PEPPA: Platform microbEnchmarks—Performance Playground for functions-as-A-service

Alex Wendland
Harvard College
awendland@college.harvard.edu

Athena Kan
Harvard College
athenakan@college.harvard.edu

William Wang
Harvard College
william_wang@college.harvard.edu

ABSTRACT

Serverless computing has risen to popularity over the past few years with the release of AWS Lambda, Google Cloud Functions, and Azure Functions. However, serverless providers offer few guarantees or transparency about performance. We propose a microbenchmark framework that can be used to create a suite of microbenchmarks in order to compare latency and scalability across the various cloud provider offerings. We evaluate cold starts, warm starts, and scalability of warm starts across two different triggers: AWS API Gateway (HTTPS) and AWS Simple Queuing Service (Pub/Sub).

ACM Reference Format:
Alex Wendland, Athena Kan, and William Wang. 2019. PEPPA: Platform microbEnchmarks—Performance Playground for functions-as-A-service. In *Proceedings of Oops, Office of Kohler and Yu's conference (OOKAY'19).* ACM, New York, NY, USA, Article 42, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Serverless computing, also known as Functions-as-a-Service (FaaS), is a type of cloud computing platform that abstracts away server management from tenants. Instead, programmers write event-driven functions and can modify a small set of configurations. The FaaS platform then handles dependencies, compilation, virtual machine configuration, and resource allocation. Moreover, FaaS platforms bill only for the compute time the user consumes, metering by timeslices on the order of 100ms [14]. After a function handles a request, its virtual machine is put to sleep, so users do not pay for idle or unused resources. FaaS platforms therefore offer certain advantages compared to traditional Infrastructure-as-a-Service (IaaS) cloud computing platforms. There are, however, certain limitations of FaaS platforms compared to traditional IaaS cloud computing platforms such as lack of state, run time cutoffs, invocation overhead, and limited logging tools for the user.

Although originally intended for simple workloads, FaaS platforms have since been used by tenants to handle a wide variety of services including video processing, data analytics, and machine learning. Some companies even rely entirely on FaaS to power their backend [27]. Despite the diversity of these workloads, all of them take advantage of one key benefit of FaaS over traditional cloud computing: the scaling of each FaaS deployment is automatically handled by the FaaS cloud provider rather than the tenant. One of the main marketing points of FaaS is that it is able to automatically handle server management so that users do not have to deal with complex resource allocation problems. However, the resulting opaqueness from cloud providers regarding startup latency and scalability makes it difficult for users to compare FaaS platforms to each other. Even when a cloud provider gives scalability guarantees, users have few tools to test that those guarantees are acted upon. This means that users and providers cannot easily compare a FaaS platform to other FaaS offerings or even previous versions of the same platform.

To address these concerns, we introduce a microbenchmarking framework along with a suite of microbenchmarks intended to be generalizable across FaaS providers to measure startup latency and scalability. Our microbenchmarking framework is capable of modeling various workloads in order to test differences in performance for cold and warm start at various degrees of scale. Through preliminary tests on AWS Lambda, we are able to measure cold start, warm start, and scalability. These preliminary results show that our benchmarking framework is able to measure core FaaS metrics, and it is generalizable to other serverless platforms. We hope these results can guide FaaS users in deciding which provider is best for their specific workload.

1.1 Paper Structure

We structure the rest of the paper as follows: section 2 gives an overview of related work and benchmarking best practices, section 3 reviews motivating serverless use cases, section 4.1 outlines the metrics we use to measure serverless platforms, section 4.2 describes our benchmarking framework, section 4.3 discusses performance concerns we addressed, section 4.4 explains the benchmarks we chose to run, section 5 is the evaluation of AWS Lambda with our benchmarking suite, section 6 discusses shortcomings in our research and opportunities for future work, and section 7 wraps up our discussions.

2 RELATED WORK

As serverless computing is such a new field, literature on the topic is still nascent. We expanded our review to include both informal discussions of serverless performance and use cases. We also added formal benchmarking literature in non-serverless fields, such as operating systems.

2.1 Benchmarking Literature

In designing a benchmarking suite, we first conducted a literature review to discover what qualities make a benchmarking suite useful for researchers and developers; below we provide a summary of some of the key research done on common benchmarking fallacies and pitfalls. We also address how our benchmarking framework will address them.

OOKAY'19, May 2019, Cambridge, Massachusetts USA
2019. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$245,260.r
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Benchmarking Crimes [43]: Heisner discusses many of the unintentional mistakes that authors commit when creating their benchmarks. Although we do not evaluate our microbenchmarks using this framework, below we discuss four particularly notable "crimes" that influenced our benchmark design.

First, authors often neglect part of the evaluation space when creating a benchmark. This allows them to demonstrate improvement on one metric, while avoiding results that show degradation on another. With this in mind, we evaluate both the *progressive criterion* in serverless—improved scalability—as well as the *conservative criterion*—degraded request handling rate. We hope to elucidate the trade-offs here that serverless offerings make. Moreover, we provide multiple trigger mechanisms and scale options in order to conduct our benchmarks on as wide a search space as possible.

Second, authors may attempt to make macro claims from microbenchmarks, ie suggesting the microbenchmarks will be representative of performance in practice. While we are putting forth a microbenchmarking suite, we hope to temper our claims about the results while still providing valuable insights to our readers. Primarily, we are focused on the elements on serverless that are the most opaque and difficult to test. We anticipate that users will leverage our benchmarking suite to create additional custom benchmarks for their unique applications and workloads.

