Formal Foundations of Serverless Computing

Abhinav Jangda Donald Pinckney Samuel Baxter

Joseph Spitzer Breanna Devore-McDonald Yuriy Brun

Arjun Guha
University of Massachusetts Amherst

Abstract

A robust, large-scale web service can be difficult to engineer. When demand spikes, it must configure new machines and manage load-balancing; when demand falls, it must shut down idle machines to reduce costs; and when a machine crashes, it must quickly work around the failure without losing data. In recent years, *serverless computing*, a new cloud computing abstraction, has emerged to help address these challenges. In serverless computing, programmers write *serverless functions*, and the cloud platform transparently manages the operating system, resource allocation, load-balancing, and fault tolerance. In 2014, Amazon Web Services introduced the first serverless platform, *AWS Lambda*, and similar abstractions are now available on all major clouds.

Unfortunately, the serverless computing abstraction exposes several low-level operational details that make it hard for programmers to write and reason about their code. This paper sheds light on this problem by presenting $\lambda_{\mathbb{S}}$, an operational semantics of the essence of serverless computing. Despite being a small core calculus (less than one column), $\lambda_{\mathbb{S}}$ models all the low-level details that serverless functions can observe. To show that $\lambda_{\mathbb{S}}$ is useful, we present three applications. First, to make it easier for programmers to reason about their code, we present a simplified semantics of serverless execution and precisely characterize when the simplified semantics and $\lambda_{\mathbb{S}}$ coincide. Second, we augment $\lambda_{\mathbb{S}}$ with a key-value store, which allows us to reason about stateful serverless functions. Third, since a handful of serverless platforms support serverless function composition, we show how to extend $\lambda_{\mathbb{S}}$ with a composition language. We have implemented this composition language and show that it outperforms prior work.

1 Introduction

Serverless computing, also known as *functions as a service*, is a new approach to cloud computing that allows programmers to run event-driven functions in the cloud without the need to manage resource allocation or configure the runtime environment. Instead, when a programmer deploys a *serverless function*, the cloud platform automatically manages dependencies, compiles code, configures the operating system, and manages resource allocation. Unlike virtual machines or containers, which require low-level system and resource management, serverless computing allows programmers to focus entirely on application code. If demand for the function suddenly increases, the cloud platform may transparently load another instance of the function on a new machine and manage load-balancing without programmer intervention. Conversely, the platform will transparently terminate underutilized instances when demand for the function falls. In fact, the cloud provider may shutdown all running instances of the function if there is no demand for an extended period of time. Since the platform completely manages the operating system and resource allocation in this manner, serverless computing is a language-level abstraction for programming the cloud.

An economic advantage of serverless functions is that they only incur costs for the time spent processing events. Therefore, a function that is never invoked incurs no cost. By contrast, virtual machines incur costs when they are idle, and they need idle capacity to smoothly handle unexpected increases in demand. Serverless computing allows cloud platforms to more easily optimize utilization across several customers, which increases hardware utilization and lowers costs (e.g., by lowering energy consumption).

Amazon Web Services introduced the first serverless computing platform, *AWS Lambda*, in 2014, and similar abstractions are now available from all major cloud providers [1, 14, 22, 26, 29, 31]. Serverless computing has seen rapid adoption [11], and programmers now often use serverless computing to write short, event-driven computations, such as web services, backends for mobile apps, and aggregators for IoT devices.

In the research community, there is burgeoning interest in developing new programming abstractions for serverless computing, including abstractions for big data processing [4, 18, 28], modular programming [7], information flow control [2], chatbot design [8], and virtual network functions [40]. However, serverless computing has significant, peculiar characteristics that prior work does not address.

The serverless computing abstraction, despite its many advantages, exposes several low-level operational details that make it hard for programmers to write and reason about their code. For example, to reduce latency, serverless platforms try to reuse the same function instance to process

multiple requests. However, this behavior is not transparent, and it is easy to write a serverless function that produces incorrect results or leaks confidential data when reused. A related problem is that serverless platforms abruptly terminate function instances when they are idle, which can lead to data loss if the programmer is not careful. To tolerate network and system failures, serverless platforms automatically re-execute functions on different machines. However, it is up to the programmer to ensure that their functions perform correctly when they are re-executed, which may include concurrent re-execution when a transient failure occurs. These problems are exacerbated when an application is composed of several functions. In summary, *serverless functions are not functions* in any typical sense of the word.

Our contributions. This paper presents a formal foundation for serverless computing. Based on our experience with several major serverless computing platforms (Google Cloud Functions, Apache OpenWhisk, and AWS Lambda), we have developed a detailed, operational semantics of serverless platforms, called $\lambda_{\mathbb{S}}$, which manifests the essential low-level behaviors of these serverless platforms, including failures, concurrency, function restarts, and instance reuse. The details of $\lambda_{\mathbb{S}}$ matter because they are observable by programs, but programmers find it hard to write code that correctly addresses all of these behaviors. By elucidating these behaviors, $\lambda_{\mathbb{S}}$ can guide the development of programming tools and extensions to serverless platforms. To demonstrate that $\lambda_{\mathbb{S}}$ is useful, this paper presents three distinct case studies that employ it.

First, we address the problem that it can be hard for programmers to reason about the complex, low-level behavior of serverless platforms, which $\lambda_{\mathbb{S}}$ models. We present an alternative, *naive semantics* of serverless computing, that elides these complex behaviors, and precisely characterize when it is safe for a programmer to use the naive semantics instead of $\lambda_{\mathbb{S}}$. Specifically, we establish a weak bisimulation between $\lambda_{\mathbb{S}}$ and the naive semantics that is conditioned on simple semantic criteria. This result helps programmers confidently abstract away the low-level details of serverless computing.

Second, we address the fact that serverless functions frequently use a cloud-hosted key-value store to share state. We have designed $\lambda_{\mathbb{S}}$ to be an extensible semantics, so we extend it with a model of a key-value store. Using this extension, we precisely characterize idempotence for serverless functions, which allows programmers to reason using an extended naive semantics, where each event is processed in isolation. The recipe that we follow to add a key-value store to $\lambda_{\mathbb{S}}$ could also be used to augment $\lambda_{\mathbb{S}}$ with models of other cloud computing services.

Third, we account for serverless platforms that go beyond running individual serverless functions and also support serverless function orchestration. We extend $\lambda_{\mathbb{S}}$ to support a *serverless programming language* (SPL), which has a suite of I/O primitives, data processing operations, and composition operators that can run safely and efficiently without the need for operating system isolation mechanisms. To perform operations that are beyond the scope of SPL, we allow SPL programs to invoke existing serverless functions as black boxes. We implement SPL, evaluate its performance, and find that it can outperform a popular alternative in certain common cases. Using case studies, we show that SPL is expressive and easy to extend with new features.

Our hope is that $\lambda_{\mathbb{S}}$ will be a foundation for further research on language-based abstractions for serverless computing. The case studies that we present are detailed examples that show how to to design new abstractions and study existing abstractions for serverless computing, using $\lambda_{\mathbb{S}}$.

The rest of this paper is organized as follows. §2 presents an overview of serverless computing, and a variety of issues that arise when writing serverless code. §3 presents $\lambda_{\mathbb{S}}$, our formal semantics of serverless computing. §4 presents a simplified semantics of serverless computing and proves exactly when it coincides with $\lambda_{\mathbb{S}}$. §5 augments $\lambda_{\mathbb{S}}$ with a key-value store. §6 and §7 extend $\lambda_{\mathbb{S}}$ with a language for serverless orchestration. Finally, §8 discusses related work and §9 concludes.

2 Overview of Serverless Computing

To motivate the need for a formal foundation of serverless computing, consider the serverless function in Figure 1a.[1] During deployment, the programmer can specify the types of events that will *trigger* the function, such as, messages on a message-bus, updates to a database, or web requests to a URL. The function receives the event as a JSON-formatted object. In this example, the *type* field of the event determines whether the function deposits money to an account or transfers money between accounts. The function also receives a callback, which it must call when event processing is complete. The callback allows the function to run asynchronously, although our example is presently synchronous. For brevity, we have elided authentication, authorization, error handling, and most input validation. However, this function suffers several other problems because it is not properly designed for serverless execution.

Ephemeral state. The first problem arises after a few minutes of inactivity: all updates to the accounts global variable are lost. This problem occurs because the serverless platform runs the function in an ephemeral container, and silently shuts down the container when it is idle. The exact timeout depends on overall load on the cloud provider's infrastructure, and is not known to the developer. Moreover, the function does not receive a notification before a shut down

[1] The examples in this paper are in JavaScript — the language that is most widely supported by serverless platforms — and are written for Google Cloud Functions. However, it is easy to port our examples to other languages and serverless platforms.

```
let accounts = new Map();
exports.bank = function(req, res) {
  if (req.body.type === 'deposit') {
    accounts.set(req.body.name, req.body.value);
    res.send(true);
  } else if (req.body.type === 'transfer') {
    let { from, to, amnt } = req.body;
    if (accounts.get(from) >= amnt) {
      accounts.set(to, accounts.get(to) + amnt);
      accounts.set(from, accounts.get(from) - amnt);
      res.send(true);
    } else {
      res.send(false); }}};
```

(a) An incorrect implementation that does not address low-level details of serverless execution.

```
let Datastore = require('@google-cloud/datastore');
exports.bank = function(req, res) {
  let ds = new Datastore({ projectId: 'bank-app' });
  let dst = ds.transaction();
  dst.run(function() {
    let tId = ds.key(['Transaction', req.body.transId]);
    dst.get(tId, function(err, trans) {
      if (err || trans) {
        dst.rollback(function() { res.send(false); });
      } else if (req.body.type === 'deposit') {
        let to = ds.key(['Account', req.body.to]);
        dst.get(to, function(err, acct) {
          acct.balance += req.body.amount;
          dst.save({ key: to, data: acct });
          dst.save({ key: tId, data: {} });
          dst.commit(function() { res.send(true); });
        });
      } else if (req.body.type === 'transfer') {
        let amnt = req.body.amount;
        let from = ds.key(['Account', req.body.from]);
        let to = ds.key(['Account', req.body.to]);
        dst.get([from, to], function(err, accts) {
          if (accts[0].balance >= amnt) {
            accts[0].balance -= amnt;
            accts[1].balance += amnt;
            dst.save([{ key: from, data: accts[0] },
              { key: to, data: accts[1] }]);
            dst.save({ key: tId, data: {} });
            dst.commit(function() { res.send(true); });
          } else {
            dst.rollback(function() { res.send(false);
          });}});}});});};
```

(b) A correct implementation that addresses instance termination, concurrency, and idempotence.

