

calls and ensuring that they do not escape transaction boundaries, (ii) hides state management from the user by allowing them to represent the application state as Python objects, and (iii) provides formal correctness guarantees for the generated implementations.

Framework Layers: Figure 2 gives an overview of μ2sls, showing the three different abstraction layers, together with their features (on the left) and the system state (on the right). Users describe their application by writing a service specification (Section 4) for each of their services (like the ones in Figure 1). Our framework formally defines service specifications in Section 4, and gives them semantics using a labeled transition system. The transition system state is partitioned for each service, and the per-service state contains its pending requests and responses, a local state map for each request that is being processed (visualized as a thread in Fig. 2), and the service's persistent state. Service specifications are an abstraction that users should be able to reason about, and therefore their semantics abstract away implementation aspects of serverless, namely faults, re-executions, and the separation of storage in an external service. These aspects are captured by serverless implementations (Section 6), which our framework formalizes by building on the serverless execution model introduced by Jangda et al. [2019]. We extend their work to support asynchronous calls in the context of transactions, i.e., their formalization does not support the Frontend service shown in Figure 1. The bottom right of Figure 2 shows the state of a serverless implementation. The local state map is replaced by processing instances, which could be duplicated and could crash, the store is separated from the processing instances, and transitions are stored in a log to tolerate faults and re-executions.

Correctness: We connect service specifications with serverless implementations by formalizing a relationship between the two and then proving that our framework produces correct serverless implementations, given a service specification. To do so, we introduce an intermediate implementation layer (Section 5) that does not include faults and re-executions but captures the store separation by exposing low-level state management using gets and sets, which corresponds to the standard API offered by key-value stores. The correctness proof is then broken up into two steps: first, the intermediate implementation is proven correct with respect to the service specification, and second, the serverless implementation is proven correct with respect to the intermediate implementation. We use a standard notion of correctness for both steps, that of observational refinement, namely that every observable behavior by the implementation is allowed by the specification. We then prove for each step that the implementation system simulates the specification one.

Challenges: In order to generate a correct implementation from a given service specification, our framework has to address two subtle issues that were uncovered by our formalization and not addressed by prior work. First, asynchronous calls decouple invocation from response handling, potentially violating *a call-graph well-formedness* property, i.e., a call can happen inside a transaction, but the response might return after the transaction has already been committed, which could lead to half-committed results. To address this, μ2sls tracks asynchronous calls in a transaction and ensures that they have all been awaited before the transaction completes (committing or aborting). Second, asynchronous calls introduce concurrency while processing a single transaction, allowing for multiple requests to be processed at the same time by a single service. For example, if both `Hotel` and `Flight` services (Figure 1) called the same backend `Storage` service to handle their persistent data, two requests in the same transaction would be processed concurrently by `Storage` for every request to `Frontend`. Due to the low-level API of external stores (and the fact that they offer atomic reads or writes but not arbitrary atomic updates), this could lead to a violation of the atomicity of persistent state update methods, which is provided by service specifications, i.e., the high-level semantics. Our framework addresses this by introducing two-level locks that ensure the atomicity of persistent field updates even in the context of multi-request concurrency in a single transaction.

Serverless implementations introduce crashes and re-executions leading to incorrect executions in the context of side effects or KV store accesses. To hide these issues from the user, µ2sls logs system transitions to avoid performing them a second time if the serverless platform re-executes the same request. Our framework avoids unnecessary overheads by re-executing deterministic request-local transitions instead of logging them—still guaranteeing correct execution.

Prototype and Evaluation: We have implemented a prototype of µ2sls (Section 7) that orchestrates service specifications in Python and generates serverless implementations that run on Knative [Knative 2022], an open-source serverless platform, and use FoundationDB [Apple 2022] as the storage service. Our prototype can also deploy service specifications locally to facilitate debugging and end-to-end application testing. The µ2sls prototype optimizes the formalized translation described in our formalization to improve concurrency, reduce logging overhead, and better utilize the API of FoundationDB. We develop three microbenchmarks to evaluate several aspects of our prototype implementation (Section 8), namely the overheads caused by logging, transactions, and the improvements by one of our optimizations. We also evaluate the feasibility of using µ2sls for real microservice applications by implementing two large-scale applications from DeathstarBench [Gan et al. 2019], a state-of-the-art benchmark suite for microservice applications.

4 STATEFUL SERVICE APPLICATION SPECIFICATION

This section introduces a formal model for applications P that consist of multiple services with persistent state and support transactions and asynchronous calls. Applications are defined using a set of *service specifications* (Section 4.1), one for each application service, and we give semantics to them using a labeled transition system $T(P)$ (Sections 4.2 and 4.3). The goal is to give a high-level semantics for such applications, i.e., from the perspective of the user, and therefore service specifications and their semantics does not expose implementation aspects, like interactions with external persistent stores, logging, re-executions, and crashes.

4.1 Service Specifications

Service Specification	SS	:=	init; stmt	
Statement	stmt	:=	var = e	Local Update
		|	var = pupdate(e)	Persistent Update
		|	req(f, e)	Async service calls
		|	var = wait	Wait for async call response
		|	ret(e)	Return a value for request
		|	stmt; stmt	Sequential composition
		|	if test then {stmt} else {stmt}	Conditional
		|	while test do {stmt}	Iteration
		|	txn {stmt}	Transaction
Initialization	init	:=	...	
Local Expression	e	:=	...	
Test Expression	test	:=	...	

Fig. 3. Service Specification Language.

Figure 3 shows the language used to define services. This closely mirrors the programming model that our prototype framework supports, but is slightly simplified for clarity. More precisely, some components are left abstract (like initialization, local, and test expressions) since their syntactic

$$\begin{array}{rrcl@{\qquad}rrcl}
\text{System} & C & := & \{\mathcal{F}, \ldots\} & \text{Requests} & \mathcal{R} & := & \{\mathbb{R}(x, v), \ldots\} \\
\text{Service} & \mathcal{F} & := & \langle f, \mathcal{M}, p, \mathcal{R}, \mathcal{S} \rangle & \text{Responses} & \mathcal{S} & := & \{\mathbb{S}(x, v), \ldots\} \\
\text{Service name} & f & := & \ldots & \text{Request Header} & x & := & \mathrm{R}(r_{id}, \overline{f}, \overline{i}, j) \\
\text{State Map} & \mathcal{M} & := & \{x \mapsto \sigma\} & \text{Request Id} & r_{id} & := & \ldots \\
\text{Request State} & \sigma & := & \langle s, \mathrm{stmt} \rangle & \text{Call Id} & i & := & \ldots \\
\text{Persistent State} & p & := & \ldots & \text{Transaction Id} & j & := & \ldots
\end{array}$$

Fig. 4. States of transition system $T(P)$ where P is an application defined using service specifications.

form is of no interest. Each service is defined using an initialization function init, and a program that handles the incoming request and returns a result. The initialization function is given the value of the incoming request and it initializes the local state, e.g., by storing the value of the request in a variable. Each service also has access to a persistent state object, which has an initial value and supports a persistent update method. Persistent state survives across requests, while local state lives during a single request.

The service handler is defined using a standard imperative language, supporting assignments, expressions applied on the local state (e.g., accessing variables and arithmetic), sequencing, conditionals, iteration, and returning a response. The handler can also update the service's persistent state, using the pupdate(e) method, which represents all the methods that an object might support, e.g., Python dictionaries support `.keys()` and `.get()` among others. For clarity, our formal model only allows a single persistent object for each service, but our prototype supports multiple persistent fields for each service. Furthermore, a service can perform asynchronous calls to other services using req(f, e), where f represents the remote service and e is a local expression representing the input argument. Services can then wait for the response of any prior asynchronous call using wait. Waiting on any response is not restrictive, since the results of a call that the handler is not interested in yet can be stored in the local state and used later. It is straightforward to perform synchronous calls by combining a call with a subsequent wait, i.e., req$(f, e); t =$ wait.

Finally, services can perform transactions, using txn {stmt}, that provide ACID guarantees for persistent object accesses across different services in the whole application. For simplicity, our formal model does not support custom abort handlers (like the example in Section 3), which is not restrictive since the caller of the service that aborted can handle it, e.g., by retrying the request.

4.2 Execution Semantics

This section gives the semantics of applications P defined using service specifications by defining a labeled transition system $T(P)$. Figure 4 shows the states of the system. We then describe the transitions, first ignoring transactions, and then introducing them to give the complete semantics. Several of our notation decisions are inspired by Jangda et al. [2019] and their formalization.

Application State: The global state of the application is a map from service names f to their states $\mathcal{F}$. Each service state is separate from the rest and contains a state map $\mathcal{M}$ (that we also call a scheduler), a persistent state p, a set of requests $\mathcal{R}$, and a set of responses $\mathcal{S}$. The scheduler maps a request header x to a pair $\langle s, \mathrm{stmt} \rangle$ where s is a local state that assigns values to variables, and stmt is the continuation of the program, i.e., the remaining program to be executed. Request headers x are generated for each request by the caller—either a client or some other service. The set of requests $\mathcal{R}$ contains requests $\mathbb{R}(x, v)$ with header x together with a value v. Responses $\mathbb{S}(x, v)$ also contain a request header so that the request handler that corresponds to them can identify them. The request header contains a request identifier r_{id}, a list of caller services $\overline{f}$ (together with call identifiers $\overline{i}$) to be able to correctly match responses to their respective caller, and an optional transaction identifier if a request is part of a transaction. We need to keep a list of all the callers and call identifiers since call chains between services can be arbitrarily long. External requests have empty caller services and call identifiers lists and no transaction identifier. Predicates ext() and int() check if a request is external or internal respectively. We will ignore persistent state and transactions for now and will come back to them later in Section 4.3. In the starting state of the system, the state map, requests, and responses are all empty.

