The Serverless Trilemma

Function Composition for Serverless Computing

Ioana Baldini
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
ioana@us.ibm.com

Perry Cheng
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
perry@us.ibm.com

Stephen J. Fink
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
sjfink@us.ibm.com

Nick Mitchell
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
nickm@us.ibm.com

Vinod Muthusamy
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
vmuthus@us.ibm.com

Rodric Rabbah
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
rabbah@us.ibm.com

Philippe Suter
Two Sigma Investments, LP
New York, NY, USA
philippe.suter@gmail.com

Olivier Tardieu
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
tardieu@us.ibm.com

Abstract

The field of serverless computing has recently emerged in support of highly scalable, event-driven applications. A serverless application is a set of stateless *functions*, along with the events that should *trigger* their activation. A serverless runtime allocates resources as events arrive, avoiding the need for costly pre-allocated or dedicated hardware.

While an attractive economic proposition, serverless computing currently lags behind the state of the art when it comes to *function composition*. This paper addresses the challenge of programming a composition of functions, where the composition is itself a serverless function.

We demonstrate that engineering function composition into a serverless application is possible, but requires a careful evaluation of trade-offs. To help in evaluating these trade-offs, we identify three competing constraints: functions should be considered as *black boxes*; function composition should obey a *substitution principle* with respect to synchronous invocation; and invocations should not be *double-billed*.

Furthermore, we argue that, if the serverless runtime is limited to a *reactive core*, *i.e.* one that deals only with dispatching functions in response to events, then these constraints form the *serverless trilemma*. Without specific runtime support, compositions-as-functions must violate at least one of the three constraints.

Finally, we demonstrate an extension to the reactive core of an open-source serverless runtime that enables the sequential composition of functions in a trilemma-satisfying way. We conjecture that this technique could be generalized to support other combinations of functions.

CCS Concepts • **Software and its engineering → Cloud computing**; *Organizing principles for web applications*;

Keywords cloud, serverless, functional, composition

ACM Reference Format:
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 *Proceedings of 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!'17).* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3133850.3133855

1 Introduction

Under economic pressure to innovate ever more rapidly, organizations routinely exploit cloud computing rather than purchase hardware and operate data centers. Serverless computing, also known as *functions-as-a-service*, has recently emerged in support of highly scalable, event-driven applications in the cloud. It allows developers to write short-running, stateless *functions* that can be triggered by events generated from middleware, sensors, services, or users.

The serverless paradigm was pioneered by Amazon with the introduction of Lambda [Cross 2016], and today every major cloud provider offers a serverless platform [Apache 2016; Google 2016; Microsoft 2016]. The model appeals to many developers since it lets them focus on their application

Onward!'17, October 25–27, 2017, Vancouver, Canada
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5530-8/17/10...$15.00
https://doi.org/10.1145/3133850.3133855

logic, and shift infrastructure concerns to the platform. In return for writing short-running, stateless functions, the platform ensures timely and secure execution of those functions. The runtime takes care of resource management, including function activation, isolation, data flow, and logging.

In addition to lowering the management burdens of maintaining an application, serverless platforms promise an economically attractive approach to resource allocation. The cost of idling is zero: for periods of time in which no events reach the application, it costs nothing. During periods of activity, resource allocation can be metered at a fine time granularity. For example, it is common for the major serverless platforms to bill for function invocations in 100 millisecond time slices. Furthermore, the degree of resource concurrency can be automatically adjusted on a function-by-function basis, rather than for the application as a whole.

Beyond simple examples, serverless applications will quickly break the mold of single functions. Complex serverless applications will need to be coded as the composition of simpler functions; e.g. sequential composition, $f_3 = f_2 \circ f_1$.

This paper explores the class of compositions-of-functions that can be implemented as serverless functions: *compositions-as-functions*. This property of self-similarity is natural when considering composition of functions, across multiple domains [Janneck 2003; Nierstrasz and Meijler 1995].

We begin by exploring the limits of existing serverless platforms, when it comes to expressing compositions as serverless functions. The foundation of all major serverless offerings is functions, scheduled in reaction to events. We refer to this as the *reactive core* of serverless programming.

We identify four structures of compositions that can be expressed using only this reactive core, and demonstrate that three constraints are exposed by these approaches. These constraints help us navigate the challenges in engineering serverless applications, and in developing solutions to the composition-as-function problem.

Composition via Reflection If $f_3 = f_2 \circ f_1$ is a serverless function that synchronously invokes f_1 followed by f_2, we say that it follows a reflective invocation structure. Solutions of this kind violate a *double billing constraint*, because the running time of f_1 will be billed twice: once as f_1 and once as part of f_3. As long as a function is active, charges accrue.

Composition via Fusion If instead f_3 is a function that inlines the code of the sequence members, we say that it follows a fusion structure. Schedulers that rely on fusion violate a *black box constraint*; e.g. they assume availability of source code, and that functions are monoglot.

Composition with Asyncs The reactive core permits a fire-and-forget model of composition. If we define an event and wire it to invoke f_1, then f_3 can asynchronously schedule f_1 in such a way that f_3 does not remain active. Solutions of this sort will violate a *substitution principle*. f_3 is no longer a composable serverless function, in that its return value, when invoked synchronously, does not have the desired value of f_2's result.

Composition on the Client If f_3 is written to operate outside of the serverless runtime, we say it follows a client-scheduled structure. We dismiss such structures as true solutions to compositions-as-functions. Namely, such approaches do not satisfy the substitution principle, in that these compositions cannot be *nested* inside of other compositions that are unaware of that client.

Amazon's Step Functions [Amazon 2016] augment their Lambda [Cross 2016] reactive core with workflow scheduling, and are thus an undeniably valuable addition. However, they follow this client-scheduled structure. Neither is a step function a step, nor is it a Lambda function.

In this paper, we conjecture that these constraints form the *serverless trilemma* (ST). That is, when using the reactive core to implement compositions, for any current serverless platform, only two of the three constraints can be satisfied.

Finally, we show that, by careful extension of this reactive core, in a way that exposes the internal scheduling of function invocation in the style of continuation passing, a serverless runtime can enable the sequential composition of functions in a trilemma-satisfying way.

Contributions This paper introduces:

- a formulation of the serverless trilemma, and a discussion of how it affects programming model choices,
- a programming model with the ability to build new serverless functions by composing existing serverless functions, illustrated with several examples,
- a solution to the trilemma for the sequential composition of functions, and
- the implementation in Apache OpenWhisk, an open-source serverless runtime that is also deployed by a major cloud vendor, used by thousands of customers, and which has run millions of serverless functions.

2 A Primer on Serverless Computing

To introduce the concepts, and to further motivate the value proposition of serverless computing, we start with an example. Typical of many applications, this example creates a new service out of two pre-existing ones. Its job is to connect them via an event-driven data pipeline: the application consumes events generated by the first service, processes them, and sends the resulting data to the second service.

This example, illustrated in Figure 1, monitors the Travis continuous integration tool [Meyer 2014] for build failures, and posts notifications via the Slack messaging service [Lin et al. 2016]. The goal is to notify those developers who break the build. Travis is a popular platform that automates integration tests, helping teams maintain stability in the face of frequent changes to their code. This section will illustrate

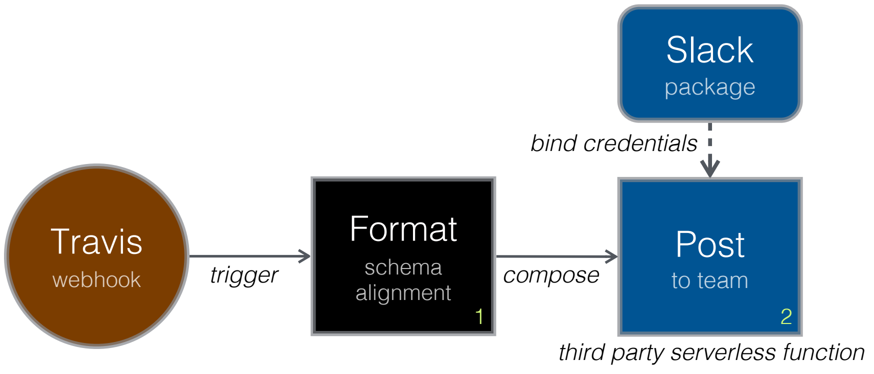


Figure 1. A motivating example: using the Slack messaging platform to notify developers of build failures from the Travis continuous integration tool. Useful features that OpenWhisk also provides (but are beyond the scope of this paper) are packages, which allow for grouping and namespacing, and currying, which lets us, in this example, extend a shared package with a binding of credentials.

that it is possible to implement this kind of server-side glue logic, without having to develop or maintain any persistent services. Serverless functions, and any compositions thereof, become a service immediately upon creation,

2.1 Functions as a Service

When a build completes, Travis generates an event whose schema does not align with that expected by Slack. The application therefore needs a schema alignment function, `Format` in Figure 1, to project the relevant fields from the Travis event, and produce a record that conforms to Slack's API. A second function, `Post`, consumes this record and invokes a Slack endpoint.

