SAND: Towards High-Performance Serverless Computing

Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein
Klaus Satzke, Andre Beck, Paarijaat Aditya, Volker Hilt
Nokia Bell Labs

Abstract

Serverless computing has emerged as a new cloud computing paradigm, where an application consists of individual functions that can be separately managed and executed. However, existing serverless platforms normally isolate and execute functions in separate containers, and do not exploit the interactions among functions for performance. These practices lead to high startup delays for function executions and inefficient resource usage.

This paper presents SAND, a new serverless computing system that provides lower latency, better resource efficiency and more elasticity than existing serverless platforms. To achieve these properties, SAND introduces two key techniques: 1) application-level sandboxing, and 2) a hierarchical message bus. We have implemented and deployed a complete SAND system. Our results show that SAND outperforms the state-of-the-art serverless platforms significantly. For example, in a commonly-used image processing application, SAND achieves a 43% speedup compared to Apache OpenWhisk.

1 Introduction

Serverless computing is emerging as a key paradigm in cloud computing. In serverless computing, the unit of computation is a function. When a service request is received, the serverless platform allocates an ephemeral execution environment for the associated function to handle the request. This model, also known as Function-as-a-Service (FaaS), shifts the responsibilities of dynamically managing cloud resources to the provider, allowing the developers to focus only on their application logic. It also creates an opportunity for the cloud providers to improve the efficiency of their infrastructure resources.

The serverless computing paradigm has already created a significant interest in industry and academia. There have been a number of commercial serverless offerings (e.g., Amazon Lambda [1], IBM Cloud Functions [5], Microsoft Azure Functions [11], and Google Cloud Functions [15]), as well as several new proposals (e.g., OpenLambda [24] and OpenFaaS [19]).

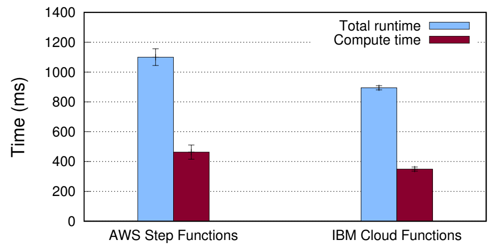


Figure 1: Total runtime and compute time of executing an image processing pipeline with four functions on existing commercial serverless platforms. Results show the mean values with 95% confidence interval over 10 runs, after discarding the initial (cold) execution.

Although existing serverless platforms work well for simple applications, they are not well-suited for more complex services due to their overheads, especially when the application logic follows an execution path spanning multiple functions. Consider an image processing pipeline that executes four consecutive functions [51]: extract image metadata, verify and transform it to a specific format, tag objects via image recognition, and produce a thumbnail. We ran this pipeline using AWS Step Functions [10] and IBM Cloud Functions with Action Sequences [29], both of which provide a method to connect multiple functions into a single service.[1] On these platforms, we found that the total runtime is significantly more than the actual time required for function executions (see Figure 1), indicating the overheads of executing such connected functions. As a result of these overheads, the use and adoption of serverless computing by a broader range of applications is severely limited.

We observe two issues that contribute to these over-

[1]As of this writing, other major serverless providers, e.g., Microsoft and Google, do not support Python, which was used for this pipeline.

heads and diminish the benefits of serverless computing. Our first observation is that most existing serverless platforms execute each application function within a separate container instance. This decision leads to two common practices, each with a drawback. First, one can start a new, 'cold' container to execute an associated function every time a new request is received, and terminate the container when the execution ends. This approach inevitably incurs long startup latency for each request. The second practice is to keep a launched container 'warm' (i.e., running idle) for some time to handle future requests. While reducing the startup latency for processing, this practice comes at a cost of occupying system resources unnecessarily for the idle period (i.e., resource inefficiency).

Our second observation is that existing serverless platforms do not appear to consider interactions among functions, such as those in a sequence or workflow, to reduce latency. These platforms often execute functions wherever required resources are available. As such, external requests (e.g., user requests calling the first function in a workflow) are treated the same as internal requests (e.g., functions initiating other functions during the workflow execution), and load is balanced over all available resources. This approach causes function interactions to pass through the full end-to-end function call path, incurring extra latency. For example, in Apache OpenWhisk [5] (i.e., the base system of IBM Cloud Functions [29]), even for functions that belong to the same workflow defined by the Action Sequences, all function calls pass through the controller. We also observe that, for functions belonging to the same state machine defined by AWS Step Functions [10], the latencies between functions executing on the same host are similar to the latencies between functions running on different hosts.

In this paper, we design and prototype a novel, high-performance serverless platform, `SAND`. Specifically, we propose two mechanisms to accomplish both low latency and high resource efficiency. First, we design a *fine-grained application sandboxing mechanism* for serverless computing. The key idea is to have two levels of isolation: 1) isolation between different applications, and 2) isolation between functions of the same application. This distinction enables `SAND` to quickly allocate (and deallocate) resources, leading to low startup latency for functions and efficient usage of cloud resources. We argue that stronger mechanisms (e.g., containers) are needed only for the isolation among applications, and weaker mechanisms (e.g., processes and light-weight contexts [37]) are well-suited for the isolation among functions within the same application.

Second, we design a *hierarchical message queuing and storage mechanism* to leverage locality for the interacting functions of the same application. Specifically, `SAND` orchestrates function executions of the same application as local as possible. We develop a local message bus on each host to create shortcuts to enable fast message passing between interacting functions, so that functions executing in a sequence can start almost instantly. In addition, for reliability, we deploy a global message bus that serves as a backup of locally produced and consumed messages in case of failures. The same hierarchy is also applied for our storage subsystem in case functions of the same application need to share data.

With these two mechanisms, `SAND` achieves low function startup and interaction latencies, as well as high resource efficiency. Low latencies are crucial for serverless computing and play a key role in broadening its use. Otherwise, application developers would merge functions to avoid the latency penalty, making the applications less modular and losing the benefits of serverless computing. In addition, the cloud industry is increasingly interested in moving infrastructure to the network edge to further reduce network latency to users [18, 26, 36, 50, 52]. This move requires high resource efficiency because the edge data centers typically have fewer resources than the core.

We implemented and deployed a complete `SAND` system. Our evaluation shows that `SAND` outperforms state-of-the-art serverless platforms such as Apache OpenWhisk [5] by $8.3\times$ in reducing the interaction latency between (empty) functions and much more between typical functions. For example, in a commonly-used image processing application, these latencies are reduced by $22\times$, leading to a 43% speedup in total runtime. In addition, `SAND` can allocate and deallocate system resources for function executions much more efficiently than existing serverless platforms. Our evaluation shows that `SAND` improves resource efficiency between $3.3\times$ and two orders of magnitude compared to the state-of-the-art.