System Transitions: We describe the system behavior using smallstep operational semantics and an asynchronous execution model, meaning that at each system step, a single service takes a step. System steps can produce a label l, which models the externally visible behavior of the system. There are two relevant labels, clientReq(f, x, v) and clientRes(x, v), that correspond to client requests and responses. The complete system steps using the following rules.

$$\boxed{C \xrightarrow{l} C'} \qquad \textsc{ClientReq}\ \frac{\text{fresh } x \qquad \text{ext}(x) \qquad \mathcal{R}' = \mathcal{R} \cup \mathbb{R}(x, v)}{C \cup \{\langle f, \mathcal{M}, p, \mathcal{R}, \mathcal{S}\rangle\} \xrightarrow{\text{clientReq}(f,x,v)} C \cup \{\langle f, \mathcal{M}, p, \mathcal{R}', \mathcal{S}\rangle\}}$$

$$\textsc{ProcessReq}\ \frac{x \notin \text{dom}(\mathcal{M}) \qquad s = [\![\text{init}_f]\!](v) \qquad \mathcal{M}' = \mathcal{M}[x \mapsto \langle s, \text{stmt}_f\rangle]}{C \cup \{\langle f, \mathcal{M}, p, \mathcal{R} \cup \mathbb{R}(x, v), \mathcal{S}\rangle\} \to C \cup \{\langle f, \mathcal{M}', p, \mathcal{R}, \mathcal{S}\rangle\}}$$

ClientReq describes an incoming request x for service f with value v, that gets stored in the pending requests of that service. Incoming requests need to have a fresh header x, with a request identifier r_{id} that has not occurred before. ProcessReq describes how the service starts processing incoming requests. The initial local state is determined using the initialization function init$_f$, which has type $[\![\text{init}_f]\!] : V \longrightarrow \Sigma$, and the handler for that specified service stmt$_f$. Note that the request is removed from the request set, ensuring that each request is processed exactly once.

$$\textsc{ReqLocalSched}\ \frac{\mathcal{M}[x] = \sigma \qquad x : \sigma \to_r \sigma' \qquad \mathcal{M}' = \mathcal{M}[x \mapsto \sigma']}{C \cup \{\langle f, \mathcal{M}, p, \mathcal{R}, \mathcal{S}\rangle\} \to C \cup \{\langle f, \mathcal{M}', p, \mathcal{R}, \mathcal{S}\rangle\}}$$

$$\textsc{Schedule}\ \frac{\mathcal{M}[x] = \sigma \qquad f, x : \sigma, p, \mathcal{S} \xrightarrow{l} \sigma', p', \mathcal{S}', \mathcal{R}_s, \mathcal{S}_s \qquad C' = \text{add}(C, \mathcal{R}_s, \mathcal{S}_s)}{C \cup \{\langle f, \mathcal{M}, p, \mathcal{R}, \mathcal{S}\rangle\} \xrightarrow{l} C' \cup \{\langle f, \mathcal{M}[x \mapsto \sigma'], p', \mathcal{R}, \mathcal{S}'\rangle\}}$$

ReqLocalSched and Schedule are two *disjoint* rules that describe how a service performs a transition for a pending request that is enabled. The first describes local request processing and only updates the local state σ, while the second describes arbitrary processing steps that affect the whole service (with new calls $\mathcal{R}_s$, responses $\mathcal{S}_s$, and updates to persistent state p). The auxiliary function add$(C, \mathcal{R}_s, \mathcal{S}_s)$ in Schedule adds the relevant new requests and responses to the sets of the corresponding services.

Request Processing: Each service progresses by executing a step for a single request at a time with the following rules. These rules describe how a handler processing request x with (i) local state σ, (ii) persistent service state p, and (iii) available responses $\mathcal{S}$, takes a step updating its local state, persistent state, and available responses, as well as producing requests $\mathcal{R}_s$ and responses $\mathcal{S}_s$ for other services. The service f and request identifier x are read-only and often ignored in the rules when not needed.

$$\textsc{InternalReq}\ \frac{\text{fresh } i \qquad \mathcal{R}_s = \{\langle f', \mathbb{R}(\text{extend}(x, f, i), [\![e]\!](s))\rangle\}}{f, x : \langle s, \text{req}(f', e)\rangle, p, \mathcal{S} \to \langle s, \text{skip}\rangle, p, \mathcal{S}, \mathcal{R}_s, \cdot}$$

InternalReq describes a call to a service f'. The type of the local expression is $[\![e]\!] : \Sigma \to V$. The request identifier of the new request contains: (i) the original x; (ii) the caller service f, to be able to route the response; and (iii) a fresh identifier i that is generated to ensure that different calls to the same service will have different request headers. Request header extension is defined as:

$$\mathrm{extend}(\mathrm{R}(r_{id}, \overline{f}, \overline{i}, j), f, i) = \mathrm{R}(r_{id}, \overline{f} \cdot f, \overline{i} \cdot i, j)$$

There are two rules (ClientRes and Res) that correspond to responses ($\mathrm{ret}(e)$). Both return an empty local state since the processing of the request is complete, and only the internal returns a response to be added to the system. Finally, the Wait transition blocks until there is a response for a request that was made while processing x, which it identifies by matching on its header x'.

$$\textsc{ClientRes}\ \frac{\mathrm{ext}(x) \qquad v = [\![e]\!](s)}{x : \langle s, \mathrm{ret}(e)\rangle, p, \mathcal{S} \xrightarrow{\mathrm{clientRes}(x,v)} \bot, p, \mathcal{S}, \cdot, \cdot} \qquad \textsc{Res}\ \frac{\mathrm{int}(x) \qquad \mathcal{S}_s = \mathbb{S}(x, [\![e]\!](s))}{x : \langle s, \mathrm{ret}(e)\rangle, p, \mathcal{S} \longrightarrow \bot, p, \mathcal{S}, \cdot, \mathcal{S}_s}$$

$$\textsc{Wait}\ \frac{x' = \mathrm{extend}(x, f, \_) \qquad s' = s \cup \{\mathrm{var} \mapsto v\}}{f, x : \langle s, \mathrm{var} = \mathrm{wait}\rangle, p, \mathcal{S} \cup \mathbb{S}(x', v) \longrightarrow \langle s', \mathrm{skip}\rangle, p, \mathcal{S}, \cdot, \cdot}$$

Request-Local Transitions: Request handling can also step in a local way, without affecting anything other than the local state of that particular request. The following rule shows one such transition: a local variable assignment (the rest are shown in the appendix of the extended version of this paper).

$$\textsc{Local}\ \frac{s' = s[\mathrm{var} \mapsto [\![e]\!](s)]}{x : \langle s, \mathrm{var} = e\rangle \longrightarrow_r \langle s', \mathrm{skip}\rangle}$$

4.3 Persistent State and Transactions

We now focus on state and transactions. Figure 5 unfolds the persistent state component that was kept abstract in Figure 4 and describes the transaction context. The persistent state contains: (i) the committed value of the persistent state, (ii) the speculative value of the persistent state if a request is in the middle of a transaction, and (iii) an ownership field that indicates the transaction identifier that owns the state and can perform updates. The system needs to keep track of both the speculative and the committed value of the state in case the transaction aborts, e.g., when attempting to modify a piece of state owned by a different transaction. The speculative state is shared between all requests because multiple requests might be in the same transaction, e.g., if their caller started a transaction and then made two calls in sequence to the same service.

Persistent State	p	:=	$\langle v, v, j\rangle$
Transaction Id	j	:=	$\ldots$
Transaction Context	txn	:=	$\mathrm{tx}_j(\tau, w)$
Services To Inform	τ	:=	$\{\ldots\}$
Waiting on	w	:=	$\ldots$

Fig. 5. Persistent State.

The transaction context is independent for each request being processed (it is kept in the local state, assigned to a special variable called txn, so that it can be accessed from the request handler) and contains the transaction identifier j, the set τ of services f participating in the transaction, i.e., the services that have processed at least one request which was made as part of this transaction, and thus need to be informed to commit or abort, and the number w of invocations for whom responses need to be awaited (necessary to ensure that a transaction commits or aborts as a whole).

We now give the rules that describe transactional execution, beginning and committing transactions, as well as performing requests and receiving responses. The parts of the rules colored in teal indicate the changes with respect to the non-transactional symmetrical rules shown in Section 4.2.

At a high level, transactions in our framework are implemented using a standard non-blocking two-phase locking protocol, using only exclusive locks (and no shared/read-only locks). Our transaction model builds on a recent framework for transactions for serverless workflows [Zhang et al. 2020a], which we extend with (i) support for asynchronous calls, and (ii) formal semantics. To support asynchronous calls without leading to half-committed state, the system keeps track of pending asynchronous invocations that have not been waited using w, and ensures that calls that were made in a transaction have always returned before the transaction is completed.

Initiating transactions and processing transaction requests: Rule BeginTx initializes the transaction context and stores it in the local state. Processing an internal request that is part of a transaction is described with the rule TxProcessReq, which initializes a transaction context in addition to what ProcessReq does.

$$\textsc{BeginTx}\ \frac{s[\text{txn}] = \bot \qquad \text{fresh } j \qquad s' = s[\text{txn} \mapsto \text{tx}_j(\emptyset, 0)]}{f, x : \langle s, \text{txn}\ \{\text{stmt}_1\}\rangle, p, \mathcal{S} \to \langle\langle s', \text{stmt}_1\rangle\rangle, p, \mathcal{S}, \cdot, \cdot}$$

$$\textsc{TxProcessReq}\ \frac{\begin{array}{c} x \notin \text{dom}(\mathcal{M}) \qquad x = \text{R}(\_, \_, \_, j) \\ s = [\![\text{init}_f]\!](v) \qquad s' = s[\text{txn} \mapsto \text{tx}_j(\emptyset, 0)] \qquad \mathcal{M}' = \mathcal{M}[x \mapsto \langle s', \text{stmt}_f\rangle] \end{array}}{C \cup \{\langle f, \mathcal{M}, p, \mathcal{R} \cup \mathbb{R}(x, v), \mathcal{S}\rangle\} \to C \cup \{\langle f, \mathcal{M}', p, \mathcal{R}, \mathcal{S}\rangle\}}$$

TxInternalReq describes an invocation to another service. The difference with the original InternalReq rule is that the caller increments its w counter and also extends the request with the transaction identifier so the callee is aware of the transaction context. When given a transaction identifier j, extend() sets the transaction identifier of the request header.

$$\textsc{TxInternalReq}\ \frac{\begin{array}{c} s[\text{txn}] = \text{tx}_j(\tau, w) \qquad \text{fresh } i \\ s' = s[\text{txn} \mapsto \text{tx}_j(\tau, w+1)] \qquad \mathcal{R}_s = \{\langle f', \mathbb{R}(\text{extend}(x, f, i, j), [\![e]\!](s))\rangle\} \end{array}}{f, x : \langle s, \text{req}(f', e)\rangle, p, \mathcal{S} \to \langle s', \text{skip}\rangle, p, \mathcal{S}, \mathcal{R}_s, \cdot}$$

Internal responses TxRes return the set τ of all the services that have participated in the transaction by processing subrequests (in addition to the return value) so that the request that initiated the transaction can inform all relevant services to commit or abort their changes, and TxWait extends its service-to-inform set using this value τ' and decrements the w counter.

$$\textsc{TxRes}\ \frac{\text{int}(x) \qquad s[\text{txn}] = \text{tx}_j(\tau, 0) \qquad \mathcal{S}_s = \mathbb{S}(x, ([\![e]\!](s), \tau))}{x : \langle s, \text{ret}(e)\rangle, p, \mathcal{S} \to \bot, p, \mathcal{S}, \cdot, \mathcal{S}_s}$$

$$\textsc{TxWait}\ \frac{s[\text{txn}] = \text{tx}_j(\tau, w) \qquad v \neq \text{abort} \qquad s' = s[\text{txn} \mapsto \text{tx}_j(\tau \cup \tau', w-1), \text{var} \mapsto v]}{f, x : \langle s, \text{var} = \text{wait}\rangle, p, \mathcal{S} \cup \mathbb{S}(\text{extend}(x, f', i), (v, \tau')) \to \langle s', \text{skip}\rangle, p, \mathcal{S}, \cdot, \cdot}$$