The code for `Format` appears in Listing 1. This function, written in JavaScript, receives a single argument `args` as input. `args` is a dictionary (manifested as a JSON object, in this JavaScript example) containing the `author` property. The function produces as output a new dictionary containing the property `text`, as is required for the Slack notification.

Listing 1 A serverless function: **Format** from Figure 1

```
let main = args => ({
  text: `A Travis integration build failed,
         due to a change by ${args.author}`
})
```

In order to run `Format` in OpenWhisk, we must first deploy it. Having stored the source in a file `format.js`, this can be achieved using the command line interface (CLI) called `wsk`; OpenWhisk refers to serverless functions as *actions*:

```
% wsk action create Format format.js
Ok, your endpoint is https://.../Format
```

Without provisioning any infrastructure, this command deploys a cloud-based endpoint. With the right credentials, one can invoke it, e.g. with the cURL HTTP client:

```
% curl https://.../Format?author=Uli
{ text: "A Travis integration build failed,
 due to a change by Uli" }
```

2.2 Event-Driven Invocation

Manual invocation is useful for initial unit testing, but, in the real application, the functions should be invoked based on Travis *events*. When a Travis build completes, the functions should be invoked.

This kind of triggered invocation is accomplished as with publish-subscribe systems generally. In pub-sub models, the production and consumption of events is mediated by a topic [Eugster et al. 2003]. Consumers subscribe to a topic of interest, and producers create messages filed with that topic. In OpenWhisk, a *trigger* represents a named topic:

```
% wsk trigger create BuildDoneTopic
Your endpoint is https://.../BuildDoneTopic
```

One can then configure Travis so that, upon build completion, it makes an HTTP call to the endpoint of that newly created trigger. In turn, this results in the activation of any registered consumers of the underlying trigger. To register an action as a new consumer of a trigger, one creates an OpenWhisk *rule*. For the moment, let us assume the developer has crafted a function `FormatThenPost` by merging the two actions and deploying the composition:

```
% wsk rule create TravisToSlackAutomation
BuildDoneTopic FormatThenPost
Your automation is live
```

With the trigger, rule, and actions in place, the application operates autonomously in reaction to Travis events. Notably absent from this discussion is the injection of the secrets necessary to make an authenticated call to the Slack service endpoint. OpenWhisk supports this style of extending an existing action via curried function application.

2.3 Componentization and Composition

Rather than create a single monolithic function, it is often desirable to separate the concerns of schema alignment and notification. Doing so enables code reuse, and results in small and focused functions. The `Post` method in particular has clear value across many applications. Furthermore, once a developer deploys the `Post` function, it can be reused across any number of entirely unrelated applications. In a conventional server deployment, it would be inconceivable to devote an entire server deployment to such a small unit of functionality.

To support this decomposition, the serverless runtime must facilitate function *composition* into larger data flows. The data flowing out of an invocation of `Format` should be routed to the input of a second invocation, of `Post`. The mechanics of this orchestration is the main challenge of this paper.

3 The Core (Reactive) Model

To lay the groundwork for the rest of the paper, this section introduces the core OpenWhisk programming model. The core of OpenWhisk consists of a reactive scheduler of monolithic actions, and a reflective capability. By reflective invocation, one action can control the scheduling of other actions. Section 4 demonstrates the power and limits of this kind of *external* scheduler, where an entity outside of the OpenWhisk core manages invocations. Otherwise, a program entirely utilizes only the core scheduler; we refer to these as *static* or *statically scheduled* programs.

Table 1 describes the grammar for an abstract syntax of the model. From this model, there is a straightforward mapping to the concrete syntax of the CLI, in part presented earlier. For static programs, the OpenWhisk runtime can interpret a parse tree, and schedule the invocation of actions appropriately.

To represent events, the model includes *triggers*, a first-class representation of topics in an abstract message queue. A *program* consists of one or more *rules*; a rule is constructed by the *when* combinator over a given trigger and action; and so on. This section steps through the types and the semantics of the combinators of this grammar.

π	:=	R | R ; π	(program)
R	:=	T.**when**(A)	(rule)
A	:=	a | P:a | A.**with**(D)	(action)
P	:=	p | P.**with**(D)	(package)
T	:=	t	(trigger)
D	:=	{ k:v, . . . }	(dictionary)

Table 1. The language of static composition includes three types (rules, triggers, and actions), and two combinators (`when` and `with`). Here, t and a denote identifiers.

3.1 OpenWhisk Actions

An *action* is a stateless function, uniquely identified by a name. This name will prove useful, when manually or reflectively invoking an action by name. For example, Section 2.1 used the `curl` command-line HTTP client to invoke an action by name.

The input to an action is a Dictionary, *i.e.* a set of key-value pairs. The output of an action is either a Dictionary, if the action completed successfully, or an indication of failure. Adopting Scala's `Try` type, where instances of `Try[T]` are either of type `T` or of the system's error type, we express the type of invocations of an action `a` as:

```
a.invoke: Dictionary → Try[Dictionary]
```

In addition to their normal response, actions may also produce a side channel of log output. The logs of an invocation are considered to be a possibly empty list of log records $(l_1, \ldots, l_n)$. To permit performant implementations, the log records are restricted so as to be accessible only post-mortem.

3.1.1 Statelessness

An action cannot assume that lexical state persists, from one invocation to the next. Statelessness allows for simpler solutions to scheduling and scaling [Momtahan et al. 2006]: *e.g.* serialization is confined to the peripheries, and the complexity of temporal scheduling scales linearly with the number of actors [Stoica 2004].

If any state needs to be communicated to future invocations, the action is responsible for orchestrating suitable out-of-band mechanisms. This is possible, while maintaining the serverless property of an application. For example, state can be externalized into a hosted document store such as S3 by Amazon [2008], or IBM's Cloudant [Bienko et al. 2015].

3.1.2 At-Most-Once Invocation Semantics

It is impossible to guarantee *exactly-once* delivery in a distributed system [Fischer et al. 1985]. For this reason, implementations need only guarantee *at-most-once* semantics with respect to action invocation.

3.2 Curried Function Application

Actions are given limited access to non-lexical state, by using the `with` combinator. Given an action a and a mapping (set of key-value pairs) M, we say $a.\textbf{\textit{with}}(M)$ is the action a' that results from currying a according to the variable assignment of M. We nickname a' to be a *binding* of a.

Operationally, we consider the semantics of binding in terms of precedence. For $a.\textbf{\textit{with}}(M).\textbf{\textit{invoke}}(D)$, i.e. an action curried with mapping M and invoked with actual parameter mapping D, then, to resolve the value for variable x, first precedence is given to $M[x]$, and second precedence to $D[x]$.

3.3 Packages: Namespaces and Bulk Currying

It is often convenient to group actions together, under distinct namespaces. An OpenWhisk *package* P is a set of actions A_p to which `with` bindings can be combined. For a package P and action $a \in A_p$, the invocation:

$$\texttt{P.}\textbf{with}(M_p)\texttt{.a.}\textbf{with}(M_a)\texttt{.}\textbf{invoke}\texttt{(P)}$$

uses the same precedence rules as above, except for treating the variable assignment M_p as a tertiary source of values, after actual parameters and action-level currying. Package currying provides a useful abstraction, especially when dealing with credentials and other secrets. Binding allows programs to isolate credentials, so that they are exposed only to the action that needs them.

3.4 OpenWhisk Triggers

The programming model manifests a message queue to programs as two operations over named topics, called *triggers*. Via the type constructor $Trigger[t]$, a new subtype of trigger may be constructed, where t is the string that identifies a topic in the application.