Figure 1. Serverless functions for banking.

occurs. Similarly, the platform automatically starts a new container when a event eventually arrives, but all state is lost. Therefore, the function must serialize all state updates to a persistent store. In serverless environments, the local disk is also ephemeral, therefore our example function must use a network-attached database or storage system.

Implicit parallel execution. The second problem arises when the function receives multiple events over a short period of time. When a function is under high load, the serverless platform transparently starts new *instances* of the function and manages load-balancing to reduce latency. Each instance runs in isolation (in a container), there is no guarantee that two events from the same requester will be processed by the same instance, and instances may process events in parallel. Therefore, a correct implementation should use transactions to correctly manage this implicit parallelism.

At-least-once execution. The third problem arises when failures occur in the serverless computing infrastructure. Serverless platforms are distributed systems that are designed to re-invoke functions when a failure is detected.[2] However, if the failure is transient, a single event may be processed to completion multiple times. Most platforms run functions *at least once* in response to a single event, which can cause problems when functions have side-effects [3, 23, 30, 32]. In our example function, this would duplicate deposits and transfers. We can fix this problem in several ways. A common approach is to require each request to have a unique identifier, maintain a consistent log of processed identifiers in the database, and ignore requests that are already in the log.

Figure 1b fixes the three problems mentioned above. This implementation uses a cloud-hosted key-value store for persistence, uses transactions to address concurrency, and requires each request to have a unique identifier to support safe re-invocation. This version of the function is more than twice as long as the original, and requires the programmer to have a deep understanding of the serverless execution model. The next section presents an operational semantics of serverless platforms ($\lambda_{\mathbb{S}}$) that succinctly describes its peculiarities. Using $\lambda_{\mathbb{S}}$, the rest of the paper precisely characterizes the kinds of properties that are needed for serverless functions, such as our example, to be correct.

3 Semantics of Serverless Computing

We now present our operational semantics of serverless platforms ($\lambda_{\mathbb{S}}$) that captures the essential details of the serverless execution model, including concurrency, failures, execution retries, and function instance reuse. Figure 2 presents $\lambda_{\mathbb{S}}$ in full, and is divided into two parts: (1) a model of serverless functions, and (2) a model of a serverless platform that receives requests (i.e., events) and produces responses by running serverless functions. The peculiar features of serverless computing are captured by the latter part.

Serverless functions. Serverless platforms allow programmers to write serverless functions in a variety of source languages, but platforms themselves are source-language agnostic. Most platforms only require that serverless functions operate asynchronously, process a single request at a time,

[2] Even if the serverless platform does not re-invoke functions itself, there are situations where the external caller will re-invoke a function to workaround a transient failure. For example, this arises when functions are triggered by HTTP requests.

Serverless Functions $\langle f, \sigma_0, \Sigma, \text{recv}, \text{step}\rangle$

Function name	$f := \cdots$	
Internal states	$\Sigma := \cdots$	
Initial state	$\sigma_0 \in \Sigma$	
Receive event	$\text{recv} \in v \times \Sigma \to \Sigma$	
Internal step	$\text{step} \in \Sigma \to \Sigma \times t$	With effect t
Values	$v := \cdots$	JSON, HTTP, etc.
Commands	$t := \varepsilon$	
	$\mid \textbf{return}(v)$	Return value

Serverless Platform

Request ID	$x := \cdots$	
Instance ID	$y := \cdots$	
Execution mode	$m := \textbf{idle}$	Idle
	$\mid \textbf{busy}(x)$	Processing x
Transition labels	$\ell :=$	Internal transition
	$\mid \text{start}(x, v)$	Receive v
	$\mid \text{stop}(x, v)$	Respond v
Components	$C := \mathbb{F}(f, m, \sigma, y)$	Function instance
	$\mid \mathbb{R}(f, x, v)$	Apply f to v
	$\mid \mathbb{S}(x, v)$	Respond with value v
Component set	$\mathbb{C} := \{C_1, \cdots, C_n\}$	

$\boxed{\mathbb{C} \overset{\ell}{\mapsto} \mathbb{C}}$

$$\textsc{Req}\ \frac{x \text{ is fresh}}{C \xRightarrow{\text{start}(x, v)} C\mathbb{R}(f, x, v)}$$

$$\textsc{Cold}\ \frac{y \text{ is fresh} \qquad \text{recv}(v, \sigma_0) = \sigma}{C\mathbb{R}(f, x, v) \Rightarrow C\mathbb{R}(f, x, v)\mathbb{F}(f, \textbf{busy}(x), \sigma, y)}$$

$$\textsc{Warm}\ \frac{\text{recv}(v, \sigma) = \sigma'}{C\mathbb{R}(f, x, v)\mathbb{F}(f, \textbf{idle}, \sigma, y) \Rightarrow C\mathbb{R}(f, x, v)\mathbb{F}(f, \textbf{busy}(x), \sigma', y)}$$

$$\textsc{Hidden}\ \frac{\text{step}(\sigma) = (\sigma', \varepsilon)}{C\mathbb{F}(f, \textbf{busy}(x), \sigma, y) \Rightarrow C\mathbb{F}(f, \textbf{busy}(x), \sigma', y)}$$

$$\textsc{Resp}\ \frac{\text{step}(\sigma) = (\sigma', \textbf{return}(v'))}{C\mathbb{R}(f, x, v)\mathbb{F}(f, \textbf{busy}(x), \sigma, y) \xRightarrow{\text{stop}(x, v')} C\mathbb{F}(f, \textbf{idle}, \sigma, y)\mathbb{S}(x, v')}$$

$$\textsc{Die}\ \frac{}{C\mathbb{F}(f, m, \sigma, y) \Rightarrow C}$$

Figure 2. $\lambda_{\mathbb{S}}$: An operational model of serverless platforms.

and usually consume and produce JSON values. These are the features that our abstract model includes (Figure 2).

In our model, the operation of a serverless function (f) is defined by two functions: a function that receives a request (recv), and a function that takes an internal step (step) that may produce a command for the serverless platform (t). For now, the only valid command is **return**(v), which indicates that the response to the last request is the value v.[3] The state of a serverless function (σ) is abstract to the serverless platform. However, there is a distinguished initial state (σ_0) in which the platform launches a serverless function. In practice, the initial state depends on the source language. For example, if f were written in JavaScript then σ_0 would be the initial JavaScript heap.

Serverless platform. $\lambda_{\mathbb{S}}$ is an operational semantics of a serverless platform that processes several concurrent requests. $\lambda_{\mathbb{S}}$ is written in a process-calculus style, where the state of the platform consists of a collection of running or idle functions (known as *function instances*), pending requests, and responses. A new request may arrive at any time for a serverless function f, and each request is given a globally unique identifier x (the Req rule). However, the platform does not process requests immediately. Instead, at some later step, the platform may either *cold-start* a new instance of f (the Cold rule), or it may *warm-start* by processing the request on an existing, idle instance of f (the Warm rule). The internal steps of a function instance are unobservable (the Hidden rule). The only observable command that an instance can produce is to respond to a pending request (the Resp rule). When an instance responds to a request, $\lambda_{\mathbb{S}}$ produces a response object, an observable response event, and marks the function instance as idle, which allows it to be reused. Finally, a function instance may die at any time without notification (the Die rule).

$\lambda_{\mathbb{S}}$ captures several subtle behaviors that occur in serverless platforms. (1) The platform produces an observable event ($\text{start}(x, v)$) when it receives a request (Req), and not when it starts to process the request. This is necessary because the platform may start several instances for a single event x, for example, if the platform detects a potential failure. (2) During a warm-start, it does *not* re-initialize the state. Instead, it applies recv to whatever last state the instance had. (3) The platform can cold-start or warm-start a function instance for any pending request at any time, even if the request is currently being processed. In practice, this behavior occurs during a transient failure, when an instance becomes temporarily unreachable. (4) After responding to a request, the platform does not discard function state (Respond), which allows functions to cache results across invocations. (5) If two instances are processing a single request x and one function responds, the other function will eventually get stuck because there is no longer a request for it to respond to. The stuck function will eventually terminate (Die). (6) Instance termination is not observable and may occur due to failure, or when the platform reclaims the resources held by an instance that is idle or stuck (Die). (7) There is no rule in $\lambda_{\mathbb{S}}$ to restart function execution after it dies. Instead, the semantics can nondeterministically launch a new instance at any time.

