Characterizing Serverless Platforms with ServerlessBench

Tianyi Yu
Shanghai Jiao Tong University
yutianyi@sjtu.edu.cn

Qingyuan Liu
Shanghai Jiao Tong University
lqyuan980413@sjtu.edu.cn

Dong Du
Shanghai Jiao Tong University
Dd_nirvana@sjtu.edu.cn

Yubin Xia
Shanghai Jiao Tong University
xiayubin@sjtu.edu.cn

Binyu Zang
Shanghai Jiao Tong University
byzang@sjtu.edu.cn

Ziqian Lu
Ant Financial
ziqian.lzq@antfin.com

Pingchao Yang
Ant Financial
pingchao.ypc@alibaba-inc.com

Chenggang Qin
Ant Financial
chenggang.qcg@alibaba-inc.com

Haibo Chen
Shanghai Jiao Tong University
haibochen@sjtu.edu.cn

ABSTRACT

Serverless computing promises auto-scalability and cost-efficiency (in "pay-as-you-go" manner) for high-productive software development. Because of its virtue, serverless computing has motivated increasingly new applications and services in the cloud. This, however, also presents new challenges including how to efficiently design high-performance serverless platforms and how to efficiently program on the platforms.

This paper proposes ServerlessBench, an open-source benchmark suite for characterizing serverless platforms. It includes test cases exploring characteristic metrics of serverless computing, e.g., communication efficiency, startup latency, stateless overhead, and performance isolation. We have applied the benchmark suite to evaluate the most popular serverless computing platforms, including AWS Lambda, OpenWhisk, and Fn, and present new serverless implications from the study. For example, we show scenarios where decoupling an application into a composition of serverless functions can be beneficial in cost-saving and performance, and that the "stateless" property in serverless computing can hurt the execution performance of serverless functions. These implications form several design guidelines, which may help platform designers to optimize serverless platforms and application developers to design their functions best fit to the platforms.

SoCC '20, October 19–21, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8137-6/20/10…$15.00
https://doi.org/10.1145/3419111.3421280

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; • **Software and its engineering** → **Cloud computing**.

ACM Reference Format:
Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. 2020. Characterizing Serverless Platforms with ServerlessBench. In *ACM Symposium on Cloud Computing (SoCC '20), October 19–21, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3419111.3421280

1 INTRODUCTION

In recent years, serverless computing becomes the new trending paradigm in cloud computing. Many public cloud platforms have been providing serverless computing services, including AWS Lambda [5], IBM Cloud Function [4], Microsoft Azure Functions [10], Google Cloud Functions [15], and more private cloud platforms [52]. Users favor serverless computing for three reasons. First, it helps developers focus on the core application logic, as the serverless platforms support the infrastructure-related properties (e.g., auto-scalability) and take over the server management. Second, serverless users can save costs with the pay-as-you-go model in serverless computing, i.e., serverless functions will only run and charge when there are requests. Third, designing an application with multiple serverless functions provides modularity benefits. Using serverless computing also benefits the cloud providers, since they can manage their resources more efficiently [1].

Despite the newfound popularity of serverless computing, designing an efficient, scalable, and user-friendly serverless

platform remains challenging for system software designers. The challenge arises because serverless computing represents a significant departure from the way cloud platforms are traditionally designed. For example, startup latency is usually not a critical factor in prior cloud platforms, since most applications are long-running and do not frequently restart. However, in serverless computing, the platform creates a new function instance for each incoming request if there are no idle ones to reuse; therefore, function startup affects the end-to-end latency of each request and thus affects the user-experience [26, 34, 38, 40, 43]. Besides, serverless application developers face challenges on how to architect a serverless application that is both performant and cost-efficient. The real-world use cases for serverless computing that the developers can refer to are limited, especially for complex applications that consist of cooperating serverless functions. We examine AWS serverless application repository [6], but the applications in the repository are mainly simple and demonstrative serverless functions that are meant to be used as a tool or a supportive component in a real-world system. Therefore, the impact of serverless computing model is still unclear for sophisticated use of serverless computing, and the design guidelines for complex serverless systems are sorely needed.

Therefore, a benchmark suite that can reveal the critical metrics of serverless computing and characterize serverless platforms is significant and necessary for both serverless platform designers and serverless application developers. Unfortunately, such a benchmark suite for serverless computing is still missing. The most related work is DeathStar [32], an open-source microservice benchmark, which does not consider some essential metrics in serverless platforms, e.g., the startup latency, auto-scalability, and overheads caused by stateless execution. Other works [34, 36] present tests to show the benefits of serverless computing over traditional paradigms. However, these tests fall short in exploring new implications to guide serverless platform optimization or serverless application design.

This paper proposes ServerlessBench, a general and open-source benchmark suite for serverless computing that helps system developers design and evaluate their serverless systems and provides useful hints for application developers to architect their serverless applications. ServerlessBench has three key distinctions compared with prior benchmark suites. First, it identifies critical metrics in serverless computing ranging from communication latency and startup latency to performance isolation. Second, it provides a set of test cases customized to evaluate the critical metrics, including real-world serverless workloads that cover popular cloud scenarios, e.g., image processing, Alexa skill application [2], online compiling system [31] and data analysis. Third, ServerlessBench is general to use in different kinds of serverless platforms, e.g., container-based systems [4], virtual machine-based systems [53], or recently proposed lightweight virtualization systems [12, 16, 25, 41, 42, 47, 54].

We conduct the evaluation on a commercial serverless platform (AWS Lambda [5]), two open-source serverless platforms (OpenWhisk [4] and Fn [13]), and one in-production private cloud (Ant Financial). The paper then presents the discovered implications on characteristic metrics of serverless computing. For example, we present scenarios where decoupling an application into a composition of serverless functions can be beneficial in both cost-saving and performance (**Implication I** and **Implication II**). We call the serverless platform designers' attention to the implicit states (e.g., Just-in-Time profile) that are typically lost across requests due to the "stateless" property in serverless computing, as we discover that their absence might introduce performance penalty to the execution of serverless functions (**Implication X**). The implications can guide the design of serverless platforms and applications. We present a case study to optimize serverless startup with inspiration from the implications. ServerlessBench is open-sourced at https://github.com/SJTU-IPADS/ServerlessBench.

The main contributions of this paper are as follows:

- A detailed analysis of critical metrics for serverless computing systems.
- A pertinent benchmark suite with customized test cases targeting different aspects of serverless computing.
- An evaluation of existing serverless platforms using ServerlessBench and characteristics of different serverless platforms found from the evaluation.
- Implications on serverless computing that can guide the design of serverless platforms and serverless applications.

2 BACKGROUND

Serverless platform. In serverless computing, the computation unit is a *function*. Application developers send their functions to a serverless platform, which provides sandboxed execution environments (containers [11, 44] or virtual machines [12, 16, 17]) for the functions to run. The platform compiles the serverless functions offline together with a runtime, then initializes the environment and invokes the handler function when requests arrive. A requirement for the function is being *stateless*—a function should not rely on states across requests. One reason for being stateless is to implement auto-scalability, as a serverless platform is responsible for scaling the instances of functions according to the request traffic.

Function startup. In existing serverless platforms, a function instance can handle a request with either *cold start* or *warm start*, depending on whether there are available idle sandboxes. The first execution of a function always begins with a cold

start, which needs to prepare function image (e.g., Docker image), create sandboxes (e.g., Docker container), and load function codes. The cold start usually causes long latency. When a function instance finishes execution, the sandbox might pause for a specified period, so that it can serve the subsequent requests for the same function by unpausing, i.e., warm start.

Function compositions. Serverless application developers tend to decouple a complex application into a composition of loosely-coupled serverless functions in pursuit of fine-grained scalability and modular development and deployment. There are two function composition types: *sequence function chain* and *nested function chain*. Typically, a platform-provided or third-party coordinator (e.g., IBM Action Sequences [20] and AWS Step Functions [7]) is responsible for conducting a sequence function chain. Each function in a sequence chain finishes their work by transferring the result data to the next function in the sequence. In the nested chain, a function invokes a callee function and waits for the callee's results before it can return. These composition methods provide different flexibility and performance characteristics, thus should be carefully chosen according to the application needs.