Third, authors may not report a sufficient breadth of statistics about, or do not conduct, multiple benchmark trial runs. When averages are reported without metrics such as variance, readers are unable to draw clear conclusions about the results. We address this by constructing benchmarks that produce composable datasets—and therefore don't require composing summary statistics after the fact—or clearly summarizing our benchmark results with complete distribution summaries whenever possible.

Fourth, authors may fail to clearly specify the environment that they are using for the evaluations. One of the primary focuses of our benchmark is providing reproducibility and freshness of results. We structure our framework to allow for low-cost (in both time and money) execution of the suite so that readers can verify the results or produce new results for unexplored environment configurations.

Seven Deadly Sins of Cloud Computing Research [34]: Schwarzkopf et al. identify many unfortunate tendencies in cloud computing research, some of which undoubtedly apply to new serverless offerings. In particular, we hope to provide a benchmarking suite that helps developers avoid three of the evaluation mistakes that Schwarzkopf et al. highlight.

First, Schwarzkopf et al mandate that cloud computing research report results based on multiple benchmark runs and report variance over multiple trials over a reasonable time period. We conduct multiple trials over time, and the automated nature of our benchmark suite allows others to replicate results. We anticipate that performance characteristics will change over time, on short timescales due to tenant resource contention, as well as long timescales as cloud providers opaquely update their serverless implementations.

Second, Scharzkopf et al. note that cloud computing research sometimes operated in isolation; researchers would compare new approaches against traditional, outdated methods, instead of more recent, and harder to beat, proposals. We address this deficit by selecting microbenchmarks and developing a benchmarking framework that can be extended to any new serverless provider. We hope our benchmarking framework will provide a unified, up-to-date overview of the state of the ecosystem.

Finally, Scharzkopf et al. call into question standard assumptions that scalability is infinite in the cloud. We agree that this is infeasible, and therefore seek to explore what the finite bounds on scalability are for serverless offerings. We intend to probe this limit through increasingly large load capacities and monitoring the serverless platform for signs of queuing or failure.

Ten Simple Rules for Reproducible Computational Research [32]: We note that the rules applicable to our microbenchmarks include: tracking how results were gathered, recording all intermediate results, storing raw data, avoiding manual data manipulation, and providing public access. To achieve the latter, we make our benchmarking suite publicly accessible.[1] We address the other requirements in our design section. We have a logging system that records all raw data generated by the benchmarking suite as well a script to generate graph based on that data. This allows our data to be publicly accessible on Github for easy reproduction.

Open Versus Closed: A Cautionary Tale [33]: Schroeder et al. disambiguate two important variants of request-based benchmarking: open versus closed. In closed systems, a specified number of actors performs sequential requests against a system, with new requests coming from an actor only once that actor has finished its previous request. In open systems, new requests arrive independently of request completion. As we will discuss in section five, we chose an open system for our microbenchmarks.

2.2 Serverless Literature

We reviewed available serverless literature to better understand the current deficits and complications in the ecosystem. Because the space is nascent, we also looked at several informal writeups in order to understand the broader serverless developer community.

Peeking Behind the Curtains of Serverless Platforms [44]: Wang et al. explore the implementation, performance, and security characteristics of several major serverless providers: AWS Lambda, Azure Functions, and Google Cloud Functions. They focus on understanding the implementation underlying the serverless platforms, and perform substantial investigation into the co-location of different function instances and the network performance that results from those decisions. Excitingly, Wang et al. noticed a substantial performance change between the initial exploration they conducted and a subsequent review they conducted several months later; for example, Microsoft Azure coldstart latency improved by a factor of fifteen. We hope to create a system that will unveil these types of changes over time in a reliable, ongoing, and easily accessible manner. Unlike Wang et al., we accept the opaqueness of the serverless implementations and instead focus all of our attention on the developer-facing aspects of the serverless platforms.

[1]Source code is available on GitHub at awendland/faas-benchmarking. We will be improving documentation as the interfaces stabilize.

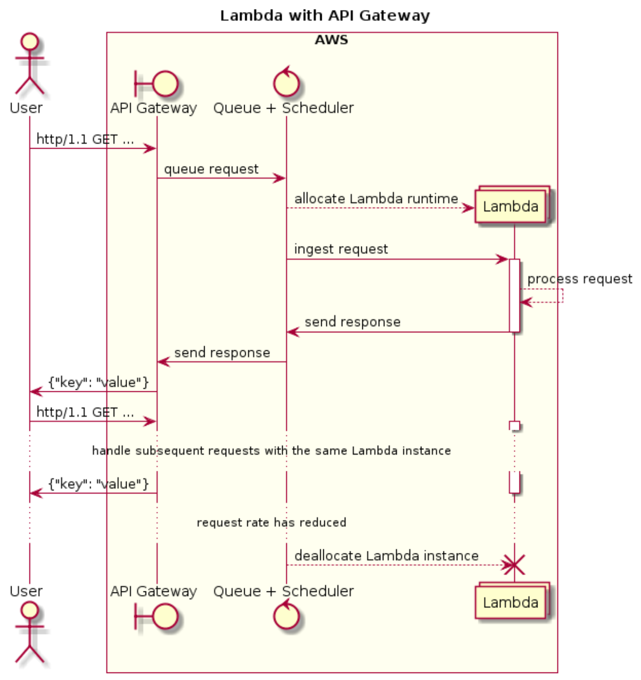

Figure 1: A request to a serverless function will traverse several intermediaries before actually being executed. We focus on the traversal of these intermediaries—such as "queue request," "ingest request," and "send response"—as opposed to the function's actual execution (i.e. "process request").