Finally, when committing (CommitTx) the request handler informs all relevant services by sending them a request to commit a transaction. The commit message is then processed by services like a standard request (ProcCommit), and it commits the speculative state only if the transaction id j corresponds to the ownership of its persistent state. During commit, the committed state is replaced with the speculative, and the ownership of the persistent state is dropped. The reason why committing is processed per service and not per request is that different requests could be part of a single transaction, and the changes done by all of them need to be committed at once.

$$\textsc{CommitTx}\quad \frac{s[\mathrm{txn}] = \mathrm{tx}_j(\tau, 0) \qquad s' = s[\mathrm{txn} \mapsto \bot] \qquad \mathcal{R}_s = \{\langle f', \mathbb{R}(j, \mathrm{commit})\rangle \mid \forall f' \in \tau\}}{\langle s, \mathrm{commit}\rangle, \langle v_1, v_2, j\rangle, \mathcal{S} \longrightarrow \langle\langle s', \mathrm{skip}\rangle\rangle, \langle v_2, \bot, \bot\rangle, \mathcal{S}, \mathcal{R}_s, \cdot}$$

$$\textsc{ProcCommit}\quad \frac{}{C \cup \{\langle f, \mathcal{M}, \langle v, v_s, j\rangle, \mathcal{R} \cup \mathbb{R}(j, \mathrm{commit}), \mathcal{S}\rangle\} \longrightarrow C \cup \{\langle f, \mathcal{M}, \langle v_s, \bot, \bot\rangle, \mathcal{R}, \mathcal{S}\rangle\}}$$

Note that we need to extend the symmetric rules in the non-transactional context, i.e., InternalReq, Res, Wait, with a requirement for the transaction variable to be empty in the starting local state so that symmetric rules are not both enabled at the same time.

Persistent state updates: The following transition rules describe how services access and modify their persistent state inside or outside of a transaction. In all cases, updating happens through the persistent object method pupdate(e).

$$\textsc{Update}\quad \frac{s[\mathrm{txn}] = \bot \qquad (v', v'_p) = [\![\mathrm{pupdate}(e)]\!](v_p) \qquad s' = s[\mathrm{var} \mapsto v']}{f, x : \langle s, \mathrm{var} = \mathrm{pupdate}(e)\rangle, \langle v_p, \bot, \bot\rangle, \mathcal{S} \longrightarrow \langle s', \mathrm{skip}\rangle, \langle v'_p, \bot, \bot\rangle, \mathcal{S}, \cdot, \cdot}$$

Update describes an atomic update of the state outside of a transaction. Non-transactional updates cannot fail and wait until no one has the lock to update their field. Applying $[\![\mathrm{pupdate}(e)]\!]$ to the value of the persistent state v_p, returns two values, a return value v' to be assigned to var, and the new value of the persistent state v'_p.

$$\textsc{UpdOwn}\quad \frac{s[\mathrm{txn}] = \mathrm{tx}_j(\tau, w)}{f, x : \langle s, \mathrm{var} = \mathrm{pupdate}(e)\rangle, \langle v, \bot, \bot\rangle, \mathcal{S} \longrightarrow \langle s, \mathrm{var} = \mathrm{pupdate}(e)\rangle, \langle v, v, j\rangle \mathcal{S}, \cdot, \cdot}$$

$$\textsc{UpdSuccess}\quad \frac{\begin{array}{c} s[\mathrm{txn}] = \mathrm{tx}_j(\tau, w) \\ (v', v'_s) = [\![\mathrm{pupdate}(e)]\!](v_s) \qquad s' = s[\mathrm{txn} \mapsto \mathrm{tx}_j(\tau \cup \{f\}, w), \mathrm{var} \mapsto v'] \end{array}}{f, x : \langle s, \mathrm{var} = \mathrm{pupdate}(e)\rangle, \langle v_p, v_s, j\rangle, \mathcal{S} \longrightarrow \langle s', \mathrm{skip}\rangle, \langle v_p, v'_s, j\rangle, \mathcal{S}, \cdot, \cdot}$$

Updates in transactions are split in two rules, UpdOwn and UpdSuccess, where the first tries to acquire the lock of the persistent state, while the second performs the update on the speculative state (which will later be committed or aborted). Note that a function is only added in the τ set if it updates the state successfully, in order to avoid unnecessary commit or abort messages.

$$\textsc{UpdFail}\quad \frac{s[\mathrm{txn}] = \mathrm{tx}_j(\tau, w) \qquad \mathrm{lock} \notin \{\bot, j\} \qquad s' = s[\mathrm{txn} \mapsto \mathrm{tx}_j(\tau, w)]}{f, x : \langle s, \mathrm{var} = \mathrm{pupdate}(e)\rangle, \langle v, v_s, \mathrm{lock}\rangle, \mathcal{S} \longrightarrow \langle s', \mathrm{abort}\rangle, \langle v, v_s, \mathrm{lock}\rangle, \mathcal{S}, \cdot, \cdot}$$

Finally, UpdFail shows how a transaction aborts if the request tries to update the persistent state when it is owned by a different transaction.

Aborting a transaction: When aborting, a request handler first waits until all of its callees have returned (to be able to collect their τ sets), and then lets its caller know (if it wasn't the initiator of the transaction).

$$\textsc{AbortTxTop}\quad \frac{\begin{array}{cc} s[\mathrm{txn}] = \mathrm{tx}_j(\tau, 0) & p = \langle v, \bot, \bot\rangle \text{ if } j' = j \text{ else } \langle v, v_s, j'\rangle \\ s' = s[\mathrm{txn} \mapsto \bot] & \mathcal{R}_s = \{\langle f', \mathbb{R}(j, \mathrm{abort})\rangle \mid \forall f' \in \tau\} \end{array}}{f, \mathrm{R}(r_{id}, \overline{f}, \overline{i}, \bot) : \langle s, \mathrm{abort}\rangle, \langle v, v_s, j'\rangle, \mathcal{S} \longrightarrow \langle s', \mathrm{ret}(\mathrm{abort})\rangle, p, \mathcal{S}, \mathcal{R}_s, \cdot}$$

AbortTxTop shows the final aborting transition for the initiator of the transaction j, i.e., the head of the transaction call-tree; the request is the transaction initiator because its request header

$\text{R}(r_{id}, \overline{f}, \overline{i}, \perp)$ does not contain a transaction identifier. The request handler that initiated the transaction informs the services in τ to abort and throws away any speculative persistent state and the lock if it was associated with transaction j. The rest of the rules that are not shown here are in the appendix of the extended version of this paper. They include stepping rules for sequence, conditionals, and iteration (mirroring standard IMP semantics [Pierce et al. 2010]), and the rest of the aborting rules.

5 KEY-VALUE STORE IMPLEMENTATION MODEL

Our overarching goal is to give semantics to the serverless implementations of stateful service applications, and show that these implementations are correct with respect to the high-level semantics defined in Section 4. In order to simplify the presentation and proofs, we do not directly describe serverless implementations but rather gradually expose implementation details using an intermediate KV implementation level that exposes state management details that were abstracted by service specifications. In this section, we introduce the *service implementation language* that can be used to define applications P^I that keep their persistent state in a key-value store that provides a get/set API, and give semantics to them using a labeled transition system $T^{\text{KV}}(P^I)$ (Section 5.1). The KV semantics is still abstract in the sense that it does not expose crashes and re-executions. We then describe how our framework translates an application P, defined using service specifications, to a KV application $P^I = \textbf{translate}(P)$ defined using service implementations. The translation is a local replacement of all pupdate(e) statements to instead use the get/set API while ensuring atomicity in and out of transactions; the rest of the statements are left intact. Finally, we prove that any KV application P^I produced by our framework is correct with respect to P, namely, that $T^{\text{KV}}(P^I)$ simulates $T(P)$ (Section 5.2).

5.1 Key-Value Store Implementations

We first describe the service implementations language. Figure 6 shows the language extensions compared to service specifications (defined in Figure 3). The persistent state method pupdate(e) is replaced with get and set statements. The difference between get and get$_c$ is that the latter does not abort if it tries to read from the persistent state when locked by a different transaction, but rather returns a "get-failed" message. Our translation uses get$_c$ to ensure that persistent state accesses will not abort when the specification is not in a transaction, by repeatedly trying to access the state while get$_c$ fails (see Figure 8).

$$\begin{array}{lcl} \text{stmt} & := & \ldots \\ & | & \text{var} = \text{get} \\ & | & \text{var} = \text{get}_c \\ & | & \text{set}(e) \end{array}$$

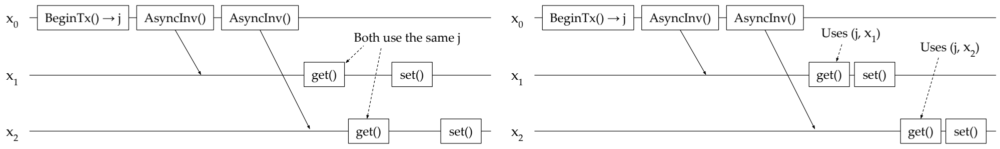
Fig. 6. Service Impl. Language

KV store semantics: We describe the transitions of the labeled transition system of KV applications $T^{\text{KV}}(P^I)$. Below follow the transition rules for the new constructs; our translation ensures that get and set are only called inside transactions and thus we only give semantics for that.

$$\textsc{GetOwn}\ \frac{s[\text{txn}] = \text{tx}_j(\tau, w) \qquad s' = s[\text{var} \mapsto v]}{f, x : \langle s, \text{var} = \text{get}\rangle, \langle v, \perp, \perp\rangle, \mathcal{S} \longrightarrow \langle s', \text{skip}\rangle, \langle v, v, j\rangle, \mathcal{S}, \cdot, \cdot}$$

$$\textsc{Get}\ \frac{s[\text{txn}] = \text{tx}_j(\tau, w) \qquad s' = s[\text{var} \mapsto v_s]}{f, x : \langle s, \text{var} = \text{get}\rangle, \langle v, v_s, j\rangle, \mathcal{S} \longrightarrow \langle s', \text{skip}\rangle, \langle v, v_s, (j, x)\rangle, \mathcal{S}, \cdot, \cdot}$$

$$\textsc{Set}\ \frac{s[\text{txn}] = \text{tx}_j(\tau, w) \qquad v_s' = [\![e]\!](s) \qquad s' = s[\text{txn} \mapsto \text{tx}_j(\tau \cup \{f\}, w)]}{f, x : \langle s, \text{set}(e)\rangle, \langle v, v_s, (j, x)\rangle, \mathcal{S} \longrightarrow \langle s', \text{skip}\rangle, \langle v, v_s', j\rangle, \mathcal{S}, \cdot, \cdot}$$



(a) Without lock promotion (atomicity violation).

(b) With lock promotion.