2 Background

In this section, we give an overview of existing serverless platforms, their practices, and the implications of these practices. We summarize how these platforms deploy and invoke functions in parallel as well as in sequence. Our observations are based on open-source projects, but we think they still reflect many characteristics of commercial platforms. For example, the open-source Apache OpenWhisk is used in IBM Cloud Functions [29]. In addition, we augment these observations with our experiences using these platforms and with publicly available information from the commercial providers.

2.1 Function Deployment

In serverless computing, the cloud operator takes the responsibility of managing servers and system resources to run the functions supplied by developers. To our knowledge, the majority of serverless platforms use containers and map each function into its own container to achieve this goal. This mapping enables the function code to be portable, so that the operator can execute the function wherever there are enough resources in the infrastructure without worrying about compatibility. Containers also provide virtually isolated environments with namespaces separating operating system resources (e.g., processes, filesystem, networking), and can isolate most of the faulty or malicious code execution. Serverless platforms that employ such a mapping include commercial platforms (e.g., AWS Lambda, Microsoft Azure Functions, Google Cloud Functions, and IBM Cloud Functions) as well as open-source platforms (e.g., Apache OpenWhisk, OpenLambda [24], Greengrass [7], and OpenFaaS [19]).

Note that there are also other platforms that employ NodeJS [45] to run functions written in JavaScript [25, 49, 53, 57]. These platforms offer alternatives for serverless computing, but are not as widely used as the container-based platforms. For this reason, hereafter we describe in detail serverless platforms that employ containers and use them in our evaluation.

2.2 Function Call

The most straightforward approach to handle an incoming request is to start a new container with the associated function code and then execute it. This approach, known as 'cold' start, requires the initialization of the container with the necessary libraries, which can incur undesired startup latency to the function execution. For example, AWS Lambda has been known to have delays of up to a few seconds for 'cold' function calls [8]. Similarly, Google has reported a median startup latency of 25 seconds on its internal container platform [54], 80% of which are attributed to library installation. Lazy loading of libraries can reduce this startup latency, but it can still be on the order of a few seconds [23].

To improve startup latency, a common practice is to reuse launched containers by keeping them 'warm' for a period of time. The first call to a function is still 'cold', but subsequent calls to this function can be served by the 'warm' container to avoid undesired startup latency. To also reduce the latency of the first function call, Apache OpenWhisk can launch containers even before a request arrives via the 'pre-warming' technique [5].

The above 'warm' container practice, however, unnecessarily occupies resources with idling containers. Note that this practice also relaxes the original isolation guarantee provided by the containers, because different requests may be handled inside the same container albeit sequentially (i.e., one execution at a time).

2.3 Function Concurrency

Another aspect in which various serverless platforms can differ is how they handle concurrent requests. Apache OpenWhisk and commercial platforms such as AWS Lambda [1], Google Cloud Functions and Microsoft Azure Functions allow only one execution at a time in a container for performance isolation. As a result, concurrent requests will either be handled in their individual containers and experience undesired startup latencies for each container, or the requests will be queued for a 'warm' container to become available and experience queuing delays. In contrast, OpenFaaS [19] and OpenLambda [24] allow concurrent executions of the same function in a single container.

2.4 Function Chaining

Application logic often consists of sequences of multiple functions. Some existing serverless platforms support the execution of function sequences (e.g., IBM Action Sequences [29], AWS Step Functions [10]). In a function sequence, the events that trigger function executions can be categorized as external (e.g., a user request calling a function sequence) and internal (e.g., a function initiating other functions during the workflow execution). Existing serverless platforms normally treat these events the same, such that each event traverses the full end-to-end function call path (e.g., event passing via a unified message bus or controller), incurring undesired latencies.

3 `SAND` Key Ideas and Building Blocks

This section describes the key ideas and building blocks of `SAND`. We first present the design of our application sandboxing mechanism (§3.1) that enables `SAND` to be resource-efficient and elastic as well as to achieve low-latency function interactions. Then, we describe our hierarchical message queuing mechanism (§3.2) that further reduces the function interaction latencies.

3.1 Application Sandboxing

The key idea in our sandboxing design is that we need two levels of fault isolation: 1) isolation between different applications, and 2) isolation between functions of the same application. Our reasoning is that different applications require strong isolation from each other. On the other hand, functions of the same application may not need such a strong isolation, allowing us to improve the performance of the application. Note that some existing serverless platforms reuse a 'warm' container to execute calls to the same function, making a similar trade-off to improve the performance of a single function.

To provide a two-level fault isolation, one can choose from a variety of technologies, such as virtual machines (VMs), LightVMs [40], containers [21], unikernels [38, 39], processes, light-weight contexts [37] and threads. This choice will impact not only performance but also the dynamic nature of applications and functions as well as the maintenance effort by cloud operators. We discuss these implications in §9.

In SAND, we specifically separate applications from each other via containers, such that each application runs in its own container. The functions that compose an application run in the same container but as separate processes. Upon receiving an incoming request, SAND forks a new process in the container to execute the associated function, such that each request is handled in a separate process. For example, Figure 2 shows that the two functions f_1 and f_2 of the same application are run in the same application sandbox on a host, but different applications are separated.

Our application sandboxing mechanism has three significant advantages. First, triggering a function execution by forking a process within a container incurs short startup latency, especially compared to launching a separate container per request or function execution — up to three orders of magnitude speedup (§6.1). Second, the libraries shared by multiple functions of an application need to be loaded into the container only once. Third, the memory footprint of an application container increases in small increments with each incoming request and decreases when the request has been processed, with the resources allocated for a single function execution being released immediately (i.e., when the process terminates). As a result, the cloud operator can achieve substantially better resource efficiency and has more flexibility to divert resources not only among a single application's functions but also among different applications (i.e., no explicit pre-allocation). This effect becomes even more critical in emerging edge computing scenarios where cloud infrastructure moves towards the network edge that has only limited resources.

3.2 Hierarchical Message Queuing

Serverless platforms normally deploy a unified message bus system to provide scalable and reliable event dispatching and load balancing. Such a mechanism works well in scenarios where individual functions are triggered via (external) user requests. However, it can cause unnecessary latencies when multiple functions interact with each other, such that one function's output is the input to another function. For example, even if two functions of an application are to be executed in a sequence and they reside on the same host, the trigger message between the two functions still has to be published to the unified message bus, only to be delivered back to the same host.

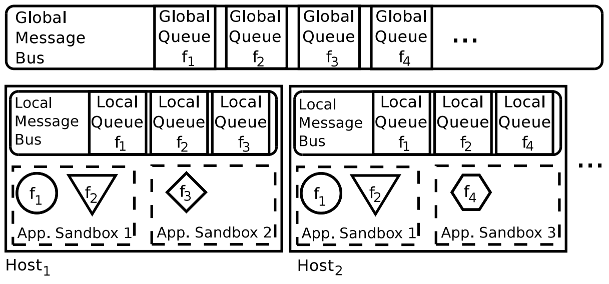


Figure 2: SAND's key building blocks: application sandboxing and hierarchical message queuing.