Cold Start Influencing Factors in Function as a Service [30] and **Performance Evaluation of Heterogeneous Cloud Functions** [24]: The benchmarking suites by Manner et al. and Figiela et al. both are for serverless functions, cover the primary providers (AWS Lambda, Azure Functions, Google Cloud Functions, and IBM OpenWhisk), and attempt to set up a framework for repeatable results. We believe there are deficits in both pursuits. Figiela et al's bechmarking suite consists of an integer-based CPU intensive benchmark, instance lifetime measurement, and data transfer benchmark. It also uses HPL Linpack, a CPU-intensive benchmark that uses a linear system of equations to measure floating point performance and assesses cost and heterogeneity of hardware. While we believe that CPU-intensive benchmarks are useful metrics for users, as we will expand on in section 3, we believe that the primary performance concern with serverless platforms is in request ingestion and response, not in the processing of the request by the function instance (see Figure 1 for an overview of the serverless invocation process). Manner et al. focus on cold starts—similarly to us–however, we believe that isolated cold starts are only one factor in a larger pool of relevant performance characteristics. Therefore, we expand our benchmarking suite to look at not only cold start performance but also warm start request latency and function instance allocation at scale. As we will discuss later, we believe that these metrics present a comprehensive picture for understanding how serverless handles request load.

Serverless Benchmark 2.0 - Part I[42] [41]: Strehl proposes an informal benchmarking suite comparing AWS Lambda, Google Cloud Functions, Azure Functions, IBM Cloud Functions, and Cloudflare Workers. The tests are continuous, measuring overhead and cold start time every hour at various concurrency levels. We augment this suite by adding other important benchmarks and being more rigorous about collecting sufficient data, timing function responses more precisely, and creating a reproducible framework.

CF Serverless: Attempts at a Benchmark for Serverless Computing [28] and **How does language, memory and package size affect cold starts of AWS Lambda?** [23] are another two informal serverless benchmarking attempts. Kaviani and Maximilien provide an initial specification for SPECserverless to add to the SPEC series of benchmarks. The authors suggest comparing workloads that vary in invocation distribution (random, spiked or periodic), size of payload, and concurrency. With these dimensions, they hope to examine performance CPU-heavy jobs, memory-heavy jobs, jobs that rely on databases, and jobs that invoke external network services. As of May 2019, results have not yet been published. Cui examines how language choice, size of the deployment package, and memory allocation affect execution time of the function. However, neither examine cold or warm start latency; metrics we propose to be the main cost introduced by serverless.

3 SERVERLESS USE CASES

We review publicly-documented use cases of serverless platforms to understand how developers leverage them today. Some of these use cases relate directly to serverless computing's proposed strengths—such as quick scalability—however, others perform functions that are traditionally handled by non-serverless platforms and are less-perfect cases for serverless. We summarize these in table 1 and grouped use cases into 4 broad categories:

(1) **Web apps** - these applications perform traditional web servers functionality, such as returning dynamically generated HTML based on a browser request. They are traditionally monolithic applications.
(2) **HTTP backends** - these applications perform simplified web server roles, such as returning a JSON response to a client request. They may be "RESTful" or communicate over some other HTTP RPC.
(3) **DevOps** - these applications perform support functionality for developers or system engineers, and usually do not directly perform client facing functions. For example, they may manage infrastructure deployments or perform code analysis. They may be triggered by internal events or periodically.
(4) **Data processing** - these applications perform ETL (extract, transform, load) operations on data sets. They may be triggered by internal events or periodically.

Jonas et al [26] also presented a categorization for use cases of serverless computing which we believe supports our 4 main groups: 32% of serverless computing use cases are for web and API serving, 21% are for data processing, 17% are for integrating 3rd party services, 16% are for internal tooling, 8% are for chat bots, and 6% are for Internet of Things. Unlike us, they then investigate

Table 1: Collection of publicly-discussed serverless use cases. *Scale* refers to the number of requests received by the system per second (Low ≈ <0.1 rps, Medium ≈ <1 rps, High ≈ >1 rps). *Burstiness* refers to the potential volatility of the request rate. *Latency* refers to how much time the system has to address each request (>1 sec, >500ms, <500ms). Scores are intended for use case summary purposes, not for detail analysis.

Category	Name	Description	Trigger	Scale	Burstiness	Latency
Web apps	Zappa[12]	Converts existing Python websites to run on FaaS	HTTPS	Medium	Medium	High
HTTP Backends	MoonMail[19]	Send infinite emails with FaaS	HTTPS	Medium	High	Low
	Coca-Cola[1]	Vending machine credit-card payments, loyalty programs, etc.	HTTPS	Low	Low	High
	iRobot[7]	Consumer vacuum controls	HTTPS	Medium	Low	Medium
	Hello, Retail![18]	Nordstrom's serverless ecommerce demo	HTTPS	High	High	High
	Bustle[2]	Online lifestyle magazine	HTTPS	Medium	Medium	High
	Reuters[5]	User messaging	HTTPS	Medium	Medium	Low
DevOps	SSH certs[20]	Netflix/Lyft's SSH certificate management	Direct	Low	Low	Low
	Automated recovery[29]	Netflix's infrastructure health-check system	Periodic	Low	Medium	Low
	Serverless Artillery[16]	Nordstrom's high-throughput HTTP load generator	Direct	High	High	High
	Prisma E2E Testing[21]	Parallelized headless browser integration testing	Direct	Medium	Low	Low
	Expedia's CI/CD[13]	Continuous integration/deployment at Expedia	HTTPS	Low	Low	Medium
Data Processing	StreamAlert[35]	AirBnB's real-time log analysis framework	File	High	Low	Low
	MLB's User Metrics[29]	Major League Baseball's user metric processing pipeline	Pub/Sub	High	High	Low
	BinaryAlert[15]	AirBnB's framework for analyzing millions of malware files daily	File	High	Low	Low

specific research projects using serverless computing and chronicle limitations to serverless as a platform in those applications. Several examples that they discuss:

