Formal Foundations of Serverless Computing

ABHINAV JANGDA, University of Massachusetts Amherst, United States
DONALD PINCKNEY, University of Massachusetts Amherst, United States
YURIY BRUN, University of Massachusetts Amherst, United States
ARJUN GUHA, University of Massachusetts Amherst, United States

Serverless computing (also known as *functions as a service*) is a new cloud computing abstraction that makes it easier to write robust, large-scale web services. In serverless computing, programmers write what are called *serverless functions*, which are programs that respond to external events. When demand for the serverless function spikes, the platform automatically allocates additional hardware and manages load-balancing; when demand falls, the platform silently deallocates idle resources; and when the platform detects a failure, it transparently retries affected requests. In 2014, Amazon Web Services introduced the first serverless platform, *AWS Lambda*, and similar abstractions are now available on all major cloud computing platforms.

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 $\lambda_{\mathbb{A}}$, an operational semantics of the essence of serverless computing. Despite being a small (half a page) core calculus, $\lambda_{\mathbb{A}}$ models all the low-level details that serverless functions can observe. To show that $\lambda_{\mathbb{A}}$ is useful, we present three applications. First, to ease reasoning about code, we present a simplified *naive semantics* of serverless execution and precisely characterize when the naive semantics and $\lambda_{\mathbb{A}}$ coincide. Second, we augment $\lambda_{\mathbb{A}}$ with a key-value store to allow reasoning about stateful serverless functions. Third, since a handful of serverless platforms support serverless function composition, we show how to extend $\lambda_{\mathbb{A}}$ with a composition language and show that our implementation can outperform prior work.

CCS Concepts: • **Software and its engineering → Distributed programming languages**.

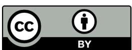
Additional Key Words and Phrases: serverless computing, distributed computing, formal language semantics

ACM Reference Format:
Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. 2019. Formal Foundations of Serverless Computing. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 149 (October 2019), 26 pages. https://doi.org/10.1145/3360575

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

Authors' addresses: Abhinav Jangda, University of Massachusetts Amherst, United States; Donald Pinckney, University of Massachusetts Amherst, United States; Yuriy Brun, University of Massachusetts Amherst, United States; Arjun Guha, University of Massachusetts Amherst, United States.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2019 Copyright held by the owner/author(s).
2475-1421/2019/10-ART149
https://doi.org/10.1145/3360575

platform may transparently load another instance of the function on a new machine and manage load-balancing without programmer intervention. Conversely, if demand for the function falls, the platform will transparently terminate underutilized instances. 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 resource allocation 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 [Akkus et al. 2018; Ellis 2018; Google 2018b; Hendrickson et al. 2016; Microsoft 2018b; OpenWhisk 2018a]. Serverless computing has seen rapid adoption [Conway 2017], 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 [Ao et al. 2018; Fouladi et al. 2017; Jonas et al. 2017], modular programming [Baldini et al. 2017], information flow control [Alpernas et al. 2018], chatbot design [Baudart et al. 2018], and virtual network functions [Singhvi et al. 2017]. However, serverless computing has several peculiar characteristics that prior work has not addressed.

Shortcomings of serverless computing. 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, the responsibility falls on 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, programmers face three key challenges when writing serverless functions:

(1) Reasoning about the correctness of serverless functions is hard, because of the exposed low-level behavior of the underlying serverless platform, which affects behavior of programmer-written functions.
(2) Interacting with external services, such as cloud-hosted databases, is also challenging, because failures and retries can be visible to external services, which increases the likelihood of data errors.
(3) Composing and orchestration of serverless functions makes reasoning even harder because most serverless platforms do not natively support serverless function composition.

Our contributions. This paper presents a formal foundation for serverless computing that makes progress towards addressing the above three challenges. 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, called λ_{λ}, which manifests the essential low-level behaviors of these serverless platforms, including failures, concurrency, function restarts, and instance reuse. The details of λ_{λ} 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, λ_{λ} can guide the development of programming tools and extensions to serverless platforms.

We design λ_{λ} to simplify reasoning about serverless programs and to be extensible to aid programmers using the serverless abstraction in more complex ways. We evaluate λ_{λ} and demonstrate its utility in three ways:

First, reasoning about serverless programs and the complex, low-level behavior of serverless platforms is hard. We derive a simplified *naive semantics* of serverless computing, which elides these complex behaviors and helps programmers reason about their programs. We precisely characterize when it is safe for a programmer to use the naive semantics instead of λ_{λ} and prove that if a serverless function satisfies a simple safety property, then there exists a weak bisimulation between λ_{λ} and the naive semantics. This result helps programmers confidently abstract away the low-level details of serverless computing. We provide canonical examples of these safety properties and examples of ill-behaved serverless functions that are unsafe, which programmers could easily write by mistake without the naive semantics. Our theorem can serve as the foundation for future work on building serverless functions that are verifiably safe.

Second, we demonstrate that λ_{λ} can compose with models of external services. External services matter, because the serverless abstraction is quite limited by itself. However, the interaction between serverless functions and external services can be quite complicated, due to the low-level behaviors that serverless functions exhibit. Specifically, we compose λ_{λ} with a model of a cloud-hosted key-value store with shared state. Using this extension, we precisely characterize what it means for a serverless function to be idempotent, which is necessary to avoid corrupting data in the key-value store. The recipe that we follow to add a key-value store to λ_{λ} could also be used to augment λ_{λ} with models of other cloud computing services.

Finally, we demonstrate that λ_{λ} can be extended to model richer serverless programming abstractions, such as serverless function orchestration. We extend λ_{λ} 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.

We hope that λ_{λ} will be a foundation for further research on language-based abstractions for serverless computing. The case studies we present are detailed examples that show how to design new abstractions and study existing abstractions for serverless computing, using λ_{λ}.