To address this problem, we design a *hierarchical* message bus for SAND. Our basic idea is to create shortcuts for functions that interact with each other (e.g., functions of the same application). We describe the hierarchical message bus and its coordination with two levels.[2]

In a two-level hierarchy, there is a *global* message bus that is distributed across hosts and a *local* message bus on every host (Figure 2). The global message bus serves two purposes. First, it delivers event messages to functions across different hosts, for example, when a single host does not have enough resources to execute all desired functions or an application benefits from executing its functions across multiple hosts (e.g., application sandbox 1 in Figure 2). Second, the global message bus also serves as a backup of local message buses for reliability.

The local message bus on each host is used to deliver event messages from one function to another if both functions are running on the same host. As a result, the interacting functions (e.g., an execution path of an application spanning multiple functions, similar to the application sandbox 1 in Figure 2) can benefit from reduced latency because accessing the local message bus is much faster than accessing the global message bus (§6.2).

The local message bus is first-in-first-out, preserving the order of messages from one function to another. For global message bus, the order depends on the load balancing scheme: if a shared identifier of messages (e.g., key) is used, the message order will also be preserved.

Coordination. To ensure that an event message does not get processed at the same time on multiple hosts, the local and global message buses coordinate: a backup of the locally produced event message is published to the global message bus with a *condition flag*. This flag indicates that the locally produced event message is being processed on the current host and should not be delivered to another host for processing. After publishing the backup message, the current host tracks the progress of the forked process that is handling the event message and

[2]This hierarchy can be extended to more than two levels in a large network; we omit the description due to space limit.

updates its flag value in the global message bus accordingly (i.e., 'processing' or 'finished'). If the host fails after the backup message is published to the global message bus but before the message has been fully processed, another host takes over the processing after a timeout.

This coordination procedure is similar to write-ahead-logging in databases [56], whereby a locally produced event message would be first published to the global message bus before the local message bus. While guaranteeing that there are no lost event messages due to host failures, this 'global-then-local' publication order can add additional latency to the start of the next (locally available) function in a sequence. In SAND, we relax this order, and publish event messages to the global message bus asynchronously with the publication to the local message bus in parallel. In serverless computing, functions are expected to, and usually, finish execution fast [9, 12]. In case of a failure, SAND can reproduce the lost event messages by re-executing the functions coming after the last (backup) event message seen in the global message bus. Note that, in SAND, the output of a function execution becomes available to other functions at the end of the function execution (§4.1). As such, the coordination and recovery procedures work for outputs that are contained within SAND. SAND does not guarantee the recovery of functions that make externally-visible side effects *during* their executions, such as updates to external databases.

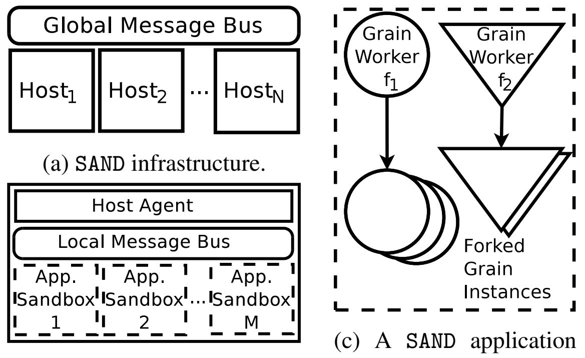
4 SAND System Design

This section presents the detailed design of SAND utilizing the aforementioned key ideas and building blocks. We also illustrate how an example application runs on SAND, and describe some additional system components.

4.1 System Components

The SAND system contains a number of hosts, which can exchange event messages via a global message bus (Figure 3a). Figure 3b shows the system components on a single host. Here, we describe these components.

Application, Grain, and Workflow. In SAND, a function of an application is called a *grain*. An *application* consists of one or multiple grains, as well as the *workflows* that define the interactions among these grains. The interaction between grains can be *static* where the grains are chained (e.g., Grain$_2$'s execution always follows Grain$_1$'s execution), or *dynamic* where the execution path is determined during the execution (e.g., the execution of Grain$_2$ and/or Grain$_3$ may follow Grain$_1$'s execution, according to Grain$_1$'s output). The grain code and workflows are supplied by the application developer. A grain can be used by multiple applications by copying it to the respective application sandboxes.



(a) SAND infrastructure.

(b) A SAND host.

(c) A SAND application sandbox with two grains.

Figure 3: High-level architecture of SAND.

Sandbox, Grain Worker, and Grain Instance. An application can run on multiple hosts. On a given host, the application has its own container called *sandbox*. The set of grains hosted in each sandbox can vary across hosts as determined by the application developer[3], but usually includes all grains of the application.

When a sandbox hosts a specific grain, it runs a dedicated OS process called *grain worker* for this grain. The grain worker loads the associated grain code and its libraries, subscribes to the grain's queue in the host's local message bus, and waits for event messages.

Upon receiving an associated event message, the grain worker forks itself to create a *grain instance* that handles the event message (Figure 3c). This mechanism provides three significant advantages for SAND. First, forking grain instances from the grain worker is quite fast and lightweight. Second, it utilizes OS mechanisms that allow the sharing of initialized code (e.g., loaded libraries), thus reducing the application's memory footprint. Third, the OS automatically reclaims the resources assigned to a grain instance upon its termination. Altogether, by exploiting the process forking, SAND becomes fast and efficient in allocating and deallocating resources for grain executions. As a result, SAND can easily handle load variations and spikes to multiplex multiple applications (and their grains) elastically even on a single host.

When a grain instance finishes handling an event message, it produces the output that includes zero or more event messages. Each such message is handled by the next grain (or grains), as defined in the workflow of the application. Specifically, if the next grain is on the same host, the previous grain instance directly publishes the output event message into the local message queue that is subscribed to by the next grain worker, which then forks a grain instance to handle this event message. In parallel, a backup of this message is asynchronously published to the global message bus.

[3]Or automatically by SAND via strategies or heuristics (e.g., a sandbox on each host should not run more than a certain number of grains).

Local and Global Message Buses. A *local message bus* runs on each host, and serves as a shortcut for local function interactions. Specifically, in the local message bus, a separate message queue is created for each grain running on this host (Figure 2). The local message bus accepts, stores and delivers event messages to the corresponding grain worker when it polls its associated queue.

