Analyzing Tail Latency in Serverless Clouds with STeLLAR

Citation for published version:
Ustiugov, D, Amariucai, T & Grot, B 2022, Analyzing Tail Latency in Serverless Clouds with STeLLAR. in 2021 IEEE International Symposium on Workload Characterization (IISWC'21). Institute of Electrical and Electronics Engineers, pp. 51-62, 2021 IEEE International Symposium on Workload Characterization, 7/11/21. https://doi.org/10.1109/IISWC53511.2021.00016

Digital Object Identifier (DOI):
10.1109/IISWC53511.2021.00016

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
2021 IEEE International Symposium on Workload Characterization (IISWC'21)

In Proceedings of the 2021 IEEE International Symposium on Workload Characterization (IISWC'21)

Analyzing Tail Latency in Serverless Clouds with STeLLAR

Dmitrii Ustiugov[§]
University of Edinburgh

Theodor Amariucai[§]
University of Edinburgh

Boris Grot
University of Edinburgh

***Abstract*—Serverless computing has seen rapid adoption because of its instant scalability, flexible billing model, and economies of scale. In serverless, developers structure their applications as a collection of functions invoked by various events like clicks, and cloud providers take responsibility for cloud infrastructure management. As with other cloud services, serverless deployments require responsiveness and performance predictability manifested through low average and tail latencies. While the average end-to-end latency has been extensively studied in prior works, existing papers lack a detailed characterization of the effects of tail latency in real-world serverless scenarios and their root causes.**

In response, we introduce STeLLAR, an open-source serverless benchmarking framework, which enables an accurate performance characterization of serverless deployments. STeLLAR is provider-agnostic and highly configurable, allowing the analysis of both end-to-end and per-component performance with minimal instrumentation effort. Using STeLLAR, we study three leading serverless clouds and reveal that storage accesses and bursty function invocation traffic are key factors impacting tail latency in modern serverless systems. Finally, we identify important factors that do *not* contribute to latency variability, such as the choice of language runtime.

***Index Terms*—serverless, tail latency, benchmarking**

I. Introduction

Serverless computing, also known as Function-as-a-Service (FaaS), has emerged as a popular cloud paradigm, with the serverless market projected to grow at the compound annual growth rate of 22.7% from 2020 to 2025 [1]. With serverless, developers structure their application logic as a collection of functions triggered by events (e.g., clicks). The number of instances of each function active at any given time is determined by the cloud provider based on instantaneous traffic load directed at that particular function. Thus, developers benefit from serverless through simplified management and pay-per-actual-usage billing of cloud applications, while cloud providers achieve higher aggregate resource utilization which translates to higher revenues.

Online services have stringent performance demands, with even slight response-time hiccups adversely impacting revenue [2], [3]. Hence, providing not only a low average response time but also a steady tail latency is crucial for cloud providers' commercial success [2].

The question we ask in this paper is what level of performance predictability do industry-leading serverless providers offer? Answering this question requires a benchmarking tool for serverless deployments that can precisely measure latency across a span of load levels, serverless deployment scenarios, and cloud providers.

While several serverless benchmarking tools exist, we find that they all come with significant drawbacks. Prior works have characterized the throughput, latency, and application characteristics of several serverless applications in different serverless clouds; however, these works lack comprehensive tail latency analysis [4]–[9]. These works also do not study the underlying factors that are responsible for the long tail effects, the one exception being function cold starts, which have been shown to contribute significantly to end-to-end latency in a serverless setting [8], [10], [11].

In this work, we introduce STeLLAR[1], an open-source provider-agnostic benchmarking framework for serverless systems' performance analysis, both end-to-end and per-component. To the best of our knowledge, our framework is the first to address the lack of a toolchain for tail-latency analysis in serverless computing. STeLLAR features a provider-agnostic design that is highly configurable, allowing users to model various aspects of load scenarios and serverless applications (e.g., image size, execution time), and to quantify their implications on the tail latency. Beyond end-to-end benchmarking, the framework supports user-code instrumentation, allowing the accurate measurement of latency contributions from different cloud infrastructure components (e.g., storage accesses within a cross-function data transfer) with minimal instrumentation effort.

Using STeLLAR, we study the serverless offerings of three leading cloud providers, namely AWS Lambda, Google Cloud Functions, and Azure Functions. We configure STeLLAR to pinpoint the inherent causes of latency variability inside cloud infrastructure components, including function instances, storage, and the cluster scheduler. With STeLLAR, we also assess the delays induced by data communication and bursty traffic and their impact on the tail latency.

Our analysis reveals that storage accesses and bursty function invocations are the key factors that cause latency variability in today's serverless systems. Storage accesses include the retrieval of function images during the function instance start-up as well as inter-function data communication that happens via a storage service. Bursty traffic stresses the serverless infrastructure by necessitating rapidly scaling up the number of function instances, thus causing a significant increase in both median and tail latency. We also find that the scheduling

[§] These authors contributed equally to this work.

[1] STeLLAR stands for Serverless Tail-Latency Analyzer. The source code is available at https://github.com/ease-lab/STeLLAR.

policy, specifically whether multiple invocations may queue at a single function instance, can significantly increase request completion time by up to two orders of magnitude, particularly for functions with long execution times. Our analysis also reveals factors that, somewhat surprisingly, contribute little to latency variability; one such factor is the choice of language runtime.

II. Background

A. *Serverless Computing Basics*

Over the last decade, serverless has emerged as the next dominant programming paradigm and cloud architecture, known as Function-as-a-Service (FaaS). Serverless offers many advantages over conventional cloud computing, resulting in growing interest among service developers. In this paradigm, cloud service developers compose their applications as a collection of jobs (i.e., functions), connecting function invocations to specific events, e.g., HTTP requests. At the same time, serverless providers are solely responsible for the function execution.

Providers deliver the natural scalability of each deployed function - from zero to, virtually, infinity - charging service developers in a convenient pay-as-you-go manner. Application developers deploy functions by providing function handlers, written in a high-level language of their choice, that providers then integrate into HTTP servers. These HTTP servers, called function instances, will process any invocation for the corresponding functions. The developers pay only for the amount of resources (CPU time and utilized memory) used by the instances of their functions during the processing of function invocations [12], [13]. To achieve function resources scalability, the providers allocate these resources on demand, launching and tearing down function instances in response to load changes. Often packaged as binary archives or container images, both further referred to as function images, function instances can be launched on any node in a serverless cluster, with the cloud provider responsible for instance placement and steering of both requests and data to the instances.

This division of labor, where serverless developers implement functions and providers manage their resources, leads to the following serverless cloud design principles. First, functions are decoupled from cloud resources allocation, which happens at the granularity of function instances. Second, function instances are ephemeral and stateless, enabling processing of any invocation by any instance. Third, instances can run on any node in a serverless cluster, requiring the provider to ship function images and steer requests to the appropriate node.

B. *A Lifecycle of a Function Invocation*

We describe the serverless infrastructure organization (Fig. 1), summarizing available information about the leading serverless provider, AWS Lambda [10], and the state-of-the-art open-source research framework for serverless experimentation, vHive [8]. First, a function invocation, e.g., triggered by an external source like a click, arrives as an RPC or HTTP request at one of the servers of a scale-out *front-end* fleet that authenticates this request and its origin ①. The request is then forwarded to the *load balancer* that routes invocations to physical hosts, called *Workers*, that have instances of the function currently running ②.[2] Request routing is based on the load in front of currently active instances.