In summary, $\lambda_{\mathbb{S}}$ succinctly models the low-level details of serverless platforms. The semantics manifests the subtleties that make serverless programming hard. The next section uses $\lambda_{\mathbb{S}}$ to present a simpler model to make serverless programming easier.

[3] §5 extends our model with new commands.

Naive function state $\mathcal{A} := \langle f, m, \vec{\sigma} \rangle$

$\boxed{\mathcal{A} \overset{\ell}{\mapsto} \mathcal{A}}$

$$\text{N-Start}\ \frac{x \text{ is fresh} \qquad \text{recv}(v, \sigma_0) = \sigma'}{\langle f, \textbf{idle}, \vec{\sigma} \rangle \overset{\text{start}(x,v)}{\longmapsto} \langle f, \textbf{busy}(x), [\sigma_0, \sigma'] \rangle}$$

$$\text{N-Step}\ \frac{\text{step}(\sigma) = (\sigma', \varepsilon)}{\langle f, \textbf{busy}(x), \vec{\sigma} +\!\!+ [\sigma] \rangle \mapsto \langle f, \textbf{busy}(x), \vec{\sigma} +\!\!+ [\sigma, \sigma'] \rangle}$$

$$\text{N-Stop}\ \frac{\text{step}(\sigma) = (\sigma', \textbf{return}(v))}{\langle f, \textbf{busy}(x), \vec{\sigma} +\!\!+ [\sigma] \rangle \overset{\text{stop}(x,v)}{\longmapsto} \langle f, \textbf{idle}, [\sigma] \rangle}$$

Figure 3. A naive semantics of serverless functions.

4 A Simpler Serverless Semantics

A natural way to make serverless programming easier is to simplify the model. For example, we could execute functions exactly once, or avoid reusing state. Unfortunately, implementing these changes is likely to be expensive (and, in many situations, beyond our control). Therefore, the approach that we take in this section is to define a simpler, *naive semantics* of serverless computing. Moreover, using a weak bisimulation theorem, we precisely characterize when the naive semantics and $\lambda_{\mathbb{A}}$ coincide. This theorem addresses the low-level details of serverless execution once and for all, thus freeing programmers to reason using the naive semantics instead of $\lambda_{\mathbb{A}}$.

In the naive serverless semantics (Figure 3), serverless functions (f) are the same as the serverless functions in $\lambda_{\mathbb{A}}$. However, the operational semantics of the naive platform is much simpler: the platform runs a single function f on one request at a time, without any concurrency or failures. At each step of execution, the naive semantics either (1) starts processing a new event if the platform is idle, (2) takes an internal step if the platform is busy, or (3) responds and returns to being idle. The state of a naive platform consists of (1) the function's name (f); (2) its execution mode (m); and (3) a trace of function states ($\vec{\sigma}$), where the last element of the trace is the current state, and the first element was the initial state of the function instance when it received the current request. The trace is a convenience that helps us relate the naive semantics to $\lambda_{\mathbb{A}}$, but has no effect on execution.

Serverless function correctness. Note that the naive semantics is an idealized model and is *not correct* for arbitrary serverless functions. However, we can precisely characterize the exact conditions when it is safe for a programmer to reason with the naive semantics, even if their code is running on a full-fledged serverless platform (i.e., using $\lambda_{\mathbb{A}}$). We require the programmer to define a *safety relation* ($\mathcal{R}$) over the state of the serverless function. At a high-level, the programmer must ensure that (1) $\mathcal{R}$ is an equivalence relation, (2) pairs of equivalent states step to pairs of equivalent states, and (3) that the serverless function produces the same observable command (if any) on equivalent states. The safety relation is formally defined as follows.

Definition 4.1 (Safety Relation)**.** For a serverless function $\langle f, \sigma_0, \Sigma, \text{recv}, \text{step} \rangle$, the relation $\mathcal{R} \subseteq \Sigma \times \Sigma$ is a *safety relation* if:

1. $\mathcal{R}$ is an equivalence relation,
2. for all $(\sigma_1, \sigma_2) \in \mathcal{R}$ and v, $(\text{recv}(v, \sigma_1), \text{recv}(v, \sigma_2)) \in \mathcal{R}$,
3. for all $(\sigma_1, \sigma_2) \in \mathcal{R}$, if $(\sigma_1', t_1) = \text{step}(\sigma_1)$ and $(\sigma_2', t_2) = \text{step}(\sigma_2)$ then $(\sigma_1', \sigma_2') \in \mathcal{R}$ and $t_1 = t_2$.

Weak bisimulation. We now prove a weak bisimulation between the naive semantics and $\lambda_{\mathbb{A}}$, conditioned on the serverless functions satisfying the safely relation defined above. We prove a weak (rather than a strong) bisimulation because $\lambda_{\mathbb{A}}$ models serverless execution in more detail. Therefore, a single step in the naive semantics may correspond to several steps in $\lambda_{\mathbb{A}}$. The theorem below states that the naive semantics and $\lambda_{\mathbb{A}}$ are indistinguishable to the programmer, modulo unobservable steps.

To establish the weak bisimulation, we define a relation ($\mathcal{A} \approx \mathbb{C}$) that precisely describes when naive state ($\mathcal{A}$) is equivalent to a $\lambda_{\mathbb{A}}$ state ($\mathbb{C}$), as follows.

Definition 4.2 (Bisimulation Relation)**.** $\mathcal{A} \approx \mathbb{C}$ is defined as:

1. $\langle f, \textbf{idle}, \vec{\sigma} \rangle \approx \mathbb{C}$, and
2. If for all $\mathbb{F}(f, \textbf{busy}(x), \sigma, y) \in \mathbb{C}$, $\exists \sigma'.\sigma' \in \vec{\sigma}$ such that $(\sigma, \sigma') \in \mathcal{R}$ then $\langle f, \textbf{busy}(x), \vec{\sigma} \rangle \approx \mathbb{R}(f, x, v)\ \mathbb{C}$.

This relation formally captures several key ideas that are necessary to reason about serverless execution. (1) A single naive state may be equivalent to multiple distinct $\lambda_{\mathbb{A}}$ states. This may occur due to failures and restarts. (2) Conversely, a single $\lambda_{\mathbb{A}}$ state may be equivalent to several naive states. This occurs when a serverless platform is processing several requests. In fact, we require all $\lambda_{\mathbb{A}}$ states to be equivalent to all *idle* naive states, which is necessary for $\lambda_{\mathbb{A}}$ to receive requests at any time. (3) The $\lambda_{\mathbb{A}}$ state may have several function instances evaluating the same request. (4) Due to warm starts, the state of a function may not be identical in the two semantics; however, they will be equivalent (per $\mathcal{R}$). (5) Due to failures, the $\lambda_{\mathbb{A}}$ semantics can "fall behind" the naive semantics during evaluation, but the state of any function instance in $\lambda_{\mathbb{A}}$ will be equivalent to some state in the execution history of the naive semantics. The weak bisimulation theorem accounts for failures, by specifying the series of $\lambda_{\mathbb{A}}$ steps needed to then catch up with the naive semantics before an observable event occurs.

An important simplification in the naive semantics is that it executes a single request at a time. Therefore, to relate a naive execution to a $\lambda_{\mathbb{A}}$ execution, we need to filter out events that are generated by other requests. To do so, we

define $x(\vec{\ell})$ as the sub-sequence of ℓ that only consists of events labeled x.

We can now state the weak bisimulation theorem.

Theorem 4.3 (Weak Bisimulation). *For a serverless function f with a safety relation $\mathcal{R}$, for all $\mathcal{A}$, $\mathbb{C}$, ℓ:*

1. *For all $\mathcal{A}'$, if $\mathcal{A} \overset{\ell}{\mapsto} \mathcal{A}'$ and $\mathcal{A} \approx \mathbb{C}$ then there exists $\vec{\ell_1}$, $\vec{\ell_2}$, $\mathbb{C}'$, $\mathbb{C}_i$ and $\mathbb{C}_{i+1}$ such that $\mathbb{C} \overset{\vec{\ell_1}}{\Rightarrow} \mathbb{C}_i \overset{\ell}{\Rightarrow} \mathbb{C}_{i+1} \overset{\vec{\ell_2}}{\Rightarrow} \mathbb{C}'$, $x(\vec{\ell_1}) = \varepsilon$, $x(\vec{\ell_2}) = \varepsilon$, and $\mathcal{A}' \approx \mathbb{C}'$*
2. *For all $\mathbb{C}'$, if $\mathbb{C} \overset{\ell}{\Rightarrow} \mathbb{C}'$ and $\mathcal{A} \approx \mathbb{C}$ then there exists $\mathcal{A}'$ such that $\mathcal{A}' \approx \mathbb{C}'$ and $\mathcal{A} \overset{\ell}{\mapsto\!\!\!\!\rightarrow} \mathcal{A}'$.*

Proof. By Theorems A.4 and A.5 in the appendix (part of the submitted supplemental material). □

The first part of this theorem states that every step in the naive semantics corresponds to some sequence of steps in $\lambda_{\mathbb{S}}$. We can interpret this as the sequence of steps that a serverless platform needs to execute to faithfully implement the naive semantics. On the other hand, the second part of this theorem states that any arbitrary step in $\lambda_{\mathbb{S}}$— including failures, retries, and warm starts — corresponds to a (possibly empty) sequence of steps in the naive semantics.