On the other hand, the *global message bus* is a distributed message queuing system that runs across the cloud infrastructure. The global message bus acts as a backup for locally produced and consumed event messages by the hosts, as well as delivers event messages to the appropriate remote hosts if needed. Specifically, in the global message bus, there is an individual message queue associated with each grain hosted in the entire infrastructure (see Figure 2). Each such message queue is partitioned to increase parallelism, such that each partition can be assigned to a different host running the associated grain. For example, the widely-used distributed message bus Apache Kafka [3] follows this approach.

Each host synchronizes its progress on the consumption of the event messages from their respective partitions with the global message bus. In case of a failure, the failed host's partitions are reassigned to other hosts, which then continue consuming the event messages from the last synchronization point.

Host Agent. Each host in the infrastructure runs a special program called *host agent*. The host agent is responsible for the coordination between local and global message buses, as well as launching sandboxes for applications and spawning grain workers associated with the grains running on this host. The host agent subscribes to the message queues in the global message bus for all the grains the host is currently running. In addition, the host agent tracks the progress of the grain instances that are handling event messages, and synchronizes it with the host's partitions in the global message bus.

4.2 Workflow Example

The SAND system can be best illustrated with a simple workflow example demonstrating how a user request to a SAND application is handled. Suppose the application consists of two grains, Grain_1 and Grain_2. Host_x is running this application with the respective grain workers, GW_1 and GW_2. The global message bus has an individual message queue associated with each grain, GQ_1 and GQ_2. In addition, there is an individual partition (from the associated message queue in the global message bus) assigned to each of the two grain workers on Host_x, namely $\text{GQ}_{1,1}$ and $\text{GQ}_{2,1}$, respectively.

Assume there is a user request for Grain_1 (Step 0 in Figure 4), and the global message bus puts this event message into the partition $\text{GQ}_{1,1}$ within the global message queue GQ_1, according to a load balancing strategy. As a result, the host agent on Host_x can retrieve this event message (Step 1) and publish it into the local queue LQ_1 (associated with Grain_1) in Host_x's local message bus (Step 2). The grain worker GW_1, which is responsible for Grain_1 and subscribed to LQ_1, retrieves the recently added event message (Step 3) and forks a new grain instance (i.e., a process) to handle the message (Step 4).

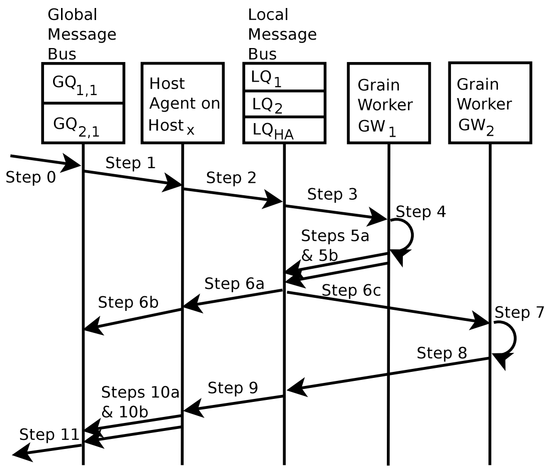


Figure 4: Handling of a user request to a simple application that consists of two grains in a workflow.

Assume Grain_1's grain instance produces a new event message for the next grain in the workflow, Grain_2. The grain instance publishes this event message directly to Grain_2's associated local queue LQ_2 (Step 5a), because it knows that Grain_2 is locally running. A copy of the event message is also published to the local queue LQ_{HA} for the host agent on Host_x (Step 5b). The host agent retrieves the message (Step 6a) and publishes it as a backup to the assigned partition $\text{GQ}_{2,1}$ in Grain_2's associated global queue with a condition flag 'processing' (Step 6b).

Meanwhile, the grain worker GW_2 for Grain_2 retrieves the event message from the local queue LQ_2 in the local message bus on Host_x (Step 6c). GW_2 forks a new grain instance, which processes the event message and terminates after execution (Step 7). In our example, GW_2 produces a new event message to the local queue of the host agent LQ_{HA} (Step 8), because Grain_2 is the last grain in the application's workflow and there are no other locally running grains to handle it. The host agent retrieves the new event message (Step 9) and directly publishes it to the global message bus (Step 10a). In addition, the finish of the grain instance of Grain_2 causes the host agent to update the condition flag of the locally produced event message that triggered Grain_2 with a value 'finished' to indicate that it has been successfully processed (Step 10b). The response is then sent to the user (Step 11).

4.2.1 Handling Host Failures

In the previous description, all hosts are alive during the course of the workflow. Suppose the processing of the event message for Grain_2 (Step 7 in Figure 4) failed due to the failure of Host_x. When the global message bus detects Host_x's failure, it will reassign Host_x's associated partitions (i.e., $\text{GQ}_{1,1}$ and $\text{GQ}_{2,1}$). Suppose there is another host Host_y taking over these two partitions. In our example, only Grain_2's grain instances were triggered via the locally produced event messages, meaning that the condition flags were published to $\text{GQ}_{2,1}$ (i.e., Host_x's partition in Grain_2's global message queue).

When Host_y starts the recovery process, it retrieves all event messages in $\text{GQ}_{2,1}$ (and also $\text{GQ}_{1,1}$). For each message, Host_y's host agent checks its condition flag. If the flag indicates that an event message has been processed successfully (i.e., 'finished'), this message is skipped because Host_x failed after processing this event message. If the flag indicates that the processing of the event message has just started (i.e., 'processing'), Host_y processes this event message following the steps in Figure 4.

It is possible that Host_y fails during the recovery process. To avoid the loss of event messages, Host_y continuously synchronizes its progress on the consumed messages from the reassigned partitions with the global message bus. It does not retrieve any new messages from the partitions until all messages of the failed host have been processed successfully. As a result, each host replacing a failed host deals with smaller reassigned partitions.

4.3 Additional System Components

Here, we briefly describe a few additional system components that complete SAND.

Frontend Server. The *frontend server* is the interface for developers to deploy their applications as well as to manage grains and workflows. It acts as the entry point to any application on SAND. For scalability, multiple frontend servers can run behind a standard load balancer.

Local and Global Data Layers. Grains can share data by passing a reference in an event message instead of passing the data itself. The *local data layer* runs on each host similar to the local message bus, and enables fast access to the data that local grains want to share among themselves via an in-memory key-value store. The *global data layer* is a distributed data storage running across the cloud infrastructure similar to the global message bus. The coordination between the local and global data layers is similar to the coordination between the local and global message buses (§3.2). Each application can only access its own data in either layer.

To ensure the data produced by a grain instance persists, it is backed up to the global data layer during the (backup) publication of the locally produced event message. This backup is facilitated with another flag value (between 'processing' and 'finished' described in §3.2) to indicate the start of the data transfer to the global data layer. This value contains the data's metadata (i.e., name, size, hash), which is checked during the recovery process to decide whether an event message needs to be processed: if the metadata in the flag matches the metadata of the actual data in the global data layer, the event message was successfully processed but the 'finished' flag could not be published by the failed host; otherwise, this event message needs to be processed again.