A trigger $Trigger[t]$ represents a class of messages. When convenient and unambiguous, we will denote this as simply t. We consider messages to be of type Dictionary, as is the case for the input and output of actions. For example, the Travis-to-Slack application from Section 2 installs a Travis build hook. Travis messages are a class of events: $Trigger$[“build_done”].

For a given topic t and payload D, we denote $t.\textbf{\textit{fire}}(D)$ as the operation that constructs a new message with this payload, and notifies the message queue of its arrival. For example, to create a message on the “build_done” topic, indicating a successful build:

$$Trigger[\text{“build\_done”}].\textbf{\textit{fire}}(status \mapsto \text{“success”})$$

3.5 Reactive Invocation via Triggers

Triggers and actions may be combined via `when`, which links the availability of a message to the invocation of an action. We use the name *rule* for an association from trigger to action. For a trigger t and action a, the `when` combinator constructs a new rule: $t.\textbf{\textit{when}}(a)$. Each rule has a mutable `enabled` status bit. The status of a rule governs whether or not messages associated with t will result in an invocation of a.

We define rule semantics operationally. Suppose trigger t is associated with n rules $r_1, \ldots, r_n$. Upon receipt of a message $t.\textbf{\textit{fire}}(D)$, first the subset of enabled rules is determined; say this set is of size $m \leq n$. Then, each of the m associated actions will be invoked once, with input D.

Triggers and rules, both being asynchronous operations, have a return type of `Unit`. Their intended use is for loose coupling of services.

Rules allow programs to dissociate themselves from managing the schedule of action invocations. Consider Figure 3, which uses three rules to connect two event sources and two actions. This program notifies team members when either a new GitHub issue or pull request is created; for pull requests only, the program will also persist a record in a data store. Using the mutable status bit, either one of these rules can be switched off, without having to tear down or reconfigure settings with GitHub.

`a.`**`invoke`**`(d)`	reflective action invocation
`t.`**`fire`**`(d)`	reflective message creation
`Action``(code, name)`	action constructor
`Trigger``(name)`	trigger constructor
`Rule``(name,a,t)`	rule constructor

Table 2. Operations for dynamic composition.

3.6 Deployment and Reflection

Given its source, an action can be *deployed*. Only then will an action become invokable. Every deployed action has a distinct remote `invoke` endpoint. Similarly, once deployed, a trigger receives a distinct remote `fire` endpoint. Using these remote endpoints, it is thus possible for one action, during its invocation, to invoke another. We refer to this prospect of actions invoking actions as *reflective* invocation.

These introspective facilities allow for experimentation with new kinds of combinators, beyond those provided by the core. By introspecting over the properties of static compositions, one can program new *combinators as serverless actions*, using the full power of a general purpose language.

Table 2 lists the introspective functions offered by the core model. The `action`, `trigger`, and `rule` functions represent constructors for the respective entities. These reflective constructors allow for metaprogramming, a scenario we will explore in more detail.

In the rest of this paper, we will assume that the introspective API is manifested as a JavaScript library made available to serverless actions implemented in JavaScript. For clarity of presentation, we will use the notation suggested by Table 2.

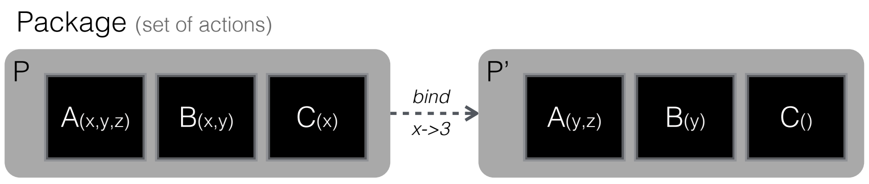


Figure 2. OpenWhisk supports *packages*, which allow convenient grouping and namespacing of actions, and curried function application. A value bound to a package is inherited by all enclosed actions.

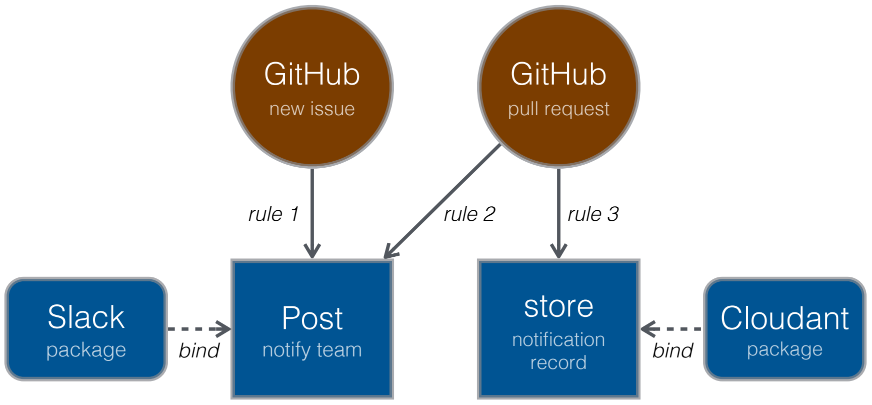


Figure 3. Rules enable programs to program event-driven invocation of actions. Here, three rules connect two triggers to two actions. Each action comes from a *bound* package.

For example, the JavaScript `a.invoke(d)` should be read to mean that a reflective invocation is intended.

4 The Serverless Trilemma

This section explores the limits of expressivity of the core model, when it comes to programs that compose other programs. We explore three implementations of sequential composition, and, in doing so, expose three constraints inherent in supporting compositions in a serverless runtime.

Sequential composition, so far missing from the core model, allows programs to construct a new action from two given actions. The meaning of invoking this new action will be defined precisely by the semantics of `flatMap` monadic composition [Wadler 1990]: sequences behave as data pipelines, where the only implicit communication is at the inter-action junctions, and execution short-circuits at the first failing action.

We refer to this desired sequential combinator as `then`. For example, the composition $a.\textbf{\textit{then}}(b).\textbf{\textit{then}}(c)$ should be equivalent to a new action s such that the input to s becomes the input to a, and the output of s is identical to the output of the expression $c(b(a()))$.

4.1 Composition by Reflective Action Invocation (Double Billing Constraint)

As a first attempt, we can use the JavaScript introspective toolkit from Table 2 to implement `then` as a dynamic composition. Given a list of actions L, Listing 2 reflectively invokes each action in turn, passing the data from one stage of the pipeline to the next. It serves as an *external* scheduler, in that it takes charge of orchestrating timing and dataflow, by relying only on the core primitives.

Listing 2 Sequential composition using reflective invocation

```
let main = L => args =>
  L.reduce((pipe, f) => f.invoke(pipe), args)
```

This example highlights the expressivity afforded by introspection, but this power comes with a compromise. In return for the use of an external reflective scheduler, the composition may substantially increase the cost of invoking the underlying actions.

As illustrated in Figure 4, the scheduler must remain active for at least as long as each of the sequenced actions is active. Having a scheduler action active concurrently with its components has economic implications. Any active action consumes a shared resource, and thus must carry a cost. For

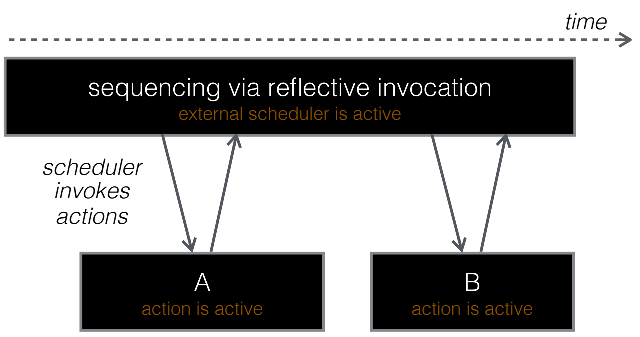

time
sequencing via reflective invocation
external scheduler is active
scheduler invokes actions
A
action is active
B
action is active

Figure 4. The *double billing* problem. If we implement a serverless function that reflectively invokes other serverless functions, we pay twice: once for the scheduler, and a second time for the constituents of the sequence.

example, it is common in environments with shared infrastructure to bill for the integral of reserved memory over the duration of the reservation.