Serverless billing. The pay-as-you-go manner billing in serverless computing is a significantly attractive factor for serverless users. The execution bill (the major cost in serverless computing) can be calculated by *C * Time * Resource*, in which *C* is a platform-specific constant, *Time* is the function execution time with millisecond-level granularity (100ms in AWS Lambda [5], IBM Cloud Function [4], and Google Cloud Functions [15], 1ms in Azure Functions [10]), *Resource* typically represents the memory and CPU resources provisioned to the function (in rarer cases such as Azure Functions it represents the average memory usage). Existing serverless platforms, including AWS Lambda, Google Cloud Functions and IBM Cloud Functions, usually allocate CPU computing resources in proportion to the provisioned memory size for a function.

3 OVERVIEW

This section illustrates the methodology that we adopt to evaluate serverless platforms and reveal implications in serverless computing. We first present four distinguishing metrics in serverless computing (§3.1). Based on the metrics, we construct ServerlessBench (Table 1), a pertinent benchmark suite for evaluating serverless computing systems. We carefully implement the applications in ServerlessBench with a variety of programming languages, resource needs, and composition patterns, to reflect the diversity in existing serverless applications, and thus enhance the representativeness of ServerlessBench. ServerlessBench can characterize serverless platforms and help expose defects in different subsystems (e.g., CPU, memory, OS, etc.). We use ServerlessBench to evaluate existing serverless platforms with the evaluation environment explained in §3.2.

3.1 Serverless Metrics

We now analyze four distinguishing metrics in serverless computing.

Metric1: Communication performance. A complex serverless application is typically composed of several interacting functions and other cloud services. There are different function-interacting models to initiate the inter-function communication, such as using a function coordinator (sequence chain model) or using the serverless provider's SDK (nested chain model).

Metric2: Startup latency. The startup overhead is a unique challenge in serverless computing [36, 44, 52]. Contrary to traditional cloud services which are typically long-running, waiting or polling for incoming requests, serverless functions are initiated on-demand. Thus, the processing latency of each request contains the startup overhead (including sandbox preparation, function loading and initialization), which eventually affects the user experience. Besides, as the execution unit (serverless function) is typically small and short-lived (in seconds or even milliseconds) in serverless computing, the second-level startup can be a considerable overhead. Moreover, the possible high concurrency in a serverless platform makes it harder to achieve low-latency startup.

Metric3: Stateless execution. Although *stateless* is one of the natures of serverless computing, it may compromise the application performance in two ways: 1) data transmission overhead to maintain the states needed by the function logic in external storage services (e.g., AWS S3 [3]); 2) loss of system states which contain useful information that can help improve performance (e.g., Just-in-Time profile and session cache).

Metric4: Resource efficiency and performance isolation. Co-locating serverless functions (of different patterns and priorities) as well as other computing workloads on the same machine is normal in the cloud for better resource utilization, especially for private serverless platforms. However, co-location may seriously damage the Service Level Agreement (SLA) if without strong performance isolation.

3.2 Methodology

We conduct evaluations on existing serverless platforms with ServerlessBench to explore serverless implications and inspire serverless platform designs and best practices. We evaluate a commercial serverless platform (AWS Lambda [5]), two open-source serverless platforms (OpenWhisk [4] and

Table 1: Test cases with ServerlessBench. "TC" is short for "Test Case". The "Platforms" means whether the test case is applied in OpenWhisk, Fn, AWS, and Ant Financial, respectively, e.g., "● ○ ○ ○" means a test case is only evaluated on OpenWhisk.

Test name	Metrics	Functions	Description	Platforms
TC1: Varied resource needs	Communication	Alu	Analyze resource-efficiency in function composition.	○ ○ ● ○
TC2: Parallel composition	Communication	Alu	Analyze performance with parallel functions.	○ ○ ● ○
TC3: Long function chain	Communication	Node.js Array Sum	Analyze the performance of a long function chain.	● ○ ○ ○
TC4: Application breakdown	Communication	Four representative Apps	Breakdown the latency of real-world applications.	● ○ ○ ○
TC5: Data transfer costs	Communication	Node.js Image	Evaluate the efficiency of cross-function data transfer.	● ● ● ○
TC6: Startup breakdown	Startup latency	Hello, App	Breakdown the OpenWhisk and Fn startup latency.	● ● ○ ○
TC7: Sandbox comparison	Startup latency	Four languages	Analyze four serverless sandbox systems.	\
TC8: Function size	Startup latency	Python PackageImporter	Analyze the startup latency with different function sizes.	○ ○ ● ○
TC9: Concurrent startup	Startup latency	Java Hello, C Hello	Analyze the startup latencies of concurrent requests.	● ● ○ ○
TC10: Stateless costs	Stateless	Java ImageResize	Evaluate the costs of stateless execution.	● ● ○ ○
TC11: Memory bandwidth	Perf isolation	DB-cache, MemBandwidth	Analyze the performance isolation on memory resource.	○ ○ ○ ●
TC12: CPU contention	Perf isolation	DB-cache, Alu	Analyze the performance isolation on CPU resource.	○ ○ ○ ●

Fn [13]), and one in-production private cloud (Ant Financial). The two evaluated open-source platforms run on a x86-64 machine with an 80-core Intel Xeon (@2.00GHz) CPU and 192GB memory. We elaborate our evaluation around the above four metrics from §4 to §7 and present the implications for serverless usage and serverless platform design inferred from the evaluations.

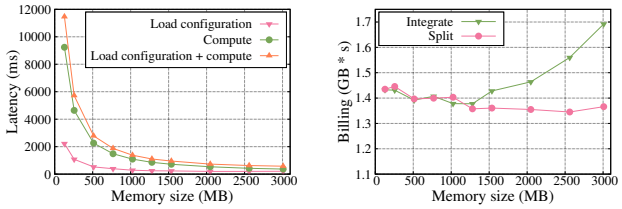
4 FUNCTION COMPOSITION

As the name "Function-as-a-Service" implies, the execution units in serverless computing are typically simple and short-lived functions. Therefore, it is natural that a complex serverless application is composed of several loosely-coupled functions. Existing serverless platforms support several function coordination mechanisms, allowing two composition types: *nested function chain* and *sequence function chain*. Serverless application developers have to make design choices such as **(1) how to split a complex serverless application into functions**, **(2) which composition method to adopt**, and **(3) what intermediate states to pass across interacting functions**.

Due to the lack of references and guidelines, serverless application developers can be confused about how to compose a performant and resource-efficient serverless application. Moreover, our evaluation (Figure 6) shows that communication overhead can take up as much as 70% of the end-to-end latency for an online compiling application, which implies a significant optimization opportunity for platforms. We try to fill this void with a set of implications on serverless composition design, which are drawn from a thorough analysis of serverless composition methods based on our evaluation using ServerlessBench on existing serverless platforms.

4.1 Function Granularity

Splitting a standalone application into a swarm of coordinated serverless functions introduces extra startup and communication overhead. Therefore, serverless application developers might be tempted to include as much logic as possible in a single function for performance concerns. However, putting all the logic in a single function is not a universal solution for all scenarios. We analyze two scenarios where function granularity can affect the serverless billing and performance in TestCase1 and TestCase2, respectively. As the resource allocation is largely configurable in a self-controlled environment, we conduct the tests on AWS Lambda to explore the billing and performance behaviors on black-box commercial systems.



(a) Execution time. **(b) Billing.**

Figure 1: The execution time and billing of integrated and splitted Alu application with different provisioned memory sizes. (a) Different phases have different resource needs. (b) The X-axis represents memory size provisioned for the "Compute" function, while the memory allocation for the "Load configuration" function stops growing at 1024MB.