5 Implementation

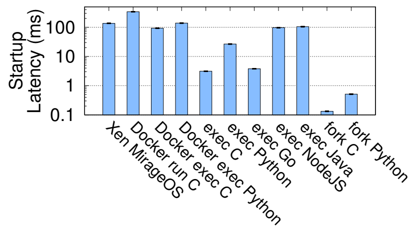
We implemented a complete SAND system with all components described in §4. Our system uses Docker [17] for application sandboxes, Apache Kafka [3] for the global message bus, Apache Cassandra [2] for the global data layer, and nginx [44] as the load balancer for the frontend server instances. We use these components off-the-shelf.

In addition, we implemented the host agent (7,273 lines of Java), the Python grain worker (732 lines of Python) and the frontend server (461 lines of Java). The host agent coordinates the local and global message buses and data layers, as well as manages application sandboxes and grains, by interacting with Kafka, Cassandra and Docker. The grain worker becomes dedicated to a specific grain after loading its code and necessary libraries, interacts with the local message bus, and forks grain instances for each associated event message. We use Apache Thrift [6] to automatically generate the interfaces for our Java implementations of the local message bus and the local data layer. The frontend server accepts connections handed over by the nginx load balancer, interacts with Kafka to deliver user requests into SAND and return application responses back to users. The frontend server embeds Jetty [32] as the HTTP endpoint and employs its thread pool to handle user requests efficiently.

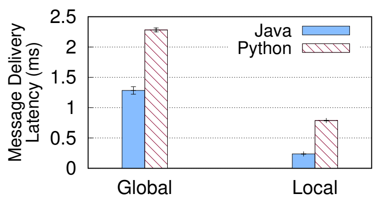
For easy development and testing, we also implemented a SAND emulator (764 lines of Python) that supports SAND's API and logging for debugging. Developers can write their grains and workflows, and test them using the emulator before the actual deployment.

6 Evaluation

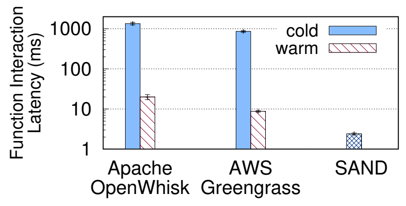
We evaluate SAND and compare it to Apache OpenWhisk [5] and AWS Greengrass [7]. We choose these two systems because we can run local installations for a fair comparison. We first report on microbenchmarks of alternative sandboxing mechanisms and SAND's hierarchical message bus. We then evaluate function interaction



(a) Function startup latencies.



(b) Message delivery latencies.



(c) Python function interaction latencies.

Figure 5: Measurements regarding function startup latencies, message delivery latencies and Python function interaction latencies, with error bars showing the 95% confidence interval.

latencies and the memory footprints of function executions, as well as investigate the trade-off between allocated memory and latency. Finally, we revisit the image processing pipeline, which was discussed as a motivational example in §1. We conducted all experiments on machines equipped with Intel Xeon E5520 with 16 cores at 2.27GHz and 48GB RAM, unless otherwise noted.

6.1 Sandboxing and Startup Latency

There are several alternative sandboxing mechanisms to isolate the applications and function executions (see §9). Here, we explore the startup latency of these alternatives.
Methodology. We measured the startup time until a function starts executing, with various sandboxing mechanisms including Docker (CE-17.11 with runc 1.0.0) and unikernel (Xen 4.8.1 with MirageOS 2.9.1), as well as spawning processes in C (gcc 4.8.5), Go (1.8.3), Python (2.7.13), NodeJS (6.12) and Java (1.8.0.151). We used an Intel Xeon E5-2609 host with 4 cores at 2.40GHz and 32GB RAM running CentOS 7.4 (kernel 4.9.63).
Results. Figure 5a shows the mean startup latencies with 95% confidence interval. Starting a process in a running, warm container via the Docker client interface (Docker exec C) is much faster than launching both the container and the process (Docker run C). Nonetheless, Docker adds significant overhead to function starts compared to starts without it (exec C, exec Python). Function starts with a unikernel (Xen MirageOS) are similar to using a container. Not surprisingly, spawning processes with binaries (exec C, exec Go) are faster than interpreted languages (exec Python, exec NodeJS) and Java, and forking processes (fork C, fork Python) is fastest among all.

6.2 Hierarchical Message Bus

Instead of a single unified message bus, SAND utilizes a local message bus on every host for fast function interactions. Here, we show the benefits of this approach.
Methodology. We created two processes on the same host that communicate in a producer-consumer style under load-free conditions. With Python and Java clients, we measured the latency for a message delivered via the global message bus (Kafka 0.10.1.0, 3 hosts, 3 replicas, default settings) and via our local message bus.
Results. Figure 5b shows the mean message delivery latencies with 95% confidence interval. The Python client (used by our grain instances) can deliver an event message to the next grain via the local message bus 2.90× faster than via the global message bus. Similarly, the Java client (used by our host agent) gets a 5.42× speedup.

6.3 Function Interaction Latency

Given two functions in a workflow, the function interaction latency is the time between the first function's finish and the second function's start.
Methodology. We created two Python functions, F_1 and F_2, such that F_1 produces an event message for F_2 to consume. We logged high-resolution timestamps at the end of F_1 and at the start of F_2. We used an Action Sequence in OpenWhisk [48], matching MQTT topic subscriptions in Greengrass [58] and SAND's workflow description. We then triggered F_1 with a request generator.

Recall that SAND uses a single, running container for multiple functions of an application. For a fair comparison, we measure function interaction latencies in OpenWhisk and Greengrass with warm containers. For completeness, we also report their cold call latencies, where a function call causes a new container to be launched.
Results. Figure 5c shows that SAND incurs a significantly shorter (Python) function interaction latency. SAND outperforms OpenWhisk and Greengrass, both with warm containers, by 8.32× and 3.64×, respectively. Furthermore, SAND's speedups are 562× and 358× compared to OpenWhisk and Greengrass with cold containers.

6.4 Memory Footprint

Concurrent calls to a function on today's serverless computing platforms are handled by concurrent execution instances. These instances are served either by launching

Table 1: Workloads and burst parameters.

Load	Rate (calls/min)	Duration (s)	Frequency (s)
A	1,000	8	240
B	250	8	240
C	1,000	30	120
D	1,000	8	120
E	250	30	120

new containers or by assigning them to warm containers if available. Because concurrently-running containers occupy system resources, we examine the memory footprint of such concurrent function executions.