Fig. 7. An example execution of two asynchronous invocations to the same service in a transaction j.

$$
\begin{aligned}
&\textbf{translate}(\text{var} = \text{pupdate}(e)) := \\
&\quad \text{if } (\text{txn} \neq \bot) \text{ then } \{\ \text{var}_1 = \text{get}; \text{var}_2 = e_p(\text{var}_1); \text{set}(\text{snd}(\text{var}_2)); \text{var} = \text{fst}(\text{var}_2); \\
&\quad \}\ \text{else}\ \{\ \text{txn}\ \{\ \text{var}_1 = \text{“get-failed”}; \text{while } (\text{var}_1 == \text{“get-failed”}) \text{ do } \{\text{var}_1 = \text{get}_c\}; \\
&\qquad\quad \text{var}_2 = e_p(\text{var}_1); \text{set}(\text{snd}(\text{var}_2)); \text{var} = \text{fst}(\text{var}_2); \text{commit} \\
&\quad \}\}
\end{aligned}
$$

Fig. 8. Translation of persistent state updates; the rest of the statements are left intact.

Using a get returns the value of the speculative persistent state (Get) and acquires the lock if needed (GetOwn). Similarly, set modifies the speculative state (Set) and adds the service f that is processing it to the τ set. Requests can fail if the persistent state is owned by a different transaction identifier as shown in the transition rules GetFail and GetCheckFail below.

$$
\text{GetFail}\ \frac{s[\text{txn}] = \text{tx}_j(\tau, w) \qquad \text{lock} \notin \{\bot, j, (j, \_)\}}{f, x : \langle s, \text{var} = \text{get}\rangle, \langle v, v_s, \text{lock}\rangle, \mathcal{S} \longrightarrow \langle s', \text{abort}\rangle, \langle v, v_s, \text{lock}\rangle, \mathcal{S}, \cdot, \cdot}
$$

$$
\text{GetCheckFail}\ \frac{s[\text{txn}] = \text{tx}_j(\tau, w) \qquad \text{lock} \notin \{\bot, j, (j, \_)\} \qquad s' = s[\text{var} \mapsto \text{“get-failed”}]}{f, x : \langle s, \text{var} = \text{get}_c\rangle, \langle v, v_s, \text{lock}\rangle, \mathcal{S} \longrightarrow \langle s', \text{skip}\rangle, \langle v, v_s, \text{lock}\rangle, \mathcal{S}, \cdot, \cdot}
$$

Lock Promotion: As mentioned in Section 3.2, asynchronous calls within a transaction could lead to a race, violating the atomicity of persistent state updates as guaranteed by the service specification semantics. Figure 7a shows an example where the updates in two asynchronous calls in a transaction with identifier j violate atomicity due to them using the same transaction identifier. To address that, get accesses, when successful (see Get transition above), promote the ownership lock to include their request identifier, i.e., from j to (j, x), so that only the set corresponding to the same update call in the service specification can proceed, at which point it demotes the lock back to a normal transaction lock (see Set transition above). Figure 7b shows an example execution of two asynchronous calls in a transaction with lock promotion.

Persistent update translation: In order to hide the store interactions from the user and allow them to develop as if their persistent state was a local Python object, our framework translates the persistent updates in service specifications to gets and sets; all other parts of the service specification are left as they are. Figure 8 shows the translation for pupdate(e), where $\text{var}_1, \text{var}_2, \ldots$ are temporary fresh variables, the local expressions fst and snd are extracting the first and second element from a tuple respectively, and $e_p(\text{var}_1)$ is a local expression that corresponds to pupdate(e), namely $[\![e_p(\text{var})]\!](s) = [\![\text{pupdate}(e)]\!](\text{var})$ and $\text{var} \in \text{dom}(s)$. Translation can be directly lifted to applications P that consist of multiple services.

The translation ensures that if the request handler is not in a transaction, then a transaction has to be started to guarantee the atomicity of the update, and since the update cannot fail, it retries getting its value until it succeeds. After the value of the persistent state is retrieved successfully, the state is owned, and then we perform the update and set the state. Note that the set() cannot fail in the generated implementation since the get has to have succeeded to reach it.

5.2 KV implementation correctness

We want to show that any KV application implementation $P^I = \mathbf{translate}(P)$ is correct with respect to its specification P, namely that any observable behavior exhibited by the translated application can also be exhibited by the original one. We prove this using a forward simulation technique, and for that, we first need to define a simulation relation $\sim$ that describes a correspondence between the states of $T^{\text{KV}}(\mathbf{translate}(P))$ and $T(P)$ transition systems.

Definition 5.1 (Simulation Relation). A state C^{KV} is similar $\sim$ to a state C^{Spec}, iff both states have the same services, $\text{dom}(C^{\text{KV}}) = \text{dom}(C^{\text{Spec}})$, and for all services f the following holds:

1. Requests and Responses of the KV service are the same as the Spec, $\mathcal{R}^{\text{KV}}, \mathcal{S}^{\text{KV}} = \mathcal{R}^{\text{Spec}}, \mathcal{S}^{\text{Spec}}$.
2. Persistent state of the KV service is the same as the Spec, $p^{\text{KV}} = p^{\text{Spec}}$.
3. The KV and the Spec scheduler contains the same requests, $\text{dom}(\mathcal{M}^{\text{KV}}) = \text{dom}(\mathcal{M}^{\text{Spec}})$.
4. The local states for all requests in both the KV and the Spec scheduler are the same, except if the Spec state is updating the persistent state, when the KV state can be in any part of the translated handler. For all x, with $\mathcal{M}^{\text{KV}}[x] = \langle s^{\text{KV}}, \text{stmt}^{\text{KV}} \rangle$ and $\mathcal{M}^{\text{Spec}}[x] = \langle s^{\text{Spec}}, \text{stmt}^{\text{Spec}} \rangle$:
   - either, $s^{\text{KV}} = s^{\text{Spec}}$ and $\text{stmt}^{\text{KV}} = \mathbf{translate}(\text{stmt}^{\text{Spec}})$,
   - or if $\text{stmt}^{\text{Spec}} = (\text{var} = \text{pupdate}(e))$, the KV state can be in any part of the translated handler (Figure 8). We only show one alternative: $\text{stmt}^{\text{KV}} = \text{var}_2 = e_p(\text{var}_1); \text{set}(\text{snd}(\text{var}_2)); \text{var} = \text{fst}(\text{var}_2)$, i.e., the KV model has just executed the get in the first branch of the translated code, $s^{\text{KV}}[\text{txn}] = \text{tx}_j(\tau, w)$ (the KV model is in a transaction), $s^{\text{KV}} = s^{\text{Spec}}[\text{var}_1 \mapsto \text{snd}(p)]$, and $\text{thrd}(p) = (j, x)$ (where thrd retrieves the third item of a tuple), ensuring that the persistent state is owned by the correct transaction id j and request.

Before describing the main correctness theorem, we introduce a lemma that describes valid executions of the KV LTS that are necessary for the proof.

Lemma 5.2 (Call-graph Well-formedness). *Given a labeled transition system $T^{KV}(P^I)$, the following holds for all states C^{KV} that are reachable from its starting state C_0^{KV}. For all services $\langle f, \mathcal{M}, p, \mathcal{R}, \mathcal{S} \rangle \in C^{KV}$, for all requests $x \in dom(\mathcal{M})$, where $\mathcal{M}[x] = \langle s, \text{stmt} \rangle$ and $s[\text{txn}] = \text{tx}_j(\tau, w)$ and for all $\mathbb{R}(j', msg) \in \mathcal{R}$ where $msg = commit \lor abort$, $j' \neq j$.*

Lemma 5.2 addresses the half-committed state issue (a request that was made in a transaction completes after the transaction was committed), and it describes that a service cannot have received a commit or abort request if it is still processing a request that is in that transaction. Our framework guarantees this property by monitoring the initiated requests inside a transaction using w, ensuring that a commit, abort, or return can only happen when all sub-requests have completed ($w = 0$).

We are now ready to define and prove the correctness theorem.

Theorem 5.3. *Given an application P defined using service specifications our framework produces an implementation $P^I = \boldsymbol{translate}(P)$ that is correct with respect to P, namely the LTS of the implementation $T^{KV}(P^I)$ simulates $\sim$ the LTS of the specification $T(P)$. To show that, we need to show that their starting states are related $C_0^{KV} \sim C_0^{Spec}$ and for all C^{KV}, C^{Spec} for which C^{KV} is reachable from C_0^{KV} and $C^{KV} \sim C^{Spec}$ the following holds:*

- *if $C^{KV} \xrightarrow{l} C^{KV\prime}$, then there exists $C^{Spec\prime}$, such that $C^{Spec} \xrightarrow{l} C^{Spec\prime}$ and $C^{KV\prime} \sim C^{Spec\prime}$.*
- *if $C^{KV} \rightarrow C^{KV\prime}$, then either $C^{KV\prime} \sim C^{Spec}$, or there exists $C^{Spec\prime}$, such that $C^{Spec} \rightarrow C^{Spec\prime}$ and $C^{KV\prime} \sim C^{Spec\prime}$.*

Proof. We provide a sketch of the proof. The start states are trivially related; the values of the persistent state are equal and the state map, requests, and responses are empty. The rest of the proof proceeds using a case analysis on the transitions of the KV system. The interesting states are the ones where at least one request x is in the process of updating, i.e., x is processing $\text{var} = \text{pupdate}(e)$ in the specification system. The rest of the cases are straightforward since the KV system exactly mirrors the specification system. If only the currently stepping request x_0 is in the process of updating the persistent state, then the destination state directly satisfies the simulation relation.

Before focusing on the interesting cases, notice the "independence" of the simulation relation w.r.t. requests, namely that similarity of two local states is completely independent to the rest of the state, except for the requirement that $\text{var}_1 \mapsto \text{snd}(p)$ when a request is in the middle of updating after having already read the value from the persistent state. Therefore, we only need to check the cases where the current transition affects the speculative part of p, and therefore might invalidate the simulation for some other x in the same service f. This can happen in two cases, (i) if the transition is ProcCommit or ProcAbort , or (ii) if the current request x_0 is performing a Set transition. For both cases, let a request x be in transaction j, having already read the value of persistent state and therefore owning the p lock with j.

(i) The commit (or abort) message cannot refer to the same j due to Lemma 5.2 that ensures that all requests for a transaction (except for the initiator) have completed before finalizing the commit or abort. The commit also cannot refer to a different j since x holds the lock for p having read the value of the persistent state. If the abort is for a different transaction j', then it cannot affect the speculative state.

(ii) This is a contradiction. The simulation relation $\sim$ precludes x_0 from being ready to perform a Set transition if another x has not completed its Set transition due to lock promotion $\text{thrd}(p) = (j, x)$, regardless of whether they have the same j or not.

□

6 SERVERLESS IMPLEMENTATION MODEL

In this section, we introduce a serverless implementation transition system T^{SLS} exposing the implementation challenges related to serverless deployments, namely failures, re-executions, and concurrent executions of the same request (Section 6.1). The serverless implementation also includes the runtime orchestration that our framework performs to guarantee that execution is correct despite the implementation challenges. We then prove that the serverless implementation of any application P^I produced by our framework is correct with respect to the KV implementation, namely, that $T^{\text{SLS}}(P^I)$ simulates $T^{\text{KV}}(P^I)$ (Section 6.2).