TestCase1: Varied resource needs. As mentioned in §2, serverless platforms typically charge the execution bill in proportion to the provisioned resources (e.g. memory and CPU resources), which is usually decided by the peak resource needs during the function execution. Thus, in the low-resource-needs phases during the execution, the resources are over-provisioned and overcharged. We evaluate a CPU-intensive Alu application on AWS Lambda and analyze its execution bill when it executes as a single function or splits into two functions. The application consists of two execution phases: in the first phase, it connects to S3 and requests for a number N (**Load configuration phase**); in the second phase, it spawns 100 threads to conduct N times of arithmetic calculations (**Compute phase**). We construct a split version of the

application with a sequence function chain of two functions, handling the two execution phases in the original application, respectively. As AWS Lambda provides vCPUs in proportion to the provisioned memory size, we adjust the provisioned memory sizes to control the vCPU allocation in different execution phases.

Figure 1 (a) suggests that the "Compute" phase is more CPU-intensive, observing performance improvement with provisioned memory size increasing all the way to 3GB, while the execution time for the "Load configuration" phase does not change much when the provisioned memory is larger than 1GB. We then compare the billing in the integrated and split cases with different provisioned memory sizes. For the split case, We provision at most 1GB memory to the "Load configuration" function, as the performance does not improve with larger memory size. Figure 1 (b) shows that the billing (represented by *Memory * Time*) grows drastically in the integrated case after the provisioned memory size is larger than 1GB, while remains steady in the split case. These results suggest that splitting the application and provision appropriate amount of resources to the composing functions can save costs with a small additional communication overhead. [1] The underlying explanation is that the billing model refers to the provisioned resources instead of the actual resource needs. [2]

Implication I

> *Decoupling a serverless application with varied resource needs across execution phases might save costs.*

Listing 1 Function code of Alu.

```
1: def Alu(loopTimes):
2:     result = 0
3:     for i in range(loopTimes):
4:         result += doAlu()
5:     return result
```

TestCase2: Parallel composition. While TestCase1 presents an example of cost-saving with a little overhead added to the function composition, in this test case, we illustrate a scenario where splitting an application into interacting functions improves the performance with parallel execution. We demonstrate with the Alu application in ServerlessBench (Listing 1). The Alu application performs abundant independent arithmetic calculations, with the core handling logic capable of parallel execution. We rewrite the application to adopt parallel execution in two ways: using in-function multi-threading (Figure 2 (b)), or exploiting parallelization between function instances (Figure 2 (c)).

[1] The added communication latency is typically less than 100ms, as evaluated in Figure 7.

[2] Azure Functions charges the resource consumption fee according to the average memory usage, with additional resource tracing mechanisms such as memory usage sampling employed by the platform [9].

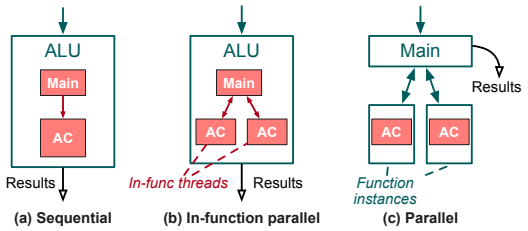
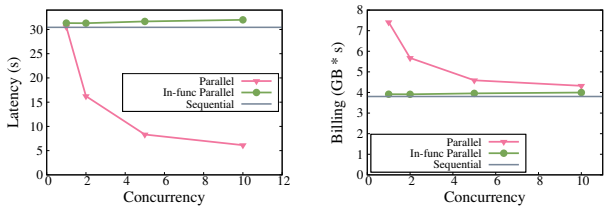


Figure 2: Different function execution patterns. "AC" is short for "Arithmetic Calculation".



(a) Latency. **(b) Billing.**

Figure 3: The latency and billing of parallel-executed Alu application. The "Latency" denotes the end-to-end latency of Main function.

We implement the original application and two of its parallel variants and evaluate their performance on AWS lambda. The results in Figure 3 (a) show that splitting an application with a parallelizable part and executing with concurrent functions outperforms the original standalone application. The request processing latency decreases with added parallel-executed functions, and the performance can improve by 5x with ten concurrent functions. Notice that the in-function parallelization does not perform as well as inter-function parallelization. This is because serverless platforms restrict the computing resources (e.g., vCPUs) that can be allocated to a function instance. Therefore, the concurrency level in a single function instance is confined, and regardlessly running many threads in a single function can even hurt performance. Figure 3 (b) shows less billing with higher function concurrency. This is because the parallel application is composed with nested function chain which faces double-billing problem (analzyed in TestCase4, §4.2). Therefore, with higher concurrency level the double-billed time (execution time for the Arithmetic Calculation) is shorter, so the total bill is also reduced.

Implication II

> *Decoupling the parallelizable part in an application might help boost the overall performance with parallel-executed serverless functions.*

4.2 Composition Method

We now analyze the two main composition methods supported by existing serverless platforms: *sequence function*

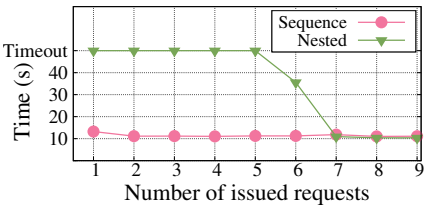


Figure 4: End-to-end latency of long function chains.

chain and *nested function chain*. For sequence function chain, platforms or third-parties provide coordination mechanisms to take care of the workflow and pass the intermediate states across interacting functions (e.g., AWS Step Functions [7], Azure Durable Functions [8]). For nested function chain, the caller function calls another function by platform-provided SDK or direct network requests. Compared to the sequence function chain method, the nested function chain is more widely supported as it does not need additional mechanisms to conduct; instead, any serverless platform allowing network requests from function codes can support the nested function chain. We analyze the two composition methods with precise breakdown on the open-source platform (OpenWhisk).

TestCase3: Long function chain. To explore the performance implications with the two composition methods, we evaluate two variants of the Array Sum application in ServerlessBench. Array Sum is a serverless application with a configurable number of chained functions (the chain length is set to 100 in our test). We implement the two variants of the application with the sequence function chain method and the nested function chain method, respectively. We send the requests successively and record the processing latency of each request.

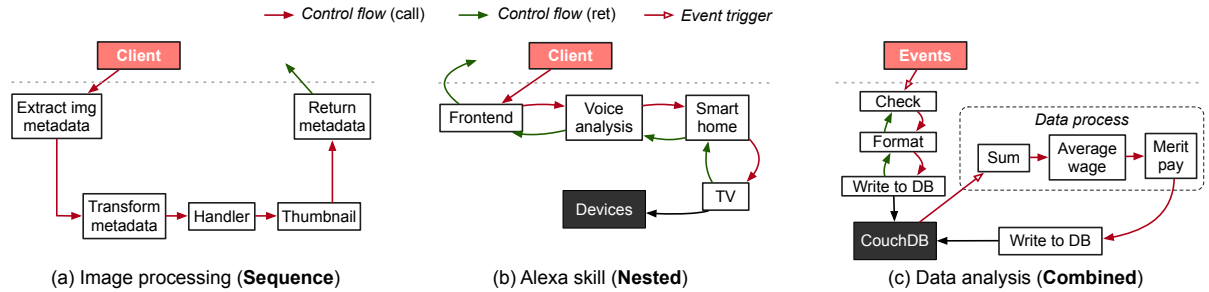
Figure 4 shows that the performance of the sequence chain and the nested chain differs significantly. For the sequence chain, the first invocation triggers a cold start for the first function in the chain, leading to higher latency for the first request. However, since a subsequent function in a sequence chain can reuse the container holding prior functions, the cold start overhead is still minor as it is amortized by all the functions in the chain. For the nested chain, the first five requests fail with a timeout error. The reason is that the first function in a nested chain waits for all the following 99 functions to finish before it can return, so it easily exceeds the timeout value (default to 60s in OpenWhisk), especially when all the functions execute with cold starts. From the 6th request, there are enough warmed containers to allow the nested chain to finish request handling without a timeout, and the performance improves with more warmed functions. We learn a lesson from the evaluation that the nested function chain method faces a higher timeout risk than the sequence function chain method, as the timeout limit of the first function is enforced on the whole chain.