Methodology. We made up to 50 concurrent calls to a single Python function. We ensured that all calls were served in parallel, and measured each platform's memory usage via `docker stats` and `ps` commands.

Results. We find that both OpenWhisk and Greengrass show a linear increase in memory footprint with the number of concurrent calls.[4] Each call adds to the memory footprint about 14.61MB and 13.96MB in OpenWhisk and Greengrass, respectively. In SAND, each call only adds 1.1MB on top of the 16.35MB consumed by the grain worker. This difference is because SAND forks a new process inside the same sandbox for each function call, whereas OpenWhisk and Greengrass use separate containers for concurrent calls.

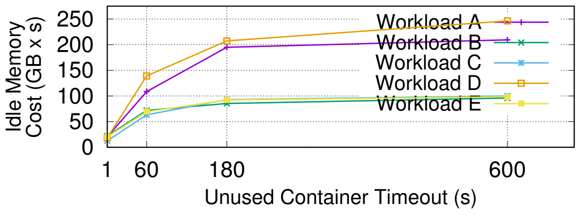
6.5 Idle Memory Cost vs. Latency

Many serverless platforms use warm containers to prevent cold startup penalties for subsequent calls to a function. On the other hand, these platforms launch new containers when there are concurrent calls to a function but no warm containers available (§6.4). These new containers will also be kept warm until a timeout, occupying resources. Here, we investigate the trade-off between occupied memory and function call latency.

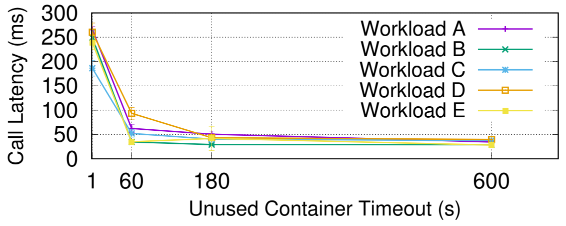
Methodology. We created 5 synthetic workloads each with 2,000 function calls. In all workloads, the call arrival time and the function processing time (i.e., busy wait) follow a Poisson distribution with a mean rate of 100 calls per minute and a mean duration of 600ms. To see how the serverless platforms behave under burst, we varied three parameters as shown in Table 1: 1) burst rate, 2) burst duration, and 3) burst frequency.

We explored 4 different unused-container timeouts in OpenWhisk. Unfortunately, this timeout cannot be modified in Greengrass, so we could not use it in this experiment. We computed the idle memory cost by multiplying the allocated but unused memory of the container instances with the duration of their allocations (i.e., total idle memory during the experiment). We reused the memory footprint results from §6.4. In OpenWhisk, the number of container instances depends on the concurrency, whereas SAND uses a single application sandbox.

[4] In Greengrass, this relationship continues until 25 concurrent calls, after which calls get queued as shown in system logs.



(a) Idle memory cost with different container timeouts.



(b) Call latency with different container timeouts.

Figure 6: Idle memory cost vs. function call latency.

Results. Figures 6a and 6b show the effects of container timeouts in OpenWhisk on the idle memory cost and the function call latency, respectively. We observe that a long timeout is not suited for bursty traffic, with additional containers created to handle concurrent calls in a burst but are not needed afterwards. Even with a relatively short timeout of 180 seconds, the high idle memory costs suggest that containers occupy system resources without using them during the majority of our experiment. We also observe that a shorter timeout lowers the idle memory cost but leads to much longer function call latencies due to the cold start effect, affecting between 18.15%–33.35% of all calls in all workloads with a 1 second timeout. Interestingly, the frequent cold starts cause OpenWhisk to overestimate the number of required containers, partially offsetting the lowered idle memory cost achieved by shorter timeouts.

In contrast, SAND reduces idle memory cost from 3.32× up to two orders of magnitude with all workloads without sacrificing low latency (15.87–16.29 ms). SAND, by its sandboxing mechanism, handles concurrent calls to a function (or multiple functions) on a single host by forking parallel processes inside a container; therefore, SAND does not suffer from cold startup penalties. With higher load, SAND would amortize the penalty of starting a new sandbox on another host by using it both for multiple executions of a single function and for different functions. Our ongoing work includes intelligent monitoring and scheduling for additional sandboxes.

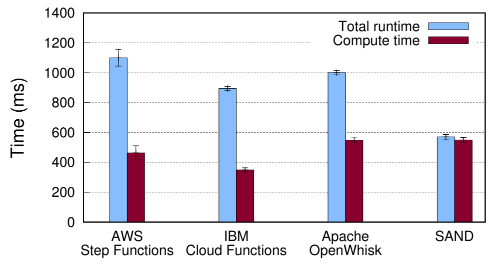


Figure 7: Total runtime and compute time of the image processing pipeline. Results show the mean values with 95% confidence interval over 10 runs, after discarding the initial (cold) execution.

6.6 Image Processing Pipeline

Here, we revisit our motivational example used in §1, i.e., the image processing pipeline. This pipeline consists of four consecutive Python functions, and is similar to the reference architecture used by AWS Lambda [51]. It first extracts the metadata of an image using ImageMagick [30]. A subset of this metadata is then verified and retained by the next function in the pipeline. The third function recognizes objects in the image using the SqueezeNet deep learning model [28] executed on top of the MXNet framework [4, 43]. Names of the recognized objects are appended to the extracted metadata and passed to the final function, which generates a thumbnail of the image and stores the metadata in a separate file.

Methodology. Each function recorded timestamps at the start and end of its execution, which we used to produce the actual compute time. The difference between the total time and the compute time gave each platform's overhead. The image was always read from a temporary local storage associated with each function call. We ran the pipeline on AWS Step Functions, IBM Cloud Functions, Apache OpenWhisk with Action Sequences, and SAND.

Results. Figure 7 shows the runtime breakdown of these platforms. Compared to other platforms, we find that SAND achieves the lowest overhead for running a series of functions. For example, SAND reduces the overhead by 22.0× compared to OpenWhisk, even after removing the time spent on extra functionality not supported in SAND yet (e.g., authentication and authorization). We notice that OpenWhisk re-launches the Python interpreter for each function invocation, so that libraries are loaded before a request can be handled. In contrast, SAND's grain worker loads a function's libraries only once, which are then shared across forked grain instances handling the requests. The difference in compute times can be explained by the difference across infrastructures: SAND and OpenWhisk ran in our local infrastructure and produced similar values, whereas we had no control over AWS Step Functions and IBM Cloud Functions.

7 Experience with SAND

During and after the implementation of SAND, we also developed and deployed several applications on it. Here, we briefly describe these applications to show that SAND is general and can serve different types of applications.