Here, OpenWhisk is no different than any cloud service, in that use is billed when it requires a reservation of resources. In serverless environments, a function invocation reserves resources, and the function return relinquishes this reservation.

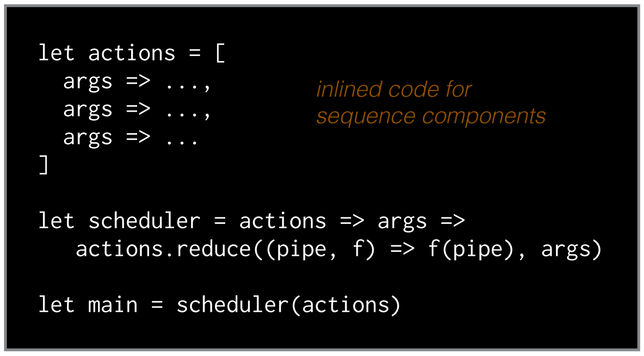
It seems desirable to avoid this overhead. We refer to this desire as the *double billing constraint.*

Another challenge with this style of composition is the fact that serverless runtimes impose a maximum duration T for any single function invocation. This is the total time a function is allowed to reserve its alloted resources. The reason for this time-limitation is rooted in the operational tradeoffs the platform must balance. For Amazon Lambda and OpenWhisk, T is capped at 5 minutes, for example. The consequence of this limit is that any external scheduler cannot compose actions where any single activation lasts for time T, or, where the total duration of all function activations in the composition is T or more.

4.2 Composition by Fusion of Actions (Black-Box/Polyglot Constraint)

It is possible to avoid the double billing penalty, by specializing an external scheduler for a given list of actions [Jones et al. 1993]. For example, we can code an action that fuses the sequenced actions into the source code of a new action [Fradet and Ha 2004]. The implementation in Figure 5 first fetches the source code for each action in the sequence. With the source to each action in hand, it operates analogously to our first attempt, Listing 2, except it can directly invoke each action as a JavaScript function call rather than as a reflective call to the serverless runtime.

This solution to sequential composition works well, as long as the source to every action in the sequence is available, and written in the same language. For example, in Figure 5,


```
let actions = [
  args => ...,
  args => ...,
  args => ...
]

let scheduler = actions => args =>
   actions.reduce((pipe, f) => f(pipe), args)

let main = scheduler(actions)
```

Figure 5. The *black box/polyglot* constraint. An action that uses fusion to schedules the actions in a sequential composition has limited applicability. In this example, if the source to any of the actions in the composition were not available to the fuser, or if any were written in a language other than JavaScript, the fused composition would not be possible.

if any of the actions in the sequence are not JavaScript, this implementation fails.

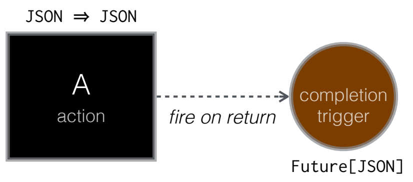
In some cases, such as performance optimization, source access is helpful. However, it seems desirable to avoid these two restrictions (source availability and monoglot sequences) as a strict requirement for composing actions into larger constructs. An action author should be able to share their action without sharing the source. Furthermore, as of this writing, the development community is strongly pluralistic in its choice of languages: the top 10 languages on GitHub explain just over 60% of the projects; numbers from a recent academic study are similar [Ray et al. 2014].

We refer to this desire to avoid a strict requirement for source code and monoglot compositions as the *black box constraint.* It is also worth noting that composition by fusion is also subject to the maximum duration imposed on invoking an action, as was the case with composition by an external scheduler.

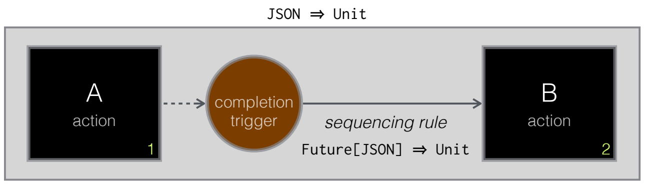
4.3 Interlude: The Serverless Substitution Principle

The first two attempts at implementing sequential composition use a scheduler action to invoke other actions synchronously. They take a list of N actions, and implement action $N + 1$ as the scheduler, which invokes those actions in a request-response/blocking style — *e.g.*, in the case of fusion, a simple JavaScript function call. Despite their pitfalls, the first two approaches generate compositions that are actions, and so behave as actions must.

Compositions-as-actions conform to the the JSON in, JSON out protocol of actions. For compositions, this protocol implies a single-entry/single-exit structure, no matter the internal structure of a composition's scheduled dataflow. Any dataflow graph with a single post-dominator node has a well-defined synchronous invocation behavior: wait for the



(a) Use triggers to *lift* actions to a Future form



(b) Use rules to implement sequential composition

Figure 6. Triggers, actions, and rules, at least on their own, are insufficient to implement a continuation style of composition, due to a violation of the substitution principle. Compositions, in this scheme, would not behave like actions, because rules asynchronously spawn their action, and return `Unit`.

post-dominator invocation to complete, and respond with that invocation's result JSON, akin to the `async/await` pattern [Davies 2012; Haller and Miller 2013; Okur et al. 2014] or Scala's `yield` [Odersky et al. 2004].

It is desirable, then, for compositions of actions themselves to be actions, and thus to have a well-defined interpretation for blocking invocation. We refer to this desire for synchronous self-similarity as the *serverless substitution principle*; there are parallels here to Janneck's notion of *safety* when composing actors [Janneck 2003], and, more distantly, behavioral subtyping [Liskov and Wing 1994].

4.4 Composition by Triggered Invocation (Substitution/Synchronous Constraint)

The first approach, using reflective invocation, suffers double billing because the scheduler action remains active for the duration of its constituents. Consider a possible alternative that avoids double billing by adopting a style akin to continuation passing. A sequence is scheduled by having actions invoke a continuation, which is the remainder of the sequence.

Let us attempt to implement a continuation passing style, by scheduling actions using triggers and rules. We first describe the approach, which attempts to use only the core features from Section 3, then argue that this approach necessarily violates the substitution principle.

To begin our attempt at continuation passing, convert each action to a Future form (c.f. monadic lifting [Liang et al. 1995]), as shown in Figure 6(a). This can be accomplished by associating a completion trigger with every action. Then, when an invocation of an action completes, the invocation fires the associated completion trigger, making sure to pass the return value of the invocation to the trigger payload.

With this structure in place, we can use rules to sequence one action after the other. For a desired sequence $a.\boldsymbol{then}(b)$, where the completion trigger of action a is denoted $ct(a)$, create a rule $ct(a).\boldsymbol{when}(b)$ — when the completion trigger of a fires, schedule an invocation of action b. The resulting composition properly flows data from input to output. Each action will return the correct result, for a given input.

However, this class of rule-trigger sequences violates the substitution principle. In OpenWhisk, an action maps dictionary to dictionary, and a rule maps future of dictionary to `Unit`. The latter mapping follows from the asynchronous nature of rule scheduling. Therefore, a sequence composed of triggers and rules also has a return type of `Unit`.

Moreover, the OpenWhisk core from Section 3 does not specify that a completion trigger should be fired on return, the black box constraint forbids modifying an action to do so, and the double billing constraints forbids wrapping actions to add this capability.

4.5 Client-Side Scheduling (Abnegation)

To implement a sequential composition, one may decide to use a client-side scheduler; *i.e.* a scheduler not implemented as an action. This fourth approach turns its back on the OpenWhisk world. Since this style of scheduler runs on the client, there are no black box or double billing violations. Furthermore, as long as the client implements all requisite compositions *on the client*, this approach satisfies the substitution principle.

Certainly there are times when client-side scheduling makes sense, just as, sometimes, fusion makes sense. However, we do not consider this approach to have solved the problem of serverless composition.

5 Trilemma-Safe Sequential Composition

We have motivated that a serverless core must offer more than actions, rules, and triggers, if it wants to support compositions that do not violate the serverless trilemma (ST-safe). This section describes a solution that supports ST-safe sequential composition. Future work will expand this to other forms of compositions.

5.1 Overview of the OpenWhisk Invocation Flow

Since any solution to ST-safe compositions must expand the reactive core of OpenWhisk, we first briefly summarize the handling of invocations in the OpenWhisk implementation. Section 7 presents the implementation in more detail.