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 λ_{λ}, our formal semantics of serverless computing. §4 presents a simplified semantics of serverless computing and proves exactly when it coincides with λ_{λ}. §5 augments λ_{λ} with a key-value store. §6 and §7 extend λ_{λ} with a language for serverless orchestration. Finally, §8 discusses related work and §9 concludes. Our implementation is available at plasma.cs.umass.edu/lambda-lambda.

```
1 let accounts = new Map();
2 exports.bank = function(req, res) {
3   if (req.body.type === 'deposit') {
4     accounts.set(req.body.name, req.body.value);
5     res.send(true);
6   } else if (req.body.type === 'transfer') {
7     let { from, to, amnt } = req.body;
8     if (accounts.get(from) >= amnt) {
9       accounts.set(to, accounts.get(to) + amnt);
10      accounts.set(from, accounts.get(from) - amnt);
11      res.send(true);
12    } else {
13      res.send(false); }}};
```

Fig. 1. A serverless function for banking that does not address low-level details of serverless execution. Therefore, it will exhibit several kinds of faults. The correct implementation is in Figure 2.

```
1 let Datastore = require('@google-cloud/datastore');
2 exports.bank = function(req, res) {
3   let ds = new Datastore({ projectId: 'bank-app' });
4   let dst = ds.transaction();
5   dst.run(function() {
6     let tId = ds.key(['Transaction', req.body.transId]);
7     dst.get(tId, function(err, trans) {
8       if (err || trans) {
9         dst.rollback(function() { res.send(err || trans); });
10      } else if (req.body.type === 'deposit') {
11        let to = ds.key(['Account', req.body.to]);
12        dst.get(to, function(err, acct) {
13          acct.balance += req.body.amount;
14          dst.save({ key: to, data: acct });
15          dst.save({ key: tId, data: true });
16          dst.commit(function() { res.send(true); });
17        });
18      } else if (req.body.type === 'transfer') {
19        let amnt = req.body.amount;
20        let from = ds.key(['Account', req.body.from]);
21        let to = ds.key(['Account', req.body.to]);
22        dst.get([from, to], function(err, accts) {
23          if (accts[0].balance >= amnt) {
24            accts[0].balance -= amnt;
25            accts[1].balance += amnt;
26            dst.save([{ key: from, data: accts[0] },
27              { key: to, data: accts[1] }]);
28            dst.save({ key: tId, data: true });
29            dst.commit(function() { res.send(true); });
30          } else {
31            dst.rollback(function() { res.send(false);
32          });}});}}});});};
```

Fig. 2. A serverless function for banking that addresses instance termination, concurrency, and idempotence.

2 OVERVIEW OF SERVERLESS COMPUTING

To motivate the need for a formal foundation of serverless computing, consider the serverless banking function in Figure 1.[1] This function processes two types of requests: (1) a request to deposit

[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.

new funds into an account and (2) a request to transform funds from one account to another, which will fail if the source account has insufficient funds.

During deployment, the programmer can specify the types of events that will *trigger* the function, e.g., messages on a message bus, updates to a database, or web requests to a URL. The function receives two arguments: (1) the event as a JSON-formatted object, and (2) 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 programmer. Moreover, the function does not receive a notification before a shut down occurs. Similarly, the platform automatically starts a new container when an 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 source will be processed by the same instance, and instances may process events in parallel. Therefore, a correct implementation must 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 [Amazon 2018; Google 2018a; Microsoft 2018a; OpenWhisk 2018b]. 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 2 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 ($\lambda_{\mathbb{A}}$) that succinctly describes its peculiarities. Using $\lambda_{\mathbb{A}}$, the rest of the paper precisely characterizes the kinds of properties that are needed for serverless functions, such as our example, to be correct.

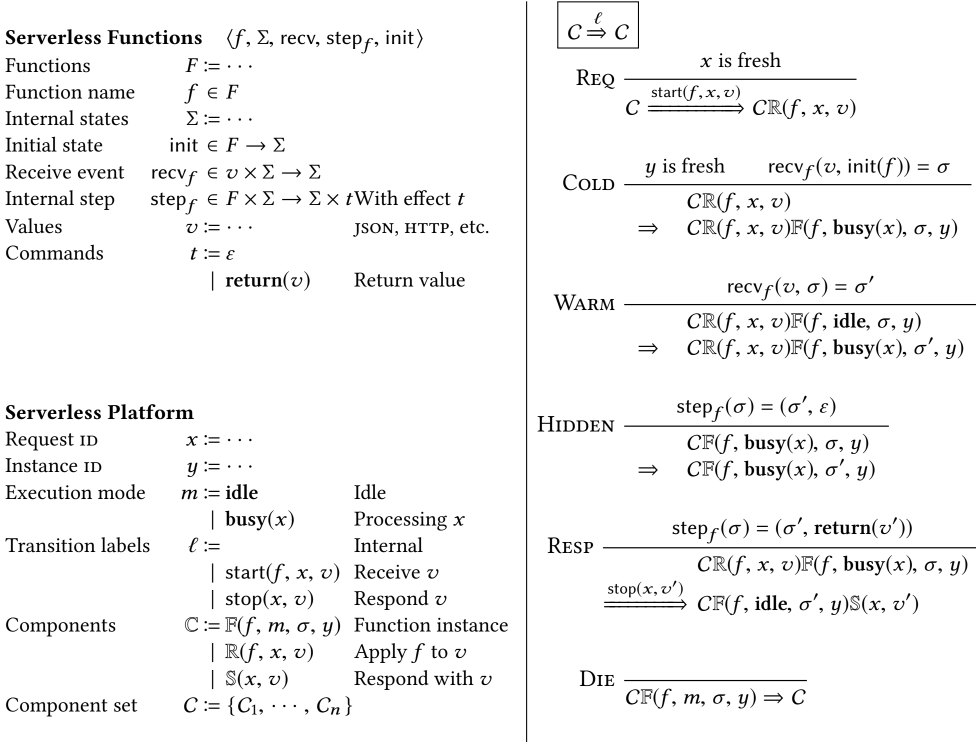


Fig. 3. $\lambda_{\mathbb{A}}$: An operational model of serverless platforms.

3 SEMANTICS OF SERVERLESS COMPUTING

We now present our operational semantics of serverless platforms ($\lambda_{\mathbb{A}}$) that captures the essential details of the serverless execution model, including concurrency, failures, execution retries, and function instance reuse. Figure 3 presents $\lambda_{\mathbb{A}}$ 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, and usually consume and produce JSON values. These are the features that our platform model includes. In our model, the operation of a serverless function (f) is defined by three functions:

(1) A function that produces the initial state of a function (init). The state of a serverless function (σ) is abstract to the serverless platform, and in practice the state is source language dependent. For example, if the serverless function f were written in JavaScript, then each state would be the state of the JavaScript VM and init(f) would be the initial JavaScript heap.
(2) A function that receives a request (recv_f) from the platform.

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

(3) A function that takes an internal step (step_f) that may produce a command for the serverless platform (t). For now, the only valid command is $\textbf{return}(v)$, which indicates that the response to the last request is the value v. §5 extends our model with new commands.

Serverless platform. $\lambda_{\mathbb{A}}$ is an operational semantics of a serverless platform that processes several concurrent requests. $\lambda_{\mathbb{A}}$ 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 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, $\lambda_{\mathbb{A}}$ 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). These rules are sufficient to capture several subtleties of serverless execution, as discussed below.