The first application we developed is a simple web server serving static content (e.g., html, javascript, images) via two grains in a workflow. The first grain parses user requests and triggers another grain according to the requested file type. The second grain retrieves the file and returns it to our frontend server, which forwards it to the user. Our SAND web server has been in use since May 2017 to serve our lab's website. The second SAND application is the management service of our SAND system. The service has 19 grains, connects with the GUI we developed, and enables developers to create grains as well as to deploy and test workflows.

In addition, we made SAND available to researchers in our lab. They developed and deployed applications using the GUI and the management service. One group prototyped a simple virus scanner, whereby multiple grains executing in parallel check the presence of viruses in an email. Another group developed a stream analytics application for Twitter feeds, where grains in a workflow identify language, remove links and stop words, and compile a word frequency list to track the latest news trend.

8 Related Work

Despite its recency, serverless computing has already been used in various scenarios including Internet of Things and edge computing [7, 16], parallel data processing [33, 34], data management [14], system security enhancement [13], and low-latency video processing [20]. Villamizar et al. [55] showed that running applications in a serverless architecture is more cost efficient than microservices or monoliths. One can expect that serverless computing is going to attract more attention.

Beside commercial serverless platforms [1, 11, 15, 25, 29, 31], there have also been academic proposals for serverless computing. Hendrickson et al. [24] proposed OpenLambda after identifying problems in AWS Lambda [1], including long function startup latency and little locality consideration. McGrath et al. [42] also investigated latencies in existing serverless frameworks. These problems are important for serverless application development, where function interaction latencies are crucial. In SAND, we address these problems via our application sandboxing approach, as well as the hierarchical message queuing and storage mechanisms.

Other approaches also targeted the long startup latency problem. Slacker [23] identifies packages that are critical when launching a container. By prioritizing these packages and lazily loading others, it can reduce the container startup latency. This improvement would benefit serverless computing platforms that launch functions with cold starts. In SAND, an application sandbox is launched once per host for multiple functions of the same application, which amortizes a container's startup latency over time.

Pipsqueak [46] and its follow-up work SOCK [47] create a cache of pre-warmed Python interpreters, so that functions can be launched with an interpreter that has already loaded the necessary libraries. However, many functions may need the same (or a similar) interpreter, requiring mechanisms to pick the most appropriate one and to manage the cache. SAND does not use any sharing nor cache management schemes; a SAND grain worker is a dedicated process for a single function and its libraries.

McGrath et al. [41] proposed a queuing scheme with workers announcing their availability in warm and cold queues, where containers can be reused and new containers can be created, respectively. Unlike SAND, this scheme maps a single container per function execution.

9 Discussion & Limitations

Performance Isolation & Load Balancing. In this paper, we reduce function interaction latencies via our sandboxing mechanism as well as the hierarchical message queuing and storage. SAND executes multiple instances of an application's functions in parallel as separate processes in the same container. This sandboxing mechanism enables a cloud operator to run many functions (and applications) even on a single host, with low idle memory cost and high resource efficiency. However, it is possible that grains in a sandbox compete for the same resources and interfere with each other's performance. A single host may also not have the necessary resources for multiple sandboxes. In addition, SAND's locality-optimized policy with the hierarchical queuing and storage might lead to sub-optimal load balancing.

SAND currently relies on the operating system to ensure that the grains (and sandboxes) running in parallel will receive their fair share of resources. As such, CPU time (or other resource consumption) rather than the wall clock time could be used for billing purposes. Nevertheless, competing grains and sandboxes may increase the latency an application experiences.

Non-fork Runtime Support. SAND makes a trade-off to balance performance and isolation by using process forking for function executions. The downside is that SAND currently does not support language runtimes without native forking (e.g., Java and NodeJS).

Alternative Sandboxing Mechanisms. SAND isolates applications with containers. Virtual machines (VMs), HyperContainers [27] and gVisor [21] are viable alternatives. VMs provide a stronger isolation than containers, but may increase the maintenance effort for each application's custom VM and have long launch times. Unikernels [38, 39] can also be used to isolate applications with custom system software compiled with the desired functionality. However, dynamically adding/removing a function requires a recompilation, affecting the flexibility of function assignment to a host. In contrast, containers provide fast launch times, flexibility to dynamically assign functions, and low maintenance effort because of the OS being shared among all application containers on the host. Recently open-sourced gVisor [22] provides a stronger fault isolation than vanilla containers.

For function executions, SAND uses separate processes. Unikernels [35, 38, 39] have also been proposed to isolate individual functions in serverless environments. A bare-bones unikernel-based VM (e.g., LightVM [40]) can launch faster than a container to execute a function; however, its image size depends on the libraries loaded by each function, and thus, may impact startup latency. Other alternatives include light-weight contexts (LWCs) [37] and threads. Particularly, LWCs may provide the best of both worlds by being lighter than processes, but achieving stronger isolation than threads by giving a separate view of resources to each LWC. We plan to extend SAND with these alternative approaches.

10 Conclusion & Future Work

This paper introduced SAND, a novel serverless computing platform. We presented the design and implementation of SAND, as well as our experience in building and deploying serverless applications on it. SAND employs a new sandboxing approach, whereby stronger mechanisms such as containers are used to isolate different applications and lighter OS concepts such as processes are used to isolate functions of the same application. This approach enables SAND to allocate and deallocate resources for function executions much faster and more resource-efficiently than existing serverless platforms. Combined with our hierarchical message bus, where each host runs a local message bus to enable fast triggering of functions running on the same host, SAND reduces function interaction latencies significantly.

For future work, we plan to address the limitations discussed in §9. In particular, we plan to intelligently distribute application functions and sandboxes across many hosts to better balance the system load without sacrificing application latency.

11 Acknowledgments

We are grateful to the anonymous reviewers and our shepherd, Patrick Stuedi, for their insightful comments.

References

[1] Amazon. AWS Lambda - Serverless Compute. `https://aws.amazon.com/lambda/`.

[2] Apache Cassandra. `https://cassandra.apache.org/`.

[3] Apache Kafka. `https://kafka.apache.org/`.

[4] MXNet: A Scalable Deep Learning Framework. `http://mxnet.incubator.apache.org/`.

[5] Apache OpenWhisk is a serverless, open source cloud platform. `http://openwhisk.apache.org/`.

[6] Apache Thrift. `https://thrift.apache.org/`.

[7] AWS Greengrass. `https://aws.amazon.com/greengrass/`.

[8] AWS Lambda for Java Quodlibet Medium. `https://medium.com/@quodlibet_be/aws-lambda-for-java-5d5e954d3bdf`.

[9] AWS Lambda Limits - AWS Lambda. `https://docs.aws.amazon.com/lambda/latest/dg/limits.html`.

[10] What is AWS Step Functions? `http://docs.aws.amazon.com/step-functions/latest/dg/welcome.html`.