As illustrated in Figure 7, the invocation flow in OpenWhisk consists of four components. An incoming invocation request is managed by the *controller*. It first selects an *invoker*

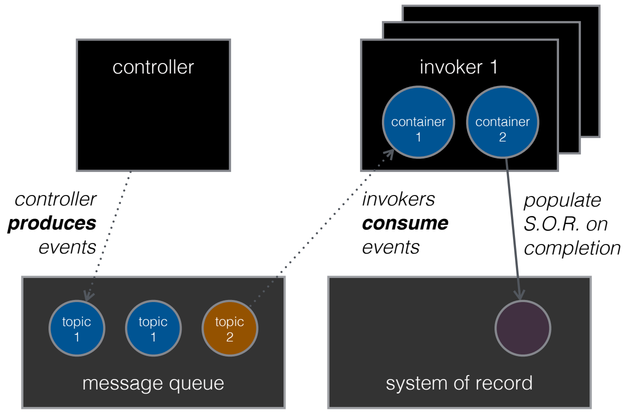


Figure 7. In the OpenWhisk architecture, requests are handled by a *controller*, which uses a message queue to line up work for a family of *invokers*. Each invoker hosts a number of containers (used for isolation), and listens for messages that the controller has directed to it. Section 7 describes the implementation of this architecture in greater detail.

from a pool of hosts, and then queues up the invocation request on a *message queue*. Each invoker is host to a number of containers, which are used to isolate the invocations from each other. Invokers subscribe to events directed at them, and, when resources become available, inject the invocation request into a container. At this point, the invocation commences. When the invocation completes, the host invoker stores the return value in a semi-permanent *system of record*, and indicates to the controller that it may now respond to the client.

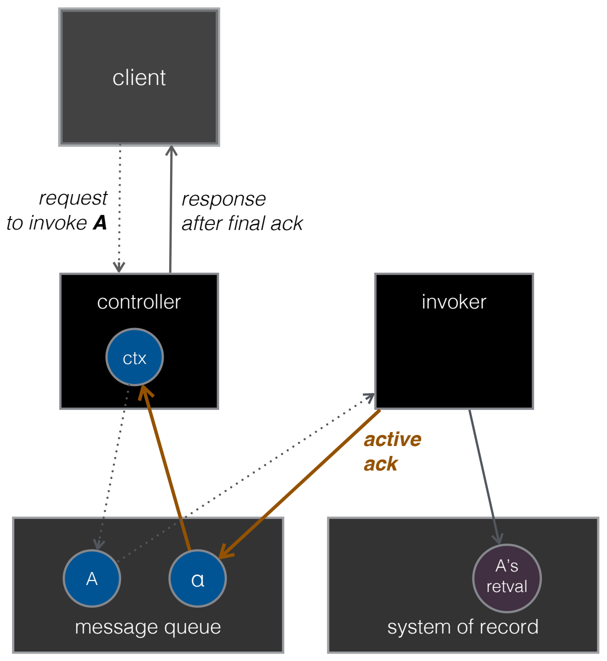
5.2 Realizing Completion Triggers with "active ack"

To introduce the notion of completion triggers, and to reduce the latency of request-response invocations, OpenWhisk adopts the microarchitectural strategy of pipeline bypass [Hennessy and Patterson 2011]. In OpenWhisk, this is nicknamed *active ack*. As illustrated in Figure 8(a), invokers signal completion in a way that bypasses the system of record. Instead, active ack uses the message queue as a scoreboard to forward results from the invoker to the controller, so that the controller can act quickly on invocation completion.

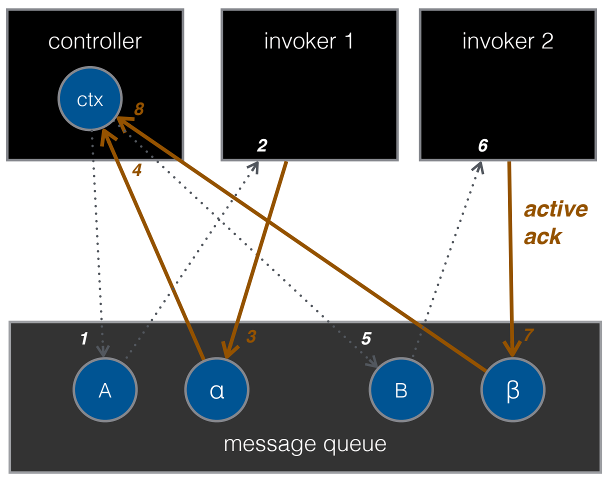
In this active ack scheme, the controller, upon receipt of an invocation request, creates a topic in the message queue, and registers as a consumer. Along with the normal invocation payload, this topic is passed to the invoker. When the invocation completes, the invoker produces a message on the given topic. Finally, the message queue notifies the controller, which can respond to the invoking client.

5.3 ST-Safe Sequences with active ack

It is now straightforward to employ active ack to implement sequential composition. Two changes are required. First,



(a) The "active ack" flow, used to orchestrate and optimize invocations. When an invocation completes, the invoking container sends a message back to the waiting controller. Dotted lines indicate request/invoke flows, and solid lines indicate response/return flows. Compared to using the system of record to communicate return values, active ack reduces the overhead of blocking calls by 18x.



(b) You can follow the numbered flow of edges to discern the sequence of events unleashed by the invocation of a given sequence $a.\mathbf{then}(b)$.

Figure 8. OpenWhisk employs an "active ack" strategy to schedule sequences. Active ack gives the completion trigger illustrated in Figure 6, and identified, in Section 4.4, as missing from the OpenWhisk core.

when creating an action, the author must signify the action to be of type Sequence, and specify the component OpenWhisk actions that form the composition. Second, the controller must handle the invocation of a Sequence action differently, as illustrated in Figure 8(b). When handling the invocation of a sequence, upon receipt of an active ack, the controller lines up the invocation of the next action in the sequence. This loop of enqueued invocation and active ack messages is repeated until the controller has reached the end of the list of actions. At this point, it responds to the client with the final active ack payload.

The controller implementation is conceptually similar to the composition via reflection (refer to Section 4.1). However, because it is internalized into the system runtime instead of the application code, the user does not get double billed. The system runtime uses conventional thread and actor scheduling, compared to the more heavy-weight resources that must be reserved for an action invocation. As a result, internalizing the continuation has negligible system overhead while also permitting serverless compositions as functions.

5.4 Performance Benefits

An important side effect of the active ack bypassing approach is latency reduction. With this approach, the latency for the result passing from the invoker to the controller is about 1-2ms on average. In contrast, storing and then fetching a document with the system of record takes, on average, 26ms and 10ms respectively, with outliers in the hundreds of milliseconds. These numbers are from a deployment that uses a dedicated allocation of Cloudant [Bienko et al. 2015], a cloud-based CouchDB database [Anderson et al. 2010].

6 Looking to the Future: New Combinators

In recognition of the need for composition patterns beyond sequences, this section discusses three such combinators: Event-Condition-Action (ECA), retry, and data forwarding. Though not yet implementable in a ST-safe way in OpenWhisk (nor any contemporary serverless runtime, to the authors' knowledge), it is valuable to begin exploring these more complex patterns.[1] This paper begins the discussion by demonstrating their implementation in terms of the OpenWhisk reflective calls. The discussion will be framed as an exploration of the engineering decisions one must make, trading off one pillar of the serverless trilemma against the others.

The combinators are discussed in the context of an expanded Travis-to-Slack example, using the simpler application from Section 2 as a starting point. The expanded application post-processes build notifications into user notifications using the pipeline shown in Figure 9. The figure numbers the steps from 1–7.

[1]Amazon's Step Functions [Amazon 2016] support some of these patterns, however, as discussed earlier, this is an example of client-side scheduling and therefore not composable.

This full application amends the functionality of the Travis notification webhook with the capability to notify a variable user (so that we can notify the user that broke the build), and with a log analysis step (so that we can provide details of what broke). We highlight three of the steps here, demonstrating the three proposed serverless design patterns. The source code of the example application is available online.[2]

6.1 ECA: Static Composition versus Combinator

The notification message from Travis includes extraneous details. Thus, the application, in step 1, projects out the relevant fields: the build identifier and build status. Conditional on build failure (step 2), the application proceeds to step 3.