ExCamera [25] uses a framework to orchestrates parallel computations with serverless computing to encode video in real-time. It uses an encoder that can pass named intermediate states to encode small chunks of the videos in a parallelized fashion. It then stitches the chunks together serially. Ultimately, ExCamera performs 60x faster than state-of-the-art encoders with similar bitrates and quality. Numpywren [39] is a system to perform large scale linear algebra with serverless computing. For certain linear algebra computations that are highly paralellizable, numpywren's performance was within 33% of ScaLAPACK's and saved 240% in compute efficiency. Numpywren also demonstrated that a limitation of serverless was inability to use locality across cores in a machine. In particular, they find FaaS is an extremely poor fit for databases and other state-heavy applications.

However, the cloud computing sin addressed by Schwarzkopf et al. most relevant to our benchmarking framework is the requirement that the workload we run through the benchmark is representative of real-world use cases. Though workloads such as ExCamera and Numpywren depend on serverless computing's automatic scaling and fine grained billing, they are highly-stateful and uncommon in enterprise deployments (which makes them interesting for research, but that's for others to address). Instead, most serverless use cases perform simple object fetching, transforming, and sending; with connections to databases, other web APIs, or object stores (like Amazon S3). As discussed by the users or documentation of the applications in 1, these applications are mostly concerned with quick elastic scaling and, in many cases, low end-to-end response times. These systems usually consume serverless functions in a platform agnostic way, such as over HTTP, and therefore must proxy their requests through intermediaries, such as AWS API Gateway[2] for HTTPS. Due to this, in our evaluations we will include the latency from API Gateway, and other triggers, as part of the serverless framework.

4 BENCHMARK

4.1 Benchmark Metrics

We focus on two core metrics in our benchmarking framework, motivated by the needs of the use cases mentioned previously.

4.1.1 Latency. We define the latency of a serverless platform as the time between when we begin sending the request and the time when we receive a response (see 1 for a visual overview). We calculate latency by recording the time when the request was sent and the time when the response was received; this period of time will include:

[2]AWS API Gateway is a load balancer, Layer 7 router, and proxy that translates HTTPS requests to Lambda invocations

- Time required to traverse the network to the serverless function's trigger endpoint
- Time required for the trigger to notify the serverless function scheduler
- Time required to allocate the resources and runtime for the serverless function
- Time required for the function to run
- Time required for the function to send back the response
- (In some cases) Time required for the response to be transformed/proxied to the client

We then remove running time of the function. For our benchmark, we either run a function that returns immediately or a function that sleeps for a pre-determined amount of time before returning. The function response includes a self-reported running time which is removed from the latency calculation. Thus, the three significant portions of latency included in the measurement are network time, trigger activation time, and VM (Virtual Machine) assignment time. We believe all of these factor into a holistic measurement of latency as consumers will necessarily be affected by all of them. We measure two kinds of latency; cold start and warm start latencies. We define a cold start latency as the response latency described above reported by a FaaS instance that had not previously returned a result (i.e. the first response sent by a FaaS instance is categorized as a cold start latency). We define a warm start latency as any response reported by a FaaS instance that had previously returned a result. We expect cold start latencies to be on average greater than warm start latencies as a cold started response will include the time required to provision, boot up, and configure the FaaS instance to handle a request whereas a warm start is sent by a FaaS instance that had already been configured to handle requests.

4.1.2 Scale. We measure scale in two ways. The first way we measure scalability is checking how many FaaS instances are used to handle requests. For this test, we can either maintain a constant request rate or incrementally increase the request rate. In this way, we are able to test a serverless platform under various loads by changing the rate of our requests and the running time of our serverless function (with a sleep as mentioned above). We expect to see an increase in FaaS instances proportional to both the running time of the serverless function and the rate of requests. In fact, AWS Lambda publicly states that the number of FaaS instances should be equal to `(invocations per second)` $*$ `(average execution duration in seconds)` [11]. We aim for our benchmarking framework to support validating this claim.

The second way we will measure scalability is by checking the average latency as we increase the number of requests over the course of the benchmark. This test checks whether or not performance degrades as the number of requests increases. We expect there to be minimal change in latency over time as serverless platforms are designed to handle multiple tenants with our benchmark being just a single user. However, we would be impressed to see if there was lower latency over time as that may indicate that the platform has some kind of optimization in place to pre-emptively scale up.

4.2 Benchmarking Framework

In this section, we describe how our benchmarking framework launches requests for each trigger mechanism. We focused our implementation on AWS Lambda, but we selected approaches that are generalilzable to other serverless platforms as well.[3] We discuss several alternative approaches that we pursued but did not eventually adopt in order to justify the final approach taken by the benchmark framework (uninterested readers can skip sections 4.2.1 and 4.3).