Instance launches are not observable. $\lambda_{\mathbb{A}}$ produces an observable event ($\text{start}(f, x, v)$) 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.

State reuse during warm starts. When a function instance responds to a request (the Resp rule), the instance becomes idle, but its state is not reinitialized and may be reused to process another request (the Warm rule). In the following example, the function instance receives the second request (x_2) when its state is σ_1, which may not be identical to the initial state of the function.

$$\begin{array}{lll}
 & \mathbb{R}(f, x_1, v_1)\mathbb{F}(f, \textbf{busy}(x_1), \sigma_0, y) & \\
\xrightarrow{\text{stop}(x_1, v_1')} & \mathbb{F}(f, \textbf{idle}, \sigma_1, y) & \text{By Resp} \\
\xrightarrow{\text{start}(f, x_2, v_2)} & \mathbb{R}(f, x_2, v_2)\mathbb{F}(f, \textbf{idle}, \sigma_1, y) & \text{By Req} \\
\Longrightarrow & \mathbb{R}(f, x_2, v_2)\mathbb{F}(f, \textbf{busy}(x_2), \sigma_2, y) & \text{By Warm}
\end{array}$$

Function instance termination is not observable. A function instance may terminate at any time. Moreover, termination is not an observable event. In practice, there are several reasons why termination may occur. (1) An instance may terminate if there is a software or hardware failure on its machine. (2) The platform may deliberately terminate the instance to reclaim idle resources. (3) The platform may deliberately terminate an instance if it takes too long to respond to a request. In $\lambda_{\mathbb{A}}$, we model all kinds of termination with the Die rule.

Function instances may start at any time. The platform is free to cold-start or warm-start a function instance for any pending request at any time, even if an existing function instance is processing the request. Therefore, several function instances may be processing a single request at once. This occurs in practice when a transient fault makes an instance temporarily unreachable. However, cold-starts and warm-starts are not observable events, thus programmers cannot directly observe the number of instances that are processing a single request. In the example below, a single

$$\begin{array}{ll}\text{Naive function state} & \mathcal{A} := \langle f, m, \vec{\sigma}, \mathcal{B}\rangle \\ \text{Response buffer} & \mathcal{B} \subseteq 2^{(x,v)}\end{array}$$

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

$$\text{N-START}\ \frac{x \text{ is fresh} \qquad \sigma_0 = \mathsf{init}(f) \qquad \mathsf{recv}_f(v, \sigma_0) = \sigma'}{\langle f, \textbf{idle}, \vec{\sigma}, \mathcal{B}\rangle \xmapsto{\mathsf{start}(f,x,v)} \langle f, \textbf{busy}(x), [\sigma_0, \sigma'], \mathcal{B}\rangle}$$

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

$$\text{N-BUFFER-STOP}\ \frac{\mathsf{step}_f(\sigma) = (\sigma', \textbf{return}(v))}{\langle f, \textbf{busy}(x), \vec{\sigma} +\!\!+ [\sigma], \mathcal{B}\rangle \mapsto \langle f, \textbf{idle}, [\sigma'], \mathcal{B} \cup \{(x, v)\}\rangle}$$

$$\text{N-EMIT-STOP}\ \langle f, \textbf{idle}, \vec{\sigma}, \mathcal{B} \cup \{(x, v)\}\rangle \xmapsto{\mathsf{stop}(x,v)} \langle f, \textbf{idle}, \vec{\sigma}, \mathcal{B}\rangle$$

Fig. 4. A naive semantics of serverless functions.

request cold-starts two function instances.

$$\begin{array}{lll} \xRightarrow{\mathsf{start}(f,x,v)} & \mathbb{R}(f,x,v) & \text{By REQ} \\ \Longrightarrow & \mathbb{R}(f,x,v)\mathbb{F}(f,\textbf{busy}(x),\mathsf{init}(f),y_1) & \text{By COLD-START} \\ \Longrightarrow & \mathbb{R}(f,x,v)\mathbb{F}(f,\textbf{busy}(x),\mathsf{init}(f),y_1)\mathbb{F}(f,\textbf{busy}(x),\mathsf{init}(f),y_2) & \text{By COLD-START}\end{array}$$

This example also shows why the request $\mathbb{R}(f,x,v)$ is not consumed after the first instance is launched. We need may need the request to launch additional instances in the future, particular when a failure occurs.

Single response per request. Although a single request may spawn several function instances, each request receives one response from a single function instance. Other instances processing the same request will eventually get stuck because they cannot respond. However, stuck instances will eventually terminate. In the following example, two instances start by processing the same request, the first instance then responds and becomes idle, and finally, the second instance terminates because it is stuck.

$$\begin{array}{lll} & \mathbb{R}(f,x,v)\mathbb{F}(f,\textbf{busy}(x),\sigma_1,y_1)\mathbb{F}(f,\textbf{busy}(x),\sigma_1,y_2) & \\ \xRightarrow{\mathsf{stop}(x,v')} & \mathbb{F}(f,\textbf{idle},\sigma_1',y_1)\mathbb{F}(f,\textbf{busy}(x),\sigma_1,y_2) & \text{By RESP} \\ \Longrightarrow & \mathbb{F}(f,\textbf{idle},\sigma_1',y_1) & \text{By DIE}\end{array}$$

Summary. In summary, $\lambda_{\mathbb{A}}$ succinctly and faithfully models the low-level details of serverless platforms, and makes manifest the subtleties that make serverless programming hard. The rest of this paper demonstrates that $\lambda_{\mathbb{A}}$ is useful in a variety of ways. The next section shows how to use $\lambda_{\mathbb{A}}$ to rigorously define a simpler semantics of serverless programming that is easier for programmers to understand, §5 shows that $\lambda_{\mathbb{A}}$ is easy to extend with a model of another cloud service, and §6 and §7 shows how to extend $\lambda_{\mathbb{A}}$ to model new serverless programming abstractions.

4 A SIMPLER SERVERLESS SEMANTICS

A natural way to make serverless programming easier is to implement a simpler execution model than $\lambda_{\mathbb{A}}$. For example, we could execute functions exactly once, or eliminate warm starts to avoid reusing state. Unfortunately, implementing these changes is likely to be expensive (and, in many situations, beyond our control). Therefore, this section gives programmers a simpler programming

model in the following way. First, we define a simpler *naive semantics* of serverless computing that eliminates most unintuitive behaviors of $\lambda_{\mathbb{A}}$. Second, 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 allowing programmers to reason using the naive semantics, even when their code is running on a full-fledged serverless platform.