To implement this kind of Event-Condition-Action (ECA) composition, we can start with a Condition-Action combinator. Listing 3 shows a combinator that is given a condition and action as input. Given an input dictionary (args), it reflectively invokes the condition. Recall from Section 3.1 that actions return a Try[Dictionary]. If the condition action fails, this combinator will return a selected default value (akin to Unit in Scala).

Listing 3 Condition-Action Combinator

```
let main = (condition, action) => args
  try { return condition.invoke(args)
          && action.invoke(args)
  } catch { return {} }
```

This combinator will result in double billing. If this constraint violation is unacceptable, then, with different compromises, ECA can be expressed using static *when* and *then* combinators.

Given a trigger *e*, and two actions *c* and *a* that implement the condition and action, e.**when**(c.**then**(a)) can implement ECA.

This static composition will not result in double billing. However, two other compromises must be accepted. First, conditions must be written to obey a convention, that failure encodes a false condition. Second, one must be willing to accept spurious failures in the system of record; in particular, any summaries of usage will now show that actions have failed, when rather they are manifestation of this convention.

6.2 Retry as Metaprogram

Step 4 verifies the location of the log content. For a short period of time, Travis stores logs within its own infrastructure. Soon after a build completes, Travis shuttles logs to a third party storage provider, Amazon's S3 [Amazon 2008; Robinson 2008]. However, there is a window where neither

[2]GitHub repository link available upon request.

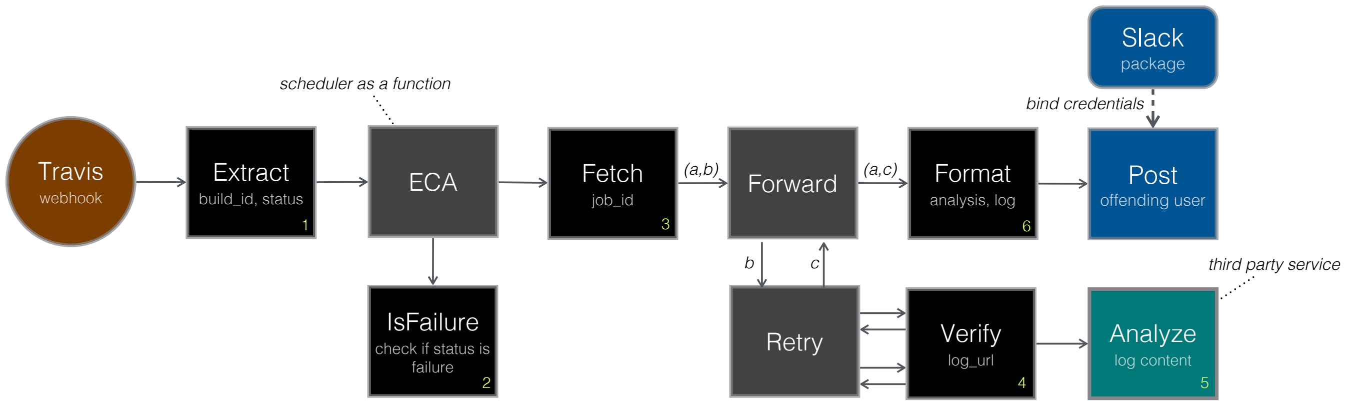


Figure 9. Case Study: the full Travis-to-Slack application, introduced in part in Section 2, includes three new composition patterns: Event-Condition-Action (ECA) for guarded invocation, Retry (fetching logs is a subtask that might fail), and Forward (analyzing the logs for a detailed description of the failure is done via third-party code).

location is valid. Step 4 may therefore need to retry in order to determine a valid location for the logs.

Listing 4 Step 4 of Travis-to-Slack: Verify

```
let probe = url => request(url)
    .then({ log_url: url }))

let main = args =>
    probe(travisURL(args.job_id))
        .catch(probe(S3URL(args.job_id)))
```

Listing 4 implements the logic for verifying log location. The code for Analyze is not shown — it invokes a third-party log analysis service, with the log location as input. The program needs to retry the sequence Verify.**then**(Analyze). First, we can manifest this static composition:

```
% wsk action create VerifyThenAnalyze
       --sequence Verify Analyze
```

Listing 5 Retry metaprogram

```
let main = (N, A) => args => {
  let tryInvoke = i =>
    i < N && (A.invoke(args) || iter(i + 1))

  return tryInvoke(0)
}
```

To retry, we can code a metaprogram that takes a function as input, and calls it until success. Listing 5 illustrates this retry metaprogram. Our desired VerifyAndAnalyze function can now be expressed as a binding of Retry, where the action argument A is bound to VerifyThenAnalyze.[3]

[3]**bind** is the CLI command for the **with** combinator of Table 1, and creates a curried version of an action by partially filling in parameters.

```
% wsk bind Step45 Retry
       -p N 5
       -p A VerifyThenAnalyze
```

The bound action can now be invoked, and it will be dynamically scheduled to behave like VerifyThenAnalyze.

6.3 Forward as Metaprogram

The output of step 3 in Figure 9 is a pair of objects (denoted (a,b) in the figure). One of the two is used as input to steps 4 and 5, while the other needs to be forwarded around steps 4 and 5. This is because the return value of the Retry is that of step 5, which is an action we do not own. We can implement this forwarding pattern as a metaprogram, as shown in Listing 6.

Listing 6 Forward Metaprogram

```
let main = (action, keysToFwd) => args =>
  action.invoke(args).then(retval =>
    Object.keys(args)
          .filter(keysToFwd.includes)
          .forEach(x => retval[x] = args[x])
```

This Forward function is curried on the action to forward over, and the keys that should be forwarded around that action invocation.

7 The OpenWhisk Implementation

This section summarizes the implementation of the open-source OpenWhisk project. The project source is available for general consumption: http://openwhisk.org.

7.1 Operational Tradeoffs

OpenWhisk deployments must allow applications to scale up and down based on offered load, allocating resources as needed. In order to do so economically, OpenWhisk exploits multi-tenancy, sharing a pool of resources across many users. By multiplexing the underlying resources, OpenWhisk need not bill for allotments. Rather, OpenWhisk can offer a billing structure based on resource usage, subject to some chosen minimum billable time unit.

For these reasons, OpenWhisk imposes a hard limit T on function running time. This constraint allows the system to schedule functions with certain guarantees on future behavior. In particular, the system can guarantee that a particular function will run for a bounded time. So, the system knows that resources will free up by a certain deadline, with no danger of user code monopolizing resources indefinitely. Thus, the simplest serverless operational contract defines two fundamental parameters, which specify the shared expectations regarding action execution:

- **Action Timeout** Any single action invocation may run for no longer than duration T.
- **Latency Bound** The system may not queue invocation requests for longer than duration L.

Together, these parameters dictate the tradeoffs a serverless platform provider must balance. If the service is under-provisioned in terms of the execution resources available, it must either reject new invocation requests or it must internally queue them until resources are available. The worst case execution time for an action is therefore $WCET(a) \leq T + L$. A level of internal queueing is acceptable but must not be exceedingly long lest the platform become undesirable to use. In other words, $L \ll T$ and in practice the expectation is that L is small even under normal load.

One way to lower L is to over-provision the platform, or have the ability to deploy additional execution capacity with low latency (in this way L represents the system overhead related to providing horizontal scaling). A lower T implies greater capacity since in a given amount of time, more requests can run on a given resource. But lowering the timeout also makes the platform undesirable.

7.2 Isolation and Multitenancy

Since the platform must run untrusted user code, isolation rises to a top-level security concern. OpenWhisk uses Docker containers [Merkel 2014] as the basic unit of isolation. Using such containers, the system can rely on Linux kernel protections to isolate user processes from each other and from the system. Having container isolation is helpful, though imperfect — undisclosed or unpatched kernel vulnerabilities may be exploited to escape containment. For this reason, production OpenWhisk deployments must rely on multiple layers of monitors and network firewalls to provide a layered defense.

Beyond isolation, the choice of Docker allows OpenWhisk to host arbitrary containers as serverless actions. Actions can be implemented with any technology, from shell scripts to Haskell.

7.3 Handling Requests

The system provides a REST API that supports Create-Read-Update-Delete (CRUD) management, action invocation, and trigger fire operations.