[11] Azure FunctionsServerless Architecture — Microsoft Azure. `https://azure.microsoft.com/en-us/services/functions/`.

[12] Best Practices for Azure Functions — Microsoft Docs. `https://docs.microsoft.com/en-us/azure/azure-functions/functions-best-practices`.

[13] Bila, N., Dettori, P., Kanso, A., Watanabe, Y., and Youssef, A. Leveraging the serverless architecture for securing linux containers. In *1st International Workshop on Serverless Computing* (2017), pp. 401–404.

[14] Chard, R., Chard, K., Alt, J., Parkinson, D. Y., Tuecke, S., and Foster, I. T. Ripple: Home automation for research data management. In *1st International Workshop on Serverless Computing* (2017), pp. 389–394.

[15] Cloud Functions - Serverless Environment to Build and Connect Cloud Services — Google Cloud Platform. `https://cloud.google.com/functions/`.

[16] de Lara, E., Gomes, C. S., Langridge, S., Mortazavi, S. H., and Roodi, M. Hierarchical serverless computing for the mobile edge. In *IEEE/ACM Symposium on Edge Computing* (2016).

[17] Docker - Build, Ship and Run Any App, Anywhere. `https://www.docker.com/`.

[18] Why Edge Computing Market Will Grow 30 Percent by 2022. `http://www.eweek.com/networking/why-edge-computing-market-will-grow-30-percent-by-2022`.

[19] Ellis, A. Functions as a Service (FaaS). `https://blog.alexellis.io/functions-as-a-service/`, 2017.

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

[21] Google/gVisor: Container Runtime Sandbox. `https://github.com/google/gvisor`.

[22] Google open sources gVisor, a sandboxed container runtime. `https://techcrunch.com/2018/05/02/google-open-sources-gvisor-a-sandboxed-container-runtime/`.

[23] Harter, T., Salmon, B., Liu, R., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Slacker: Fast distribution with lazy docker containers. In *14th USENIX Conference on File and Storage Technologies (FAST 16)* (2016).

[24] Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Serverless computation with openlambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)* (2016).

[25] hook.io. `https://hook.io/`.

[26] Hu, Y. C., Patel, M., Sabella, D., Sprecher, N., and Young, V. Mobile edge computinga key technology towards 5g. *ETSI white paper 11*, 11 (2015), 1–16.

[27] Hyper: Make VM run like Container. `https://hypercontainer.io/`.

[28] Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., and Keutzer, K. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size. *arXiv:1602.07360* (2016).

[29] Cloud Functions - Overview — IBM Cloud. `https://www.ibm.com/cloud/functions`.

[30] Convert, Edit, Or Compose Bitmap Images @ ImageMagick. `https://www.imagemagick.org/`.

[31] Iron.io - DevOps Solutions from Startups to Enterprise. `https://www.iron.io/`.

[32] Jetty - Servlet Engine and Http Server. `http://www.eclipse.org/jetty/`.

[33] Jonas, E. Microservices and Teraflops. `http://ericjonas.com/pywren.html`.

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

[35] Koller, R., and Williams, D. Will serverless end the dominance of linux in the cloud? In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems* (2017), HotOS '17.

[36] Introducing Lambda@Edge in Preview Run Lambda functions at AWSs edge locations closest to your users. `https://aws.amazon.com/about-aws/whats-new/2016/12/introducing-lambda-at-edge-in-preview-run-lambda-function-at-aws-edge-locations-closest-to-your-users/`.

[37] Litton, J., Vahldiek-Oberwagner, A., Elnikety, E., Garg, D., Bhattacharjee, B., and Druschel, P. Light-weight contexts: an os abstraction for safety and performance. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016).

[38] Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., and Crowcroft, J. Unikernels: Library operating systems for the cloud. In *ASPLOS* (2013).

[39] Madhavapeddy, A., and Scott, D. J. Unikernels: the rise of the virtual library operating system. *Communications of the ACM* (2014).

[40] Manco, F., Lupu, C., Schmidt, F., Mendes, J., Kuenzer, S., Sati, S., Yasukata, K., Raiciu, C., and Huici, F. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), SOSP '17.

[41] McGrath, G., and Brenner, P. R. Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)* (2017).

[42] McGrath, G., Short, J., Ennis, S., Judson, B., and Brenner, P. Cloud event programming paradigms: Applications and analysis. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)* (2016).

[43] GitHub - awslabs/mxnet-lambda: Reference Lambda function that predicts image labels for a image using an MXNet-built deep learning model. The repo also has pre-built MXNet, OpenCV libraries for use with AWS Lambda. `https://github.com/awslabs/mxnet-lambda`.

[44] nginx news. `https://nginx.org/`.

[45] Node.js. `https://nodejs.org/en/`.

[46] Oakes, E., Yang, L., Houck, K., Harter, T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Pipsqueak: Lean lambdas with large libraries. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)* (2017).

[47] Oakes, E., Yang, L., Zhou, D., Houck, K., Harter, T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (2018).

[48] OpenWhisk Action Sequences - Create and invoke Actions. `https://console.bluemix.net/docs/openwhisk/openwhisk_actions.html#openwhisk_create_action_sequence`.

[49] PubNub: Making Realtime Innovation Simple. `https://www.pubnub.com/`.

[50] Satyanarayanan, M. The emergence of edge computing. *Computer 50*, 1 (2017), 30–39.

[51] GitHub - awslabs/lambda-refarch-imagerecognition: The Image Recognition and Processing Backend reference architecture demonstrates how to use AWS Step Functions to orchestrate a serverless processing workflow using AWS Lambda, Amazon S3, Amazon DynamoDB and Amazon Rekognition. `https://github.com/awslabs/lambda-refarch-imagerecognition/`.

[52] Shi, W., and Dustdar, S. The promise of edge computing. *Computer 49*, 5 (2016), 78–81.

[53] UnitCluster - Make and run scripts in the cloud. `https://unitcluster.com/`.

[54] Verma, A., Pedrosa, L., Korupolu, M. R., Oppenheimer, D., Tune, E., and Wilkes, J. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), EuroSys '15.

[55] Villamizar, M., Garces, O., Ochoa, L., Castro, H. E., Salamanca, L., Verano, M., Casallas, R., Gil, S., Valencia, C., Zambrano, A., and Lang, M. Infrastructure cost comparison of running web applications in the cloud using AWS lambda and monolithic and microservice architectures. In *CCGrid* (2016), pp. 179–182.

[56] Write-ahead logging. `https://en.wikipedia.org/wiki/Write-ahead_logging`.

[57] Webtask. `https://webtask.io/`.

[58] What is AWS Greengrass? `https://docs.aws.amazon.com/greengrass/latest/developerguide/what-is-gg.html`.