6.1 SLS Implementations

We first describe how the system state is modified for SLS. Figure 9 shows the modified state components. The service state is modified in two ways. First, the scheduler $\mathcal{M}$ is modified to be a multiset instead of a set. This is necessary to model that serverless platforms do not guarantee at-most-once execution of each request, and therefore can be processing multiple executions of the same request at the same time. Second, it is extended to contain a log $\mathbb{L}$ of non-deterministic actions. This log is used by our framework to store the results of some transitions so that they can be replayed instead of re-executed after a crash. The state contains an additional step number y that is incremented every transition and then used together with request identifiers x to index the log.

$$\begin{array}{rcll}
\text{Service} & \mathcal{F} & := & \langle f, \mathcal{M}, p, \mathbb{L}, \mathcal{R}, \mathcal{S} \rangle \\
\text{Instances} & \mathcal{M} & := & \{(x, \langle \sigma, y \rangle), \ldots\} \\
\text{Log} & \mathbb{L} & := & \{(x, y) \mapsto \sigma\} \\
\text{Step Number} & y & := & \ldots
\end{array}$$

Fig. 9. Serverless implementation state.

Serverless platform execution: First, there are two modifications corresponding to the execution characteristics of serverless platforms, namely that instances processing requests might crash at any point in time and that requests are not guaranteed to be processed exactly once (also described by Jangda et al. [2019]). These modifications are captured using a new transition rule named Crash, and by modifying the ProcessReq rule.

$$\textsc{Crash}\ \frac{\mathcal{M} = \mathcal{M}_0 \uplus (x, \langle \sigma, y \rangle)}{C \cup \{\langle f, \mathcal{M}, p, \mathbb{L}, \mathcal{R}, \mathcal{S} \rangle\} \to C \cup \{\langle f, \mathcal{M}_0, p, \mathbb{L}, \mathcal{R}, \mathcal{S} \rangle\}}$$

$$\textsc{ProcessReq}\ \frac{\mathbb{R}(x, v) \in \mathcal{R} \qquad s = [\![\mathrm{init}_f]\!](v) \qquad \mathcal{M}' = \mathcal{M} \uplus (x, \langle \langle s, \mathrm{stmt}_f \rangle, 0 \rangle)}{C \cup \{\langle f, \mathcal{M}, p, \mathbb{L}, \mathcal{R}, \mathcal{S} \rangle\} \to C \cup \{\langle f, \mathcal{M}', p, \mathbb{L}, \mathcal{R}, \mathcal{S} \rangle\}}$$

A Crash rule is added that simply removes a request from the scheduler, simulating the failure of a processing instance that can happen at any time. Transition ProcessReq differs from the specification transition system since it does not remove the request when its processing begins. Serverless platforms achieve that by storing incoming requests in a persistent queue, and then re-executing them until they determine that they have been completed. In order to have better performance and scalability, determining that a request has completed might happen much later than the actual completion of the request, and to model that, we never remove a request from the request set. In addition, ProcessReq adds the request to the multiset of instances even if it already exists there. Our framework extends all requests with a step number y that is initialized to 0 and is then used to store and retrieve transitions from the log.

Managing crashes and re-executions: Second, there are the extensions that have to do with the runtime orchestration of our framework, and how it guarantees that requests are executed exactly once in the context of crashes and re-executions. Scheduling the processing of a request is split into three disjoint cases, instead of just two in the Spec semantics.

$$\textsc{Sched}\ \frac{\begin{array}{c} \mathcal{M} = \mathcal{M}_0 \uplus (x, \langle \sigma, y \rangle) \qquad (x, y) \notin \mathrm{dom}(\mathbb{L}) \qquad f, x : \sigma, p, \mathcal{S} \xrightarrow{l} \sigma', p', \mathcal{S}', \mathcal{R}_s, \mathcal{S}_s \\ \mathcal{M}' = \mathcal{M}_0 \uplus (x, \langle \sigma', y + 1 \rangle) \qquad \mathbb{L}' = \mathbb{L}[(x, y) \mapsto \sigma'] \qquad C' = \mathrm{add}(C, \mathcal{R}_s, \mathcal{S}_s) \end{array}}{C \cup \{\langle f, \mathcal{M}, p, \mathbb{L}, \mathcal{R}, \mathcal{S} \rangle\} \xrightarrow{l} C' \cup \{\langle f, \mathcal{M}', p', \mathbb{L}', \mathcal{R}, \mathcal{S}' \rangle\}}$$

The rule Sched captures the standard processing of a request, checking that this transition has not happened before by looking at the existence of the pair (x, y) in the log $\mathbb{L}$. If the transition hasn't happened in the past, then it simply proceeds normally, and is then stored in the log while also incrementing the step number y for the next transition.

$$\textsc{Replay}\ \frac{\mathcal{M} = \mathcal{M}_0 \uplus (x, \langle \sigma, y \rangle) \qquad \mathbb{L}[(x, y)] = \sigma' \qquad \mathcal{M}' = \mathcal{M}_0 \uplus (x, \langle \sigma', y + 1 \rangle)}{C \cup \{\langle f, \mathcal{M}, p, \mathbb{L}, \mathcal{R}, \mathcal{S} \rangle\} \to C \cup \{\langle f, \mathcal{M}', p, \mathbb{L}, \mathcal{R}, \mathcal{S} \rangle\}}$$

$$\textsc{ReqLocalSched}\ \frac{\mathcal{M} = \mathcal{M}_0 \uplus (x, \langle \sigma, y \rangle) \qquad x : \sigma \xrightarrow{l}_r \sigma' \qquad \mathcal{M}' = \mathcal{M}_0 \uplus (x, \langle \sigma', y + 1 \rangle)}{C \cup \{\langle f, \mathcal{M}, p, \mathbb{L}, \mathcal{R}, \mathcal{S} \rangle\} \xrightarrow{l} C \cup \{\langle f, \mathcal{M}', p, \mathbb{L}, \mathcal{R}, \mathcal{S} \rangle\}}$$

If that specific transition exists in the log, then it is simply replayed without actually executing it (Replay). Note that our prototype implementation doesn't save the complete next state (the whole memory and remaining program), but just saves the result of the current transition on the state, e.g., the result of retrieving the value of persistent state from the store. If the transition is replayed, the runtime does not modify the persistent state, pending requests, and pending responses, since these changes have happened already when the transition was first executed.

The ReqLocalSched transition captures request-local transitions that need not be logged and are simply always re-executed.

$$\textsc{ProcCommit}\ \frac{\mathcal{R} = \mathcal{R}_0 \cup \mathbb{R}(j, \text{commit}) \qquad j \notin \text{dom}(\mathbb{L}) \qquad \mathbb{L}' = \mathbb{L}[j \mapsto \bot]}{C \cup \{\langle f, \mathcal{M}, \langle v, v_s, j\rangle, \mathbb{L}, \mathcal{R}, \mathcal{S}\rangle\} \rightarrow C \cup \{\langle f, \mathcal{M}, \langle v_s, \bot, \bot\rangle, \mathbb{L}', \mathcal{R}_0, \mathcal{S}\rangle\}}$$

Finally, ProcCommit ensures that each commit happens exactly once using the log $\mathbb{L}$.

6.2 Serverless Implementation Correctness

We now want to show that the serverless implementation of any application is correct with respect to the KV implementation, namely that any observable behavior exhibited by the serverless implementation can be exhibited by the KV implementation. We prove this by defining a simulation relation $\approx$ that describes a correspondence between serverless and KV implementation states.

Before defining the simulation relation, we need to define a helper relation $x, \mathbb{L} : y, \sigma \rightsquigarrow \sigma'$ that describes what kind of local states can be reached using only request-local and replay transitions.

$$\boxed{x, \mathbb{L} : y, \sigma \rightsquigarrow \sigma'} \qquad \textsc{Base}\ \frac{}{x, \mathbb{L} : y, \sigma \rightsquigarrow \sigma} \qquad \textsc{Local}\ \frac{x : \sigma \rightarrow_r \sigma' \qquad x, \mathbb{L} : y+1, \sigma' \rightsquigarrow \sigma''}{x, \mathbb{L} : y, \sigma \rightsquigarrow \sigma''}$$

$$\textsc{Replay}\ \frac{\mathbb{L}[(x, y)] = \sigma' \qquad x, \mathbb{L} : y+1, \sigma' \rightsquigarrow \sigma''}{x, \mathbb{L} : y, \sigma \rightsquigarrow \sigma''}$$

This relation is essential since the SLS states can lag behind due to instance failures that completely delete the local state of a request. Therefore, we need a way to express that a request is able to catch up with the KV local state; even though it is lagging behind currently.

Definition 6.1 (SLS Simulation Relation). A state C^{SLS} is similar $\approx$ to a state C^{KV}, iff both states have the same services ($\text{dom}(C^{\text{SLS}}) = \text{dom}(C^{\text{KV}})$) and for all services f the following holds:

(1) The request identifiers of the SLS are either in the pending requests of the KV, or the scheduler of the KV or finished processing, $\{x \mid \mathbb{R}(x, v) \in \mathcal{R}^{\text{KV}}\} \cup \text{dom}(\mathcal{M}^{\text{KV}}) \cup \{x \mid \exists y, \mathbb{L}[(x, y)] = \bot\} = \{x \mid \mathbb{R}(x, v) \in \mathcal{R}^{\text{SLS}}\}$.

(2) The pending requests in the KV model, the requests in the KV scheduler, and the completed requests in SLS are pairwise disjoint. Let $X_1 = \{x \mid \mathbb{R}(x, v) \in \mathcal{R}^{\text{KV}}\}$, $X_2 = \text{dom}(\mathcal{M}^{\text{KV}})$, and $X_3 = \{x \mid \exists y, \mathbb{L}[(x, y)] = \bot\}$. Then $X_1 \cap X_2 = X_1 \cap X_3 = X_2 \cap X_3 = \emptyset$.

(3) The KV pending requests are a subset of the SLS requests, $\mathcal{R}^{\text{KV}} \subseteq \mathcal{R}^{\text{SLS}}$.

(4) Commit (and abort) requests have either been processed or logged, $\{j' \mid \mathbb{R}(j', \text{commit}) \in \mathcal{R}^{\text{KV}}\} \cup \{j' \mid j' \in \mathbb{L}\} = \{j' \mid \mathbb{R}(j', \text{commit}) \in \mathcal{R}^{\text{SLS}}\}$ (similarly for abort requests).

(5) Responses of the SLS service are the same as the ones of the KV, $\mathcal{S}^{\text{KV}} = \mathcal{S}^{\text{Spec}}$.

(6) Persistent state of the SLS service are the same as the original one, $p^{\text{SLS}} = p^{\text{KV}}$.