Edge Proxy The system mediates the REST API through a set of NGINX [Reese 2008] reverse proxy servers. The edge proxies direct each request to a *controller* host.

Action Invocation Requests When receiving a request to invoke an action, the controller passes the request to a load balancer subsystem. The load balancer chooses a slave (called an *invoker*) to handle the request. Having selected an invoker, the load balancer posts a message to the Kafka message queuing system [Kreps et al. 2011], directed to the chosen worker. Invokers listen for invocation request messages from the message queue.

Trigger Fire Requests The implementation maintains a persistent trigger map $\mathcal{T}$, from trigger name to a *rule set*. A rule set is a set of pairs (a, e), where a is some action for which there exist a rule $r = t \Rightarrow a$, and e is the boolean-valued *enabled bit for* r.

A creation request for trigger t results in the update $\mathcal{T}[t] = \{\}$, *i.e.* adding an initially empty rule set for t. A request to create a rule $t \Rightarrow a$ results in the update $\mathcal{T}[t] \mathrel{+}= (a, true)$. In this way, rules are initially enabled.

Upon receipt of a fire request for a trigger t, the controller looks up the set of enabled actions $\{a \mid (a, true) \in \mathcal{T}[t]\}$ and invokes them.

7.4 The Invoker

Each slave that can run actions runs a microservice called an *invoker*. The invoker listens to the message queue for requests to run actions, as generated by the load balancer.

As discussed earlier, the system relies on Linux Containers (Docker) to provide standard, isolated environments for each action. The invoker manages a pool of such containers, dispatches work to containers, and processes action results.

Cold Starts In the simplest possible flow, we consider the case where the invoker receives a request to run an action, but it has no container already provisioned suitable to handle the invocation. In this case, the invoker must spin up a new container to service the request. We call this a *cold start*.

Our system currently implements *cold start* with `docker run` command, starting a container via the Docker daemon running on the host. On Docker 1.12, we observe starting a container with minimal options (`docker run -d hello-world`) on an unloaded system to impose latency of about 200ms. This figure rapidly rises to 500ms when security, network, quota, and file system options are employed.

To reduce the impact of cold starts, the OpenWhisk implementation introduces two optimization strategies: stem cell containers and container caching.

Container Pooling: Stem Cell Containers The first optimization strategy keeps a pool of running containers. We say that a *stem cell for runtime* R is a running Docker container that has been created from the base image for R, but for which the action source has not yet been provided. Once started, a stem cell container waits for an invocation request. That request defines the body for the invoked action, at which point the invoker attempts to find a free stem cell, then injects the body into that running container, and finally proceeds normally to run the action.

Container Caching To take load off the stem cell pool, the invoker uses the Docker pause/unpause facility to cache containers. When a container completes an invocation, the invoker will pause it, in the hopes that an invocation request for that same action will arrive in the near future. Docker pause and unpause take approximately 50-60ms — well below the 200-500ms latency for start a new container. For safety, we further restrict container reuse to invocations by the same user. As these idle containers can not be kept around indefinitely, the invoker discards stale containers based on capacity conflicts or a configurable timeout.

8 Related Work

The serverless landscape is relatively new, but it builds on long-running trends in both distributed systems and programming languages.

In 2006, Fotango hosted a short-lived commercial platform, Zimki, that began the movement towards functions as a service [Velte et al. 2010].[4] Seven years later, Amazon more successfully pioneered the serverless space with their Lambda offering on Amazon Web Services [Cross 2016]. Other commercial offerings include Google Cloud Functions [Google 2016], the IBM Bluemix installation of Apache OpenWhisk [Apache 2016],[5] Iron.io [IronIO 2016], and Microsoft Azure Functions [Microsoft 2016]. There is currently no published academic work on either the programming model or the runtime implementations of any of these commercial systems.

In contrast, OpenLambda [Hendrickson et al. 2016] is an ongoing academic effort to design and build a serverless runtime. We view the present work on OpenWhisk as a partial answer to some of the challenges outlined in the OpenLambda vision paper [Hendrickson et al. 2016].

The serverless platforms can be viewed as an evolution of Platform-as-a-Service (PaaS), as provided by platforms such as Cloud Foundry [Bernstein 2014], Heroku [Middleton et al. 2013], and Google App Engine [Sanderson 2009]. The main differentiators of serverless platforms, as compared to previous PaaS, include transparent auto-scaling and fine-grained resource charging. Much previous work has addressed auto-scaling technology [Amza et al. 2005; Bunch et al. 2012; Cecchet et al. 2011; Ferraris et al. 2012; Mao et al. 2010], but the serverless model of fully-automatic instant scaling is qualitatively different in practice.

The event-driven programming model has many precedents [Etzion and Niblett 2010]. Notable previous event-driven models include actor models [Agha 1986], reactive programming [Bainomugisha et al. 2013; Wan and Hudak 2000], active database systems [Paton and Díaz 1999], and even earlier work on Kahn Process Networks [Kahn 1974]. OpenWhisk naturally expresses common event-driven programming concepts: triggers provide a unified representation of event sources, actions (and action sequences) can act as sinks, and rules enable channels.

OpenWhisk relies heavily on prior work for lightweight isolated execution environments. The current implementation exploits technologies developed for Linux Containers [Merkel 2014]. Other relevant technologies include Hyper-V [Velte and Velte 2010], and unikernel approaches, such as MirageOS [Madhavapeddy et al. 2013] or OSv [Briggs et al. 2015]. Additionally, many systems provide higher-level container management solutions, such as Kubernetes [Brewer 2015] and Mesos [Hindman et al. 2011].

8.1 Relation to Job/Workflow Scheduling

Serverless functions provide the building blocks for composing larger, more complex applications. Arguably, a natural way of expressing such application is to model functions as tasks which are coordinated via a workflow execution engine [Islam et al. 2012; Yu et al. 2008]. AWS Step Functions [Amazon 2016] is an example of composing functions as *steps*, and describing a state machine for the overall orchestration of a large application.

In relation to this paper, workflows are examples of client-side orchestration. We provide in this paper a foundation for realizing such orchestrations using the OpenWhisk reactive core. In doing so, we can explore the intriguing possibility of executing workflows in a serverless manner, where execution of the workflow engine only consumes resources when it must make a decision for the next step in the execution. This in turn eschews the costs for waiting for state transitions, and makes the workflows further composable, in keeping with good software engineering practices.

9 Conclusion

Serverless computing offers the capability to host fine-grained function invocations as a managed service. All contemporary serverless platforms, including Apache OpenWhisk, Amazon Lambda, Google Cloud Functions, and Microsoft Azure Functions, provide the ability to schedule the invocation of these hosted functions based on triggering events. Given the design point of these platforms, *i.e.* the execution of stateless functions, it is natural to ask how one might compose stateless functions into larger functional pipelines.

[4]Zimki did not survive long enough to leave much of a code or scholarly legacy, so it is hard to judge its relation to the contemporary platforms.

[5]In 2016, IBM donated the OpenWhisk project to the Apache foundation.

We have shown that the event-driven core of serverless, albeit useful and powerful, is not yet expressive enough to implement compositions of functions, *as serverless functions*. We introduced the *serverless trilemma*, which captures the inherent tension between economics, performance, and synchronous composition.

We demonstrated that the intuitive tack towards resolving the trilemma, that of a continuation-passing style of invocation, also cannot be expressed against the purely reactive core programming model that serverless platforms currently offer. We have shown how to extend the core in such a way as to implement at least the sequential composition of functions. The implementation of this extension is available in the open-source project Apache OpenWhisk.

In the future, we hope to provide proofs of the serverless trilemma, and to extend the core to handle a larger class of compositions. We also propose that more work is needed to describe the classes of expressivity in serverless. For example, ST-safe functions seems to be a subclass of composable serverless functions, which is likely a further subclass of all serverless functions.

References

Gul Agha. 1986. An Overview of Actor Languages. In *Proceedings of the 1986 SIGPLAN Workshop on Object-oriented Programming (OOPWORK'86)*. ACM, New York, NY, 58–67.

Amazon. 2008. S3 Simple Storage Service. (2008). https://aws.amazon.com/s3/

Amazon. 2016. AWS Step Functions. (2016). https://aws.amazon.com/step-functions/

Cristiana Amza, Alan L Cox, and Willy Zwaenepoel. 2005. A comparative evaluation of transparent scaling techniques for dynamic content servers. In *21st International Conference on Data Engineering (ICDE'05)*. IEEE, 230–241.