In the naive serverless semantics (Figure 4), 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. At each step of execution, the naive semantics either (1) starts processing a new event if the platform is idle (N-Start), (2) takes an internal step if the platform is busy (N-Step), (3) buffers a completed response (N-Buffer-Stop), or (4) responds to a past request (N-Emit-Stop). This buffering behavior is essential, thus a programmer cannot rely on a platform to process concurrent messages in-order. However, the naive semantics abstracts away the details of concurrent execution and warm starts. The state of a naive platform consists of (1) the function's name (f); (2) its execution mode (m); (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 (we write $+$ to append two traces); and (4) a buffer of responses that have yet to be returned ($\mathcal{B}$). The trace is a convenience that helps us relate the naive semantics to $\lambda_{\mathbb{A}}$, but has no effect on execution because step_f only works on the latest state in the semantics.

Naive semantics safety. 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* over the state of the serverless function. At a high-level, the safety relation is an equivalence relation on program states, which ensures that the (1) serverless function produces the same observable command (if any) on equivalent states and that (2) all final states are equivalent to the initial state. Intuitively, the latter condition ensures that warm starts and cold starts are indistinguishable from each other, and the former condition ensures that interactions between the serverless function and the external world are identical in equivalent states. The safety relation is formally defined below.

Definition 4.1 (Safety Relation). For a serverless function $\langle f, \Sigma, \mathsf{recv}_f, \mathsf{step}_f, \mathsf{init}\rangle$, 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, $(\mathsf{recv}_f(v, \sigma_1), \mathsf{recv}_f(v, \sigma_2)) \in \mathcal{R}$,
(3) for all $(\sigma_1, \sigma_2) \in \mathcal{R}$, if $(\sigma_1', t_1) = \mathsf{step}_f(\sigma_1)$ and $(\sigma_2', t_2) = \mathsf{step}_f(\sigma_2)$ then $(\sigma_1', \sigma_2') \in \mathcal{R}$ and $t_1 = t_2$, and
(4) for all σ, if $\mathsf{step}_f(\sigma) = (\sigma', \mathbf{return}(v))$ then $(\sigma', \mathsf{init}(f)) \in \mathcal{R}$.

Bisimulation relation. We now define the bisimulation relation, which is a relation between naive states ($\mathcal{A}$) and $\lambda_{\mathbb{A}}$ states (C). The bisimulation 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 proof of 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. The bisimulation relation is formally defined below.

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

(1) $\langle f, \textbf{idle}, \vec{\sigma}, \mathcal{B} \rangle \approx C$
  (a) For all $(x', v') \in \mathcal{B}$, $\mathbb{R}(f, x', v'') \in C$, and
(2) $\langle f, \textbf{busy}(x), \vec{\sigma}, \mathcal{B} \rangle \approx \mathbb{R}(f, x, v)\, C$ if:
  (a) For all $\mathbb{F}(f, \textbf{busy}(x), \sigma, y) \in C$, if $\exists \sigma'.\sigma' \in \vec{\sigma}$ such that $(\sigma, \sigma') \in \mathcal{R}$ and
  (b) For all $(x', v') \in \mathcal{B}$, $\mathbb{R}(f, x', v'') \in C$

Weak Bisimulation Theorem. We are now ready to prove a weak bisimulation between the naive semantics and $\lambda_{\mathbb{A}}$, conditioned on the serverless functions satisfying the safety 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. The first part of the theorem states that every step in the naive semantics corresponds to some sequence of steps in $\lambda_{\mathbb{A}}$. 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 the theorem states that any arbitrary step in $\lambda_{\mathbb{A}}$— including failures, retries, and warm starts — corresponds to a (possibly empty) sequence of steps in the naive semantics.

An important simplification in the naive semantics is that it executes a single request at a time. Therefore, to relate a naive trace to a $\lambda_{\mathbb{A}}$ trace, 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 $\vec{\ell}$ that only consists of events labeled x. In addition, we write $\Rrightarrow$ and $\mapsto\!\!\!\!\rightarrow$ for the reflexive-transitive closure of $\Rightarrow$ and $\mapsto$ respectively. With these definitions, we can state the weak bisimulation theorem.

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

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

Proof. By Theorems A.4 and A.5 in Jangda et al. [2019]. □

In summary, this theorem allows programmers to justifiably ignore the low-level details of $\lambda_{\mathbb{A}}$, 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 [Fragoso Santos et al. 2018; Park et al. 2015], 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. The rest of these section gives examples that illustrate the kind of reasoning needed to verify serverless function safety.

4.1 Examples of Safe and Unsafe Serverless Functions

We now give two examples of serverless functions and show that they are safe and unsafe respectively using only the definition of the safety relation.

```
1 var cache = new Map();
2 function auth(req, res) {
3   let {user, pass} = req.body;
4   if (cache.contains(user, pass)) {
5     res.write(true);
6   } else if (db.get(user) === pass) {
7       cache.insert(user, pass);
8       res.write(true);
9   } else {
10    res.write(false);
11  }
12 }
```

Username	$U := \cdots$
Password	$P := \cdots$
Cache and Database	$C, D \in U \rightharpoonup P$
Program state	$\Sigma := \text{Option}\ (U \times P) \times C \times D$

$\text{init}(f) = (\text{None}, \cdot, D)$

$\text{recv}_f((u, p), (\text{None}, c, D)) = (\text{Some}\ (u, p), c, D)$

$$\text{step}(\text{Some}(u, p), c, D) = \begin{cases} ((\text{None}, c[u \mapsto p], D), \textbf{return}(\texttt{true})) & \text{if } u \notin \text{dom}(C) \wedge D(u) = p \\ ((\text{None}, c, D), \textbf{return}(\texttt{false})) & \text{if } u \notin \text{dom}(C) \wedge D(u) \neq p \\ ((\text{None}, c, D), \textbf{return}(\texttt{true})) & \text{if } C(u) = p \end{cases}$$

Fig. 5. An authentication example that caches the recent authentications to decrease number of authentication server calls.

In-Memory Cache. Figure 5 is a serverless function that receives a username and password combination, and returns true if the combination is correct. The function queries an external database for the password. Since database requests take time, the function locally caches correct passwords to improve performance. The cache will be empty on cold starts and may be non-empty on warm starts. For simplicity, we assume that passwords do not change. (A more sophisticated example would invalidate the cache after a period of time.)

Ignoring JavaScript-specific details, this program operates in two kinds of states: (1) in the initial state, the program is idle and waiting to receive a request and (2) while processing a request, the program has a username (U) and password (P) in memory. We model the two states with the type Option($U \times P$). In both states, the program has an in-memory cache (C) and access to the database (D). Although we assume the database is read-only, the program may update the cache. Therefore, the complete type of program state is a product of these three components (Σ in Figure 5).