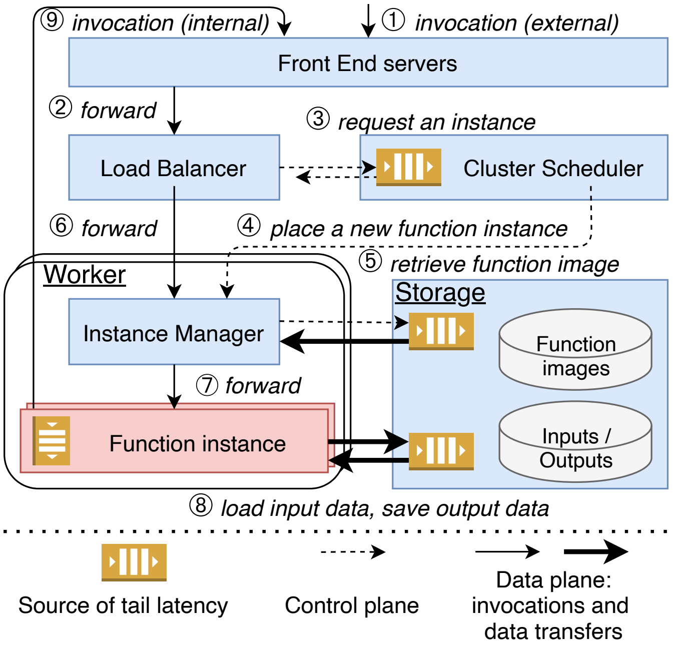


Figure 1: Serverless infrastructure overview.

If all function instances are busy upon an invocation arrival, the load balancer buffers the invocation while asking the *cluster scheduler* to spawn a new instance for the function ③. The scheduler keeps track of the entire cluster resource utilization, which informs decisions regarding function instance placement. Once the scheduler chooses a Worker to run a new function instance, it contacts the Worker's *instance manager* asking to launch a new instance ④.[3] The instance manager retrieves the necessary function state, e.g., a Docker image or an archive with sources, from a storage service and starts the instance ⑤. Note that the instance manager also acts as a part of the invocation data plane, terminating connections to the load balancer and the function instances on the Worker.

When a new instance of the function is ready, the instance manager informs the load balancer, which can then steer the invocation to the instance manager ⑥, which relays it to the instance ⑦. The function then performs language runtime initialization, after which the user-provided code of the function might retrieve the function invocation's inputs, e.g., the output produced by the corresponding caller function ⑧. Finally, the function processes the invocation. During processing, a function may call other functions with inputs that can be transferred inline inside the callee's invocation RPC arguments or by saving

[2] The load-balancer component is referred to as Worker Manager in AWS Lambda [10] and Activator in Knative [14].

[3] The instance manager is referred to as MicroVM Manager in AWS Lambda [10].

the input data in storage, which is required for larger payloads. These *internal* function invocations also need to traverse the front-end and/or the load balancer (9), effectively repeating the whole aforementioned procedure.

III. Sources of Tail Latency in Serverless Clouds

Serverless infrastructures aim to deliver a high quality of service to the majority of cloud application users. Specifically, similarly to conventional clouds, serverless infrastructures strive to minimize tail latency. Providing tail latency guarantees is a hard task for serverless clouds while delivering continuous scaling of function instances and given the necessity of running these instances on any node in a serverless cluster.

Fig. 1 shows the components that could potentially become sources of tail latency. These are: the function instances themselves, the storage services used by both the instance manager and the user code, and the cluster scheduler. For each of these components, we identify low-level application characteristics and scenarios that could hurt tail latency.

First, spawning new function instances induces a significant delay, often referred to as a cold start delay in the literature [8], [15], [16]. The key factors that contribute to cold start delays include the language runtime (chosen by the application developer), the provider's sandbox technology, and the size of a function image. These factors impact the cold start delays not only directly but also indirectly by interacting with various infrastructure components, like storage services and network switches. For example, interpreted languages, like Python, are known to have higher startup delays compared to compiled languages, like Go. Also, providers use different sandboxes for function instances; e.g., MicroVMs in AWS Lambda [10] and Google Cloud Functions [17], whereas Azure Functions run in containers atop of regular VMs [18]. Finally, large function images can take non-negligible amounts of time when retrieved from storage. Function images are usually resident in low-cost storage that is not optimized for low latency access since the majority of functions are invoked once per hour or less [16].

Second, the functions that transfer large payloads experience delays induced by the involved infrastructure components. Functions can perform data transfers by embedding their payloads inside the invocation RPC, albeit sizes are restricted to 256KB-10MB [19], [20]. For transferring larger payloads, functions have to resort to storage services suffering from the tail-adverse effects of cost-optimized services, similarly to functions that have large function images. Hence, functions that sporadically transfer large amounts of data – both inline and via storage – may encounter significant tail latency bloat.

The cluster scheduler is another important source of tail latency as prior work shows that function invocation traffic can be bursty [21]. Serverless schedulers attempt to right-size the number of active function instances by quickly reacting to changes in the invocation traffic. Prior work showed that most functions are short-running [16], which has the effect of placing a high load on the cluster scheduler which must cope with a flurry of scheduling decisions at small time intervals. One important aspect of the scheduler is the choice of the policy that deals with whether to steer multiple requests in a burst to an existing warm instance or spawn new instances. The trade-off is between inducing queuing at a warm instance or incurring long cold-start delays to avoid queuing.

Take-away: a comprehensive measurement framework for tail latency analysis must stress all the relevant infrastructure components and technologies to pinpoint, assess and compare all tail latency contributors for a given provider.

IV. STeLLAR Design

This work introduces STeLLAR, an open-source, provider-agnostic framework for serverless cloud benchmarking that enables systems researchers and practitioners to conduct comprehensive performance analysis. STeLLAR is highly configurable, allowing users to model specific load scenarios and selectively stressing distinct cloud infrastructure components. For instance, as we show in this work, STeLLAR not only helps assess the implications of the language runtime on the function startup time, like prior work [5], [22]–[24], but can selectively stress the cluster scheduler by steering frequent bursts of invocations to a set of functions. Also, STeLLAR can stress the storage layer by invoking functions with large image sizes and by configuring data transfer sizes across chains of functions. We envision STeLLAR's users to be able to configure the framework to model other scenarios and stress tests. STeLLAR uses a robust performance measurement methodology (see §V for details), tailored for tail latency analysis, supporting both end-to-end and localized in-depth performance studies.

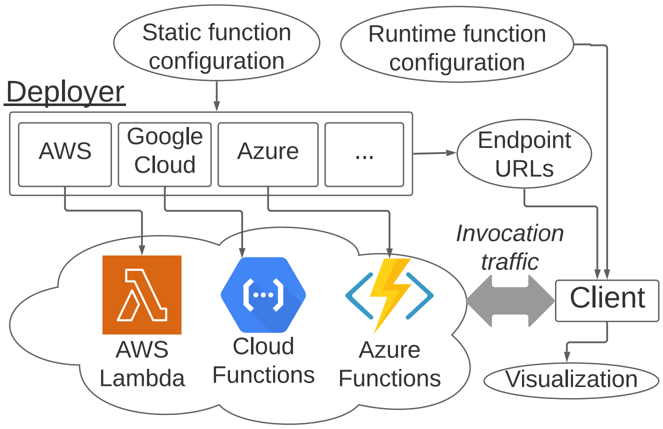


Figure 2: STeLLAR architecture overview.

STeLLAR architecture, shown in Fig. 2, is comprised of two main components, namely *deployer* and *client*. The deployer features a set of provider-specific plugins, each of which are responsible for deploying functions in the corresponding provider's cloud, using a programmatic interface offered by the providers. The deployer's logic is configured via a file with a *static function configuration* that abstracts away the details and terminology of various providers. Using this configuration file, STeLLAR users can define a wide range of static function parameters for each of the deployed functions, as follows:

- Function deployment method; currently, a ZIP archive or a Docker container;
- Maximum memory size of the deployed function's instances;
- A number of identical replicas of the function. This is particularly useful to accelerate cold-start latency measurements by invoking many identical functions in parallel instead of invoking a single function, then waiting for the provider to shut down its corresponding idle instance before repeating the experiment.