(7) For all requests that are being processed in KV or are done processing, the requests with the same request header x in the scheduler and pending requests of the SLS can reach them using request-local and replay transitions. For all x, such that $\mathcal{M}^{\text{KV}}[x] = \sigma^{\text{KV}}$ or $\exists y, \mathbb{L}[(x, y)] = \bot = \sigma^{\text{KV}}$, then both:
   - for all $(x, \langle \sigma, y\rangle) \in \mathcal{M}^{\text{SLS}}$, then $x, \mathbb{L} : y, \sigma \rightsquigarrow \sigma^{\text{KV}}$.
   - for $\mathbb{R}(x, v) \in \mathcal{R}^{\text{SLS}}$, where $\sigma_0 = \langle [\![\text{init}_f(v)]\!], \text{stmt}_f\rangle$ then $x, \mathbb{L} : 0, \sigma_0 \rightsquigarrow \sigma^{\text{KV}}$.

The first four requirements of the simulation relation relate the requests of both models. In the SLS model requests are not removed once they start being processed since the serverless platform can re-execute them later (e.g., when a crash occurs). Therefore, the two request sets do not match exactly, but we still need a way to relate them. First, all SLS requests need to either be in the pending requests of the KV, or they need to be processed at the moment (in $\mathcal{M}$), or they need to have completed processing (the log contains the final step of the request). Second, the KV model cannot lag behind the SLS, i.e., a request cannot have started processing in SLS but still be in the pending requests of KV. Third, all pending requests in the KV model must also have arrived in the SLS. Together with the first requirement, this ensures that all request identifiers in the SLS that also exist in the KV correspond to the same request. Fourth, all commit and abort requests in the SLS model need to either be in the pending requests of the KV or already processed (and therefore logged). The responses and the persistent state need to be the same (similarly to $\sim$).

Finally, we relate the local states of each request that is being processed using two requirements. First, each request that is being processed in the SLS should be able to reach the local state in the KV with only request-local and log transitions. Furthermore, all SLS pending requests should also be able to reach the local state in the KV with only request-local and log transitions.

We are now ready to define and prove the theorem that captures the correctness of serverless implementations with respect to KV implementations. By combining this theorem with Theorem 5.3 we get that for any application P defined using service specifications, our framework generates a serverless implementation that is correct with respect to P, i.e., all of the observable behaviors that can be exhibited by $T^{\mathrm{SLS}}(\mathbf{translate}(P))$ can also be exhibited by $T(P)$.

Theorem 6.2. *The serverless implementation of any application P^I is correct with respect to its KV implementation, i.e., the LTS $T^{SLS}(P^I)$ simulates $\approx$ the LTS of the KV implementation $T^{KV}(P^I)$. To show that, we need to show that their starting states are related $C_0^{SLS} \approx C_0^{KV}$ and for all C^{SLS}, C^{KV} for which C^{SLS} is reachable from C_0^{SLS} and $C^{SLS} \approx C^{KV}$ the following holds:*

- *if $C^{SLS} \xrightarrow{l} C^{SLS\prime}$, then there exists $C^{KV\prime}$, such that $C^{KV} \xrightarrow{l} C^{KV\prime}$ and $C^{SLS\prime} \approx C^{KV\prime}$.*
- *if $C^{SLS} \longrightarrow C^{SLS\prime}$, then either $C^{SLS\prime} \approx C^{KV}$, or there exists $C^{KV\prime}$, such that $C^{KV} \longrightarrow C^{KV\prime}$ and $C^{SLS\prime} \approx C^{KV\prime}$.*

Proof. We provide a sketch of some interesting cases of the proof. The start states are related; the values of the persistent state are equal and the state map, requests, responses, and log are empty. The proof proceeds using a case analysis on the transitions of the SLS system C^{SLS}. For transitions Crash and Replay the destination SLS state simulates the starting KV state. We will now go through the intuition behind the proof for the Sched transition. We want to show that the KV can also take the same transition by showing that $\sigma^{\mathrm{KV}}, p^{\mathrm{KV}}, \mathcal{S}^{\mathrm{KV}} = \sigma^{\mathrm{SLS}}, p^{\mathrm{SLS}}, \mathcal{S}^{\mathrm{SLS}}$. Then we can very easily show that the simulation requirements are preserved. We already know that $p^{\mathrm{KV}}, \mathcal{S}^{\mathrm{KV}} = p^{\mathrm{SLS}}, \mathcal{S}^{\mathrm{SLS}}$ from simulation requirements (5,6) on the initial states. Therefore, we just need to show the local states are also the same.

Based on the first part of requirement (7), we also know that $x, \mathbb{L} : y, \sigma^{\mathrm{SLS}} \rightsquigarrow \sigma^{\mathrm{KV}}$. Note that if the Sched rule is enabled, then the ReqLocal rule is not enabled, and also $(x, y) \notin \mathbb{L}$. Based on that and the definition of the helper relation we know that $\sigma^{\mathrm{SLS}} = \sigma^{\mathrm{KV}}$. Therefore, the same transition can be made by x in the KV, satisfying the simulation requirements.

The rest of the cases follow using similar reasoning. □

7 PROTOTYPE IMPLEMENTATION

The µ2sls prototype implements the correct orchestration described in Sections 4 to 6 and generates a serverless implementation given an application specified as shown in Figure 1. The translation contains several components, totaling more than 1500 SLoC, and is open source and available on Github (github.com/eniac/mu2sls). In this section we briefly describe a few highlights of our implementation, including some optimizations that improve the performance of the generated serverless implementations and how it facilitates debugging and testing.

Wrapping Python Objects: The μ2sls prototype supports arbitrary Python objects as the persistent state of a service. To achieve that, it wraps a given object, and then instruments all of the method calls to it to go through the persistent store. Each instrumented method call performs a transaction where it (i) gets the object from the persistent store, (ii) deserializes it, (iii) performs the actual method call, (iv) serializes back the (potentially modified) object, (v) writes it to the store and commits the transaction. Since the framework does not know whether the method modifies the object, it needs to always write it back to the store.

Logging Transitions: The formalized framework guarantees correctness in the presence of faults and re-executions by logging the complete local state of a request every time it performs a non-request-local transition. Our implementation improves performance by not logging unnecessary information in two ways that do not affect the correctness guarantees. First, some transitions in the formalization, like BeginTx (Section 4.3), are nondeterministic only because they generate a fresh identifier. In our implementation, this identifier generation is deterministic, essentially making the transition request-local, where it is safe to reexecute in case of a crash. Similarly, in the formalization each log item contains the whole resulting local state σ of the logged transition (see Sched rule in Section 6.1), whereas our implementation only stores the necessary information to determine what part of the state changed, for example when a Get transition (Section 5) is logged, the implementation only saves the result of the store access. Both of these changes do not weaken the guarantees provided by our formal model.

Exploiting the API of the Persistent Store: Different persistent stores offer slightly different affordances through their APIs in addition to simple gets and sets. For example, some stores offer simple atomic primitives (like conditional writes) or restricted transaction guarantees that cannot support full-fledged cross-service transactions. In addition to our standard transaction mechanism, our implementation leverages FoundationDB's transactions when performing Set transitions with logging, i.e., μ2sls checks the log, writes the data and appends the new entry to the log in a single FoundationDB transaction so that FoundationDB can buffer all writes and apply them together to disk, improving latency.

Optimized Wrapper for Python Dictionaries: The default Python wrapper can support arbitrary python objects as persistent state and supports concurrency across these different objects. However, services often use collection objects to store parts of their data, which could themselves allow for concurrent accesses. In μ2sls's implementation we develop a custom wrapper for a common Python builtin collection object, dictionaries. Our implementation enables additional concurrency by partitioning the keys into different "buckets" using a deterministic hash, therefore requiring locking a single bucket for accessing a single key (instead of locking the entire dictionary). The methods that access multiple keys then need to lock multiple buckets, but this tradeoff is acceptable due to the high frequency of single key accesses. Furthermore, by knowing which methods are read-only, μ2sls avoids writing back to the persistent store after performing them. Our custom wrapper is only a first step in optimizations of that kind, since there is a vast literature on highly concurrent implementations of data structures that could be leveraged to develop highly concurrent implementations for multiple commonly used Python data structures, such as lists, arrays, and sets.

Facilitating application debugging and testing: Debuggability and testability are key challenges when developing serverless applications because the applications run in a remote environment. Remote execution minimizes observability, especially if the only available testing infrastructure is unit testing for individual functions on a few examples. Our framework addresses this issue by supporting end-to-end application deployment in a local setting, i.e., on a local Python interpreter, with an in-memory database, no faults, logging, or re-executions, where it can be debugged and tested more easily. If the application is correct, i.e., satisfies some property in the local setting, then given the correctness of our compilation framework, the serverless implementation should also satisfy the same property. The correctness of a locally deployed application can be checked using standard software correctness methods such as property-based testing or fuzz testing. This functionality proved particularly useful when we ported the applications used in our evaluation since it helped uncover typing and other bugs in our ports locally, before deploying them to the serverless platform.

8 PERFORMANCE EVALUATION

In this section, we want to answer three questions regarding the performance of the serverless implementations generated by µ2sls.

(Q1) What is the performance overhead of guaranteeing exactly-once execution in the presence of faults and re-executions, namely what is the cost introduced by our logging mechanism.

(Q2) What is the performance overhead of providing transaction guarantees using our two-phase locking protocol.

(Q3) Is the performance of implementations generated by µ2sls acceptable for real microservice applications.

To answer the first two questions, we use µ2sls to develop and generate serverless implementations for the following applications.

Stateful Counter: This is a small application with a single stateful service that handles requests that contain a single key, and then increments a counter for the number of requests that it has received with this key. To ensure correctness, the service performs a transaction to increment the counter for each key. In our experiments, we produce requests for keys that are chosen uniformly at random from 0 to 1000.

Chain Counter: This is an application containing three services that are combined in a chain pattern. The first and second services simply forward the call to the next service and forward back the results. The final service is the stateful counter described above. The input workload is the same as above.

Cross Service Txn: This is an application consisting of three services combined in a tree pattern, where the frontend handles input requests by performing a cross-service transaction and forwarding its input value to the two backends asynchronously. The two backend services are both stateful counters and the input is the same as above.

To answer the final question, we adapt two applications from the Deathstar Benchmark Suite [Gan et al. 2019], a state-of-the-art microservice application suite. The applications are:

Movie Review Service (Cf. IMDB or Rotten Tomatoes): Users can create accounts, read reviews, view the plot and cast of movies, and write their own movie reviews and articles. The complete application consists of 14 services, out of which 9 are stateful. Each client request leads to 28 persistent field method calls (which are all wrapped in transactions by our framework's translation). For our experiments, we first populate the database with 100 movies and users, and then we perform review requests by picking users and movies uniformly at random.