J Chris Anderson, Jan Lehnardt, and Noah Slater. 2010. *CouchDB: the definitive guide.* " O'Reilly Media, Inc.".

Apache 2016. OpenWhisk. (2016). https://github.com/openwhisk/openwhisk

Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. 2013. A Survey on Reactive Programming. *ACM Comput. Surv.* 45, 4, Article 52 (Aug. 2013), 34 pages.

David Bernstein. 2014. Cloud foundry aims to become the OpenStack of PaaS. *IEEE Cloud Computing* 2, 1 (2014), 57–60.

Christopher D. Bienko, Marina Greenstein, Stephen E. Holt, and Richard T. Phillips. 2015. *IBM Cloudant: Database as a Service Advanced Topics.* IBM Redbooks.

Eric A. Brewer. 2015. Kubernetes and the Path to Cloud Native. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC '15)*. ACM, New York, NY, USA, 167–167.

Ian Briggs, Matt Day, Eric Eide, Yuankai Guo, and Peter Marheine. 2015. *Performance Evaluation of OSv for Server Applications.* Technical Report. University of Utah.

Chris Bunch, Vaibhav Arora, Navraj Chohan, Chandra Krintz, Shashank Hegde, and Ankit Srivastava. 2012. A pluggable autoscaling service for open cloud PaaS systems. In *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing.* IEEE Computer Society, 191–194.

Emmanuel Cecchet, Rahul Singh, Upendra Sharma, and Prashant Shenoy. 2011. Dolly: virtualization-driven database provisioning for the cloud. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 51–62.

T. Cross. 2016. *AWS Lambda: The Ultimate Guide to Serverless Microservices - Learn Everything You Need to Know about Microservices Without Servers!* CreateSpace Independent Publishing Platform.

Alex Davies. 2012. *Async in C# 5.0.* " O'Reilly Media, Inc.".

Opher Etzion and Peter Niblett. 2010. *Event Processing in Action.* Manning Publications Co., Greenwich, CT.

Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. 2003. The Many Faces of Publish/Subscribe. *ACM Comput. Surv.* 35, 2 (June 2003), 114–131.

Filippo Lorenzo Ferraris, Davide Franceschelli, Mario Pio Gioiosa, Donato Lucia, Danilo Ardagna, Elisabetta Di Nitto, and Tabassum Sharif. 2012. Evaluating the auto scaling performance of Flexiscale and Amazon EC2 clouds. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2012 14th International Symposium on.* IEEE, 423–429.

Michael J Fischer, Nancy A Lynch, and Michael S Paterson. 1985. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)* 32, 2 (1985), 374–382.

Pascal Fradet and Stéphane Hong Tuan Ha. 2004. Network fusion. In *Asian Symposium on Programming Languages and Systems.* Springer, 21–40.

Google 2016. Cloud Functions. (2016). https://cloud.google.com/functions/

Philipp Haller and Heather Miller. 2013. RAY: Integrating Rx and Async for direct-style reactive streams. In *Workshop on Reactivity, Events and Modularity.*

Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Serverless Computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2016, Denver, CO, USA, June 20-21, 2016.*, Austin Clements and Tyson Condie (Eds.). USENIX Association.

John L. Hennessy and David A. Patterson. 2011. *Computer Architecture, Fifth Edition: A Quantitative Approach* (5th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center.. In *NSDI*, Vol. 11. 22–22.

IronIO. 2016. Iron Functions. (2016). https://github.com/iron-io/functions

Mohammad Islam, Angelo K Huang, Mohamed Battisha, Michelle Chiang, Santhosh Srinivasan, Craig Peters, Andreas Neumann, and Alejandro Abdelnur. 2012. Oozie: towards a scalable workflow management system for hadoop. In *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies.* ACM, 4.

Jörn W. Janneck. 2003. Actors and their Composition. *Formal Aspects of Computing* 15, 4 (2003), 349–369. DOI: https://doi.org/10.1007/s00165-003-0016-3

Neil D Jones, Carsten K Gomard, and Peter Sestoft. 1993. *Partial evaluation and automatic program generation.*

Gilles Kahn. 1974. The semantics of a simple language for parallel programming. In *Information processing*, J. L. Rosenfeld (Ed.). North Holland, Amsterdam, Stockholm, Sweden, 471–475.

J. Kreps, N. Narkhede, and J. Rao. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of 6th International Workshop on Networking Meets Databases (NetDB), Athens, Greece.*

Sheng Liang, Paul Hudak, and Mark Jones. 1995. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM, 333–343.

Bin Lin, Alexey Zagalsky, Margaret-Anne Storey, and Alexander Serebrenik. 2016. Why Developers Are Slacking Off: Understanding How Software Teams Use Slack. In *Proceedings of the 19th ACM Conference on Computer Supported Cooperative Work and Social Computing Companion.* ACM, 333–336.

Barbara H. Liskov and Jeannette M. Wing. 1994. A Behavioral Notion of Subtyping. *ACM Trans. Program. Lang. Syst.* 16, 6 (Nov. 1994), 1811–1841.

Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library operating systems for the cloud. *ACM SIGPLAN Notices* 48, 4 (2013), 461–472.

Ming Mao, Jie Li, and Marty Humphrey. 2010. Cloud auto-scaling with deadline and budget constraints. In *2010 11th IEEE/ACM International Conference on Grid Computing*. IEEE, 41–48.

Dirk Merkel. 2014. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.* 2014, 239, Article 2 (March 2014).

Mathias Meyer. 2014. Continuous integration and its tools. *IEEE software* 31, 3 (2014), 14–16.

Microsoft. 2016. Azure Functions. (2016). https://functions.azure.com/

Neil Middleton, Richard Schneeman, and others. 2013. *Heroku: Up and Running*. O'Reilly Media, Inc.

Lee Momtahan, Andrew Martin, and A. W. Roscoe. 2006. A Taxonomy of Web Services Using CSP. *Electron. Notes Theor. Comput. Sci.* 151, 2 (May 2006), 71–87.

Oscar Nierstrasz and Theo Dirk Meijler. 1995. *Requirements for a composition language*. Springer Berlin Heidelberg, Berlin, Heidelberg, 147–161. DOI: https://doi.org/10.1007/3-540-59450-7_9

Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. 2004. The Scala language specification. (2004).

Semih Okur, David L Hartveld, Danny Dig, and Arie van Deursen. 2014. A study and toolkit for asynchronous programming in C. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 1117–1127.

Norman W. Paton and Oscar Díaz. 1999. Active Database Systems. *ACM Comput. Surv.* 31, 1 (1999), 63–103.

Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 155–165.

Will Reese. 2008. Nginx: The High-performance Web Server and Reverse Proxy. *Linux J.* 2008, 173, Article 2 (Sept. 2008).

Donald Robinson. 2008. *Amazon Web Services Made Simple: Learn how Amazon EC2, S3, SimpleDB and SQS Web Services enables you to reach business goals faster*. Emereo Pty Ltd.

Dan Sanderson. 2009. *Programming Google App Engine: build and run scalable web apps on google's infrastructure*. O'Reilly Media, Inc.

Ion Stoica. 2004. *Stateless Core: A Scalable Approach for Quality of Service in the Internet: Winning Thesis of the 2001 ACM Doctoral Dissertation Competition*. Vol. 2979. Springer.

Anthony Velte and Toby Velte. 2010. *Microsoft Virtualization with Hyper-V* (1 ed.). McGraw-Hill, Inc., New York, NY, USA.

Anthony T Velte, Toby J Velte, Robert C Elsenpeter, and Robert C Elsenpeter. 2010. *Cloud computing: a practical approach*. McGraw-Hill New York.

Philip Wadler. 1990. Comprehending Monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming (LFP '90)*. ACM, New York, NY, USA, 61–78.

Zhanyong Wan and Paul Hudak. 2000. Functional reactive programming from first principles. In *Acm sigplan notices*, Vol. 35. ACM, 242–252.

Jia Yu, Rajkumar Buyya, and Kotagiri Ramamohanarao. 2008. Workflow scheduling algorithms for grid computing. *Metaheuristics for scheduling in distributed computing environments* (2008), 173–214.