When the program receives a request carrying a username and password, it records them in program state and leaves the cache unmodified (recv in Figure 5). After receiving a request, the JavaScript program performs a series of internal steps to check for a cached result, query the database (if needed), and update the cache. For brevity, our model of the program condenses these operations into a single step (step in Figure 5).

Given this model of the program, we define the safety relation ($\mathcal{R}$) as follows:

$$((\text{Some}(u, p), c, D), (\text{Some}(u, p), c', D)) \in \mathcal{R} \quad \text{if } c \subseteq D \wedge c' \subseteq D$$
$$((\text{None}, c, D), (\text{None}, c', D)) \in \mathcal{R} \quad \text{if } c \subseteq D \wedge c' \subseteq D$$

This relation specifies that two program states are equivalent only if they are both idle states or both processing the same request (same username and password combination). However, the relation allows the caches (c and c') in either state to be different, as long as both are consistent with the database. The latter condition is the key to ensuring that warm starts are safe.

Finally, we need to prove that the safety relation above satisfies the four criteria of Definition 4.1:

(1) It is straightforward to show that $\mathcal{R}$ is an equivalence relation.
(2) To show that recv maps equivalent states to equivalent states, note that recv is only defined when the program state does not contain a query (i.e., the first component is None). Therefore, the two equivalent input states may only be of the form (None, c, D) and (None, c', D), where $c, c' \subseteq D$. recv records the query in program state and leaves the cache unmodified, therefore the input states are related.

```
1 var process = require('process');
2 exports.main = function (req, res) {
3   pid = parseInt(process.pid);
4   if (pid > 10000) {
5     res.write ({"output": "High process id"});
6   } else {
7     res.write ({"output": "Low process id"};});}
```

Fig. 6. It is not possible to define a safety relation for this function, because it depends on the process ID.

(3) To show that step maps equivalent states to equivalent states, note that step is only defined when the program state contains a query (Some (U, P)). Moreover, for two states with queries to be equivalent, their queries must be identical. We have to consider the six combinations of step and ensure that it is never the case that one state produces **return**(`true`) while the other state produces **return**(`false`). This does not occur because the two caches are consistent with the database. We have to also ensure that the resulting states are equivalent, which is straightforward because the cache updates preserve consistency.
(4) Finally, to show that final states are related to $\mathsf{init}(f)$, note that step produces a state with None for the query, and all these states are related by $\mathcal{R}$, as long as their caches are consistent with the database.

Therefore, since $\mathcal{R}$ is a safety relation, by Theorem 4.3, the function operates the same way using $\lambda_{\mathbb{A}}$ and the naive semantics.

Unsafe Serverless Functions. Not all serverless functions are safe, thus it isn't always possible to define a safety relation. For example, the function in Figure 6 responds with the UNIX process ID, which will vary across function instances. A function can observe other low-level detail of the instance, such as its IP address or Ethernet address.

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* [Google 2018a; Microsoft 2018a; OpenWhisk 2018b]. 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 $\lambda_{\mathbb{A}}$ 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 7 augments $\lambda_{\mathbb{A}}$ 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')). An important detail here is 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 $\lambda_{\mathbb{A}}$ 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

Serverless Functions

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

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

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

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

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

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

Fig. 7. $\lambda_{\mathbb{A}}$ augmented with a key-value store.

instance that held the lock no longer exists. This may occur if the function instance dies before committing its changes.

Idempotence in the naive semantics. There are several ways to ensure that a serverless function is idempotent. A common protocol is to save 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, 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 $\lambda_{\mathbb{A}}$ 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 $\lambda_{\mathbb{A}}$ state and the naive state are equivalent, it is possible for all function instances in $\lambda_{\mathbb{A}}$ to fail. When this occurs, $\lambda_{\mathbb{A}}$ "falls behind" the naive semantics. Nevertheless, we still treat the states as equivalent, and let the weak bisimulation proof re-invoke function instances until $\lambda_{\mathbb{A}}$ catches up with the naive semantics. Unfortunately, this approach does not always work with the key-value store, since the key-value store may have changed. To address

Serverless Functions

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

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

$$\frac{\mathcal{A} \mapsto \mathcal{A}'}{\widetilde{M}, \mathcal{A} \mapsto \widetilde{M}, \mathcal{A}'} \qquad \frac{\mathcal{A} \xmapsto{\mathsf{start}(f,x,v)} \mathcal{A}'}{\cdot, \mathcal{A} \xmapsto{\mathsf{start}(f,x,v)} \cdot, \mathcal{A}'} \qquad \frac{\mathcal{A} \xmapsto{\mathsf{stop}(x,v)} \mathcal{A}'}{\textbf{commit}(v), \mathcal{A} \xmapsto{\mathsf{stop}(x,v)} \cdot, \mathcal{A}'}$$

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

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

$$\textsc{N-BeginTx}\ \frac{\mathsf{step}_f(\sigma) = (\sigma', \textbf{beginTx})}{\begin{array}{l}\cdot, \langle f, \textbf{busy}(x), \vec{\sigma} +\!\!+ [\sigma], \mathcal{B}\rangle \\ \mapsto \textbf{locked}(M, \vec{\sigma} +\!\!+ [\sigma]), \langle f, \textbf{busy}(x), \vec{\sigma} +\!\!+ [\sigma, \sigma'], \mathcal{B}\rangle\end{array}}$$

$$\textsc{N-EndTx}\ \frac{\mathsf{step}_f(\sigma) = (\sigma', \textbf{endTx})}{\begin{array}{l}\textbf{locked}(M, \vec{\sigma}'), \langle f, \textbf{busy}(x), \vec{\sigma} +\!\!+ [\sigma], \mathcal{B}\rangle \\ \mapsto \textbf{commit}(M(x)), \langle f, \textbf{busy}(x), \vec{\sigma} +\!\!+ [\sigma, \sigma'], \mathcal{B}\rangle\end{array}}$$

$$\textsc{N-Rollback}\ \frac{}{\textbf{locked}(M, \vec{\sigma}'), \langle f, \textbf{busy}(x), \vec{\sigma}, \mathcal{B}\rangle \mapsto \cdot, \langle f, \textbf{busy}(x), \vec{\sigma}', \mathcal{B}\rangle}$$

Fig. 8. The naive semantics with a key-value store.

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 is within a transaction, and the equivalent $\lambda_{\mathbb{A}}$ state takes several steps that result in a failure, followed by other updates to the key-value store. When this occurs, the naive semantics must rollback to the start of the transaction and re-execute with the updated key-value store.