Implication III

> *Nested function chain requires more resources and execution time as the caller function needs to wait for callee functions and, bares higher timeout risk compared to the sequence function chain method.*

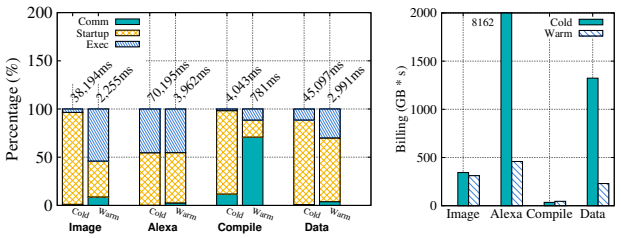
TestCase4: Serverless processing breakdown. To further analyze the impact of different composition methods on complex serverless applications, we include four real-world serverless applications with different composition patterns in ServerlessBench. The four applications are *image processing*, *Alexa skill*, *online compiling*, and *data analysis*. Image processing is one of the most widely-used serverless workloads in the cloud [18, 22]. We compose the image processing application with five functions in a sequence chain (Figure 5 (a)). Alexa skill is a representative of serverless workloads in the IoT world. It equips Alexa [2] with capabilities that the users want, e.g., reading the daily news. We implement the Alexa skill application with nested function chain (Figure 5 (b)). We implement the online compiling application based on *gg* framework [31], which leverages the auto-scaling nature of serverless computing to boost the compiling performance. Online compiling relies on an external coordinator to schedule functions (sequence function chain). Data analysis is an example of serverless applications that are triggered by a third-party event-source. The application begins with the insertion of personal wage data (implemented with nested function chain), then it detects the database change, and triggers a database analysis sequence function chain (Figure 5 (c)). We evaluate the request processing time of the four real-world serverless applications on OpenWhisk [4]. We first upload the tested application into the platform, then sequentially request the application for ten times. We send each request after the response of the last request is received. Thus, the applications serve the first request with a cold started function instance, and serve all subsequent requests with the warm start. For each application, we repeat the evaluation for ten times and calculate the average latency.

We break the end-to-end request processing latency into three parts: startup, communication, and execution. Figure 6 (a) shows the proportion of time spent on each part in cold start and warm start for the four applications. Surprisingly, we notice that in Alexa skill application, although the startup latency decreases from 38,194ms on average in cold start cases to 2,067ms in warm start cases, the proportion of the three parts in the end-to-end latency does not change much. The reason is that we construct the Alexa skill application with nested function chain method, so the execution time of the caller function includes the startup latency of the callee functions as well as the communication overhead. Similar observations occur in the data analysis application (constructed with a combination of sequence and nested function chain):



(a) Image processing (**Sequence**) (b) Alexa skill (**Nested**) (c) Data analysis (**Combined**)

Figure 5: Serverless applications with different composition methods.



(a) Processing latency breakdown (b) Execution bill

Figure 6: Real-world serverless applications on OpenWhisk.

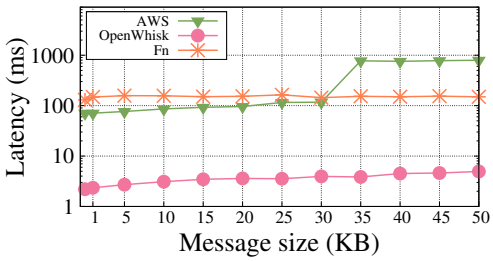


Figure 7: Communication latency with different message sizes.

with the startup latency decreases from 39,528ms to 1,975ms when switching from cold start to warm start, the execution time also decreases from 5,170ms to 900ms. This phenomenon reflects the double-billing drawback of nested function chain, i.e., the function processing time is charged more than once in a nested chain. As is shown in Figure 6 (b), the billing duration contains all the overhead between nested chain functions, including startup and communication. Thus, *the slow cold start not only affects the startup latency but can also poison the execution performance in the nested function chain, and cause a drastic rise in serverless cost*. By contrast, the execution time and overhead variation in one function do not affect the execution time of another in a sequence chain, thus free of double-billing problem.

Implication IV

> *Composition methods can significantly impact the billing in serverless computing as the platforms charge the execution time and overhead more than once in a nested function chain.*

Despite the higher timeout risk and double-billing problem, serverless application developers might still prefer the nested chain method in specific scenarios with its advantages in flexibility. First, not all serverless platforms provide sequence function chain mechanisms with semantics needed by the application. For example, Google Cloud Functions [15] does not have a platform-provided serverless orchestration mechanism. OpenWhisk provides Action Sequence [20], but only supports a single chain, i.e., common composition patterns such as branching and parallelization are not supported. Second, even for function coordinators with rich semantics such as Amazon Step Functions [7], which orchestrates functions with user-defined state machines, the supported semantics might not be flexible enough for many use cases. For example, Amazon Step Functions support the *Parallel* state type to execute tasks in parallel. However, the application developers have to list all the tasks to execute in parallel in an array in the state machine, restricting the flexibility of concurrency level. Besides, it is a waste of effort if all the tasks to execute in parallel are the same function. Third, adopting coordinators with rich semantics usually require application developers to make extra efforts on the orchestration services, e.g., the state machine for whole application using Amazon Step Functions.

4.3 Payload Size

TestCase5: Data transfer costs. The data transfer between serverless functions typically involves several steps, e.g., request routing, load balancing, and message queue operations. To examine how efficient existing serverless platforms transfer data between functions, we evaluate a Node.js serverless application which transfers images with different sizes (from 0KB to 50KB) on OpenWhisk, Fn, and AWS Lambda. We implement the chain on OpenWhisk with OpenWhisk's Action Sequence [20]. For Fn, we leverage *flow* [14], a lightweight workflow model provided by Fn project. We employ AWS Step Functions [7] for AWS Lambda. As AWS Step Functions

restricts the directly-passing payload size to less than 32KB, we upload the payload data to S3 before the caller function finishes, and let the callee function download the data before it starts the handling logic for the payload with sizes more than 32KB (this is also the suggested workaround for large payloads by AWS [21]). We record the period between the caller function finishes and the callee function ready to *unpause* as OpenWhisk's communication latency, and use the period between the end of the caller function and the start of the callee function as the communication latency in Fn and AWS Lambda with payload size under 32KB. For large payload (more than 32KB) in AWS Lambda, we record the time between the data uploading (to S3) in the caller function and the start of handling (after downloading the payload) in the callee function as the communication latency.

The results in Figure 7 show that communication latency increases very slowly within a small range of transferred image sizes (0KB–30KB) on all three platforms. Because the data passing method changes from direct data passing to indirect data passing via S3 due to the AWS Lambda data passing limit, the communication latency rises drastically at 35KB. Placing interacting functions on the same node further inspires data passing optimizations such as using local message bus [22] or shared memory [49] for data transfer. Thus, we address the observation that passing a limited size of data between functions may only add neglectable communication overhead compared to the case with no data passed between functions. This observation can motivate possible serverless execution optimization with state passing, as explored in [48].

Implication V

Passing small messages during communication incurs little additional costs, while the applicaiton developers might use the takeaway information to optimize the execution of subsequent functions.

5 STARTUP LATENCY

This section analyzes the startup latency challenge faced by serverless applications with a set of detailed evaluation in different platform settings and requesting scenarios.

TestCase6: Startup breakdown. To analyze the source of serverless startup cost, we make a detailed breakdown of the startup latency of four functions in ServerlessBench: Python-Hello, Java-Hello, Python-Django and Java-ImageResize. We examine the warm start and cold start cases on two open-source serverless platforms, OpenWhisk and Fn [13].