Provided with the static configuration file, the deployer automatically configures and deploys the requested set of functions, producing a file that contains a set of *endpoint URLs*, each of which corresponds to a single function and is assigned by the appropriate provider. The static function configuration file may define a function handler's code, a maximum memory size of a function instance, the effective image size, as well the provider the target availability zone. For some providers, the deployer supports several deployment methods, namely ZIP archive-based deployment, which is common to all providers, and a more recent container-based deployment option available in some serverless clouds, e.g., AWS Lambda [25]. With a ZIP deployment, the user needs to wrap their code with its dependencies, via each provider's programmatic or CLI interface used by STeLLAR, in an archive, which can be then deployed at the target availability zone. The container based deployment is done with the regular toolchain provided by Docker that builds a container image using a Dockerfile. The users can configure the effective function image size by instructing the deployer to add a random-content file to the corresponding ZIP archive or Docker image. Currently, the STeLLAR deployer automates deployment in AWS Lambda whereas deploying functions in Azure and Google clouds needs to be done manually.

After configuring and uploading a set of functions using the deployer, STeLLAR can drive the load to these functions, measuring their response time and visualizing the measurements with a set of plotting utilities. The client is provider-agnostic, generating function invocation traffic as HTTP requests to the endpoints that are defined in the file produced by the deployer component. The clients invokes functions from the file with the endpoints' URLs in a round-robin fashion, calling them one after another according to the specified IAT. To send a request to a deployed function, the client spawns a goroutine that sends an HTTP request to the function's URL, blocking till the function's response arrives. For each of the requests, its goroutine measures the latency between the time when the request was issued and the time when the corresponding response was received. The measurements are then aggregated in a single file for further data analysis and visualization.

The client supports further customization with a *runtime configuration file* where users may specify a number of runtime parameters, including:

- An arbitrary mix of deployed functions;
- Inter-arrival time (IAT) distribution of the function invocation traffic, with a fixed, stochastic, or bursty distribution;
- Function execution time;
- Chain length that is the number of functions in a *chain*, where each preceding function invokes its following function while transferring a payload of a configurable size;
- The type of data transfer to use for chained functions; currently, (1) inline transfers, and (2) transfers via a storage service (AWS S3 and Google Cloud Storage) are supported.

STeLLAR lets the users specify the IAT distribution (e.g., round-robin across all deployed functions) along with the number of requests issued in each step, i.e., the burst size, which is essential to evaluate cloud infrastructure efficiency in the presence of bursty traffic. Also, STeLLAR users can specify the execution time of a function with a busy-spin loop of a configurable duration. Other parameters define the data transfer behavior across chains of functions, where each function calls the next function in the chain and waits for its response before returning. The users can specify the transport for data transfers (inline arguments inside the invocation HTTP requests or a cloud storage service) and the length of function chains.

STeLLAR supports intra-function instrumentation by adding calls into Go's native Time module. For example, to capture the data transfer delays, we add a timestamp in the producer function before saving a payload to a storage service (e.g., AWS S3) and another timestamp in the consumer function after retrieving the payload from the storage. The functions' code concatenates these timestamps and passes them to STeLLAR's client as a string. The overhead of this instrumentation is sub-microsecond, as Go relies on Linux' `clock_gettime()` whose overhead has been measured to be within 30ns [26], [27].

Finally, STeLLAR can visualize latency measurements as a cumulative distribution function or as latency percentiles as a function of one of the serverless-function parameters (e.g., the payload size in a data transfer, function image size). The ability to collect both the end-to-end time and the internal timestamps, e.g., for measuring the data transfer time, allows users to cross-validate their measurements.

V. Methodology

We use STeLLAR to characterize three leading serverless cloud offerings, namely AWS Lambda, Google Cloud Functions, and Azure Functions. In the rest of this section, we discuss different aspects of performance, approaches to function deployment, the configuration of STeLLAR, and the metrics that we focus on.

Factor Analysis Vectors: To assess the impact of each of the identified tail-latency sources, we conduct studies along the following four vectors. First, we evaluate the response time of warm and cold functions under a non-bursty load (i.e., allowing no more than a single outstanding request to each function).[4] Second, we assess the cold function delays that

[4] Here, we call a function *warm* if it has at least one instance online and idle upon a request's arrival, otherwise we refer to the function as a *cold* function.

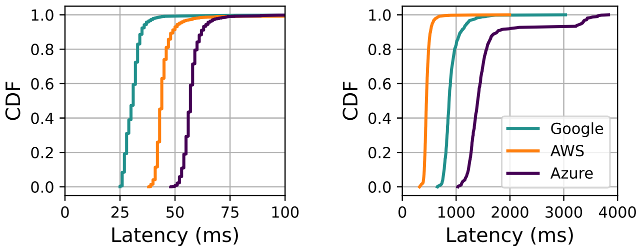
appear on the physical node that hosts these functions, varying the language runtime and the functions' image size. Third, we study the data communication delays for chained functions, where one function transmits a payload of varied size to the second function; we consider two data transfer methods: inline (i.e., inside the HTTP request) and via a cloud storage service. Lastly, we investigate the behavior of serverless clouds in the presence of bursty function invocations, varying the number of requests in a burst (further referred to as the burst size) and their inter-arrival time.

Function Deployment Configuration: We deploy the functions in datacenters located near the western coast of the USA in close proximity to the CloudLab Utah datacenter, where STeLLAR runs. The functions are configured with the maximum memory sizes, which are 1.5GB for Azure and 2GB for AWS and Google for a single CPU core per function instance [19], [20], [28]. These high-memory configurations are not subject to CPU throttling applied by the providers to low-memory instances.

Unless specified otherwise, we deploy all functions using the ZIP-based deployment method, which is supported by all studied providers. We deploy Python 3 functions for all experiments except the function image size (§VI-B2) and the data transfer (§VI-C) experiments. In those experiments, we use Golang functions to minimize the image size and increase the accuracy of the internal timestamp measurements required in the data transfer experiment. To measure the data transfer time in a chain of two functions, the first function records an initial timestamp before storing it in the transferred payload. The payload is then sent using the second function's invocation request or cloud storage. For the data transfer experiments, functions transmitted data via the same-type storage services available in the studied clouds, namely AWS S3 and Google Cloud Storage. Finally, the second function is invoked, after which it loads the data from the request or storage and then records a second timestamp. STeLLAR's client computes the effective transfer time by subtracting these timestamps. All functions return immediately, unless specified otherwise.

STeLLAR Configuration: We run the STeLLAR client on an xl170 node in CloudLab Utah datacenter which features a 10-core Intel Broadwell CPU with 64GB DRAM and a 25Gb NIC [29]. The propagation delays between the STeLLAR client deployment and the AWS, Google, and Azure datacenters in the US West region, as measured by the Linux `ping` utility, are 26, 14, and 32ms, respectively.

In all experiments, unless stated otherwise, functions return immediately with no computational phase. To study warm function invocations, the client invokes each function with a 3-second inter-arrival time (IAT), further referred to as *short* IAT, that statistically ensures that at least one function instance stays alive. To evaluate cold function invocations, the client invokes each function with a 15-min IAT, further referred to as *long* IAT, which was chosen so that the providers shut down idle instances with a likelihood of over 50%.[5] We configure the STeLLAR client to invoke each function either with a single request or by issuing a burst of requests simultaneously. A serverless request completes in >20ms, as observed by the client, which means that requests in the same burst create negligible client-side queuing. For each evaluated configuration, STeLLAR collects 3000 latency samples (each request in a burst is one measurement).