In summary, this theorem allows programmers to justifiably ignore the low-level details of $\lambda_{\mathbb{S}}$, and simply use the naive semantics, if their code satisfies Definition 4.1. There are now several tools that are working toward verifying these kinds of properties in scripting languages, such as JavaScript [19, 34], which is the most widely supported language for writing serverless functions. Our work, which is source language-neutral, complements this work by establishing the verification conditions necessary for correct serverless execution.

5 Serverless Functions and Cloud Storage

It is common for serverless functions to use an external database for persistent storage because their local state is ephemeral. But, serverless platforms warn programmers that stateful serverless functions must be *idempotent* [23, 30, 32]. In other words, they should be able to tolerate re-execution. Unfortunately, it is completely up to programmers to ensure that their code is idempotent, and platforms do not provide a clear explanation of what idempotence means, given that serverless functions perform warm-starts, execute concurrently, and may fail at any time. We now address these problems by adding a key-value store to both $\lambda_{\mathbb{S}}$ and the naive semantics, and present an extended weak bisimulation. In particular, the naive semantics still processes a single request at a time, which is a convenient mental model for programmers.

Figure 4 augments $\lambda_{\mathbb{S}}$ with a key-value store that supports transactions. To the set of components, we add exactly one key-value store ($\mathbb{D}(M, L)$), which has a map from keys to values (M) and a lock (L), which is either unlocked (**free**) or contains uncommitted updates from the function instance that holds the lock (**owned**(y, M')). Note that the lock is held by a function instance and not a request, since there may be several running instances processing the same request. We allow serverless functions to produce four new commands: **beginTx** starts a transaction, **endTx** commits a transaction, **read**(k) reads the value associated with key k, and **write**(k, v) sets the key k to value v. We add four new rules to $\lambda_{\mathbb{S}}$ that execute these commands in the natural way: BeginTx blocks until it can acquire a lock, EndTx commits changes and releases a lock, and for simplicity, the Read and Write rules require the running instance to have a lock. Finally, we need a fifth rule (DropTx) that releases a lock and discards its uncommitted changes if the function instance that held the lock no longer exists. This may occur if the function instance dies before committing its changes.

Serverless Functions

Key set	$k \coloneqq$	strings
Key-value map	$M \in k \rightharpoonup v$	
Commands	$t \coloneqq \cdots$	
	$\mid$ **beginTx**	Lock data store
	$\mid$ **read**(k)	Read value
	$\mid$ **write**(k, v)	Write value
	$\mid$ **endTx**	Unlock data store
Component set	$\mathbb{C} \coloneqq \{C_1, \cdots, C_n, \mathbb{D}(M, L)\}$	
Lock state	$L \coloneqq$ **free**	Data store is free
	$\mid$ **owned**(y, M)	Owned by y

$$\textsc{Read}\ \frac{\text{step}(\sigma) = (\sigma', \textbf{read}(k)) \qquad \text{recv}(M'(k), \sigma') = \sigma''}{\begin{array}{l} C\mathbb{F}(f, \sigma, \textbf{busy}(x), y)\mathbb{D}(M, \textbf{owned}(y, M')) \\ \Rightarrow\ C\mathbb{F}(f, \sigma'', \textbf{busy}(x), y)\mathbb{D}(M, \textbf{owned}(y, M')) \end{array}}$$

$$\textsc{Write}\ \frac{\text{step}(\sigma) = (\sigma', \textbf{write}(k, v)) \qquad M'' = M'[k \mapsto v]}{\begin{array}{l} C\mathbb{F}(f, \sigma, \textbf{busy}(x), y)\mathbb{D}(M, \textbf{owned}(y, M')) \\ \Rightarrow\ C\mathbb{F}(f, \sigma', \textbf{busy}(x), y)\mathbb{D}(M, \textbf{owned}(y, M'')) \end{array}}$$

$$\textsc{BeginTx}\ \frac{\text{step}(\sigma) = (\sigma', \textbf{beginTx})}{\begin{array}{l} C\mathbb{F}(f, \sigma, \textbf{busy}(x), y)\mathbb{D}(M, \textbf{free}) \\ \Rightarrow\ C\mathbb{F}(f, \sigma', \textbf{busy}(x), y)\mathbb{D}(M, \textbf{owned}(y, M)) \end{array}}$$

$$\textsc{EndTx}\ \frac{\text{step}(\sigma) = (\sigma', \textbf{endTx})}{\begin{array}{l} C\mathbb{F}(f, \sigma, \textbf{busy}(x), y)\mathbb{D}(M, \textbf{owned}(y, M')) \\ \Rightarrow\ C\mathbb{F}(f, \sigma', \textbf{busy}(x), y)\mathbb{D}(M', \textbf{free}) \end{array}}$$

$$\textsc{DropTx}\ \frac{\forall f\, \sigma\, x.\ \mathbb{F}(f, \sigma, \textbf{busy}(x), y) \notin C}{C\mathbb{D}(M, \textbf{owned}(y, M')) \Rightarrow C\mathbb{D}(M, \textbf{free})}$$

Figure 4. $\lambda_{\mathbb{S}}$ augmented with a key-value store.

Idempotence in the naive semantics. There are several ways to ensure that a serverless function is idempotent. A common protocol is to commit each output value, keyed by the unique request ID, to the key-value store, within a transactional update. Therefore, if the request is re-tried,

Serverless Functions

Optional key-value map $\quad \widetilde{M} := \mathbf{locked}(M, \vec{\sigma}) \mid \mathbf{commit}(v) \mid \cdot$

$$\boxed{\widetilde{M}, \mathcal{A} \overset{\ell}{\mapsto} \widetilde{M}, \mathcal{A}}$$

$$\frac{\mathcal{A} \mapsto \mathcal{A}'}{\widetilde{M}, \mathcal{A} \mapsto \widetilde{M}, \mathcal{A}'} \qquad \frac{\mathcal{A} \xrightarrow{\mathbf{start}(x,v)} \mathcal{A}'}{\cdot, \mathcal{A} \xrightarrow{\mathbf{start}(x,v)} \cdot, \mathcal{A}'} \qquad \frac{\mathcal{A} \xrightarrow{\mathbf{stop}(x,v)} \mathcal{A}'}{\mathbf{commit}(v), \mathcal{A} \xrightarrow{\mathbf{stop}(x,v)} \cdot, \mathcal{A}'}$$

$$\text{N-Read}\ \frac{\mathrm{step}(\sigma) = (\sigma', \mathbf{read}(k)) \qquad \mathrm{recv}(M(k), \sigma') = \sigma''}{\begin{array}{l}\mathbf{locked}(M, \vec{\sigma}'), \langle f, \mathbf{busy}(x), \vec{\sigma} +\!\!+ [\sigma] \rangle \\ \mapsto \mathbf{locked}(M, \vec{\sigma}'), \langle f, \mathbf{busy}(x), \vec{\sigma} +\!\!+ [\sigma, \sigma', \sigma''] \rangle\end{array}}$$

$$\text{N-Write}\ \frac{\mathrm{step}(\sigma) = (\sigma', \mathbf{write}(k, v)) \qquad M' = M[k \mapsto v]}{\begin{array}{l}\mathbf{locked}(M, \vec{\sigma}'), \langle f, \mathbf{busy}(x), \vec{\sigma} +\!\!+ [\sigma] \rangle \\ \mapsto \mathbf{locked}(M', \vec{\sigma}'), \langle f, \mathbf{busy}(x), \vec{\sigma} +\!\!+ [\sigma, \sigma'] \rangle\end{array}}$$

$$\text{N-BeginTx}\ \frac{\mathrm{step}(\sigma) = (\sigma', \mathbf{beginTx})}{\begin{array}{l}\cdot, \langle f, \mathbf{busy}(x), \vec{\sigma} +\!\!+ [\sigma] \rangle \\ \mapsto \mathbf{locked}(M, \vec{\sigma} +\!\!+ [\sigma]), \langle f, \mathbf{busy}(x), \vec{\sigma} +\!\!+ [\sigma, \sigma'] \rangle\end{array}}$$

$$\text{N-EndTx}\ \frac{\mathrm{step}(\sigma) = (\sigma', \mathbf{endTx})}{\begin{array}{l}\mathbf{locked}(M, \vec{\sigma}'), \langle f, \mathbf{busy}(x), \vec{\sigma} +\!\!+ [\sigma] \rangle \\ \mapsto \mathbf{commit}(M(x)), \langle f, \mathbf{busy}(x), \vec{\sigma} +\!\!+ [\sigma, \sigma'] \rangle\end{array}}$$

$$\text{N-Rollback}\ \frac{}{\mathbf{locked}(M, \vec{\sigma}'), \langle f, \mathbf{busy}(x), \vec{\sigma} \rangle \mapsto \cdot, \langle f, \mathbf{busy}(x), \vec{\sigma}' \rangle}$$

Figure 5. The naive semantics with a key-value store.

the function can lookup and return the saved output value. We now formally characterize this protocol, and use it to prove a weak bisimulation theorem between $\lambda_{\mathbb{K}}$ and the naive semantics, where each is extended with a key-value store. This will allow programmers to reason about serverless execution using the naive semantics, which processes exactly one request at a time, without concurrency.