Table 2 presents the time spent in each phase of the startup process in OpenWhisk. In a typical cold start with image pulling (when the function not specifies the docker image tag or uses the "latest" tag), the image pulling phase ("Image pull" in Table 2) is the primary source of cost, taking up to 81% of the total startup latency. In addition, sandbox initialization ("Docker run" and "Docker init" in Table 2) takes up 20% of the startup latency (or more than 95% when a local image is used and free of image pulling overhead), during which Docker uncompresses the function image into the local storage, prepares the isolation environment (e.g., namespaces and cgroups), and initiates the language runtime (e.g., Python and JVM) of the invoked function. The warm start takes about 135ms, which is 35x faster than a cold start with image pulling (or 7x faster than a cold start without image pulling). In the warm start, "Docker unpause" becomes the primary source of cost, composing up to 71% of the total startup latency. In this phase, the serverless platform wakes up a paused container with the same function image to handle the new requests. OpenWhisk also provides a hot start mechanism, where a request reuses the container holding a just-finished function instance without pausing/unpausing. With OpenWhisk hot start, the function startup time can reduce to less than 10ms. However, OpenWhisk pauses the function container after 50ms since it finishes, so the hot start case only applies to requests arrive within 50ms interval since a reusable function instance finishes.

Table 3 breaks down the cold start latency of a Hello Python function in Fn. Fn platform always looks for and uses the local function image in a cold start, contrasting to OpenWhisk, which only looks to the local image registry when a specific docker image tag is configured. The overall cold start latency (without image pulling) is 941ms, which slightly outperforms OpenWhisk, as the complex components involved in OpenWhisk (e.g., Kafka for message passing from OpenWhisk core controller to the function invokers and CouchDB for function lifetime management) introduce additional overheads. Fn also provides a high-performance "hot function" mechanism, which can achieve around 1ms startup latency.

The results above show the effort to optimize the startup latency by holding or caching the finished sandboxes on the platform to avoid long latency incurred by cold start for every request. However, the finished function instances can remain in hot or warm state only for a short period of time before they cool down, because the platform cannot predict when will a next request arrive, and keeping idle sandboxes wastes platform resources. There are existing methods that use user-provided information to increase the warm start probability. For example, AWS Lambda provides "provisioned concurrency" configuration that promises fast startup for the configured concurrency. However, this configuration is static and pre-defined. For more flexible and dynamic resource allocation, we suggest that an end-user who is knowledgeable about the request issuing pattern can inform the platform with some hints in each request, e.g., expected arrival time of the next request and the numbers of subsequent requests. This negotiation can benefit both the serverless provider and the user.

Table 2: OpenWhisk startup latency breakdown.

	Applications	Routing	Load balance	Msg queue	Image pull (optional)	Docker run	Docker init	Send Arg.	Sum
Cold start (ms)	Complex Java	30.7	3.0	0.8	3645.7	749.3	158.4	18.7	960.9 (+3645.7)
	Complex Python	30.3	3.0	0.9	3812.8	733.0	266.6	2.1	1035.9 (+3812.8)
	Hello Java	30.5	2.6	0.6	3609.0	744.2	159.7	18.0	955.6 (+3609.0)
	Hello Python	30.5	2.8	0.9	3763.6	741.0	259.7	1.2	1036.1 (+3763.6)
	Applications	**Routing**	**Load balance**	**Msg queue**	**Prepare container**	**Docker unpause**	**Prepare Arg.**	**Send Arg.**	**Sum**
Warm start (ms)	Complex Java	30.3	2.7	0.8	0.2	97.2	1.8	2.3	135.4
	Complex Python	30.2	2.7	0.7	0.3	96.0	1.7	3.3	135.2
	Hello Java	30.0	2.4	0.9	0.3	95.6	1.4	2.5	133.0
	Hello Python	30.1	2.6	0.8	0.3	96.7	1.7	1.8	134.4

Table 3: Fn startup latency breakdown.

	Routing	Image pull (optional)	Image validate	Docker create	Docker run	Function exec	Sum
Cold start (ms)	<1	6281.2	1.8	99.6	529.5	263.4	894.6 (+6281.2)

The serverless provider can charge the resource idling fee on the user thus avoid resource wasting, and users are willing to be charged as the warm start leads to shorter execution time and thus lower execution cost.

Implication VI

Serverless platforms can accept hints of the expected arrival time of the next request that can help adjust the resource holding time to serve requests with warm start as much as possible.

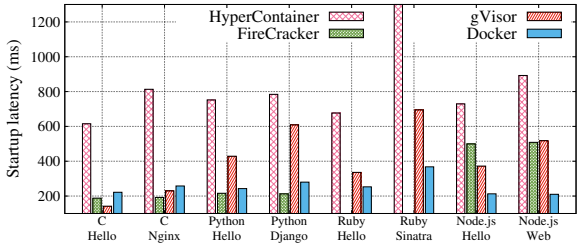


Figure 8: Startup latency with different sandboxing methods. Cold start for all cases.

TestCase7: Sandbox comparison. As illustrated in §2, different serverless platforms might employ different sandboxing methods to hold the function runtime environment. For example, both Fn and OpenWhisk adopt Docker containers as the function sandbox. The startup breakdown for OpenWhisk and Fn shows that Docker operations (e.g., Docker run, Docker unpause) are significant sources of startup overhead. We analyze how different sandboxes perform in the function booting process with four widely-used sandbox runtimes: Docker [11], FireCracker [12], gVisor [16], and HyperContainer [17]. We evaluate functions written in four programming languages, including Python, Node.js, Ruby, and C, which are the most used languages in existing serverless platforms [1]. For each language, a simple "HelloWorld" application and a real-world application are included. For C, we use Stateless-Nginx as a real-world application. We use Django for Python and Sinatra for Ruby as the real-life applications, which are both web applications. For Node.js real-life application, we use a web server.

The results (Figure 8) show that sandboxes with higher isolation levels (e.g., HyperContainer, gVisor) typically suffer from longer startup latency, as the sandbox runtime is heavier. Firecracker is a lightweight virtual machine designed and employed by AWS Lambda. It stands out with the shortest startup latency in all evaluated applications except for Node.js applications.

Implication VII

General-purpose sandboxes such as Docker containers are not efficient enough for serverless computing. Serverless platform designers should research on serverless-specific sandbox designs (e.g., FireCracker microVM) that might reduce the startup overhead and even provide stronger isolation.

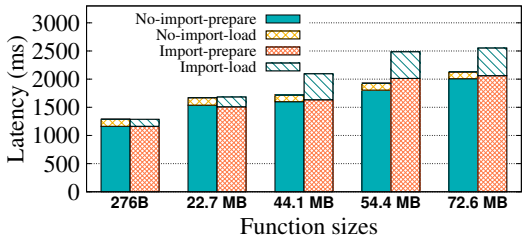


Figure 9: Startup latency with different function sizes. Cold start for all cases.

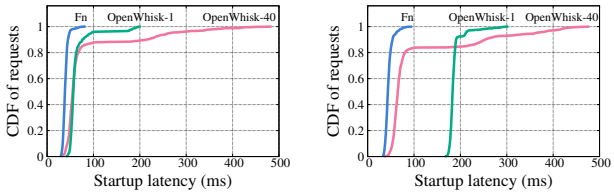
TestCase8: Function size comparison. Function code size is an often-overlooked factor in serverless use cases. Before a serverless platform can execute a function for the first time (cold start), it has to prepare the execution environment by first importing and unzipping the function package (containing the function source codes and dependency packages), then loading the function codes and dependencies for execution. In this process, larger function package might bring higher overhead to the total startup latency. We analyze the impact of function size on serverless startup latency using the Python PackageImporter application in ServerlessBench. We construct the application by packing the Python handler codes with different numbers of dependency packages, producing varying package sizes. Specifically, we add four popular PyPI packages (*mypy*, *numpy*, *django*, and *sphinx*) of sizes between 10–23MB into the function package accumulatively, producing five function packages of sizes: 276B (no dependency packages included), 22.7MB (with *mypy*), 44.1MB (with *mypy* and *numpy*), 54.4MB (with *mypy*, *numpy*, and *django*), and 72.6MB (with all four packages). For each package size, we compare the startup latency when the function handler actually imports the packages or not (the dependency packages are redundantly included in this case).