Figure 8 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 an 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 $\lambda_{\mathbb{A}}$ to be operating in lock-step with the naive semantics. However, other instances in $\lambda_{\mathbb{A}}$ that are not using the key-value store can make progress. Therefore, a transaction is not globally atomic in $\lambda_{\mathbb{A}}$, and other requests can be received and processed while some instance is in a transaction.

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

(1) If $\mathcal{A} \approx C$ then $\cdot, \mathcal{A} \approx \mathbb{D}(M, \textbf{free})C$, or

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

(3) If $\mathcal{A} = \langle f, \textbf{busy}(x), \vec{\sigma}, \mathcal{B}\rangle$, $\widetilde{M} = \textbf{commit}(v)$, and $M(x) = v$ then $\widetilde{M}, \mathcal{A} \approx \mathbb{D}(M, \textbf{free})C$.

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 safety 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 $\widetilde{M}, \mathcal{A}, C, \ell$:

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

Proof. By Theorems A.9 and A.10 in Jangda et al. [2019]. □

The theorem statements in Jangda et al. [2019] 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 $\lambda_{\mathbb{A}}$.

Example. Consider the banking serverless function (Figure 2), which uses transactions and an external key-value store. We can show that this function satisfies the safety relation using the approach presented in §4.1. We now argue that this function satisfies the three conditions of Theorem 5.2, which will allow us to reason about its execution using the naive semantics. After the function receives a request, it extracts the request's ID and checks to see if the database has a value with that ID (6 − 9) and if so immediately returns the saved value. This satisfies first condition. The function runs a transaction on lines 14 − 16 and 26 − 29, and in each case it stores the value in the database and returns the same value. This satisfies the second and third conditions.

6 SERVERLESS COMPOSITIONS

Thus far, we have used $\lambda_{\mathbb{A}}$ to reason about serverless functions in isolation, and also extended $\lambda_{\mathbb{A}}$ to reason about serverless functions that use a key-value store. In this section, we consider a different kind of extension to $\lambda_{\mathbb{A}}$, which involves a significant change to the serverless computing platform itself. The change that we make is different from, but inspired by the work of Baldini et al. [2017]. This section first motivates why we would want to change the serverless platform, and then formalizes the modified platform. The key takeaway from this section, is that the model of the modified platform extends $\lambda_{\mathbb{A}}$ as-is, and doesn't involve changing the definitions and reduction rules in Figure 3. We have also implemented this modified platform, which we discuss in §7.

6.1 The Need for Serverless Composition Languages

Serverless platforms encourage programmers to decompose large applications into several little functions (or, "microservices"). This approach has obvious benefits: smaller functions are easier to understand, and can be reusable. Amazon Web Services has a *Serverless Application Repository* that encourages programmers to reuse and share microservices.[3] For example, a serverless function to

[3]https://aws.amazon.com/serverless/serverlessrepo/

```
1 let request = require('request-promise-native');
2 exports.postStatus = function(req, res) {
3   let {state, sha, url, repo} = req.body;
4   request.post({
5     url: postStatusToGithub,
6     json: { state: state, sha: sha, url: url, repo: repo }})
7   .then(function (response, body) {
8     if (response.state === "failure") {
9       request.post({
10        url: postToSlack,
11        json: { channel: "<id>", text: "<msg>" }});}});}
```

Fig. 9. A serverless function to receive build state from Google Cloud Build, set status on GitHub using `postStatusToGitHub`, and report failures to Slack using `postToSlack`.

post messages on Slack – which is available on the AWS Serverless Application Repository – could be used to implement notifications for many different applications.

Many development teams use an application that connects a Slack channel, a GitHub repository, and a continuous integration (CI) service (e.g., TravisCI or Google Cloud Build). The CI service tests every commit to the GitHub repository. After testing completes, (1) the CI service invokes the application with the test results, (2) the application updates the build status on GitHub, and (3) the application posts a message to Slack only if testing fails. This application is a good fit for serverless computing and is easy to write by wiring together existing serverless functions that post to GitHub and Slack, as sketched in Figure 9. This example is a serverless function that acts as a coordinator (`postStatus`) that invokes two other auxiliary serverless functions (`postStatusToGitHub` and `postToSlack`).

However, as Baldini et al. [2017] point out, a major problem with this approach is that the programmer gets "double-billed" and has to pay for the time spent running the coordinator function, which is mostly idle, and for the time spent doing actual work in the auxiliary functions. 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. The only way to resolve this problem is to modify the serverless computing platform with function composition primitives.

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.

Figure 10 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 on Hughes' arrows [Hughes 2000], 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

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

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

$$\boxed{C \stackrel{\ell}{\Rightarrow} C}$$

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

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

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

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

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

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

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

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

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

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

Fig. 10. Extending $\lambda_{\mathbb{A}}$ with SPL.

(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 [Felleisen and Friedman 1986]. 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 it 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

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

Fig. 11. JSON transformation language.

```
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`.

Fig. 12. Compiling the surface syntax of SPL.

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 11). 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 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 [Paterson 2001]. 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 12a 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 12b). 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 12c. 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 12d). 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 [Baldini et al. 2017] to implement the $\ggg$ operator of SPL.

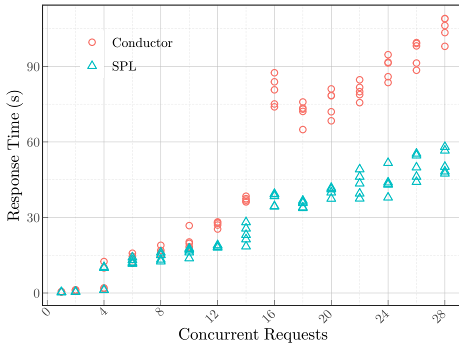
Our OpenWhisk implementation of SPL has three differences from the language presented so far. First, it supports bounded loops, which are a programming convenience. Second, instead of implementing the first operator and the JSON transformation language as independent expressions, we have a single operator 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.

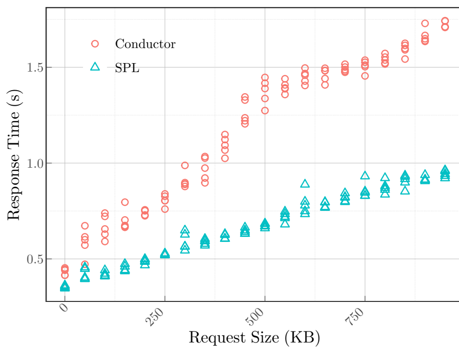
7 EVALUATION

This section first evaluates the performance of SPL using microbenchmarks, and then highlights the expressivity of SPL with three case studies.

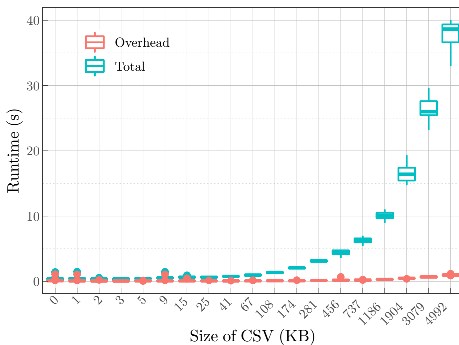
7.1 Comparison to OpenWhisk Conductor



(a) Response time versus concurrent requests.



(b) Response time versus request size.



(c) Response time and overhead.

Fig. 13. SPL benchmarks. Figures 13a and 13b show that SPL is faster than OpenWhisk Conductor when the serverless platform is processing several concurrent requests, or when the request size increases. Figure 13c shows that most of the execution time is spent in serverless functions, and not running SPL.

The Apache OpenWhisk serverless platform has built-in support for composing serverless functions using a *Conductor* [Rabbah 2017]. 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 code for a Conductor can be written in an arbitrary language (e.g., JavaScript). The key difference between SPL and Conductor, is that SPL is designed to run directly on the platform, the Conductor has to be executed in a container, 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 13a shows that SPL is slightly faster than Conductor when $N \leq 12$, but approximately twice as fast when $N > 12$. We