(a) Short IAT (b) Long IAT

Figure 3: Latency distributions for functions invoked with short and long inter-arrival times.

In all experiments, to speed up the measurements, we deploy a set of identical independent functions that the client invokes in a round-robin fashion, ensuring no client-side contention. For example, to benchmark cold functions, we deploy over 100 functions, each of which is invoked with a fixed IAT.

Latency and Bandwidth Metrics: We compare the studied cloud providers using several metrics that include the median response time, the 99-th percentile (further referred to as the tail latency), and the *tail-to-median ratio* (TMR) that we define as the 99-th percentile normalized to the median. Both median and tail latencies are reported as observed by the client, i.e., the latencies include the propagation delays between the client deployment and the target cloud datacenters. The TMR metric acts as a measure of predictability which allows the comparison of response time predictability between different providers. We consider a TMR above 10 potentially problematic from a performance predictability perspective. In the data-communication experiments, we estimate the effective data transmission bandwidth as the payload size divided by the median latency of the transfer.

VI. Results

A. Warm Function Invocations

We start by evaluating the response time of functions with warm instances by issuing invocations with a short inter-arrival time (IAT). For this study, at most one invocation to an instance is outstanding at any given time. Fig. 3a shows cumulative distribution functions (CDFs) of the response times as observed by the STeLLAR client.

We note that propagation delays to and from the datacenter (§V) constitute a significant fraction of the latency for warm invocations. Since the propagation delays to the different

[5] We found that AWS Lambda always shuts down idle function instances after 10 minutes of inactivity, which allowed us to speed up experiments on AWS by issuing requests with a long IAT of 10 minutes.

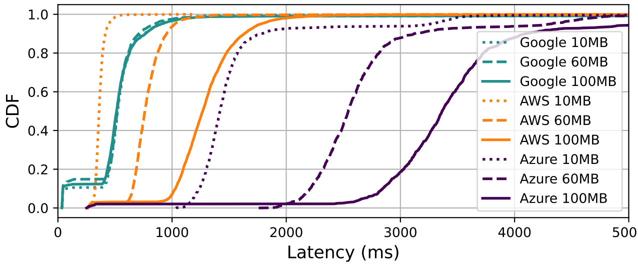


Figure 4: Cold-start latency as a function of the extra random-content file added to the function and provider.

providers' datacenters differ, we subtract them to focus on the intra-datacenter latencies. Note that this is the only place in the paper where we do so; the rest of the results (including Fig. 3a) include the propagation delays.

With propagation delays subtracted, we find that Google has the lowest median latency (17ms), followed by AWS (18ms), and Azure (25ms). In line with having the lowest median latency, Google also displays a lower tail latency (47ms) compared to the other two providers (74ms for AWS, 75ms for Azure). Despite having the highest median latency, Azure yields the lowest TMR of all (1.3), compared to 1.5 for Google and 1.7 for AWS.

Observation 1. *Invocations of warm functions impose low delays and variability, with a median latency of ≤25ms and TMRs <2.*

B. Cold Function Invocations

We perform a factor analysis of cold-start delays, starting from latencies corresponding to baseline invocations followed by an investigation of the effect of function image size, language runtime and function deployment method.

1) Baseline Cold Invocation Latency: We study the response time of cold functions by issuing invocations with the long IAT. As shown in Fig. 3b, the median and tail latencies of cold invocations are, respectively, 10-28× and 9-49× higher than their warm-start counterparts. We observe that the lowest median latency (448ms) and the lowest latency variability (TMR of 1.5) are delivered by AWS. Google ranks in the middle and displays a median latency of 870ms and a TMR of 1.8. Finally, Azure shows the highest median latency of 1401ms with the highest variability (TMR of 2.6).

2) Impact of Function Image Size: In this experiment, we assess the impact of the image size on the median and tail response times, by adding an extra random-content file to each image. We only consider ZIP-based images as these are supported by all three of our studied cloud providers.

Results are shown in Fig. 4. We observe that Google is not sensitive to the image size, with near-identical CDFs for images with added 10MB and 100MB files. It is possible that Google allows retrieving images at much higher network and/or storage bandwidth than its competitors, while masking the image retrieval with other cold-start related operations.

In contrast, AWS show considerable sensitivity to the image size: increasing the added file's size from 10MB to 100MB results in 3.5× and 4.2× increase of the median and the tail latency, respectively. Azure exhibits even higher sensitivity to the image size, with the median and tail latencies increasing by 2.4× and 1.6× when adding 100MB to the image, compared to the response time of the functions with added 10MB. For functions with larger images, the tail latency may reach 2155ms in AWS and 5723ms in Azure. However, for both AWS and Azure, the usage of large function images has a moderate impact on latency variability, with a TMR of 1.7. In contrast to AWS and Azure, Google shows considerably lower sensitivity to varying the image size. Also, for large image sizes, e.g., with a 100MB file, Azure exhibits lower median latencies (approx. 510ms) than its competitors albeit showing higher latency variability, with a TMR of 3.6.

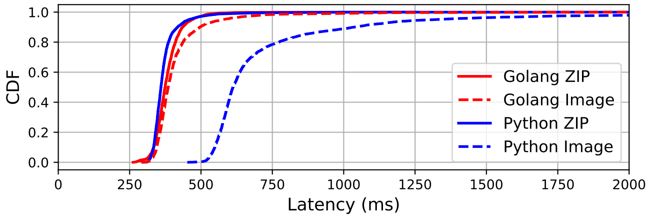


Figure 5: Cold-start latency distributions in AWS for different language runtimes and deployment methods.

Observation 2. *Invocations of cold functions may impose large delays of up to several seconds to the median response time, particularly for functions that have large images. However, variability of cold-starts is moderate, with TMR of <3.6.*

3) Impacts of Deployment Method and Language Runtime: We next study the implications of different deployment methods and language runtimes. Deployment methods refer to how a developer packages and deploys their functions, which also affects the way in which serverless infrastructures store and load a function image when an instance is cold booted. We study the two deployment methods that are in common use today: (1) ZIP archive, and (2) container-based image. With respect to language runtime, our intent is not to evaluate all possible options, but rather to focus on two fundamental classes of runtimes: compiled and interpreted. To that end, we study functions written in Python 3 (interpreted) and Golang 1.13 (compiled) deployed via ZIP and container-based images. This study is performed exclusively using AWS Lambda because, at the time of this paper's submission, Google Cloud Functions did not support container-based deployment and Azure Functions did not support Golang.

The results of the study are shown in Fig. 5, from which we draw three observations. First, for ZIP-based deployment, both Golang and Python's CDFs nearly overlap, showing median and tail latencies of 360ms and 570ms, respectively. This result demonstrates that the choice of a language runtime has negligible implications for cold-start delays. The result is surprising in that it contradicts academic works that showed that the choice of a runtime impacts cold-start latency [4], [15], [24]. However, these academic works did not study production clouds, instead focusing on in-house or open-source systems. Thus, we hypothesize that academic systems may lack important optimization employed by production systems, such as having a pool of warm generic function instances [10]. We call attention to this issue as one that requires further study.