We evaluate how function size affects startup latency on a typical commercial serverless platform (AWS Lambda). We break down the startup latency into "prepare" and "load" phases. The latency in the "prepare" phase includes time spent from the request issuing at the client-side, to the request handling and function triggering directed by AWS platform, and function code transmission from S3 storage to the compute node where the lambda function will execute on. We extract the time spent in the "load" phase from Lambda execution logs ("Init Duration" attribute provided by AWS Lambda). During this phase, the platform prepares the function execution environment on the compute node and loads the function codes into the execution environment.

Figure 9 shows the evaluation results. Comparing the "import" and "no-import" cases of the same package size, we observe similar latency on the "prepare" phase. However, the total startup latencies differ because the loading time is longer when the handler actually imports the packages. Besides, larger function packages endure longer startup latency due to larger data transmission size in the "prepare" phase. There are existing researches pointing out the package-import overhead [44] and suggesting optimizations with locally-prepared packages and cached execution environment. However, many existing serverless platforms (including AWS Lambda and OpenWhisk) expect the application developers to pack the dependencies in the function package and do not handle the dependencies on the platform side, so serverless application developers should still be careful about the function size and dependencies really needed.

Implication VIII

Functions with larger code sizes suffer from longer startup latency due to larger data transmission and package import overhead. Serverless application developers should optimize the function codes to import the minimal needed packages and pack only necessary dependencies.



(a) Java function. **(b) C function.**

Figure 10: Startup latency with different container concurrency. "OpenWhisk-40" denotes the tests on OpenWhisk with container concurrency of 40.

TestCase9: Concurrent startup. To analyze how the auto-scale property of serverless computing impacts serverless startup, we evaluate the startup latency of the Java-Hello and C-Hello function in ServerlessBench on OpenWhisk and Fn. For each function, we send 40 requests simultaneously with 40 clients. We compare the distribution of startup latencies with the maximum concurrent container number limit in OpenWhisk configured to be 1 and 40. Fn does not have a hard limit in the maximum concurrent container number on a single node, but in our tests, the actual concurrency of Fn containers is 30–40.

Figure 10 presents the cumulative distribution function (CDF) of startup latency of the Java-Hello and C-Hello functions, with different container concurrency. The C-Hello function with only one container on OpenWhisk runs much slower than other cases because of the queueing delay in the message queue (the Java-Hello function does not suffer from this significant delay because the Java-Hello function executes faster than the C-Hello function, thus shortens the wait time). We observe drastically increased startup latency and significantly longer tail at higher container concurrency on OpenWhisk. For example, the p99 latency of the Java function is 191ms for one concurrent container and 415ms for 40 concurrent containers. Compared to OpenWhisk, Fn has a significantly shorter tail. The main reason for the difference in tail latency lies in the framework components: OpenWhisk employs several complex building blocks to support the functioning of the framework (e.g., CouchDB to maintain the system states, Kafka for message passing, and ZooKeeper to manage the Kafka cluster); while Fn does not integrate the complex components in itself. This implies that the serverless platform

designers should pay attention to each supporting component's scalability in the serverless framework to preserve the auto-scaling promise for serverless users.

Implication IX

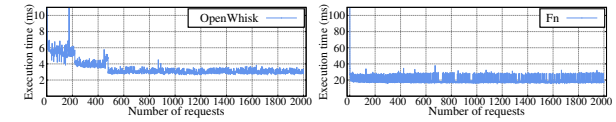
The design of supporting components for serverless platforms (such as load balancer and message queue) can affect the scalability of serverless computing.

6 STATELESS EXECUTION

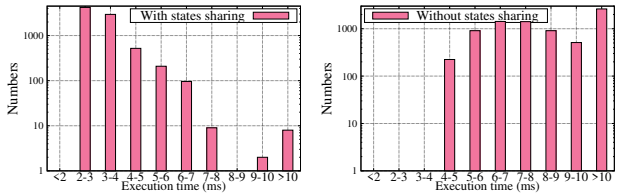
With the consensus that "stateless" is one of the build-in natures of serverless computing, we study the implications related to the states in this section. In the paper, we classify the states into two categories: *explicit states* and *implicit states*. The explicit states are the data explicitly used and managed by serverless functions, while the implicit states are the data used by the language runtime (e.g., cached data and codes, JIT profiles). As many state-of-the-art cloud applications are stateful, many recent works discuss and propose stateful serverless designs [19, 24, 49, 50]. The design of a serverless state management system must highlight two key factors: 1) states being durable beyond function life cycles; 2) state passing being efficient enough. Most researches adopt a two-layer design with a high-efficiency local data passing mechanism (using local cache [24, 50], local key-value store [19], or shared memory [49]) along with a scalable and durable global object storage.

While the platforms have put effort into supporting stateful property in serverless computing to expand serverless use cases, they usually drop the implicit states which do not affect the correctness of execution. Nevertheless, the implicit states might contain useful information that can help improve performance. For example, JVMs typically employ Just-in-Time (JIT) Compiling to improve application performance over time. A long-running application can accumulate the profile information throughout the execution, and leverage it to generate code cache for performance improvement, but scaling out requests into stateless function instances loses the information.

TestCase10: Stateless costs. To explore the effect of the implicit states on serverless computing, we use the ImageResize function in ServerlessBench to compare the execution time when the requests share the implicit states or not. We employ open-source platforms in this test to control the state-reusing behavior. First, we simulate a long-running scenario by invoking the ImageResize function one by one (up to 2000 requests). The warm start mechanism preserves all states across requests by serving each request in the same function container. Figure 11 (a) shows the execution time of each sequentially executed ImageResize function on OpenWhisk and Fn. The result on OpenWhisk reveals three significant



(a) Function execution time with states sharing across requests.



(b) Execution time distribution on OpenWhisk.

Figure 11: ImageResize application execution with or without state sharing across requests.

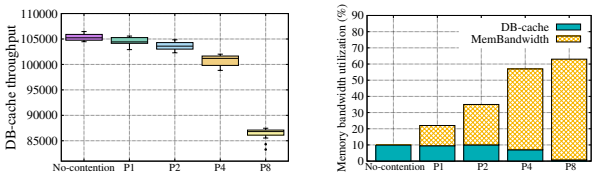
latency decreases: the first decrease from 90ms to 6ms after five function invocations, and the second from 6ms to 4ms at the 210th invocation, and to 3ms at the 478th invocation. The latency decrease at the beginning comes from the lazy initialization mechanisms in JVM, e.g., dynamic class loading, in which case the classes loaded in the prior executions save the class loading time for the subsequent executions. JIT Compilation leads to the following two significant latency decreases. Fn's result also indicates that the execution time decreases significantly after the first request, suggesting the performance degradation caused by lost system states.

We then compare the execution time of the same function without state-sharing on OpenWhisk. The left of Figure 11 (b) presents the distribution of the function execution times with state-sharing in OpenWhisk (corresponding to the left of Figure 11 (a), repeated four times). The results show that most executions finish in less than 5ms. For the non-sharing scenario, we send 2000 concurrent requests to OpenWhisk and collect execution times for each of them. We configure the maximum concurrent container number limit of OpenWhisk to 80. We repeat the test 4 times and present the execution time distribution of the 8000 requests in the right of Figure 11 (b). Compared with the distribution in the state-sharing case (the left of Figure 11 (b)), we observe longer execution time in the non-sharing scenario, i.e., >7ms for most requests.

To conclude, the stateless nature in serverless computing introduces extra latency with both the transfer of explicit states and the loss of implicit states. While the application developers have to manage the explicit states to ensure correctness, the serverless platform controls the implicit states (e.g., JIT profiles), which could be leveraged for optimizations.

Implication X

Serverless platforms could share the implicit states, e.g., cache or JIT profile, among instances of a function to improve the execution performance.