4.2.1 Time Synchronization. A primary concern for our benchmarking framework was how to address time. An ideal approach for benchmarking the overhead of a FaaS system would be to record the time when the client invokes the trigger, and then the time in the FaaS instance when it is subsequently invoked by that trigger event. However, since we are evaluating a distributed system, each FaaS instance is running on a separate, remote, compute instance, and therefore different clocks are used when recording those different times. If the system being timed was on the order of minutes, or even several seconds, the disparity between the two clocks would be negligible. Informal measurements demonstrated that the clocks on the benchmark executor (running on an EC2 instance) and the FaaS instances (running on AWS Lambda) had less than 20ms disparity, however, when cold starts are measured on the order ~200 ms, and warm starts are 10s of ms, this disparity can influence results substantially.

NTP—To reduce this disparity, we investigated several time synchronization protocols. We first explored the Network Time Protocol, NTP [8], as a mechanism for synchronizing time. NTP is a complex algorithm that makes repeated requests to time servers and performs statistical analysis on the results to converge on the "true time" even across WANs. In 10+ minutes, NTP is able to synchronize to within 2ms of a time server's time [22]. NTP is widely deployed, and synchronizes most computer systems today.

Selecting a reliable time server with consistent network latency is the next important step when using NTP. Most major cloud providers have advanced time keeping functionality, leveraging atomic clocks to provide highly accurate time sources with performant access [38]. In our testing, we found that an EC2 instance could query this clock in less than 1ms with negligible network jitter. However, this mechanism imposes new portability constraints on the system, since not all cloud providers offer these time servers. When we tested on other time server offerings, such as the public NTP pool[4] or Canonical's,[5] we found that the best NTP software we had access to[6] was unable to narrow its potential error size to less than 5ms.

Moreover, two other concerns arose with using NTP. Firstly, 10+ minutes convergence is too long for our benchmark suite since it would impose constraints on function execution time and would substantially increase the cost of running our benchmark suite.

[3]In particular, we reviewed the equivalent documentation, service offerings, and SDKs, of Google Cloud Platform and Microsoft Azure to ensure the benchmarking framework's approaches were portable
[4]`pool.ntp.org`
[5]`pool.ubuntu.com`
[6]`chrony`[4] was our software of choice for optimal NTP performance given it's endorsement by Canonical[10], AWS[38], and Redhat[3].

Given reliable time servers and stable network connections, it's likely that this time could be improved, however. Secondly, there were no reputable NTP clients for Node.js,[7] and implementing a performant, spec-compliant, complete NTP solution is not a trivial feat.

SNTP—A simplified version of NTP, called Simple Network Time Protocol, or SNTP, can be leveraged for requesting the time from a networked time server. It leverages the same data format and network protocol as NTP, but it doesn't introduce the statistical calculations and statefulness. SNTP can be used as the basis for custom time synchronization approaches. Several game engines leverage custom SNTP-based implementations that use basic outlier filters and other statistical techniques to quickly converge on shared understandings of time when network latency is consistent [17][31].

However, these implementations also take time to converge and introduce new potential sources of noise in the benchmark suite. Upon consultation with Professor Kohler, we discontinued investigation into distributed time synchronization with the justification that network overhead on the response is an appropriate element of latency that consumers of the benchmark would be curious about.

Cloud Provider Time—Another approach we briefly considered was relying on the cloud provider's logging system to record the time. Serverless providers keep execution logs with millisecond granularity in order to support 100 ms-incremented billing. However, we dismissed this approach for two reasons. Firstly, these logs only cover FaaS instance execution, and do not provide similar precision guarantees for trigger invocations. Secondly, and most importantly, we believe that minimizing our reliance on the cloud provider for any information increases the robustness of our benchmarks.

Single VM Time Keeping—Our current version of the benchmarking framework performs all time recording activity on a single VM. This removes the need to have any sense of "true time", since we are concerned primarily about durations or offsets from a fixed epoch (such as the start of the benchmark's execution). More importantly, this removes the need to have a shared time distributed across all computers participating in the benchmark execution.

In the current version, the VM sending requests to the triggers being tested records all times. The benchmark uses the `Date.now()` in Node.js, which in the Node.js 8.15.1 VM on Linux kernel 4.15.0-1039-aws become `clock_gettime(CLOCK_MONOTONIC, ...)` calls.[8] This call is appropriate for recording elapsed time because it will never jump backwards and returns swiftly.

4.2.2 Trigger Classes. We group request triggers into two classes: synchronous request triggers and asynchronous request triggers. Synchronous requests differ from asynchronous requests in that synchronous requests invoke a FaaS instance as soon as the trigger fires. With asynchronous requests, a scheduler constantly polls the event source to see if there is a trigger event it should respond to. As such, when the trigger fires, there is additional latency before the FaaS instance is invoked. On the AWS platform, the main asynchronous request triggers include SQS, S3, and DynamoDB; the main synchronous request triggers include API Gateway and Kinesis Data Firehose[37].

4.2.3 Synchronous Framework. Setting up and testing synchronous triggers follows a simple high-level model:

(1) Deploy infrastructure to be tested: FaaS configuration (source code, memory size, runtime), trigger configuration (API Gateway)
(2) Send requests to the trigger (HTTPS requests to the new endpoint on API Gateway)
(3) Record the send time and response time of each request
(4) Teardown infrastructure that was tested

This approach ensures a fresh environment for every invocation, and is portable to all serverless providers that we reviewed. However, it is quite slow, with the deployment and teardown steps taking ~2 minutes (combined) while, in the case of cold start testing, the request step takes less than 1 second. In order to improve the framework's test throughput, several test runs are batched together. For a cold start test, 58[9] FaaS configurations are deployed at once and then each FaaS is sent a single, independent, request. Further discussion about exactly how HTTP requests are sent will occur in section 4.3.1.