The challenge we face is to extend the bisimulation relation (Definition 4.2) to account for the key-value store. In that definition, when the $\lambda_{\mathbb{K}}$ state ($\mathbb{C}$) and the naive state ($\mathcal{A}$) are equivalent ($\mathcal{A} \approx \mathbb{C}$), it is possible for a failure to occur ($\mathbb{C} \Rightarrow \mathbb{C}'$), such that after the failure, $\lambda_{\mathbb{K}}$ falls behind the naive semantics. Nevertheless, we still treat the states as equivalent ($\mathcal{A} \approx \mathbb{C}'$), and let the weak bisimulation proof re-invoke a function in $\mathbb{C}'$ until it catches up with $\mathcal{A}$. Unfortunately, for an arbitrary serverless function, replay does not work because the key-value store may have changed. To address this, we need to ensure that functions that use the key-value store follow an appropriate protocol to ensure idempotence. A more subtle problem arises when the naive state ($\mathcal{A}$) is within a transaction, and the equivalent $\lambda_{\mathbb{K}}$ state takes several steps ($\mathbb{C} \Rightarrow \mathbb{C}'$) that result in a failure, followed by other updates to the key-value store. If this occurs, the naive semantics must rollback to the start of the transaction and re-execute with the key-value store in $\mathbb{C}'$.

Figure 5 shows the extended naive semantics, which addresses these issues. In this semantics, the naive key-value store ($\widetilde{M}$) goes through three states: (1) at the start of execution, it is not present; (2) when a transaction begins, the semantics selects a new mapping nondeterministically; and (3) when the transaction completes, the mapping moves to a committed state, where it only contains the final result. For simplicity, we assume that reads and writes only occur within transactions. The semantics also includes a N-Rollback rule, which allows execution to rollback to the start of transaction. However, once a transaction is complete (N-EndTx), a rollback is not possible.

The extended bisimulation relation, shown below, uses the bisimulation relation from the previous section (Definition 4.2). When the naive semantics is within a transaction, the relation requires some instance in $\lambda_{\mathbb{K}}$ to be operating in lock-step with the naive semantics. However, it allows other instances that are not using the key-value store to make progress. Therefore, a transaction is not globally atomic in $\lambda_{\mathbb{K}}$, and other requests can be received and processed while some instance is in a transaction.

Definition 5.1 (Extended Bisimulation Relation). $\widetilde{M}, \mathcal{A} \approx \mathbb{D}(M, L), \mathbb{C}$ is defined as:

1. If $\mathcal{A} \approx \mathbb{C}$ then $\cdot, \mathcal{A} \approx \mathbb{D}(M, \mathbf{free})\mathbb{C}$, or
2. If $\mathcal{A} = \langle f, \mathbf{busy}(x), \vec{\sigma} +\!\!+ [\sigma] \rangle$, $L = \mathbf{owned}(y, M')$, $\widetilde{M} = \mathbf{locked}(M', \vec{\sigma})$, and there exists $\mathbb{F}(f, \widetilde{\sigma}, \mathbf{busy}(x), y) \in \mathbb{C}$ such that $(\sigma, \widetilde{\sigma}) \in \mathcal{R}$ then $\widetilde{M}, \mathcal{A} \approx \mathbb{D}(M, L), \mathbb{C}$, or
3. If $\mathcal{A} = \langle f, \mathbf{busy}(x), \vec{\sigma} \rangle$, $\widetilde{M} = \mathbf{commit}(v)$, and $M(x) = v$ then $\widetilde{M}, \mathcal{A} \approx \mathbb{D}(M, \mathbf{free})\mathbb{C}$.

With this definition in place, we can prove an extended weak bisimulation relation.

Theorem 5.2 (Extended Weak Bisimulation). *For a serverless function f, if $\mathcal{R}$ is its safely relation and for all requests $\mathbb{R}(f, x, v)$ the following conditions hold:*

1. *f produces the value stored at key x, if it exists,*
2. *When f completes a transaction, it stores a value v' at key x, and*
3. *When f stops, it produces v',*

then, for all $\widetilde{M}, \mathcal{A}, \mathbb{C}, \ell$:

1. *For all $\widetilde{M}', \mathcal{A}'$, if $\widetilde{M}, \mathcal{A} \overset{\ell}{\mapsto} \widetilde{M}', \mathcal{A}'$ and $\mathcal{A} \approx \mathbb{C}$ then there exists $\vec{\ell_1}, \vec{\ell_2}, \mathbb{C}', \mathbb{C}_i$ and $\mathbb{C}_{i+1}$ such that $\mathbb{C} \overset{\vec{\ell_1}}{\Rightarrow} \mathbb{C}_i \overset{\ell}{\Rightarrow} \mathbb{C}_{i+1} \overset{\vec{\ell_1}}{\Rightarrow} \mathbb{C}'$, $x(\vec{\ell_1}) = \varepsilon$, $x(\vec{\ell_2}) = \varepsilon$, and $\mathcal{A}' \approx \mathbb{C}'$*
2. *For all $\mathbb{C}'$, if $\mathbb{C} \overset{\ell}{\Rightarrow} \mathbb{C}'$ and $\widetilde{M}, \mathcal{A} \approx \mathbb{C}$ then there exists $\mathcal{A}'$ such that $\mathcal{A}' \approx \mathbb{C}'$ and $\mathcal{A} \overset{\ell}{\longmapsto} \mathcal{A}'$.*

Proof. By Theorems A.9 and A.10 in the appendix (part of the submitted supplemental material). □

The theorem statement in the appendix formalizes the conditions of the bisimulation, but the less formal conditions are useful for programmers, since they are simple requirements that are easy to ensure. Therefore, this theorem gives

the assurance that the reasoning with the naive semantics is adequate, even though the serverless platform operates using $\lambda_{\mathbb{A}}$.

6 Serverless Compositions

Thus far, we have discussed the basic serverless computing abstraction, where serverless functions are black-boxes to the platform. However, there are many situations where this abstraction makes it hard to write serverless programs in a modular way [7]. This section extends $\lambda_{\mathbb{A}}$ with a significant new feature: a *serverless programming language* (SPL) for composing serverless functions that is designed to directly run on a serverless platform (i.e., without operating system virtualization). We motivate the need for SPL (§6.1), present the design of SPL (§6.2), and discuss our implementation (§6.3). Then, §7 evaluates SPL's performance.

6.1 The Need for Serverless Composition Languages

Baldini et al. [7] present several problems that arise when programmers build new serverless functions by composing existing serverless functions. The natural approach to serverless function composition is to have one serverless function act as a coordinator that invokes other functions. For example, Figure 6 shows a serverless function that deposits funds into two accounts by calling our example function (bank) from §2. A problem with this approach is that the programmer gets "double-billed" for the time spent running the coordinator function (deposit2) which is mostly idle and the time spent doing actual work in bank. An alternative approach is to merge several functions into a single function. Unfortunately, this approach hinders code-reuse. In particular, it does not work when source code is unavailable or when the serverless functions are written in different languages. A third approach is to write serverless functions that each pass their output as input to another function, instead of returning to the caller (i.e., continuation-passing style). However, this approach requires rewriting code. Moreover, some clients, such as web browsers, cannot produce a continuation URL to receive the final result. Baldini et al. show that the only way to address these problems is to add primitive composition operators to serverless platforms [7].

6.2 Composing Serverless Functions with Arrows

We now extend $\lambda_{\mathbb{A}}$ with a domain specific language for composing serverless functions, which we call SPL (*serverless programming language*). Since the serverless platform is a shared resource and programs are untrusted, SPL cannot run arbitrary code. However, SPL programs can invoke serverless functions to perform arbitrary computation when needed. Therefore, invoking a serverless function is a primitive operation in SPL, which serves as the wiring between several serverless functions.

```
let request = require('request-promise-native');
exports.deposit2 = function(req, res) {
  let { amount, tId1, tId2 } = req.body;
  if (amount > 100) {
    request.post({ url: ..., json: { type: 'deposit',
      to: 'checking', amount: amount - 100,
      transId: tId1 }})
    .then(function() {
      request.post({ url: ..., json: { type: 'deposit',
        to: 'savings', amount: 100, transId: tId2 }})
      .then(function() { res.send(true); });});
  } else {
    request.post({ url: ..., json: { type: 'deposit',
      to: 'checking', amount: amount, transId: tId1 }})
    .then(function() { res.send(true); });};
```

Figure 6. A serverless function to deposit funds, based on the amount provided, using the serverless function in Figure 1b.

Values	$v \coloneqq \cdots$	
	$\mid (v_1, v_2)$	Tuples
SPL expressions	$e \coloneqq$ invoke f	Invoke serverless function
	$\mid$ first e	Run e to first part of input
	$\mid e_1 \ggg e_2$	Sequencing
SPL continuations	$\kappa \coloneqq$ ret x	Response to request
	$\mid$ seq $e\ \kappa$	In a sequence
	$\mid$ first $v\ \kappa$	In first
Components	$C \coloneqq \cdots$	
	$\mid \mathbb{E}(e, v, \kappa)$	Running program
	$\mid \mathbb{E}(x, \kappa)$	Waiting program
	$\mid \mathbb{R}(e, x, v)$	Run program e on v

$\boxed{\mathbb{C} \overset{\ell}{\mapsto} \mathbb{C}}$

$$\text{P-NewReq}\ \frac{x \text{ is fresh}}{C \xrightarrow{\text{start}(v)} C\mathbb{R}(e, x, v)}$$

$$\text{P-Start}\ C\mathbb{R}(e, x, v) \Rightarrow C\mathbb{R}(e, x, v)\mathbb{E}(e, v, \text{ret } x)$$

$$\text{P-Respond}\ C\mathbb{E}(v', \text{ret } x)\mathbb{R}(e, x, v) \xrightarrow{\text{stop}(v')} C\mathbb{S}(x, v')$$

$$\text{P-Seq1}\ C\mathbb{E}(e_1 \ggg e_2, v, \kappa) \Rightarrow C\mathbb{E}(e_1, v, \text{seq } e_2\ \kappa)$$

$$\text{P-Seq2}\ C\mathbb{E}(v, \text{seq } e\ \kappa) \Rightarrow C\mathbb{E}(e, v, \kappa)$$

$$\text{P-Invoke1}\ \frac{x' \text{ is fresh}}{C\mathbb{E}(\text{invoke } f, v, \kappa) \Rightarrow C\mathbb{E}(x', \kappa)\mathbb{R}(f, x', v)}$$

$$\text{P-Invoke2}\ C\mathbb{E}(x, \kappa)\mathbb{S}(x, v) \Rightarrow C\mathbb{E}(v, \kappa)$$

$$\text{P-First1}\ C\mathbb{E}(\text{first } e, (v_1, v_2), \kappa) \Rightarrow C\mathbb{E}(e, v_1, \text{first } v_2\ \kappa)$$

$$\text{P-First2}\ C\mathbb{E}(v_1, \text{first } v_2\ \kappa) \Rightarrow C\mathbb{E}((v_1, v_2), \kappa)$$

$$\text{P-Die}\ C\mathbb{E}(v, \kappa) \Rightarrow C$$

Figure 7. Extending $\lambda_{\mathbb{A}}$ with SPL.