(a) Throughput. **(b) Memory utilization.**

Figure 12: Performance comparison when DB-cache application co-running with memory-intensive workload. (a) DB-cache throughput under different memory pressure. (b) Memory bandwidth utilization under different memory pressure. The series with the "No-contention" label shows the ideal case, i.e., DB-cache instances running without memory-intensive workloads. Other series show results when DB-cache instances co-run with different-pressured memory-intensive workload.

7 PERFORMANCE ISOLATION

Co-locating serverless functions with different resource needs (e.g., memory-intensive) is a common practice for resource efficiency [33]. In some cases, cloud providers co-locate the serverless functions with long-running analytics workloads on the same machine. While resource sharing happens in various co-location scenarios, the datacenter operators should carefully assess the performance isolation to preclude the possibility that other workloads break the SLAs of serverless applications.

This section explores the performance isolation issue in a private serverless platform (Ant Financial), which has been extensively co-locating the workloads to improve the resource utilization. We deploy a Redis-like cache-based serverless database application, *DB-cache*, with CPU-intensive and memory-intensive workloads, and examine sharing and isolation implications for each resource. Specifically, we bind eight DB-cache instances on 8 CPU cores on the same CPU socket (16 virtual CPUs with SMT enabled). We bind eight function invokers (responsible for sending serverless requests and collecting the responses) to cores on another CPU socket. We use an Alu application (CPU-intensive workload) and a MemBandwidth application (memory-intensive workload) to explore the CPU-related and memory-related implications, respectively. Running both the invokers and serverless functions on the same machine rules out the uncertain latency added by network transmission.

TestCase11: Memory bandwidth contention. To evaluate the performance isolation on memory bandwidth, we co-locate DB-cache application with a memory-intensive workload (MemBandwidth application). We construct cases with different memory bandwidth pressures by configuring the thread number of the MemBandwidth application to 1, 2, 4, and 8 (denoted by the ending "x" in "Px" labels in Figure 12). The more threads running by the MemBandwidth application, the higher memory bandwidth pressure it generates. Eight function invokers each sends a million SET requests to a DB-cache instance with 50 parallel connections. Figure 12 (a) shows the DB-cache throughput when co-running with the MemBandwidth application set to different thread numbers. The results show that memory-intensive workloads could severely degrade the performance of serverless applications (17% throughput loss in the "P8" case compared to the "No-contention" case). However, when the thread number of MemBandwidth application is 1 or 2 (the "P1" and "P2" cases in Figure 12), the performance of the DB-cache application is not severely affected. Figure 12 (b) suggests that in the "P1" and "P2" cases, the total memory bandwidth need is still modest, and the bandwidth consumption of the DB-cache application does not reduce. This indicates that properly enforcing bandwidth limit on the memory-intensive workloads on the same node can preserve the performance of serverless applications.

Implication XI

> *A serverless function will contend with others for memory bandwidth, which implies the platform should provide an essential isolation mechanism to guarantee sufficient bandwidth budgets.*

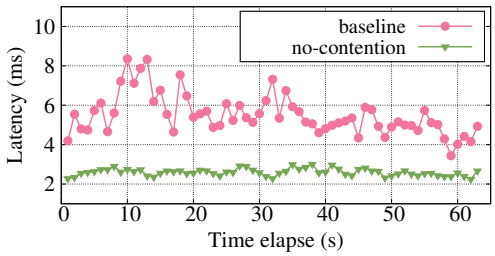


Figure 13: Performance comparison when DB-cache application co-running with CPU-intensive workload.

TestCase12: CPU contention. CPU sharing among applications is a default behavior handled by the system scheduler. Modern schedulers such as Complete Fair Scheduler (CFS) in Linux [27] have been providing priority differentiation and enforcement mechanisms since long ago, e.g., by setting CPU shares in the CPU control group. However, schedulers cannot guarantee that performance will not degrade when latency-sensitive serverless applications co-run with CPU-intensive workloads.

We conduct a comparative experiment to evaluate the performance of the DB-cache application in cases with or without CPU-intensive workload. A 16-threaded CPU-intensive workload (the Alu application) runs on the same CPU cores as the DB-cache application. To analyze the existing priority enforcement mechanism, we set the CPU shares of the

Alu application to 2, while the CPU shares for the DB-cache application remains the default value (1024). Each function invoker sends a million LRANGE_600 requests to a DB-cache instance with 50 parallel connections. We calculate the average processing latencies of the eight DB-cache instances in each second. Figure 13 shows 109% higher latency when the DB-cache instances co-run with the Alu application.

Analyzing the system resource utilization, we find that the Alu application by itself causes nearly 100% CPU utilization on all 16 virtual CPUs. While the CPU shares parameter prioritizes the DB-cache threads over the Alu threads on the same virtual CPU, the intense CPU usage from the sibling hyperthread still limits the physical computation resource available to DB-cache, thus degrading the performance of the DB-cache application.

Implication XII

Linux CPU Shares is insufficient. A more comprehensive system-level mechanism should accompany to maintain the performance of serverless functions, such as enforcing hardware thread priority across sibling virtual CPUs.

8 OPTIMIZATION CASE STUDY

The implications found with ServerlessBench can guide the design of serverless platforms and serverless applications. In this section, we briefly introduce a startup optimization work inspired by **Implication VII**, Catalyzer [28].

Catalyzer is a customized serverless sandbox design providing both strong isolation and extremely fast function startup. Instead of booting from scratch, Catalyzer achieves initialization-less by restoring from well-formed checkpoint images or forking from running template sandbox instances. Fundamentally, it removes the initialization cost by reusing state, which enables general optimizations for diverse serverless functions.

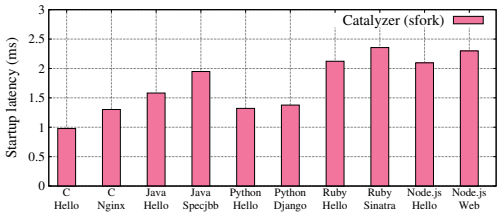


Figure 14: Startup latency with Catalyzer on gVisor.

We evaluate the startup latency with Catalyzer on gVisor to show the effectiveness. Figure 14 shows the startup latencies with different languages on Catalyzer (forking from template sandboxes), ranging from 0.9ms to 2.4ms. Comparing to results in Figure 8, Catalyzer brings orders-of-magnitude speedup on function startup.

9 RELATED WORK

DeathstarBench [32] is a benchmark suite for large-scale applications with tens of microservices. It constructs several microservice applications (e.g., social networks) to reveal the implications of microservices, e.g., network contention and QoS violations. CloudSuite [29] and DCBench [35] explore the architecture implications of scale-out cloud workloads, e.g., the organization of instruction and memory systems. These benchmark suites fail to identify performance bottlenecks of serverless platforms, which should provide high scalability as well as low latency.

In serverless-specific field, Microsoft researchers characterize the serverless workloads on Azure Functions [46]. FunctionBench [37] and SPEC-RG Vision [51] propose serverless benchmark designs to reflect various aspects of serverless computing. Previous works also evaluate and compare how serverless applications behave on different platforms [23, 30, 39]. These works focus more on the serverless workloads and the performance differences between platforms instead of analyzing the underlying implications on serverless computing. Wang et al. [53] reverse-engineer the architectures adopted by popular serverless providers with architectural-specific measurements on these platforms. Shahrad et al. [45] uncover architectural and microarchitectural impacts on serverless computing with evaluations on OpenWhisk. Our work distinguishes by three key factors. First, we introduce a set of serverless benchmarks to analyze performance metrics that are unique and significant in serverless platforms. Second, we conclude serverless implications that can inspire the designing and using of serverless systems. Third, we evaluate on different categories of serverless platforms (open-source, commercial and private cloud) with both white-box and black-box analysis.

10 CONCLUSION

This paper proposes ServerlessBench, an open-source benchmark suite for cloud platform providing serverless computing services. We have leveraged the benchmark suite to explore 12 new implications on existing platforms, which are the basis for serverless system designers to optimize the platforms and for application developers to refine their serverless applications.