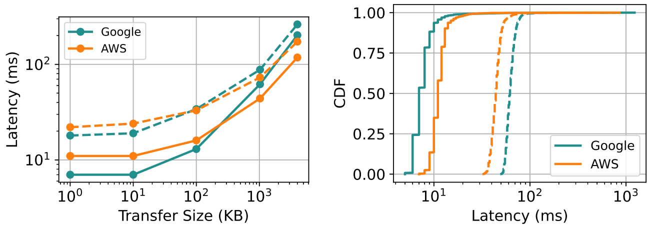
Our second observation is that the latency CDFs of Python and Golang runtimes with container-based deployment differ significantly. Python functions show considerably higher median and tail latency of 612ms and 2882ms, respectively, whereas for Golang, the latency CDF of container-based deployment is close to that of ZIP-based one. One possible explanation of this phenomenon is that a Golang program is compiled as a static binary, suggesting that both ZIP and container image comprise the same binaries that are likely to be stored in the same storage service. Meanwhile, for Python, a container-based deployment shows higher median and tail latencies, compared to the corresponding ZIP deployment. We attribute this behavior to the fact that Python imports modules dynamically, requiring on-demand accesses to multiple distinct files in the function image. When combined with a container-based deployment method, we hypothesize that this results in multiple accesses to the function image storage, since containers support splintering and on-demand loading of image chunks [30]. The additional accesses to the image store would explain the high cold start time and latency variability for Python container-based deployments.

Third, we characterize the latency variability induced by the language runtime and the deployment method selection. We observe that Golang ZIP and Python ZIP-based functions show similar TMRs of 1.5 and 1.7, indicating little impact on the tail latency. Golang container-based deployment has slightly higher variability with a TMR of 2.4. In contrast to Python ZIP deployment, Python container-based functions exhibit much higher latency variability with a TMR of 4.7.

Observation 3. *The choice of the language runtime has low impact on the cold function invocation delay with a <15ms difference between Golang and Python functions' median response time with ZIP deployment. In contrast, the deployment method strongly impacts cold-start delays for functions written in an interpreted language such as Python; compared to ZIP, container-based deployment significantly increases both median and tail latencies by 1.7× and 8.0×, respectively.*

C. Data Transfer Delays

To study the impact of the data-transfer delays on the overall response time tail latency, we deploy a producer-consumer chain of two functions in AWS and Google.[6] The producer function invokes the consumer with an accompanying payload of a specified size. The payload is transmitted in one of two ways: inline or via a storage service. We report the latency from the start of the payload transmission, including the consumer function invocation time, to the point when the payload is retrieved by the consumer function.



(a) Median (solid) and tail (dashed) latencies

(b) Latencies for 10KB (solid) and 1MB (dashed) payload transfers

Figure 6: Inline data transfer latency as a function of payload size. Note that both axes in (a) and the X-axis in (b) are logarithmic.

1) Inline Transfers: First, we investigate the transmission time of transferring the payloads inline. Note that both providers restrict the maximum HTTP request sizes, and hence the inline payload size, to 6 and 10MB in AWS [19] and Google [20], respectively. For larger data transfers, application developers must resort to storage-based transfers, discussed below.

Fig. 6a shows the median and the tail transmission latency of inline transfers. Google delivers the lowest median latency for small payloads (1-10KB), completing the transfer 1.6× faster than AWS. E.g., a 1KB transfer completes in 7ms in Google vs 11ms in AWS.

In contrast to small-payload transfers, which complete relative quickly, transferring large payloads may take hundreds of milliseconds. For instance, AWS and Google complete a 4MB transfer with median latencies of 124ms and 202ms, and TMRs of 1.4 and 1.3, respectively. For functions that run for less than 10 seconds, which account for >70% of all functions as reported in Azure Functions' trace [16], such data transfer overheads might be prohibitively high.

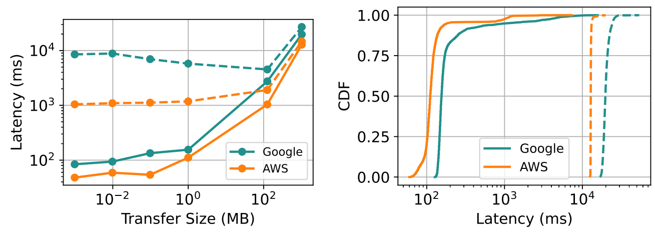
Next, we compare data transfer time variability in AWS and Google (Fig. 6b). For both providers, the variability is low, with TMRs of 1.7 and 1.4, respectively. With such low TMRs, we find that inline transfers have a fairly low impact on tail latency compared to other sources of variability.

Finally, we study the effective bandwidth of inline data transfers, which we compute by dividing the payload size by the observed median transmission time. We find that AWS and Google functions deliver a relatively meager 264Mb/s and 152Mb/s of bandwidth, respectively. This bandwidth is significantly lower than the bandwidth of commodity datacenter network cards (e.g., 10-100Gb/s in non-virtualized AWS EC2 instances [31]).

2) Storage-based Transfers: First, we evaluate the latency and the effective-bandwidth characteristics of storage-based transfers in AWS and Google, sweeping the size of the transmitted payloads from 1KB to 1GB. Fig. 7a demonstrates that the lowest median latency is delivered by AWS. For instance, a 1MB payload transfer completes 1.4× faster in AWS than in Google (111ms in AWS vs 155ms in Google).

Second, we investigate the effective transmission bandwidth of storage-based transfers and compare it to the bandwidth

[6] At the time of this writing, Azure Functions did not support Go runtime.



(a) Median (solid) and tail (dashed) latencies

(b) Latencies for 1MB (solid) and 1GB (dashed) payload transfers

Figure 7: Storage-based data transfer latency as a function of payload size. Note that both axes in (a) and the X-axis in (b) are logarithmic.

that we measure for inline transfers (§VI-C1). We observe that storage-based transfers provide significantly larger effective bandwidth than the corresponding inline transfers. For example, 1MB transfers between functions in AWS and Google yield 72Mb/s and 48Mb/s, respectively. The achieved bandwidth is much higher for >100MB payloads, reaching up to 960Mb/s and 408Mb/s for AWS and Google, respectively. Despite the higher bandwidth achieved by large transferred sizes, it is still more than an order of magnitude lower than what a low-end commodity 10Gb NIC can offer.

Finally, we assess how storage transfers contribute to latency variability in serverless. Fig. 7b shows that storage-based transfer delays exhibit large tail latency. For instance, when transferring 1MB of data, the tail latency is 1177ms in AWS and 5781ms in Google, with corresponding TMRs of 10.6 and 37.3 in AWS and Google, respectively. In contrast, transferring 1MB inside function invocation requests, i.e., inline, yields much lower TMRs of 1.7 in AWS and 1.4 in Google (§VI-C1).

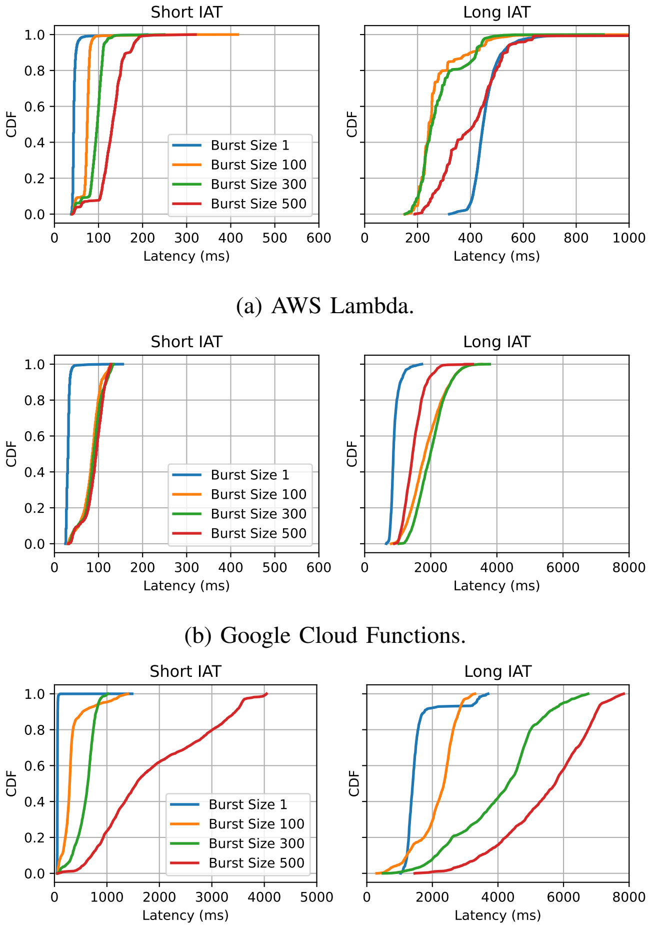
We speculate that the high variability of storage transfers is due to the fact that storage services, by design, optimize for cost rather than performance. With the lack of a fast and cheap communication alternative for large payload transfers, we identify storage as one of the key contributors to the overall response time and performance variability in serverless.