Figure 7 extends the $\lambda_{\mathbb{A}}$ with SPL. This extension allows requests to run SPL programs ($\mathbb{R}(e, x, v)$), in addition to ordinary requests that name serverless functions.[4] SPL is based

[4] In practice, a request would name an SPL program instead of carrying the program itself.

SPL expressions	$e ::= \cdots \mid p$	Run transformation
JSON values	$v ::= n \mid b \mid str \mid \text{null}$	
JSON pattern	$p ::= v$	JSON literal
	$\mid [p_1, \cdots, p_n]$	Array
	$\mid \{str_1 : p_1, \cdots, str_n : p_n\}$	Object
	$\mid p_1 \; op \; p_2$	Operators
	$\mid \texttt{if}\ (p_1)\ \texttt{then}\ p_2\ \texttt{else}\ p_3$	Conditional
	$\mid [str_1 \rightarrow p_1]$	Update field
	$\mid \text{in}\ q$	Input reference
JSON query	$q ::=$	Empty query
	$\mid .[n]q$	Array index
	$\mid .idq$	Field lookup

Figure 8. JSON transformation language.

on Hughes' arrows [27], thus it supports the three basic arrow combinators. An SPL program can (1) invoke a serverless function (invoke f); (2) run two subprograms in sequence ($e_1 \ggg e_2$); or (3) run a subprogram on the first component of a tuple, and return the second component unchanged (first e). These three operations are sufficient to describe loop- and branch-free compositions of serverless functions. It is straightforward to add support for bounded loops and branches, which we do in our implementation.

To run SPL programs, we introduce a new kind of component ($\mathbb{E}$) that executes programs using an abstract machine that is similar to a CK machine [15]. In other words, the evaluation rules define a small-step semantics with an explicit representation of the continuation (κ). This design is necessary because programs need to suspend execution to invoke serverless functions (P-INVOKE1) and then later resume execution (P-INVOKE2). Similar to serverless functions, SPL programs also execute at-least-once. Therefore, a single request may spawn several programs (P-START) and a program may die while waiting for a serverless function to response (P-DIE).

A sub-language for JSON transformations. A problem that arises in practice is that input and output values to serverless functions (v) are frequently formatted as JSON values, which makes hard to define the first operator in a satisfactory way. For example, we could define first to operate over two-element JSON arrays, and then require programmers to write serverless functions to transform arbitrary JSON into this format. However, this approach is cumbersome and resource-intensive. For even simple transformations, the programmer would have to write and deploy serverless functions; the serverless platform would need to sandbox the process using heavyweight OS mechanisms; and the platform would have to copy values to and from the process.

Instead, we augment SPL with a sub-language of JSON transformations (Figure 8). This language is a superset of JSON. It has a distinguished variable (in) that refers to the input JSON value, which may be followed by a query to select fragments of the input. For example, we can use this transformation language to write an SPL program that receives

```
a <- invoke f(in); b <- invoke g(in);
c <- invoke h({ x: b, y: a.d }); ret c;
```

(a) Surface syntax program.

```
[in, { input: in }] >>> first (invoke f) >>>
[in[1].input, in[1][a -> in[0]]] >>>
first (invoke g) >>>
[{ x: in[0], y: in[1].a.d }, in[1]] >>>
first (invoke h) >>> in[0]
```

(b) Naive translation to SPL.

```
[in, { input: in }] >>> first (invoke f) >>>
[in[1].input, { a: in[0] }] >>> first (invoke g) >>>
[{ x: in[0], y: in[1].a.d }, {}] >>>
first (invoke h) >>> in[0]
```

(c) Live variable analysis eliminates several fields.

```
[in, { input: in }] >>> first (invoke f) >>>
[in[1].input, { ad: in[0].d }] >>> first (invoke g) >>>
[{ x: in[0], y: in[1].ad }, {}] >>>
first (invoke h) >>> in[0]
```

(d) Live key analysis immediately projects a.d.

Figure 9. Compiling the surface syntax of SPL.

a two-element array as input and then runs two different serverless functions on each element:

```
first (invoke f) >>> [in[1], in[0]] >>> first (invoke g)
```

Without the JSON transformation language, we would need an auxiliary serverless function to swap the elements.

A simpler notation for SPL programs. SPL is designed to be a minimal set of primitives that are straightforward to implement in a serverless platform. However, SPL programs are difficult for programmers to comprehend. To address this problem, we have also developed a surface syntax for SPL that is based on Paterson's notation for arrows [35]. Using the surface syntax, we can rewrite the previous example as follows:

```
x <- invoke f(in[0]); y <- invoke g(in[1]); ret [y, x];
```

This version is far less cryptic than the original.

We describe the surface syntax compiler by example. At a high level, the compiler produces SPL programs in store-passing style. For example, Figure 9a shows a surface syntax program that invokes three serverless functions (f, g, and h). However, the composition is non-trivial because the input of each function is not simply the output of the previous function. We compile this program to an equivalent SPL program that uses the JSON transformation language to save intermediate values in a dictionary (Figure 9b). However, this naive translation carries unnecessary intermediate state. We address this problem with two optimizations. First, the compiler performs a live variable analysis, which produces the more compact program shown in Figure 9c. In the original program, the input reference (in) is not live after g, and

c is the only live variable after h, thus these are eliminated from the state. Second, the compiler performs a liveness analysis of the JSON keys returned by serverless functions, which produces an even smaller program (Figure 9d). In our example, f returns an object a, but the program only uses $a.d$ and discards any other fields that a may have. There are many situations where the entire object a may be significantly larger than $a.d$, thus extracting it early can shrink the amount of state a program carries.

6.3 Implementation

OpenWhisk implementation. Apache OpenWhisk is a mature and widely-deployed serverless platform that is written in Scala and is the foundation of IBM Cloud Functions. We have implemented SPL as a 1200 LOC patch to OpenWhisk, which includes the surface syntax compiler and several changes to the OpenWhisk runtime system. We inherit OpenWhisk's fault tolerance mechanisms (e.g., at-least-once execution) and reuse OpenWhisk's support for serverless function sequences [7] to implement the $\ggg$ operator of SPL.

Our OpenWhisk implementation of SPL has three differences from the language presented so far. First, it supports bounded loops, which are a programming convenience. Second, instead of implementing the first operator and the JSON transformation language as independent expressions, we have a single operator that performs the same action as first, but applies a JSON transformation to the input and output, which is how transformations are most commonly used. Finally, we implement a multi-armed conditional, which is a straightforward extension to SPL. These operators allow us to compile the surface syntax to smaller SPL programs, which moderately improves performance.

Portable implementation. We have also built a portable implementation of SPL (1156 LOC of Rust) that can invoke serverless functions in public clouds. (We have tested with Google Cloud Functions.) Whereas the OpenWhisk implementation allows us to carefully measure load and utilization on our own hardware test-bed, we cannot perform the same experiments with our standalone implementation, since public clouds abstract away the hardware used to run serverless functions. The portable implementation has helped us ensure that the design of SPL is independent of the design and implementation of OpenWhisk, and we have used it to explore other kinds of features that a serverless platform may wish to provide. For example, we have added a fetch operator to SPL that receives the name of a file in cloud storage as input and produces the file's contents as output. It is common to have serverless functions fetch private files from cloud storage (after an access control check). The fetch operator can make these kinds of functions faster and consume fewer resources.

7 Evaluation

This section first evaluates the performance of SPL using microbenchmarks, and then highlights the expressivity of SPL with three case studies. We will release our implementations as open-source and submit our experiments to the AEC.