Travel reservation (Cf. Expedia): Users can create an account, search for hotels and flights, sort them by price/distance/rate, find recommendations, and reserve hotel rooms and flights. This is the application that was shown in Figure 1. The applications consists of 10 services and the frontend performs a cross-service transaction to ensure that when a user reserves a hotel and a flight, the reservation goes through only if both succeed. Each request to the frontend leads to 5 persistent field method calls. Similar to Beldi [Zhang et al. 2020a], we extend this app to

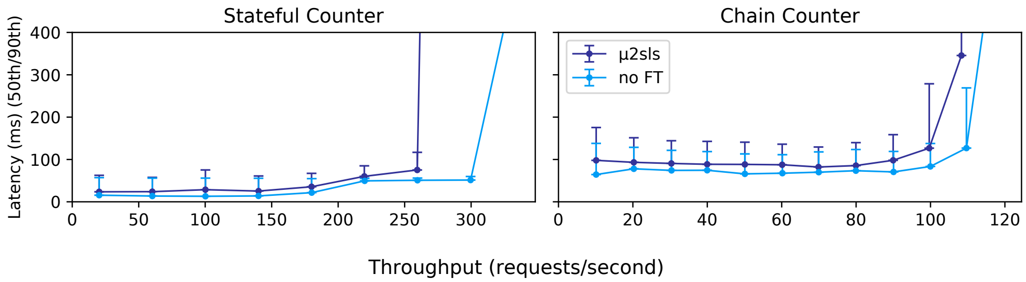


Fig. 10. **Logging overhead (Q1):** Comparison of μ2sls with no FT. no FT doesn't guarantee exactly-once semantics in the presence of faults and reexecutions.

support flight reservations, as the original implementation only supports hotel reservations. We populate the database with 100 hotels and 100 flights, which are then chosen uniformly at random in user requests.

Setup and Infrastructure: All of our experiments run on machines in the Cloudlab testbed [Duplyakin et al. 2019] (configuration: c6525-25g) with a 16-core CPU and 128GB RAM. Our prototype implementation deploys applications using the following components:

- We use Knative [Knative 2022] running on minikube [Minikube 2022] with 12 CPUs and 20GB RAM as the serverless platform. We configured it to 2 Kubernetes pods per service. Autoscaling has been disabled to avoid measuring cold starts and autoscaler overheads.
- Each service is built in a docker container, that is running Quart [Quart 2022] on hypercorn [Hypercorn 2022], an asynchronous HTTP server framework for Python, configured with 4 workers per container.
- As a persistent store, we use FoundationDB [Apple 2022] on the host machine, configured for 1 CPU and 8GBs RAM.

Input Workload and Measurements: For all experiments, we generate input HTTP requests using the open-source workload generator wrk2 [Tene 2022]. For each data point, we produce a steady load for 30s and measure median and 90th percentile latency. We gradually increase the input rate until the implementation is overloaded to evaluate the maximum throughput that the implementation can sustain with reasonable latency. For Movie and Travel applications, we use the same workload generator configuration as in the Beldi paper[Zhang et al. 2020a].

8.1 Logging Overhead (Q1)

To evaluate the logging overhead we use the Stateful Counter and the Counter Chain applications and we toggle the fault tolerance (FT) guarantees. Note that the serverless implementation without fault-tolerance guarantees will not produce correct results in the presence of faults and re-executions.

Figure 10 shows the results. In the Stateful Counter workload, under manageable load, μ2sls's median latencies range from 23ms to 75ms, and no FT's median latencies range from 13ms to 51ms. The maximum throughput of μ2sls is 260 rps, while that of no FT is 300 rps. In the Chain Counter workload, under manageable load, μ2sls's median latencies range from 82ms to 126ms, and no FT's median latencies range from 64ms to 83ms. The maximum throughput of μ2sls is 100 rps, while that of no FT is 110 rps.

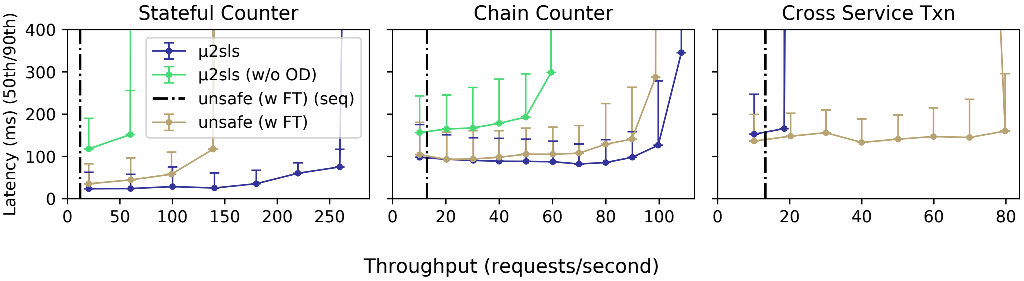


Fig. 11. **Transactions Overhead (Q2):** Comparison of µ2sls, µ2sls (w/o OD) (no optimized dictionary wrapper), and unsafe (w/ FT) (no transactional guarantees). unsafe (w/ FT) (seq) shows the throughput of the implementation without transactional guarantees when executed with sequential load.

Take-away: The overhead of guaranteeing fault tolerance for serverless implementations using logging is not prohibitive; compared to the incorrect implementation, latency is impacted by up to 1.7× when the applications are under manageable load, and the impact on maximum throughput is up to 0.86×. The differences become less pronounced when there are more stateless services, as we can see by comparing Stateful Counter and Chain Counter.

8.2 Transactions Overhead (Q2)

To evaluate the transactions overhead we use the applications that we used for **Q1** with the addition of Cross Service Transaction that includes a cross-service transaction. To evaluate the transaction overheads we compare µ2sls with an unsafe implementation that does not perform any concurrency control (but performs logging, i.e., is fault tolerant). We run this unsafe implementation both with a sequential load (one request at a time) (ref unsafe (w/ FT) (seq)) to measure its throughput, and with the normal workload (ref unsafe (w/ FT)) to measure its upper bound maximum throughput. Note that unsafe (w/ FT) produces wrong results, since incrementing the counter is not performed inside a transaction. We also include a configuration without the optimized dictionary wrapper (µ2sls (w/o OD)) to measure its benefits.

Figure 11 contains the results. The latencies for µ2sls (w/o OD) in the Cross Service Transaction are too high, so they are not visible in the plot. In the Stateful Counter workload, under manageable load, µ2sls's median latencies range from 23ms to 75ms, µ2sls (w/o OD)'s median latencies range from 117ms to 152ms, and unsafe (w/ FT)'s median latencies range from 35ms to 117ms. µ2sls's maximum throughput is 260 rps, µ2sls (w/o OD)'s is 60 rps, unsafe (w/ FT) (seq)'s is 19 rps, and unsafe (w/ FT) is 100 rps. Similar results can be seen for the Chain Counter workload. In the Cross Service workload, under manageable load, µ2sls's median latencies range from 152ms to 165ms, and unsafe (w/ FT)'s median latencies range from 133ms to 160ms. µ2sls's maximum throughput is 20 rps, unsafe (w/ FT) (seq)'s is 15.5 rps, and unsafe (w/ FT) is 80 rps.

Take-away: µ2sls's throughput is consistently higher than unsafe (w/ FT) (seq), showing that the parallelism offered by serverless can be leveraged by µ2sls's implementations. Our optimized wrapper significantly improves performance, even better than the unsafe implementation for applications that don't have cross-service transactions. OD achieves it by leveraging the knowledge that some methods do not update the dictionary state (like `.get(k)` and `.keys()`), so that µ2sls does not need to write them back, making fewer calls to the persistent store. However, in the presence of cross-service transactions, 2-phase locking and additional requests made when a transaction commits or aborts lead to significant costs. In total, transactions do not add significant overhead

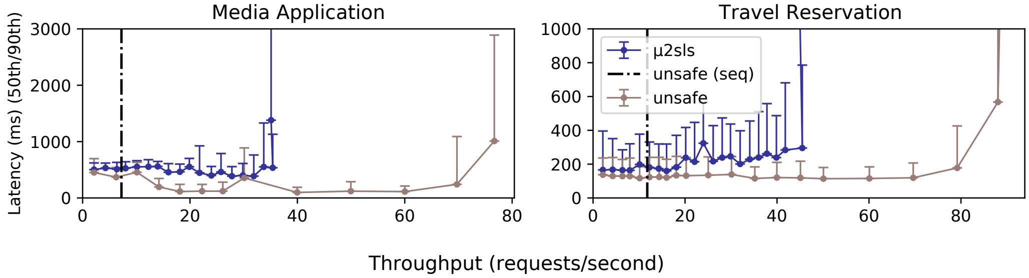


Fig. 12. **Real-world Applications (Q3):** Comparison of μ2sls and unsafe (no transactional guarantees or logging). unsafe (seq) shows the throughput of unsafe when executed with sequential load.

when using the optimized dictionary. However, cross-service transactions introduce significant overheads (1.15× on median latency, 4× on throughput) and therefore should be used carefully.

8.3 Real-world Applications (Q3)

To assess the feasibility of μ2sls for real-world applications, we use the applications adapted from the Deathstar Benchmark Suite. Since there is no framework that provides equivalent guarantees to ours that would allow for direct comparisons, we compare the implementation produced by μ2sls with an unsafe implementation that does not perform any concurrency control or logging. These unsafe implementations mirror the hand-tuned expert-written ones from the Beldi paper, except that they avoid logging and concurrency control, representing a performance upper bound. We do not include a comparison with the expert-written implementations that perform logging and concurrency because our framework's generated implementations are very similar to them, achieving similar performance. We run the unsafe implementations both with a sequential load (one request at a time) unsafe (seq) to measure their throughput, and with the normal workload unsafe to measure their upper bound maximum throughput. Note that unsafe produce wrong results in the context of concurrency or faults.

Figure 12 contains the results. In Media Application, under manageable load, μ2sls's median latencies range from 387ms to 559ms, and unsafe's median latencies range from 114ms to 455ms. μ2sls's maximum throughput is 36 rps, unsafe (seq)'s is 7 rps, and unsafe's is 70 rps. In Travel Reservation, under manageable load, μ2sls's median latencies range from 159ms to 295ms, and unsafe's median latencies range from 113ms to 177ms. μ2sls's maximum throughput is 46 rps, unsafe (seq)'s is 15 rps, and unsafe's is 80 rps.

Take-away: In conclusion, μ2sls can generate serverless implementations for real-world microservice applications that provide transaction support and exactly-once execution semantics while also leveraging the parallelization offered by serverless platforms. μ2sls consistently outperforms unsafe (seq) w.r.t. to throughput by 3-5× and its throughput is in the same order of magnitude (50-57%) from an unsafe implementation that does not provide fault tolerance or transaction guarantees.

9 RELATED WORK

Serverless Execution Platforms: Serverless execution platforms, like AWS Lambda [AWS 2022], Azure Functions [Microsoft Azure 2022], Google Cloud Functions [Google 2022], and Knative [Knative 2022], have become increasingly popular due to the promise of offloading operational concerns, such as function scheduling, request routing, and autoscaling, to the provider. All of these platforms offer limited support for state, and were primarily designed for completely stateless applications. Our framework is orthogonal to these platforms and it focuses on providing features that they lack, such as state management, logging, and transaction support. Our prototype deploys its implementations on Knative, using it as a serverless execution backend.