4.2.4 Asynchronous Framework. Asynchronous triggers require an additional component in the system. Because the FaaS instances are invoked asynchronously, they do not have a mechanism to return a response to the client through the same channel that they were triggered via. For example, in Pub/Sub the client will publish a message, and a FaaS instance will eventually be invoked to process it, but the client will immediately be returned a response from the Pub/Sub framework indicating if the message was published successfully, before the FaaS instance is even invoked.

Therefore, in order to receive responses from the FaaS instances, we create a local server that the FaaS instance can callback to. The callback server runs on the same computer as the request engine so that it shares the same clock. The callback server listens for HTTP requests on a TCP port. The trigger event contains information about the host of the callback server so the FaaS instance can send an HTTP request back to it. The trigger event also contains a unique identifier so that the initial message can be connected to the callback request. Therefore, the high-level process is:

(1) Deploy infrastructure to be tested: FaaS configuration (source code, memory size, runtime), trigger configuration (SQS)
(2) Attach the callback server to an Internet accessible port on the local machine
(3) Send N requests to the trigger (pub/sub messages to the queue in SQS)
(4) Record the send time of each message
(5) Wait until N requests have been received by the callback server, or a timeout period has elapsed
(6) Teardown infrastructure that was tested

[7]Some FaaS providers allow users to execute non-JavaScript code, however, this is not a universal feature and would limit portability.

[8]Following the rabbit trail in the source code takes you to https://github.com/v8/v8/blob/4b9b235/src/base/platform/time.cc#L556, and further strace-ing shows that what the system calls were (most of the system calls in a test Node.js program running HTTP requests in a loop were this `clock_gettime` call).

[9]58 is a limitation imposed by the CloudFormation[36] infrastructure-as-code and serverless[9] framework.

Similar throughput improvements are made to those described for synchronous invocation in section 4.2.3. This approach does add one additional constraint to portability: the serverless provider must allow FaaS instances to communicate outwards over the Internet, however, we expect this to be an existing requirement for most FaaS use cases anyways. Furthermore, this approach adds no additional constraints to serverless infrastructure portability since it only introduces additional components on the local machine.

Our asynchronous benchmark currently focuses on AWS Simple Queue Service. The benchmarking framework send messages to SQS using the AWS SDK for JavaScript, by calling the `sendMessageBatch`[10] function, with a batch size of 10 messages. The SDK translates this call into an HTTPS request to the SQS REST endpoint. The benchmarking framework records the start time for those messages immediately before the `sendMessageBatch` function is called. This approach therefore introduces the overhead of the SDK call into the latency measurements—which includes the SDK processing time as well as the time for the SDK to interact with the remote API—however, we believe this is appropriate to measure since serverless providers frequently provide proprietary triggers that are primarily consumed through SDK interfaces, and any latency they introduce in inefficient SDK implementations will manifest for their users.

4.2.5 Open vs. Closed. We implement all asynchronous benchmarks in an open manner in order to mirror real-world serverless conditions; a key benefit of serverless computing is the promise of infinite, elastic scaling (automatic, quick scaling of new FaaS instances before previous requests have been completed). An open design allows us to incorporate unfavorable scenarios, such as highly bursty request periods. We note that this approach is similar to the approach taken by SPECmail2001, SPECJ2EE, and SPECWeb96. Synchronous benchmarks are implemented in a closed manner due to limitations in our workload generator (though they are practically open for the tests we run). We hope to explore other approaches to maintaining several thousand open TCP sockets at once to allow us to implement the HTTPS benchmark in a fully open manner in the future.

4.2.6 FaaS Function. Each FaaS configuration is deployed with the same source code and Node.js 8.15.1 runtime.[11] The source code has three parts:

- *Initialization* - Immediately upon initialization the time would be recorded, then a unique identifier would be randomly generated (16 bytes of entropy).
- *Trigger entries* - Each trigger type has a specific handler that exposes the appropriate interface for that trigger. These handlers extract appropriate information/parameters from the trigger event and then invoke the core handler. However, before anything else, they first record the time they were triggered.
- *Core handler* - A shared handler is invoked with a normalized set of arguments from any trigger. This will sleep the FaaS instance if requested, send an HTTP request to the callback server, and prepare execution information for analysis. This information includes:
  - *Initialization time*
  - *Trigger time*
  - *FaaS instance ID*
  - *Processing duration* (FaaS instance recorded time between invocation and replying)
  - *Run count* (in-memory counter of the number of times the FaaS instance has run)
  - *Parameters* (sleep duration, callback server host)
  - *FaaS provider info* (provider request ID, version, etc.)

The *processing duration* is set immediately before writing the HTTP request data to the TCP socket, or immediately before the function returns to the FaaS runtime. We evaluated the runtime of the FaaS source code on an m5.xlarge EC2 VM, and found that loading the initialization process takes <10ms (including overhead of reading the file, loading it into memory, and executing it; though not including the time to start a Node.js VM). The handler pipeline, without making an HTTP request or sleeping, takes <0.01ms.[12]

The FaaS instance can perform two operations outside of simply returning execution information to the client:

The FaaS instance can be instructed to *sleep* for a set period of milliseconds. This will idle the Node.js VM until the specified amount of time has elapsed. This feature allows the benchmarking framework to customize the running time of of the FaaS instances and stress the concurrency limits of the FaaS system. We use the *processing duration* to substract the running time of the FaaS instance in order to isolate FaaS overhead. We recognize that a serverless function that does not do any meaningful work is not a realistic use case, however, since we are focused on the non-processing elements of the FaaS platforms this allowed us to remove as many factors as possible while keeping execution times short.