7.1 Comparison to OpenWhisk Conductor

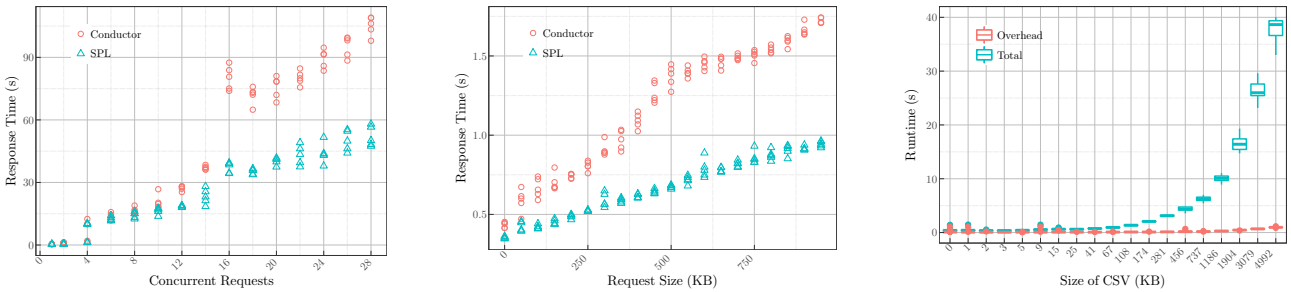
The Apache OpenWhisk serverless platform has built-in support for composing serverless functions using a *Conductor* [37]. A Conductor is special type of serverless function that, when invoked, may respond with the name and arguments of an auxiliary serverless function for the platform to invoke, instead of returning immediately. When the auxiliary function returns a response, the platform re-invokes the Conductor with the response value. Therefore, the platform interleaves the execution of the Conductor function and its auxiliary functions, which allows a Conductor to implement sequential control flow, similar to SPL. The key difference between SPL and a Conductor, is that SPL is designed to run directly on the platform, whereas the Conductor is a serverless function itself, which consumes additional resources.

This section compares the performance of SPL to Conductor with a microbenchmark that stresses the performance of the serverless platform. The benchmark SPL program runs a sequence of ten serverless functions, and we translate it to an equivalent program for Conductor. We run two experiments that (1) vary the number of concurrent requests and (2) vary the size of the requests. In both experiments, we measure end-to-end response time, which is the metric that is most relevant to programmers. We find that SPL outperforms Conductor in both experiments, which is expected because its design requires fewer resources, as explained below.

We conduct our experiments on a six-core Intel Xeon E5-1650 with 64 GB RAM with Hyper-Threading enabled.

Concurrent invocations. Our first experiment shows how response times change when the system is processing several concurrent requests. We run N concurrent requests of the same program, and then measure five response times. Figure 10a shows that SPL is slightly faster than Conductor when $N \leq 12$, but approximately twice as fast when $N > 12$. We attribute this to the fact that Conductor interleaves the conductor function with the ten serverless functions, thus it requires twice as many containers to run. Moreover, since our CPU has six hyper-threaded cores, Conductor is overloaded with 12 concurrent requests.

Request size. Our second experiment shows how response times depend on the size of the input request. We use the same microbenchmark as before, and ensure that the platform only processes one request at a time. We vary the request body size from 0KB to 1MB and measure five response times at each request size. Figure 10b shows that SPL is almost twice as fast as Conductor. We again attribute this to



(a) Response time vs. concurrent requests.

(b) Response time vs. request size.

(c) Response time and overhead.

Figure 10. SPL benchmarks. Figures 10a and 10b show that SPL is faster than OpenWhisk Conductor when the serverless platform is processing several concurrent requests, or when the request size increases. Figure 10c shows that most of the execution time is spent in serverless functions, and not running SPL.

the fact that Conductor needs to copy the request across a sequence of functions that is twice as long as SPL.

Summary. The OS-based isolation techniques that Conductor uses have a nontrivial cost. SPL, since it uses language-based isolation, is able to lower the resource utilization of serverless compositions by up to a factor of two.

7.2 Case Studies

The core SPL language, presented in §6, is a minimal fragment that lacks convenient features that are needed for real-world programming. To identity the additional features that are necessary, we have written several different kinds of SPL programs, and added new features to SPL when necessary. Fortunately, these new features easily fit the structure established by the core language. This section presents some of the programs that we've built and discusses the new features that they employ. These examples illustrate that it is easy to grow core SPL into a convenient and expressive programming language for serverless function composition.

Conditional bank deposits. Figure 11a uses SPL to rewrite the bank deposit function (Figure 6). The original program suffered the "double billing" problem, since the composite serverless function needs to be active while waiting for the bank serverless function to do actual work. In contrast, the serverless platform suspends the SPL program when it invokes a serverless function and resumes it when the response is ready. This example also shows that our implementation supports basic arithmetic, which we add to the JSON transformation sub-language in a straightforward way.

Continuous integration. Baldini et al. [7] present a compelling example of serverless composition, which we adapt for SPL. Our SPL program (Figure 11b) connects GitHub, Slack, and Google Cloud Build (which is a continuous integration tool, similar to TravisCI). Cloud Build makes it easy to run tests when a new commit is pushed to GitHub. But, it is much harder to see the test results in a convenient way. However,

```
if (in.amount > 100) {
  invoke bank({ type: "deposit", to: "checking",
           amount: in.amount - 100, transId: in.tId1 });
  invoke bank({ type: "deposit", to: "savings",
           amount: 100, transId: in.tId2 });
} else {
  invoke bank({ type: "deposit", to: "checking",
           amount: in.amount, transId: in.tId1 });
} ret;
```

(a) Receive funds, deposit $100 in savings (if feasible), and deposit the rest in checking.

```
invoke postStatusToGitHub({ state: in.state,
  sha: in.sha, url: in.url, repo: "<repo owner/name>" });
if (in.state == "failure") {
  invoke postToSlack({ channel: "<id>", text: "<msg>" });
} ret;
```

(b) Receive build state from Google Cloud Build, set status on GitHub, and report failures on Slack.

```
data <- get in.url;
json <- invoke csvToJson(data);
out <- invoke plotJson({data:json,x:in.xAxis,y:in.yAxis});
ret out;
```

(c) Receive a URL of a CSV file and two column names, download the file, convert it to JSON, then plot the JSON.

Figure 11. Example SPL programs.

Cloud Build can invoke a serverless function when tests complete, and we use it run an SPL program that (1) uses the GitHub API to add a test-status icon next to each commit message, and (2) uses the Slack API to post a message whenever a test fails. However, instead of writing a monolithic serverless function, we first write two serverless functions that post to Slack and set GitHub status icons respectively, and let the SPL program invoke them. It is easy to reuse the GitHub and Slack functions in other applications.

Data visualization. Our last example (Figure 11c) receives the URL of a CSV-formatted data file with two columns, plots

the data, and responds with the plotted image. This is the kind of task that a power-constrained mobile application may wish to offload to a serverless platform, especially when the size of the data is significantly larger than the plot. Our program invokes two independent serverless functions that are trivial wrappers around popular JavaScript libraries: *csvjson*, which converts CSV to JSON and *vega*, which creates plots from JSON data. This example uses a new primitive (get) that our implementation supports to download the data file. Downloading is a common operation that is natural for the platform to provide. Our implementation simply issues an HTTP GET request and suspends the SPL program until the response is available. However, it is easy to imagine more sophisticated implementations that support caching, authorization, and other features. Finally, this example show that our SPL implementation is not limited to processing JSON. The get command produces a plain-text CSV file, and the `plotJson` invocation produces a JPEG image.

A natural question to ask about this example is whether the decomposition into three parts introduces excessive communication overhead. We investigate this by varying the size of the input CSVs (ten trials per size), and measuring the total running time and the overhead, which we define as the time spent outside serverless functions (i.e., transferring data, running get, and applying JSON transformations). Figure 10c shows that even as the file size approaches 5 MB, the overhead remains low (less than 3% for a 5 MB file, and up to 25% for 1 KB file).

8 Related Work

Serverless computing. Baldini et al. [7] introduce the problem of serverless function composition and present a new primitive for serverless function sequencing. Subsequent work develops serverless state machines (Conductor) and a DSL (Composer) that makes state machines easier to write [37]. In §6, we present an alternative set of composition operators that we formalize as an extension to $\lambda_{\mathbb{A}}$, implement in OpenWhisk, and evaluate their performance.

Trapeze [2] presents dynamic IFC for serverless computing, and further sandboxes serverless functions to mediate their interactions with shared storage. Their Coq formalization of termination-sensitive noninterference does not model several features of serverless platforms, such as warm starts and failures, that our semantics does model.

Several projects exploit serverless computing for elastic parallelization [4, 17, 18, 28]. §6 addresses modularity and does not support parallel execution. However, it would be an interesting challenge to grow the DSL in §6 to the point where it can support the aforementioned applications without any non-serverless computing components. It is worth noting that today's serverless execution model is not a good fit for all applications. For example, Singhvi et al. [40] list several barriers to running network functions on serverless platforms.

Serverless computing and other container-based platforms suffer several performance and utilization barriers. There are several ways to address these problems, including datacenter design [20], resource allocation [9], programming abstractions [7, 37], and edge computing [5]. $\lambda_{\mathbb{A}}$ is designed to elucidate subtle semantic issues (not performance problems) that affect programmers building serverless applications.

Language-based approaches to microservices. λ_{FAIL} [38] is a semantics for horizontally-scaled services with durable storage, which are related to serverless computing. A key difference between $\lambda_{\mathbb{A}}$ and λ_{FAIL} is that $\lambda_{\mathbb{A}}$ models *warm-starts*, which occur when a serverless platform runs a new request on an old function instance, without resetting its state. Warm-starts make it hard to reason about correctness, but this paper presents an approach to do so. Both $\lambda_{\mathbb{A}}$ and λ_{FAIL} present weak bisimulations between detailed and naive semantics. However, $\lambda_{\mathbb{A}}$'s naive semantics processes a single request at a time, whereas λ_{FAIL}'s idealized semantics has concurrency. We use $\lambda_{\mathbb{A}}$ to specify a protocol to ensure that serverless functions are idempotent and fault tolerant. However, λ_{FAIL} also presents a compiler that automatically ensures that these properties hold for C# and F# code. We believe the approach would work for $\lambda_{\mathbb{A}}$. §6 extends $\lambda_{\mathbb{A}}$ with new primitives, which we then implement and evaluate.