Formal Semantics for Serverless: Jangda et al. [2019] were the first to study the semantics of serverless platforms, presenting a formal model that captures important execution characteristics of serverless. They also studied stateful serverless functions, namely ones that store their state in an external store, and function compositions by proposing a language called SPL. Our framework uses their work as a foundation, using their formalization of faults and reexecutions in serverless, but extends it to support a programming model with both asynchronous calls and transactions, see the Frontend service in Figure 1 for an example that is not supported by their work. The work by Burckhardt et al. [2021] goes deeper in the study of persistent state, giving semantics to Durable Functions (DF), a programming model for stateful serverless applications, and describing a correct—providing exactly-once execution guarantees—and efficient implementation of DF (excluding critical sections) in the context of faults, reexecutions, and compute-storage separation. The Durable Functions programming model differs from our service specifications in several aspects, e.g., in DF persistent state is stored and accessed through single-threaded stateful actors and not through data objects, and also DF supports critical sections which require a preemptive declaration of the accessed state components (in contrast to transactions that allow for dynamic accesses). At a higher level, our framework builds on both of these works [Burckhardt et al. 2021; Jangda et al. 2019]) but its focus is different, as we target microservice applications and features that are necessary to implement them, i.e., asynchronous calls and cross-service transactions, and we are the first to formalize and develop a framework that given local Python source code generates correct serverless implementations supporting all those features. The work by Ramalingam and Vaswani [2013] came before serverless but has also influenced our framework, as they propose a semantics for distributed services that execute in the presence of reliable storage, providing a translation framework using monads that guarantees correct execution in the presence of faults. Finally, our work is influenced by the rich literature on transaction formalization (e.g., [Abadi et al. 2008; Jagannathan et al. 2005; Lesani et al. 2022; Moore and Grossman 2008; Vitek et al. 2004]).

Stateful Serverless Frameworks: The advent of serverless has led to a surge of proposals for stateful serverless frameworks from both academia [Fouladi et al. 2019, 2017; Jonas et al. 2017; López et al. 2020; Sreekanti et al. 2020a,b; Zhang et al. 2020a,b] and industry [Amazon 2020; Bonér 2020; Burckhardt et al. 2021; CloudFlare 2020a,b]. These systems provide helpful abstractions for managing the state of serverless applications but usually do not provide strong reliability and formal guarantees. [Zhang et al. 2020a] propose Beldi, a runtime and library that can be used by serverless application developers to orchestrate storage accesses to provide exactly once guarantees in the presence of faults. In contrast, μ2slsis an end-to-end framework with formal correctness guarantees where developers write high-level code that is then executed on serverless infrastructure. Our framework's runtime builds on their work, but extends it to microservice applications that support true asynchronous calls that can be awaited inside transactions (the Frontend service in Figure 1 is an example that is not supported by their work), retrieving their results, and provides *formal guarantees*. Furthermore, Beldi requires manual state management (at the level of our KV model) and explicit calls from the user to their library to provide correctness guarantees in the context of faults, both of which are hidden by our programming model and taken care by our framework. Kappa [Zhang et al. 2020b] is a framework for developing stateful serverless applications that focuses on the issue of execution time limits that some serverless providers enforce. To address that, they propose a continuation based translation that breaks user code to smaller functions so that they can execute correctly without surpassing the execution time limit; this also enables functions that invoke other functions to not wait (and be billed for idle time) but rather stop executing and then continue after the response arrives. Their techniques could be used in complement to ours, e.g., to split request-processing at wait points so that a new function can be spawned to process the response.

Microservice Application Correctness: The importance of microservice applications and the lack of support for their correctness has been identified as an issue and recent work has taken steps towards addressing it using a combination of static and dynamic techniques. Whip [Waye et al. 2017] proposes a contract language that captures the higher order nature of many service APIs and develops a runtime monitoring framework to catch contract violations. Panda et al. [2017] argue that microservice applications lack adequate tooling for checking correctness and propose techniques that enable runtime checking of such application properties. Watchtower [Alpernas et al. 2021] builds on those ideas and developes a framework for the runtime verification of serverless and other cloud applications. Finally, Filibuster [Meiklejohn et al. 2021] is a tool that combines static analysis and concolic execution to identify resilience issues in microservice applications.

Actor Frameworks: Actor frameworks, like Erlang [Armstrong 1997], Akka [Haller 2012], and Orleans [Bykov et al. 2011], are often used to simplify the development of service-like applications since each actor is a single-threaded entity that processes incoming requests and holds its own state. In order to develop multi-threaded applications with actors, developers often need to carefully consider how many actors will hold the state and how to partition the state among them. They must also perform manual logging and checkpointing to guarantee exactly-once execution in the presence of faults. In contrast, state management is abstracted in our framework: the developers do not need to worry about its partitioning, and execution is guaranteed to be correct even in the presence of faults.

ACKNOWLEDGMENTS

We thank the POPL reviewers for their thoughtful comments, which improved the content and presentation of this work. We also want to thank the anonymous OOPSLA reviewers for identifying several issues in our formalism in a prior version of this work. This work was partially supported by NSF award CCF 2124184.

REFERENCES

Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. 2008. Semantics of Transactional Memory and Automatic Mutual Exclusion. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. Association for Computing Machinery, New York, NY, USA, 63–74. https://doi.org/10.1145/1328438.1328449

Kalev Alpernas, Aurojit Panda, Leonid Ryzhyk, and Mooly Sagiv. 2021. Cloud-Scale Runtime Verification of Serverless Applications. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 92–107. https://doi.org/10.1145/3472883.3486977

Amazon 2020. AWS Step Functions. https://docs.aws.amazon.com/step-functions/latest/dg/welcome.html.

Apple 2022. FoundationDB. https://www.foundationdb.org/.

Joe Armstrong. 1997. The development of Erlang. In *ICFP*, Vol. 97. 196–203.

AWS 2022. AWS Lambda. https://aws.amazon.com/lambda/.

Jonas Bonér. 2020. Towards Stateful Serverless. https://www.youtube.com/watch?v=DVTf5WQlgB8.

Sebastian Burckhardt, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, and Christopher S. Meiklejohn. 2021. Durable Functions: Semantics for Stateful Serverless. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 133 (Oct 2021), 27 pages. https://doi.org/10.1145/3485510

Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. 2011. Orleans: Cloud Computing for Everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (Cascais, Portugal) *(SOCC '11)*. Association for Computing Machinery, New York, NY, USA, Article 16, 14 pages. https://doi.org/10.1145/2038916.2038932

CloudFlare 2020a. Using Durable Objects, Cloudflare Docs. https://developers.cloudflare.com/workers/learning/using-durable-objects.

CloudFlare 2020b. Workers Durable Objects Beta: A New Approach to Stateful Serverless. https://blog.cloudflare.com/introducing-workers-durable-objects/.

Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 1–14. https://www.flux.utah.edu/paper/duplyakin-atc19

Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 475–488.

Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI'17)*. USENIX Association, USA, 363–376.

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

Google 2022. Google Cloud Functions. https://cloud.google.com/functions.

Philipp Haller. 2012. On the integration of the actor model in mainstream technologies: the Scala perspective. In *Proceedings of the 2nd Edition on Programming Systems, Languages and Applications based on Actors, Agents, and Decentralized Control Abstractions*. ACM, 1–6.

Maurice P Herlihy and Jeannette M Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492.

Hypercorn 2022. Hypercorn. https://pgjones.gitlab.io/hypercorn.

Suresh Jagannathan, Jan Vitek, Adam Welc, and Antony Hosking. 2005. A transactional object calculus. *Science of Computer Programming* 57, 2 (2005), 164–186.

Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. 2019. Formal foundations of serverless computing. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–26.

Zhipeng Jia and Emmett Witchel. 2021a. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 691–707.

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

Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*. 445–451. https://doi.org/10.1145/3127479.3128601

Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* (2019).

Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 427–444.

Knative 2022. Knative. https://knative.dev.

Mohsen Lesani, Li-yao Xia, Anders Kaseorg, Christian J Bell, Adam Chlipala, Benjamin C Pierce, and Steve Zdancewic. 2022. C4: verified transactional objects. *Proceedings of the ACM on Programming Languages* 6, OOPSLA (2022), 1–31.

Pedro García López, Aitor Arjona, Josep Sampé, Aleksander Slominski, and Lionel Villard. 2020. Triggerflow: Trigger-Based Orchestration of Serverless Workflows. In *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems* (Montreal, Quebec, Canada) *(DEBS '20)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/3401025.3401731

Christopher S. Meiklejohn, Andrea Estrada, Yiwen Song, Heather Miller, and Rohan Padhye. 2021. Service-Level Fault Injection Testing. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 388–402. https://doi.org/10.1145/3472883.3487005

Microsoft Azure 2022. Azure Functions. https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview.

Minikube 2022. Minikube. https://minikube.sigs.k8s.io.

Katherine F Moore and Dan Grossman. 2008. High-level small-step operational semantics for transactions. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 51–62.

Atif Nazir, Saqib Raza, and Chen-Nee Chuah. 2008. Unveiling facebook: a measurement study of social network based applications. In *Proceedings of the 8th ACM SIGCOMM Internet Measurement Conference*. ACM.

Aurojit Panda, Mooly Sagiv, and Scott Shenker. 2017. Verification in the age of microservices. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. 30–36.

Benjamin C Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2010. Software foundations. *Webpage: http://www.cis.upenn.edu/bcpierce/sf/current/index.html* (2010).

Quart 2022. Quart. https://pgjones.gitlab.io/quart.

Ganesan Ramalingam and Kapil Vaswani. 2013. Fault tolerance via idempotence. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 249–262. https://doi.org/10.1145/2429069.2429100

Ray 2022. Ray Core: A faster, simpler approach to parallel Python. https://ray.io/ray-core.

Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E Gonzalez, Joseph M Hellerstein, and Jose M Faleiro. 2020a. A fault-tolerance shim for serverless computing. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–15. https://doi.org/10.1145/3342195.3387535

Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Jose M Faleiro, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov. 2020b. Cloudburst: Stateful functions-as-a-service. *arXiv preprint arXiv:2001.04592* (2020).

Temporal 2022. Temporal. https://temporal.io/.

Gil Tene. 2022. wrk2. https://github.com/giltene/wrk2.

Jan Vitek, Suresh Jagannathan, Adam Welc, and Antony L Hosking. 2004. A semantic framework for designer transactions. In *European Symposium on Programming*. Springer, 249–263.

Lucas Waye, Stephen Chong, and Christos Dimoulas. 2017. Whip: Higher-Order Contracts for Modern Services. *Proc. ACM Program. Lang.* 1, ICFP, Article 36 (aug 2017), 28 pages. https://doi.org/10.1145/3110280

Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. 2020a. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 1187–1204.

Wen Zhang, Vivian Fang, Aurojit Panda, and Scott Shenker. 2020b. Kappa: A Programming Framework for Serverless Computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 328–343. https://doi.org/10.1145/3419111.3421277

Received 2022-07-07; accepted 2022-11-07