Observation 4. *Storage-based data transfers significantly contribute to both median and tail latencies. E.g., for a 1MB transfer in Google, these delays result in 155ms median and 5774ms tail latencies, yielding a high TMR of 37.3. In contrast, inline transfers are fast and predictable: e.g., for a 1MB transfer in Google, these delays result in 62ms median and 88ms tail latencies with a much lower TMR of 1.4.*

D. Bursty Invocations

We study the response time of functions in the presence of bursty invocations and assess the impact of the scheduling policy on request completion time.

Fig. 8 shows the response time for requests arriving in bursts with short and long IATs, corresponding to (mostly) warm and (mostly) cold invocations. We observe that the burst size, i.e., the number of requests sent in a single burst, impacts both median and tail latency characteristics for all three providers. However, different providers exhibit different degrees of sensitivity to the burst size. Note that a burst size of 1 corresponds to a single invocation (Fig. 3).



(a) AWS Lambda.

(b) Google Cloud Functions.

(c) Azure Functions.

Figure 8: Latency CDFs for short and long IATs for different burst sizes. Note that X-axes vary across graphs.

1) Bursty Invocations with Short IAT: Fig. 8 (left subfigures) plot the latency CDFs of the three providers when bursts are issued with short IATs. We observe that all providers exhibit the similar behavior: serving larger bursts leads to an increase in both median and tail latencies. Azure displays the highest sensitivity to the burst size: increasing the burst size from 1 to 500 leads to an increase in both median and tail latencies by 33.4× and 98.5×, respectively. Noticeably, Google shows the lowest sensitivity to increasing the burst size from 100 to 500, with the median latencies being within 15ms for different burst sizes (the tail latencies are within 50ms).

We next compare latency variability across the three providers using a burst size of 100 as a base for comparisons. We observe that Google shows the lowest variability, followed by AWS, and Azure with TMRs of 1.7, 6.2, and 7.9, respectively. Increasing the burst size from 100 to 500 results in lower variability for AWS and Azure, with TMRs of 4.4 and 3.9, but slightly goes up for Google, with a TMR of 1.9.

2) Bursty Invocations with Long IAT: Next, we compare invocations in the presence of bursts with long IAT. Results are plotted in the right-hand subfigures of Fig. 8. We find that different providers exhibit different behavior as burst size increases.

For AWS, increasing the burst size from 1 to 100 results in 1.8× and 1.3× decrease of the median and tail latencies, respectively. This latency reduction suggests that AWS optimizes retrieval of function images from storage, possibly by employing an in-memory storage-side caching. While increasing the burst size from 100 to 300 requests results in minimal changes, within 12%, in both median and tail latencies in AWS, we observe that when serving a burst of even 500 requests, both median and tail latencies continue to be lower than for an individual request.

Google's median and tail latencies in the presence of bursts are higher than in the presence of individual requests (i.e., burst size of 1). For instance, the median and tail latencies is 870ms and 1567ms for a burst size of 1 vs. 1818ms and 3095ms for a burst size of 100, respectively. Increasing the burst size to 300 results in further increase of both median and tail latencies Interestingly, increasing the burst size to 500 results in a reduction of both median and tail latencies. We hypothesize that this behavior might be attributed to the effects coming from the function image storage subsystem that might adjust aggressiveness of images caching based on load.

One can also see that AWS and Google functions' response time never drops to the range attributable to warm function invocations, i.e., 25-100ms (Fig. 3a). This suggests that these providers do not allow multiple requests to queue at an already-executing instance, and, instead, a dedicated instance services each and every request in a burst. This corroborates AWS documentation [32]. Azure exhibits a different behavior, as its CDF suggests that such queuing may occur, albeit limited to a very small fraction of requests (<5%).

Azure functions show that both median and tail latencies significantly increase when increasing the burst size. For instance, increasing the burst size from 1 to 500 increases the median and the tail latencies by 4.1× (i.e., by 4344ms) and 2.1× (i.e., by 4037ms).

Finally, we note that all three providers have low latency variability for bursts with long IAT. For the burst size of 100, AWS shows the highest variability with a TMR of 2.2. Meanwhile, Google and Azure enjoy lower TMRs of 1.7 and 1.4, respectively.

Observation 5. *For bursts arriving with a short IAT, two out of the three providers experience a moderate increase in the median latency by 3.1-3.3× and the tail latency by 4.2-8.4×, compared to to serving individual invocations. The third provider exhibits higher sensitivity with its median and tail latencies increasing by 33.5× and 98.5×, respectively. Meanwhile latency variability is moderate for all providers, with TMRs <7.9.*

Observation 6. *For bursts arriving with a long IAT, all providers show moderate latency variability with TMRs of 1.3-2.6. Despite that, the median and the tail latencies of two out of three providers increase by up to 740ms and 4344ms.*

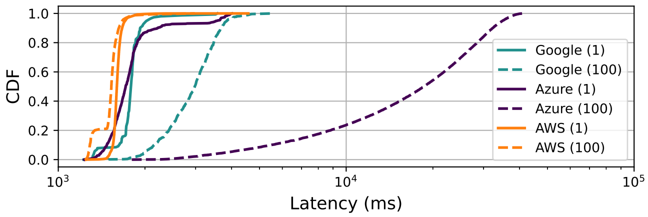


Figure 9: Latency CDFs for different request burst sizes, arriving with a long IAT, for the functions with 1-second long execution time.

3) Implications of Scheduling Policy: In this experiment, we study the scheduling policies that different providers apply in the presence of bursty invocations with a long IAT. In contrast to the previous study, where functions responded immediately (i.e., took no time other than to generate the response), here we deploy functions with an execution time of 1 second. Our goal is to understand whether providers allow concurrent requests to queue at an active instance to alleviate the lengthy cold-start delays. We chose the function execution time to be 1 second as it exceeds the median cold-start delays for all providers, as shown in §VI-B1. Moreover, the traces released by Microsoft Azure show that 50% of functions run for 1 second, on average [16]. Intuitively, with the function execution time being longer than the cold-start latency, a scheduling policy optimized exclusively for performance would cold-start a new instance for each request in a burst instead of allowing multiple requests to queue at an existing instance.

We perform the experiment with burst sizes of 1 and 100. Results are shown in Fig. 9. First, we observe that for non-bursty execution (i.e., a burst size of 1), CDFs for all providers are close to each other, as there is no potential for queuing. For bursty execution, the providers exhibit dramatically different behavior from each other. For instance, for burst sizes of 1 and 100, we observe that AWS demonstrates nearly identical latency CDFs with median and tail latencies of 1598ms and 1865ms, respectively. As both of these latencies are below 2 seconds, it is clear that all requests execute on separate instances, and no request waits for another request, which is in line with the observation we made in §VI-D2.

In contrast, Google delivers median and tail latencies of 2978ms and 4595ms, respectively, indicating that up to four requests may queue at one function instance. Meanwhile, Azure demonstrates median and tail latencies of 18637ms and 38545ms, respectively, showing that more than 30% of requests in a burst may be executed by the same instance.