```
1 if (in.amount > 100) {
2   invoke bank({ type: "deposit", to: "checking",
3           amount: in.amount - 100, transId: in.tId1 });
4   invoke bank({ type: "deposit", to: "savings",
5           amount: 100, transId: in.tId2 });
6 } else {
7   invoke bank({ type: "deposit", to: "checking",
8           amount: in.amount, transId: in.tId1 });
9 } ret;
```

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

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

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

```
1 data <- get in.url;
2 json <- invoke csvToJson(data);
3 out <- invoke plotJson({data:json,x:in.xAxis,y:in.yAxis});
4 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.

Fig. 14. Example SPL programs.

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 13b shows that SPL is almost twice as fast as Conductor. We again attribute this to 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 14a uses SPL to write a bank deposit function using the `deposit` function from Figure 2. If the received amount is greater than $100, it is split in two parts and

deposited into the checking and savings accounts, by calling `deposit`. This SPL program does not suffer from "double billing" because 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. Figure 14b uses SPL to rewrite the Continuous Integration example (Figure 9), and is based on an example from Baldini et al. [2017]. The program in Figure 9 suffered from the "double billing" problem, since the composite serverless function needs to be active while waiting for the `postStatusToGitHub` and `postToSlack` do the 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. Our SPL program (Figure 14b) 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, Cloud Build can invoke a serverless function when tests complete, and we use it to 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. 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 14c) 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 13c 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. [2017] 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 [Rabbah 2017]. 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 [Alpernas et al. 2018] 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 some features of serverless platforms, such as warm starts and failures, that our semantics does model.

Several projects exploit serverless computing for elastic parallelization [Ao et al. 2018; Fouladi et al. 2019, 2017; Jonas et al. 2017]. §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. [2017] 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 [Gan and Delimitrou 2018], resource allocation [Björkqvist et al. 2016], programming abstractions [Baldini et al. 2017; Rabbah 2017], edge computing [Aske and Zhao 2018], and cloud container design [Shen et al. 2019]. $\lambda_{\mathbb{A}}$ is designed to elucidate subtle semantic issues (not performance problems) that affect programmers building serverless applications.

Language-based approaches to microservices. SKC [Gabbrielli et al. 2019] is another formal semantics of serverless computing that models how serverless functions can interact with each other. Unlike $\lambda_{\mathbb{A}}$, it does not model certain low-level, observable details of serverless platforms, such as warm starts.

λ_{FAIL} [Ramalingam and Vaswani 2013] 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 [Waye et al. 2017] and ucheck [Panda et al. 2017] 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.

Orleans [Bernstein et al. 2014] is a programming language for developing distributed systems using virtual actors. Orleans raises the level of abstraction and provides better locality by automatically placing hot actors nearby. Orleans is complementary to $\lambda_{\mathbb{A}}$, for example, Orleans can be used to implement $\lambda_{\mathbb{A}}$ semantics to develop a serverless system.

Cloud orchestration frameworks. Ballerina [Weerawarana et al. 2018] is a language for managing cloud environments; Engage [Fischer et al. 2012] is a deployment manager that supports inter-machine dependencies; Pulumi [Pulumi 2018] is an embedded DSL for writing programs that configure and run in the cloud; and CPL [Bračevac et al. 2016] is a unified language for writing distributed cloud programs together with their distribution routines. 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., [Bakst et al. 2017; Chajed et al. 2018; Desai et al. 2018;

Drăgoi et al. 2016; Gomes et al. 2017; Guha et al. 2013; Hawblitzel et al. 2015; Sergey et al. 2017; Wilcox et al. 2015]). The serverless computation model is more constrained than arbitrary distributed systems and algorithms. This paper presents a formal semantics of serverless computing, $\lambda_{\mathbb{\lambda}}$, with an emphasis on low-level details that are observable by programs, and thus hard for programmers to get right. To demonstrate that $\lambda_{\mathbb{\lambda}}$ 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 $\lambda_{\mathbb{\lambda}}$ could be used as a foundation for future verification work.

λ_{zap} [Walker et al. 2006] is a model of computation in the presence of transient hardware faults (e.g., bit flips). To detect and recover from such faults, λ_{zap} programs replicate computations and use majority voting. Serverless computing does not address these kinds of data errors, but does address communication failures by reinvoking functions on new machines, which we model in $\lambda_{\mathbb{\lambda}}$. Unfortunately, function reinvocation and other low-level behaviors are observable by serverless functions, and its easy to write a function that goes wrong when reinvoked. Using $\lambda_{\mathbb{\lambda}}$, this paper lays out a methodology to reason about serverless functions while abstracting away low-level platform behaviors.

9 CONCLUSION

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

ACKNOWLEDGEMENTS

This work was partially supported by the National Science Foundation under grants CNS-1413985, CCF-1453474, and CNS-1513055. We thank Samuel Baxter, Breanna Devore-McDonald, and Joseph Spitzer for their work on the SPL implementation.

REFERENCES

Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *USENIX Annual Technical Conference (ATC)*.

Kalev Alpernas, Cormac Flanagan, Sadjad Fouladi, Leonid Ryzhyk, Mooly Sagiv, Thomas Schmitz, and Keith Winstein. 2018. Secure Serverless Computing Using Dynamic Information Flow Control. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (Oct. 2018).