Also, the FaaS instance can be instructed to make an HTTP request to a callback server. The FaaS instance will complete all other processing activities (sans returning to the FaaS runtime) and then open a TCP connection to a specified host and write a JSON string containing the aforementioned informational payload. This request is opened with Nagle's algorithm disabled in order to minimize any delay. Any response to the request is ignored, and the function returns to the FaaS runtime as soon as the request has been sent.

4.3 Eliminating Overhead

We dedicated most of our time on this project to minimizing interference and producing accurate test results. We ran several isolated microbenchmarks of various approaches to ensure we were testing the overhead of the FaaS system, not our benchmarking framework, as much as possible. However, we admit to several initial compromises that were made. Firstly, our benchmarking framework runs

[10]https://docs.aws.amazon.com/AWSJavaScriptSDK/latest/AWS/SQS.html#sendMessageBatch-property

[11]All FaaS providers that we reviewed offered a Node.js runtime (usually this was their first available runtime, before others were released), so therefore we selected it for its high portability. Node.js 8 is an LTS version, and, until very recently, was the most up-to-date version of Node.js available on AWS Lambda (though as of May 15th, 2019 Node.js 10 is available in beta; it is also in beta on GCP; and it is generally available on Azure.

[12]Our test demonstrates that it takes less than 0.01ms, however, this test is running in a tight loop and the Node.js VM may be performing JIT optimizations on the code path that would not be performed if the handler was invoked only once. However, no-loop tests (with a much smaller sample size) indicate that it takes <1ms and is therefore still negligible.

on the Node.js VM, which is not a traditional high performance computing language.[13] Secondly, our benchmarking framework includes network overhead in its latency results. Though we would like to remove these eventually, we believe that they are fine for an initial version of the benchmarks because FaaS users will experience them and FaaS providers do have control over their network performance. Furthermore, to emphasize the latter point, we ran all tests of AWS Lambdas from AWS EC2 machines inside the same AWS region, leaving the network performance entirely in their hands.

We will now discuss various concerns we explored.[14]

4.3.1 HTTP(S) Request Performance. Our initial implementation of the HTTPS request engine leveraged the Node.js library `got`[40], a wrapper around Node.js's `https` sockets[6]. However, time series plots showing when requests were sent out and when responses were received by the client showed two issues: 1.) a large delay between send events and responses, 2.) a peculiar pattern where only 2 to 3 responses were handled in any given millisecond, supporting a maximum rate of 2000 requests per second, a number substantially lower than Node.js HTTP server benchmarks have demonstrated.

We mitigated the initial delay through three mechanisms:

- We opened up TLS connections to the host before making the HTTP requests (therefore removing DNS resolution and TCP/TLS handshake time).
- We reused existing TLS connection between HTTP requests using the `keep-alive` TCP option.
- We disabled Nagle's algorithm, which caused ~100 ms of buffering while waited for additional data to be written to the TCP socket in order to combine requests into fewer packets.

The peculiar response handling time required a different approach. After strace-ing the application,[15] we determined that the Node.js process was aware of open sockets substantially (on the order of milliseconds) before it was reading data from them and sending it to our time recording functions. However, there were no system calls in between the `epoll_wait` responses and `read` calls. Because of this, we concluded that excessive computation at the JavaScript level was delaying any operations.

With our first attempt at improving performance,[16] we removed `got` and used Node.js's `https.request` method directly. This tripled the number of requests we could handle per second, but we still believed Node.js should be able to process requests faster. Additional testing led us to remove calls to `process.hrtime`, a nanosecond-resolution timer, which were occurring on handlers registered to every part of the HTTP/socket request lifecycle. This brought an additional 50% throughput gain. However, two additional changes made things even faster. We stopped using `https.request`, and instead used the `tls` library to create and manage TLS sockets directly. We wrote custom HTTP messages directly to this socket and handled the response data in JavaScript directly (with deferred decoding of the body until after all responses were received). Finally, we added *HTTP pipelining*, a rarely used, but frequently supported, feature of HTTP 1.1 that allows a subsequent HTTP requests to be sent on the same socket before the previous request has returned.[17] This allowed us to send over 25,000 requests per second, indicating a per millisecond response throughput that would have negligible impact on latency measurements. It's important to note, however, that *HTTP pipelining* introduces a new problem into the system: head-of-line blocking. If one request is delayed with 9 requests pipelined behind it, then the 9 other requests will have to wait until the first request is returned before they can be returned. Therefore, though HTTP pipelining may reduce the median response time of the requests, it may increase the tail latencies.

4.3.2 HTTP Server Performance. Similar approaches were taken to improve the callback server's request handling performance, allowing it reach a rate of 31,000 requests per second on average (from a client in a separate process with 500 TCP connections communicating over localhost).[18] All decoding of HTTP request body data was deferred until after all responses had been received (caching the benchmarking frameworks standard FaaS response required ~400 bytes of memory).