While it is difficult to ascertain that either Google or Azure do, in fact, allow requests to queue at an active instance, the results certainly suggest that. Indeed, doing so would be a sensible policy, particularly for shorter functions, aimed at striking a balance between function execution time and resource utilization in terms of the number of active instances. Both policies (i.e., allowing queuing or not) have pros and cons,

	AWS		Google		Azure	
Factor	MR	TR	MR	TR	MR	TR
Base warm (§VI-A)	1	2	1	2	1	1
Base cold (§VI-B1)	10	15	28	50	25	64
Image size, 100MB (§VI-B2)	29	49	17	60	59	100
Inline transfer (§VI-C1)	1	2	2	3	n/a	n/a
Storage transfer (§VI-C2)	3	27	5	187	n/a	n/a
Bursty warm (§VI-D1)	2	11	3	5	5	41
Bursty cold (§VI-D2)	6	12	59	100	41	58
Bursty long[7](§VI-D3)	12	16	64	102	309	619

Table I: *Median to base median (MR)* and *tail to base median (TR)* metrics per studied tail-latency factor across providers. Cells with MR or TR >10 highlighted in red. In the corresponding rows, the payload size of the transferred data is 1MB (for both inline and storage-based transfers), the burst size is 100 invocations.

which points to a promising optimization space for future research.

Observation 7. *The choice of scheduling policy with respect to whether multiple invocations may queue at a given function instance has dramatic implications on request completion time and resource utilization (i.e., number of active instances). For functions with long execution times, a scheduling policy that allows queuing may increase both median and tail latency by up to two orders of magnitude.*

VII. Discussion

In this section, we first recap our findings by focusing on key sources of execution time variability induced by the serverless infrastructure. We next discuss variability in actual function execution time by analyzing data from a publicly-available trace of serverless invocations in Microsoft Azure.

A. Variability due to Serverless Infrastructure

We summarize our findings in Table I. For each of the factors that we study, we compute two metrics, namely *median to base median ratio (MR)* and *tail to base median ratio (TR)*, which normalize the median and tail delays as induced by the corresponding factor to median latency of an individual warm function invocation. This normalization is done separately for each provider, i.e., the reported median or tail latency for a given experiment with a particular provider is normalized to the median latency of a warm invocation on that provider. We consider an MR or TR above 10 to be potentially problematic as it implies a high degree of variability. Such cells are highlighted in red in Table I.

We identify two trends that are common across the studied providers. First, we find storage to be a key source of long tail effects. Indeed, both cold function invocations, which require accessing the function image from storage, and storage-based data transfers induce high MR (up to 59) and high TR (up to 187). To put these numbers in perspective, a hypothetical warm function with a median execution latency of 20ms would see its median latency skyrocket to 1.18s with MR of 59 and its tail to 3.74s with TR of 187.

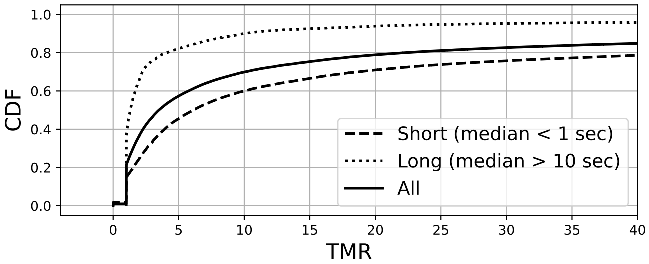


Figure 10: Tail-to-median ratio (TMR) CDFs for per-function execution times, as reported in Azure Function's trace [16].

The second trend we identify is that all studied providers exhibit high sensitivity to bursty traffic, particularly, when bursts arrive with a long IAT (rows "Bursty cold" and "Bursty long" in Table I). While part of the reason for the resulting high latencies can be attributed to storage accesses for cold invocations, we note that the scheduling policy also seems to play a significant role. For functions with a long execution duration (1s, in our experiments), if requests to a function are allowed to queue at an active instance, we observe MR and TR of 309 and 619, respectively.

B. Variability in Function Execution Time

We ask the question of how the variability induced by serverless infrastructure compares to the variability in function execution time, i.e., the useful work performed by functions. Given the many options for the choice of implementation language, the numerous ways for breaking up a given functionality into one or more functions, the actual work performed by each function and other effects that determine function execution time, we do not attempt to characterize the execution-time variability on our own. Instead, we use a publicly-available trace from Azure Functions that captures the distribution of function execution times as a collection of percentiles [16], including a 99$^{\text{-th}}$ percentile and a median, allowing us to compute the tail-to-mean ratio (TMR) for each function.

For each function, the trace captures the time between the function starting execution until it returns. Even though each function's reported execution time excludes cold-start delays, this measurement may still include some infrastructure delays, e.g., if that function invokes other functions or interacts with a storage service. Hence, the computed TMRs are the *upper bound* for the pure function execution time variability.

Fig. 10 shows the CDF of the TMRs for each of the functions in the trace. We find that 70% of all functions have a TMR less than 10, indicating moderate variability in function execution times. However, other functions exhibit significant variability, roughly in the same range is the variability induced by storage-based transfers which have a TMR of between 10.6 and 37.3. We observe that these conclusions generally stand for both short- and long-running functions captured in the trace; however, short functions exhibit higher variability in their execution time. Thus, only 60% of the functions that run for less than a second have

[7] We subtract the 1s function execution time from the measured latencies to account only for infrastructure and queuing delays in order to compute the MR and TR metrics.

a TMR of less than 10; meanwhile, 90% of the functions that run for more than ten seconds have a sub-10 TMR.

VIII. Related Work

Prior work includes a number of benchmarking frameworks and suites for end-to-end analysis of various serverless clouds. FaaSDom [7], SebS [5], and BeFaaS [6] introduce automated deployment and benchmarking platforms, along with a number of serverless applications as benchmarks, supporting many runtimes and providers. ServerlessBench [4] and FunctionBench [22], [33] present collections of microbenchmarks and real-world workloads for performance and cost analysis of various clouds [4], [22], [33]. In contrast, STeLLAR stresses the components of serverless clouds to pinpoint their implications on tail latency whereas the prior works focus on measuring performance of distinct applications or evaluate the efficiency of certain serverless test cases, e.g., invoking a chain of functions or concurrently launching function instances.

Another body of works study the performance of particular components of serverless systems. Wang et al. conducted one of the first comprehensive studies of production clouds [34], investigating a wide range of aspects, including cold start delays for different runtimes.While we analyze many more tail latency factors, we also find that some of their results in 2018 are now obsolete, e.g., in contrast to their findings, we show that the choice of runtime minimally affects the tail latency in AWS (§VI-B2). vHive is a framework for serverless experimentation and explores the cold-start delays of MicroVM snapshotting techniques [8]. Li et al. studies the throughput of the cluster infrastructure of open-source FaaS platforms in the presence of concurrent function invocations [35]. Hellerstein et al. analyzes the existing I/O bottlenecks in modern serverless systems [36]. FaaSProfiler conducts microarchitectural analysis of serverless hosts [9].

Other works investigate the efficiency of serverless systems for different classes workloads, namely ML training [37], latency-critical microservices [38], data-intensive applications [39]–[41], and confidential computations [42]. Eismann et al. categorizes open-source serverless applications according to their non-performance characteristics [43]. Shahrad et al. analyzes invocation frequency and execution time distributions of applications in Azure Functions and explores the design space of function instance keep-alive policies [16].