Amazon 2018. AWS Lambda Developer Guide: Invoke. https://docs.aws.amazon.com/lambda/latest/dg/API_Invoke.html. Accessed Jul 5 2018.

Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. 2018. Sprocket: A Serverless Video Processing Framework. In *ACM Symposium on Cloud Computing (SOCC)*.

Austin Aske and Xinghui Zhao. 2018. Supporting Multi-Provider Serverless Computing on the Edge. In *International Conference on Parallel Processing (ICPP)*.

Alexander Bakst, Klaus v. Gleissenthall, Rami Gökhan K, and Ranjit Jhala. 2017. Verifying Distributed Programs via Canonical Sequentialization. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

Ioana Baldini, Perry Cheng, Stephen J. Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, and Olivier Tardieu. 2017. 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!)*.

Guillaume Baudart, Julian Dolby, Evelyn Duesterwald, Martin Hirzel, and Avraham Shinnar. 2018. Protecting Chatbots from Toxic Content. In *ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*.

Phil Bernstein, Sergey Bykov, Alan Geller, and Jorgen Thelin. 2014. *Orleans: Distributed Virtual Actors for Programmability and Scalability*. Technical Report. https://www.microsoft.com/en-us/research/publication/orleans-distributed-virtual-actors-for-programmability-and-scalability/

Mathias Björkqvist, Robert Birke, and Walter Binder. 2016. Resource management of replicated service systems provisioned in the cloud. In *Network Operations and Management Symposium (NOMS)*.

Oliver Bračevac, Sebastian Erdweg, Guido Salvaneschi, and Mira Mezini. 2016. CPL: A Core Language for Cloud Computing. In *Proceedings of the 15th International Conference on Modularity*.

Tej Chajed, Frans Kaashoek, Butler Lampson, and Nickolai Zeldovich. 2018. Verifying concurrent software using movers in CSPEC. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

Sarah Conway. 2017. Cloud Native Technologies Are Scaling Production Applications. https://www.cncf.io/blog/2017/12/06/cloud-native-technologies-scaling-production-applications/. Accessed Jul 12 2018.

Ankush Desai, Amar Phanishayee, Shaz Qadeer, and Sanjit A. Seshia. 2018. Compositional Programming and Testing of Dynamic Distributed Systems. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

Cezara Drăgoi, Thomas A. Henzinger, and Damien Zufferey. 2016. PSync: A Partially Synchronous Language for Fault-tolerant Distributed Algorithms. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

Alex Ellis. 2018. OpenFaaS. https://www.openfaas.com. Accessed Jul 5 2018.

Matthias Felleisen and Daniel P. Friedman. 1986. 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*.

Jeffery Fischer, Rupak Majumdar, and Shahram Esmaeilsabzali. 2012. Engage: A Deployment Management System. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

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.

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 *USENIX Symposium on Networked System Design and Implementation (NSDI)*.

José Fragoso Santos, Petar Maksimović, Daiva Naudžiūnienė, Thomas Wood, and Philippa Gardner. 2018. JaVerT: JavaScript verification toolchain. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 50:1–50:33.

Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, Fabrizio Montesi, Marco Peressotti, and Stefano Pio Zingaro. 2019. No More, No Less - A Formal Model for Serverless Computing. In *Coordination Models and Languages (COORDINATION)*. 148–157.

Yu Gan and Christina Delimitrou. 2018. The Architectural Implications of Cloud Microservices. In *Computer Architecture Letters (CAL)*.

Victor B. F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. 2017. Verifying Strong Eventual Consistency in Distributed Systems. In *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*.

Google 2018a. Cloud Functions Execution Environment. https://cloud.google.com/functions/docs/concepts/exec. Accessed Jul 5 2018.

Google 2018b. Google Cloud Functions. https://cloud.google.com/functions/. Accessed Jul 5 2018.

Arjun Guha, Mark Reitblatt, and Nate Foster. 2013. Machine Verified Network Controllers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: Proving Practical Distributed Systems Correct. In *ACM Symposium on Operating Systems Principles (SOSP)*.

Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Serverless computation with OpenLambda. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*.

John Hughes. 2000. Generalising Monads to Arrows. *Science of Computer Programming* 37, 1–3 (May 2000), 67–111.

Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. 2019. Formal Foundations of Serverless Computing. https://arxiv.org/abs/1902.05870.

Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the Cloud: Distributed Computing for the 99%. In *Symposium on Cloud Computing*.

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

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

OpenWhisk 2018a. Apache OpenWhisk. https://openwhisk.apache.org. Accessed Jul 5 2018.

OpenWhisk 2018b. OpenWhisk Actions. https://github.com/apache/incubator-openwhisk/blob/master/docs/actions.md. Accessed Jul 5 2018.

Aurojit Panda, Mooly Sagiv, and Scott Shenker. 2017. Verification in the Age of Microservices. In *Workshop on Hot Topics in Operating Systems*.

Daejun Park, Andrei Stefănescu, and Grigore Roşu. 2015. KJS: A Complete Formal Semantics of JavaScript. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

Ross Paterson. 2001. A New Notation for Arrows. In *ACM International Conference on Functional Programming (ICFP)*.

Pulumi 2018. Pulumi. Cloud Native Infrastructure as Code. https://www.pulumi.com/. Accessed Jul 5 2018.

Rodric Rabbah. 2017. Composing Functions into Applications the Serverless Way. https://medium.com/openwhisk/composing-functions-into-applications-70d3200d0fac. Accessed Jul 5 2018.

Ganesan Ramalingam and Kapil Vaswani. 2013. Fault Tolerance via Idempotence. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2017. Programming and Proving with Distributed Protocols. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Van Robbert Renesse, and Hakin Weatherspoon. 2019. X-Containers: Breaking Down Barriers to Improve Performance and Isolation of Cloud-Native Containers. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

Arjun Singhvi, Sujata Banerjee, Yotam Harchol, Aditya Akella, Mark Peek, and Pontus Rydin. 2017. Granular Computing and Network Intensive Applications: Friends or Foes?. In *ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*.

David Walker, Lester Mackey, Jay Ligatti, George A. Reis, and David I. August. 2006. Static Typing for a Faulty Lambda Calculus. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*.

Lucas Waye, Stephen Chong, and Christos Dimoulas. 2017. Whip: Higher-Order Contracts for Modern Services. In *ACM International Conference on Functional Programming (ICFP)*.

Sanjiva Weerawarana, Chathura Ekanayake, Srinath Perera, and Frank Leymann. 2018. Bringing Middleware to Everyday Programmers with Ballerina. In *Business Process Management*.

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: A framework for implementing and formally verifying distributed systems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.