11 ACKNOWLEDGMENT

This work is supported in part by Key-Area Research and Development Program of Guangdong Province (No. 2020B010164003), the National Natural Science Foundation of China (No. 61972244, U19A2060, 61925206), the HighTech Support Program from Shanghai Committee of Science and Technology (No. 19511121100). Yubin Xia (xiayubin@sjtu.edu.cn) is the corresponding author.

REFERENCES

[1] [n.d.]. 2018 Serverless Community Survey: huge growth in serverless usage. https://serverless.com/blog/2018-serverless-community-survey-huge-growth-usage/. Referenced Jan 2020.

[2] [n.d.]. Amazon Alexa. https://developer.amazon.com/en-US/alexa. Referenced December 2019.

[3] [n.d.]. AMAZON. Amazon simple storage service. https://aws.amazon.com/s3. Referenced December 2019.

[4] [n.d.]. Apache OpenWhisk is a serverless, open source cloud platform. http://openwhisk.apache.org/. Referenced December 2018.

[5] [n.d.]. AWS Lambda - Serverless Compute. https://aws.amazon.com/lambda/. Referenced December 2018.

[6] [n.d.]. AWS Serverless Application Repository. https://serverlessrepo.aws.amazon.com. Referenced December 2019.

[7] [n.d.]. AWS Step Functions. https://aws.amazon.com/step-functions/. Referenced July 2019.

[8] [n.d.]. Azure Durable Functions. https://docs.microsoft.com/en-us/azure/azure-functions/durable/. Referenced May 2020.

[9] [n.d.]. Azure Functions Pricing. https://azure.microsoft.com/en-us/pricing/details/functions/. Referenced May 2020.

[10] [n.d.]. Azure Functions Serverless Architecture. https://azure.microsoft.com/en-us/services/functions/. Referenced December 2018.

[11] [n.d.]. The Docker Containerization Platform. https://www.docker.com/. Referenced December 2018.

[12] [n.d.]. Firecracker. https://firecracker-microvm.github.io/. Referenced December 2018.

[13] [n.d.]. Fn Project - The Container Native Serverless Framework. https://fnproject.io. Referenced December 2018.

[14] [n.d.]. fnproject/flow: Fn Flow Server. https://github.com/fnproject/flow. Referenced Jan 2020.

[15] [n.d.]. Google Cloud Function. https://cloud.google.com/functions/. Referenced December 2018.

[16] [n.d.]. Google gVisor: Container Runtime Sandbox. https://github.com/google/gvisor. Referenced December 2018.

[17] [n.d.]. Hyper - Make VM run like Container. https://hypercontainer.io/. Referenced December 2018.

[18] [n.d.]. lambda-refarch-imagerecognition. https://github.com/aws-samples/lambda-refarch-imagerecognition. Referenced December 2019.

[19] [n.d.]. Nimbella. The simplest way to build and run serverless applications. https://nimbella.com. Referenced May 2020.

[20] [n.d.]. openwhisk/actions.md. https://github.com/apache/openwhisk/blob/master/docs/actions.md. Referenced July 2019.

[21] [n.d.]. Use ARNs Instead of Passing Large Payloads. https://docs.aws.amazon.com/step-functions/latest/dg/avoid-exec-failures.html. Referenced May 2020.

[22] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 923–935.

[23] Timon Back and Vasilios Andrikopoulos. 2018. Using a Microbenchmark to Compare Function as a Service Solutions. In *Service-Oriented and Cloud Computing*, Kyriakos Kritikos, Pierluigi Plebani, and Flavio de Paoli (Eds.). Springer International Publishing, Cham, 146–160.

[24] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. 2019. On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures. In *Proceedings of the 20th International Middleware Conference* (Davis, CA, USA) *(Middleware '19)*. Association for Computing Machinery, New York, NY, USA, 41–54. https://doi.org/10.1145/3361525.3361535

[25] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe User-Level Access to Privileged CPU Features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, USA, 335–348.

[26] Sol Boucher, Anuj Kalia, David G. Andersen, and Michael Kaminsky. 2018. Putting the "Micro" Back in Microservice. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 645–650.

[27] Justinien Bouron, Sebastien Chevalley, Baptiste Lepers, Willy Zwaenepoel, Redha Gouicem, Julia Lawall, Gilles Muller, and Julien Sopena. 2018. The Battle of the Schedulers: FreeBSD ULE vs. Linux CFS. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 85–96.

[28] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 467–481.

[29] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. *SIGPLAN Not.* 47, 4 (2012), 37–48.

[30] Kamil Figiela, Adam Gajek, Adam Zima, Beata Obrok, and Maciej Malawski. 2018. Performance evaluation of heterogeneous cloud functions. *Concurrency and Computation: Practice and Experience* 30, 23 (2018), e4792.

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

[32] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. ACM, 3–18.

[33] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. 2019. Who Limits the Resource Efficiency of My Datacenter: An Analysis of Alibaba Datacenter Traces. In *Proceedings of the International Symposium on Quality of Service* (Phoenix, Arizona) *(IWQoS '19)*. Association for Computing Machinery, New York, NY, USA.

[34] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2018. Serverless Computing: One Step Forward, Two Steps Back. *arXiv preprint arXiv:1812.03651* (2018).

[35] Zhen Jia, Lei Wang, Jianfeng Zhan, Lixin Zhang, and Chunjie Luo. 2013. Characterizing data analysis workloads in data centers. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 66–76.

[36] Eric Jonas, Johann Schleier-Smith, Vikram Sreekantl, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl

Krauth, Neeraja Yadwadkar, Joseph E. Gonzales, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *arXiv preprint arXiv:1902.03383* (2019).

[37] J. Kim and K. Lee. 2019. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. 502–504.

[38] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, USA, 427–444.

[39] Hyungro Lee, Kumar Satyam, and Geoffrey Fox. 2018. Evaluation of production serverless computing environments. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 442–450.

[40] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. 2019. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 363–378.

[41] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, Jon Crowcroft, and Ian Leslie. 2015. Jitsu: Just-in-Time Summoning of Unikernels. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Oakland, CA) *(NSDI'15)*. USENIX Association, USA, 559–573.

[42] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 218–233.

[43] Garrett McGrath, Jared Short, Stephen Ennis, Brenden Judson, and Paul Brenner. 2016. Cloud event programming paradigms: Applications and analysis. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. IEEE, 400–406.

[44] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. 57–70.

[45] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. 2019. Architectural Implications of Function-as-a-Service Computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 1063–1075.

[46] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. *arXiv preprint arXiv:2003.03423* (2020).

[47] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. 2019. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. ACM, 121–135.

[48] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. 2016. Fast and Concurrent RDF Queries with RDMA-Based Distributed Graph Exploration. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 317–332.

[49] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. arXiv:2002.09344 [cs.DC]

[50] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Jose M. Faleiro, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful Functions-as-a-Service. arXiv:2001.04592 [cs.DC]

[51] Erwin van Eyk, Joel Scheuner, Simon Eismann, Cristina L. Abad, and Alexandru Iosup. 2020. Beyond Microbenchmarks: The SPEC-RG Vision for a Comprehensive Serverless Benchmark. In *Companion of the ACM/SPEC International Conference on Performance Engineering* (Edmonton AB, Canada) *(ICPE '20)*. Association for Computing Machinery, New York, NY, USA, 26–31.

[52] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. 2019. Replayable Execution Optimized for Page Sharing for a Managed Runtime Environment. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. ACM, 39.

[53] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking behind the curtains of serverless platforms. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 133–146.

[54] Liang Zhang, James Litton, Frank Cangialosi, Theophilus Benson, Dave Levin, and Alan Mislove. 2016. Picocenter: Supporting Long-lived, Mostly-idle Applications in Cloud Environments. In *Proceedings of the Eleventh European Conference on Computer Systems* (London, United Kingdom) *(EuroSys '16)*. ACM, New York, NY, USA, Article 37, 37:1–37:16 pages.