IX. Conclusion

Over the last decade, serverless computing has seen wide adoption by cloud service developers, attracted by its fast time to market, pay-as-you-go pricing model, and built-in scalability. Composing their services as a collection of short-running stateless functions, service developers offload infrastructure management entirely to cloud providers. This role separation challenges the cloud infrastructure that must deliver low response time to most of its customers. Hence, measuring and analyzing tail latency and its sources is crucial when designing latency-critical cloud applications. To the best of our knowledge, STeLLAR is the first open-source provider-agnostic benchmarking framework that enables tail-latency analysis of serverless systems, allowing to study performance both end-to-end and per-component. By design, STeLLAR is highly configurable and can model various load scenarios and vary the characteristics of serverless applications, selectively stressing various components of serverless infrastructure. Using STeLLAR, we perform a comprehensive analysis of tail latency characteristics of three leading serverless clouds and show that storage accesses and bursty traffic of function invocations are the largest contributors to latency variability in modern serverless systems. We also find that some of the important factors, like the choice of language runtime, have a minor impact on tail latency.

Acknowledgment

The authors thank the anonymous reviewers and the paper's shepherd, Trevor E. Carlson, as well as the members of the EASE Lab at the University of Edinburgh for the fruitful discussions and for their valuable feedback on this work. We are grateful to Michal Baczun for helping with the experiment setup. This research was generously supported by the Arm Center of Excellence at the University of Edinburgh and by EASE Lab's industry partners and donors: ARM, Facebook, Google, Huawei and Microsoft.

References

[1] Markets and Markets, "Serverless Architecture Market - Global Forecast to 2025," available at https://www.marketsandmarkets.com/Market-Reports/serverless-architecture-market-64917099.html.

[2] J. Dean and L. A. Barroso, "The Tail at Scale." *Commun. ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[3] ComputerWeekly.com, "Storage: How Tail Latency Impacts Customer-facing Applications," available at https://www.computerweekly.com/opinion/Storage-How-tail-latency-impacts-customer-facing-applications.

[4] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, "Characterizing Serverless Platforms with ServerlessBench." in *Proceedings of the 2020 ACM Symposium on Cloud Computing (SOCC)*, 2020, pp. 30–44.

[5] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, "SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing." *CoRR*, vol. abs/2012.14132, 2020.

[6] M. Grambow, T. Pfandzelter, L. Burchard, C. Schubert, M. Zhao, and D. Bermbach, "BeFaaS: An Application-Centric Benchmarking Framework for FaaS Platforms." *CoRR*, vol. abs/2102.12770, 2021.

[7] P. Maissen, P. Felber, P. G. Kropf, and V. Schiavoni, "FaaSdom: A Benchmark Suite for Serverless Computing." in *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems (DEBS)*, 2020, pp. 73–84.

[8] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, analysis, and optimization of serverless function snapshots." in *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVI)*, 2021, pp. 559–572.

[9] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural Implications of Function-as-a-Service Computing." in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019, pp. 1063–1075.

[10] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight Virtualization for Serverless Applications." in *Proceedings of the 17th Symposium on Networked Systems Design and Implementation (NSDI)*, 2020, pp. 419–434.

[11] M. Brooker, A. C. Catangiu, M. Danilov, A. Graf, C. MacCárthaigh, and A. Sandu, "Restoring Uniqueness in MicroVM Snapshots." *CoRR*, vol. abs/2102.12892, 2021.

[12] Amazon, "AWS Lambda Pricing," available at https://aws.amazon.com/lambda/pricing.

[13] Google, "Cloud Functions Pricing," available at https://cloud.google.com/functions/pricing.

[14] The Knative Authors, "Knative," available at https://knative.dev.

[15] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, "Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting." in *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXV)*, 2020, pp. 467–481.

[16] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider." in *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*, 2020, pp. 205–218.

[17] Google, "gVisor," available at https://gvisor.dev.

[18] P. Litvak, "How We Escaped Docker in Azure Functions," available at https://www.intezer.com/blog/research/how-we-escaped-docker-in-azure-functions.

[19] Amazon, "AWS Lambda Quotas," available at https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html.

[20] Google, "Google Cloud Functions Quotas," available at https://cloud.google.com/functions/quotas.

[21] A. Wang, S. Chang, H. Tian, H. Wang, H. Yang, H. Li, R. Du, and Y. Cheng, "FaaSNet: Scalable and Fast Provisioning of Custom Serverless Container Runtimes at Alibaba Cloud Function Compute," in *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*, 2021, pp. 443–457.

[22] J. Kim and K. Lee, "FunctionBench: A Suite of Workloads for Serverless Cloud Function Service." in *Proceedings of the 12th IEEE International Conference on Cloud Computing (CLOUD)*, 2019, pp. 502–504.

[23] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "SOCK: Rapid Task Provisioning with Serverless-Optimized Containers." in *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, 2018, pp. 57–70.

[24] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: Towards High-Performance Serverless Computing." in *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, 2018, pp. 923–935.

[25] Amazon, "Using Container Images with Lambda," available at https://docs.aws.amazon.com/lambda/latest/dg/lambda-images.html.

[26] Hacker News, "`clock_gettime()` Overhead," available at https://news.ycombinator.com/item?id=18519735.

[27] Cloudflare Blog, "It's Go Time on Linux," available at https://blog.cloudflare.com/its-go-time-on-linux.

[28] Azure Lessons, "How Much Memory Available For Azure Functions," available at https://azurelessons.com/azure-functions-memory-limit/.

[29] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The Design and Operation of CloudLab," in *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019, pp. 1–14.

[30] Marc Brooker at AWS re:Invent 2020, "Deep Dive into AWS Lambda Security: Function Isolation," available at https://www.youtube.com/watch?v=FTwsMYXWGB0&t=782s.

[31] Amazon, "Amazon EC2 Instance Types," available at https://aws.amazon.com/ec2/instance-types.

[32] ——, "AWS Lambda Function Scaling," available at https://docs.aws.amazon.com/lambda/latest/dg/invocation-scaling.html.

[33] J. Kim and K. Lee, "Practical Cloud Workloads for Serverless FaaS." in *Proceedings of the 2019 ACM Symposium on Cloud Computing (SOCC)*, 2019, p. 477.

[34] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. M. Swift, "Peeking Behind the Curtains of Serverless Platforms." in *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, 2018, pp. 133–146.

[35] J. Li, S. G. Kulkarni, K. K. Ramakrishnan, and D. Li, "Analyzing Open-Source Serverless Platforms: Characteristics and Performance." *CoRR*, vol. abs/2106.03601, 2021.

[36] J. M. Hellerstein, J. M. Faleiro, J. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless Computing: One Step Forward, Two Steps Back." in *Proceedings of the 9th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2019.

[37] J. Jiang, S. Gan, Y. Liu, F. Wang, G. Alonso, A. Klimovic, A. Singla, W. Wu, and C. Zhang, "Towards Demystifying Serverless Machine Learning Training." in *SIGMOD Conference*, 2021, pp. 857–871.

[38] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems." in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXIV)*, 2019, pp. 3–18.

[39] A. Klimovic, Y. Wang, C. Kozyrakis, P. Stuedi, J. Pfefferle, and A. Trivedi, "Understanding Ephemeral Storage for Serverless Analytics." in *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, 2018, pp. 789–794.

[40] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic Ephemeral Storage for Serverless Analytics." in *Proceedings of the 13th Symposium on Operating System Design and Implementation (OSDI)*, 2018, pp. 427–444.

[41] F. Romero, G. I. Chaudhry, I. Goiri, P. Gopa, P. Batum, N. J. Yadwadkar, R. Fonseca, C. Kozyrakis, and R. Bianchini, "Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications." *CoRR*, vol. abs/2104.13869, 2021.

[42] M. Li, Y. Xia, and H. Chen, "Confidential Serverless Made Efficient with Plug-in Enclaves," in *Proceedings of the 48th International Symposium on Computer Architecture (ISCA)*, 2021, pp. 14–19.

[43] S. Eismann, J. Scheuner, E. V. Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "Serverless Applications: Why, When, and How?" *IEEE Softw.*, vol. 38, no. 1, pp. 32–39, 2021.