Whip [41] and ucheck [33] are tools that check properties of microservice-based applications at run-time. These works are complementary to ours. For example, our paper identifies several important properties of serverless functions, which could then be checked using Whip or ucheck.

Cloud orchestration frameworks. Ballerina [42] is a language for managing cloud environments; Engage [16] is a deployment manager that supports inter-machine dependencies; and Pulumi [36] is an embedded DSL for writing programs that configure and run in the cloud. In contrast, $\lambda_{\mathbb{A}}$ is a semantics of serverless computing. §6 uses $\lambda_{\mathbb{A}}$ to design and implement a language for composing serverless functions that runs within a serverless platform.

Verification. There is a large body of work on verification, testing, and modular programming for distributed systems and algorithms (e.g., [6, 10, 12, 13, 21, 24, 25, 39, 43]). The serverless computation model is more constrained than arbitrary distributed systems and algorithms. This paper presents a formal semantics of serverless computing, $\lambda_{\mathbb{A}}$, with an emphasis on low-level details that are observable by programs, and thus hard for programmers to get right. To demonstrate that $\lambda_{\mathbb{A}}$ is useful, we present three applications that employ it and extend it in several ways. This paper does not address verification for serverless computing, but $\lambda_{\mathbb{A}}$ could be used as a foundation for future verification work.

9 Conclusion

We have presented $\lambda_{\mathbb{A}}$, an operational semantics that models the low-level of serverless platforms that are observable by programmers. We have also presented three applications of $\lambda_{\mathbb{A}}$. (1) We prove a weak bisimulation to characterize when programmers can ignore the low-level details of $\lambda_{\mathbb{A}}$. (2) We extend $\lambda_{\mathbb{A}}$ with a key-value store to reason about stateful functions. (3) We extend $\lambda_{\mathbb{A}}$ with a language for serverless function composition, implement it, and evaluate its performance. We hope that these applications show that $\lambda_{\mathbb{A}}$ can be a foundation for further research on language-based approaches to serverless computing.

References

[1] Akkus, I. E., Chen, R., Rimac, I., Stein, M., Satzke, K., Beck, A., Aditya, P., and Hilt, V. SAND: Towards high-performance serverless computing. In *USENIX Annual Technical Conference (ATC)* (2018).

[2] Alpernas, K., Flanagan, C., Fouladi, S., Ryzhyk, L., Sagiv, M., Schmitz, T., and Winstein, K. Secure serverless computing using dynamic information flow control. *Proc. ACM Program. Lang. 2*, OOPSLA (Oct. 2018).

[3] AWS Lambda developer guide: Invoke. https://docs.aws.amazon.com/lambda/latest/dg/API_Invoke.html. Accessed Jul 5 2018, 2018.

[4] Ao, L., Izhikevich, L., Voelker, G. M., and Porter, G. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing* (2018), SoCC '18.

[5] Aske, A., and Zhao, X. Supporting multi-provider serverless computing on the edge. In *Proceedings of the 47th International Conference on Parallel Processing Companion* (2018), ICPP '18.

[6] Bakst, A., Gleissenthall, K. v., K, R. G., and Jhala, R. Verifying distributed programs via canonical sequentialization. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)* (2017).

[7] Baldini, I., Cheng, P., Fink, S. J., Mitchell, N., Muthusamy, V., Rabbah, R., Suter, P., and Tardieu, O. The serverless trilemma: Function composition for serverless computing. In *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* (2017).

[8] Baudart, G., Dolby, J., Duesterwald, E., Hirzel, M., and Shinnar, A. Protecting chatbots from toxic content. In *2018 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (2018).

[9] Björkqvist, M., Birke, R., and Binder, W. Resource management of replicated service systems provisioned in the cloud. In *Network Operations and Management Symposium (NOMS)* (2016).

[10] Chajed, T., Kaashoek, F., Lampson, B., and Zeldovich, N. Verifying concurrent software using movers in CSPEC. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (2018).

[11] Conway, S. Cloud native technologies are scaling production applications. https://www.cncf.io/blog/2017/12/06/cloud-native-technologies-scaling-production-applications/, 2017. Accessed Jul 12 2018.

[12] Desai, A., Phanishayee, A., Qadeer, S., and Seshia, S. A. Compositional programming and testing of dynamic distributed systems. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)* (2018).

[13] Drăgoi, C., Henzinger, T. A., and Zufferey, D. Psync: A partially synchronous language for fault-tolerant distributed algorithms. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (2016).

[14] Ellis, A. OpenFaaS. https://www.openfaas.com. Accessed Jul 5 2018.

[15] Felleisen, M., and Friedman, D. P. Control operators, the SECD-machine, and the λ-calculus. In *Proceedings of the IFIP TC 2/WG 2.2 Working Conference on Formal Description of Programming Concepts* (1986).

[16] Fischer, J., Majumdar, R., and Esmaeilsabzali, S. Engage: A deployment management system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2012).

[17] Fouladi, S., Iter, D., Chatterjee, S., Kozyrakis, C., Zaharia, M., and Winstein, K. A thunk to remember: make-j1000 (and other jobs) on functions-as-a-service infrastructure, 2017.

[18] Fouladi, S., Wahby, R. S., Shacklett, B., Balasubramaniam, K. V., Zeng, W., Bhalerao, R., Sivaraman, A., Porter, G., and Winstein, K. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *USENIX Symposium on Networked System Design and Implementation (NSDI)* (2017).

[19] Fragoso Santos, J., Maksimović, P., Naudžiūnienė, D., Wood, T., and Gardner, P. JaVerT: JavaScript verification toolchain. *Proceedings of the ACM on Programming Languages 2*, POPL (2018), 50:1–50:33.

[20] Gan, Y., and Delimitrou, C. The Architectural Implications of Cloud Microservices. In *Computer Architecture Letters (CAL), vol.17, iss. 2* (Jul-Dec 2018).

[21] Gomes, V. B. F., Kleppmann, M., Mulligan, D. P., and Beresford, A. R. Verifying strong eventual consistency in distributed systems. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)* (2017).

[22] Google Cloud Functions. https://cloud.google.com/functions/. Accessed Jul 5 2018.

[23] Cloud Functions execution environment. https://cloud.google.com/functions/docs/concepts/exec. Accessed Jul 5 2018, 2018.

[24] Guha, A., Reitblatt, M., and Foster, N. Machine verified network controllers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2013).

[25] Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J. R., Parno, B., Roberts, M. L., Setty, S., and Zill, B. Ironfleet: Proving practical distributed systems correct. In *ACM Symposium on Operating Systems Principles (SOSP)* (2015).

[26] Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Serverless computation with OpenLambda. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)* (2016).

[27] Hughes, J. Generalising monads to arrows. *Science of Computer Programming 37*, 1–3 (May 2000), 67–111.

[28] Jonas, E., Pu, Q., Venkataraman, S., Stoica, I., and Recht, B. Occupy the cloud: Distributed computing for the 99%. In *Symposium on Cloud Computing* (2017).

[29] Microsoft Azure Functions. https://azure.microsoft.com/en-us/services/functions/. Accessed Jul 5 2018.

[30] Choose between Azure services that deliver messages. https://docs.microsoft.com/en-us/azure/event-grid/compare-messaging-services. Accessed Jul 5 2018, 2018.

[31] Apache OpenWhisk. https://openwhisk.apache.org. Accessed Jul 5 2018.

[32] OpenWhisk actions. https://github.com/apache/incubator-openwhisk/blob/master/docs/actions.md. Accessed Jul 5 2018, 2018.

[33] Panda, A., Sagiv, M., and Shenker, S. Verification in the age of microservices. In *Workshop on Hot Topics in Operating Systems* (2017).

[34] Park, D., Ştefănescu, A., and Roşu, G. KJS: A complete formal semantics of JavaScript. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2015).

[35] Paterson, R. A new notation for arrows. In *ACM International Conference on Functional Programming (ICFP)* (2001).

[36] Pulumi. cloud native infrastructure as code. https://www.pulumi.com/. Accessed Jul 5 2018, 2018.

[37] Rabbah, R. Composing functions into applications the serverless way. https://medium.com/openwhisk/composing-functions-into-applications-70d3200d0fac. Accessed Jul 5 2018, 2017.

[38] Ramalingam, G., and Vaswani, K. Fault tolerance via idempotence. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (2013).

[39] Sergey, I., Wilcox, J. R., and Tatlock, Z. Programming and proving with distributed protocols. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* (2017).

[40] Singhvi, A., Banerjee, S., Harchol, Y., Akella, A., Peek, M., and Rydin, P. Granular computing and network intensive applications: Friends or foes? In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks* (2017), HotNets-XVI.

[41] Waye, L., Chong, S., and Dimoulas, C. Whip: Higher-order contracts for modern services. In *ACM International Conference on Functional Programming (ICFP)* (2017).

[42] Weerawarana, S., Ekanayake, C., Perera, S., and Leymann, F. Bringing middleware to everyday programmers with ballerina. In *Business Process Management* (2018).

[43] Wilcox, J. R., Woos, D., Panchekha, P., Tatlock, Z., Wang, X., Ernst, M. D., and Anderson, T. Verdi: A framework for implementing and formally verifying distributed systems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2015).