4.3.3 Network Jitter. To ensure that network performance was at least consistent between requests (and therefore not introducing noisy amounts of error into latency results) we performed ping tests between various points of the benchmarking framework when deployed on AWS EC2 and AWS Lambda.[19] Network round-trip-times were determined by measuring the time it took to establish TCP connections to target hosts. Within AWS's `us-east-1` region, network latency averaged ~1.0 ms with a standard deviation of ~0.5 ms (over 1000 trials spaced across several days). Since this was substantially smaller than the latencies being measured, this was within acceptable bounds for our benchmarks and would have negligible impact their results.

4.4 Benchmark Tests

We implemented several benchmarks on top of our benchmarking framework that we will now overview.

4.4.1 Cold Start. To benchmark cold start latency, the benchmarking framework deploys FaaS infrastructure and then sends a single request to the appropriate trigger. It filters out any results from FaaS instances that had previously run (though this should never occur due to the freshness of the deployments being tested). We note that this cold start measurement is expected to result in lower latencies for synchronous trigger directly initiate FaaS instances. For asynchronous triggers, the delay due to trigger polling adds even more to the cold start latency. However, since we are only sending one request, we do not expect to see that massive queuing delays that we saw from warm start tests and large request volumes.

4.4.2 Warm Start. To benchmark warm state latency, the benchmarking framework operates similarly to cold start, however, it

[13]Node.js, however, is quite efficient at handling a large number of network requests, which is the primary job of our benchmarks.

[14]All measurements are reported from tests on an m5.xlarge instance on AWS EC2 running Ubuntu 18.04 with Node.js 8.15.1 VM on Linux kernel 4.15.0-1039-aws.

[15]These experiments can be found under `experiments/strace-http-engine`

[16]These experimental results can be found under `experiments/autocannon` and implementations can be found at `experiments/old-apps/lib/triggerer`

[17]We leveraged an existing library that implemented this feature, autocannon, to reduce our development burden.

[18]These experiments can be found under `experiments/callback-server-perf`

[19]These experiments can be found under `experiments/jitter-test`

sends many requests to the trigger instead of only one. Any responses indicating that they were the first response from a FaaS instance are discarded. We note that this warm start benchmark will produce results that are uncomparable to cold starts in the asynchronous case, due to the massive delays introduced by the queuing of requests before they are processed (unlike synchronous invocations which invoke FaaS instances upon receiving the request). In fact, preliminary results showed that the cold start latencies were lower than the warm start latencies for the SQS trigger due to the added queuing latency for warm start latency measurements. In order to make a fair comparison, we would have to significantly limit the number of requests sent for asynchronous warm start latency benchmarks so that request queuing delay can be minimized, but a realistic use case would likely incur the request queuing overhead.

4.4.3 Scaling Request Rates. To benchmark rate of scale and latency at scale, we introduce a variant of the warm start benchmark. For each time period, we increase the rate at which requests are sent every second by a fixed amount. However, we do not increase the rate for asynchronous triggers as the services are already rate limited (by the rate at which the event source is polled) reflected by the presence of queuing overhead at a low rate. Testing indicated that there was minimal to no difference when comparing a constant amount of requests per period and a scaling amount of requests per period, therefore we elide those results. We also do not decrease the request rate between time periods, since we are not interested in how quickly FaaS instances scale down—scale down is not immediately beneficial to FaaS users as they pay only for invocation time, not idle FaaS capacity. We add a sleep delay to our scalability tests as increasing the run time of our FaaS function will add stress to the FaaS system. As each FaaS instance will be held-up for a longer time, in order to handle the same rate of requests, more FaaS instances will need to be provisioned. This is bolstered by the fact that, as mentioned above, AWS claims the number of FaaS instances partitioned is proportional to both request rate and running time. This also relieves stress on the benchmarking system as it can achieve a high stress workload without overloading our own request/response handler.

These benchmarks provide information regarding basic latency and scalability metrics. The benchmarking framework is able to simulate many more workloads, but we focus on these workloads as a preliminary benchmark for serverless providers. These tests generate a basic set of metrics with which we can compare the performance of a cloud provider against other cloud providers and against the guarantees made by the cloud provider. We leave the generation of other benchmarks utilizing our benchmarking framework to future work we hope to pursue.

5 RESULTS

We ran our benchmarking suite only on AWS Lambda,[20] as we are mainly interested in a rigorous theoretical justification of our microbenchmark selection. Our suite is designed so that future researchers can easily port it to other FaaS providers by alterring

[20] All benchmarks were run from m5.xlarge (4 vCPU, 16 GiB RAM) instances on AWS EC2 running Ubuntu 18.04 with Node.js 8.15.1 VM on Linux kernel 4.15.0-1039-aws against Lambdas that had an account global concurrency limit of 1000. Benchmarks that approached the concurrency limit were not conducted simultaneously, and only one benchmark was ever running at a time per m5.xlarge instance.

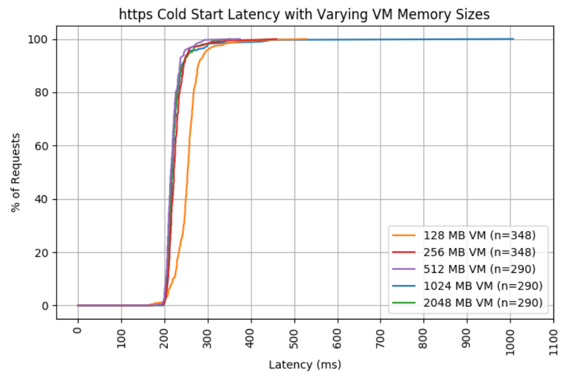

Figure 2: CDF of cold start latency when sending 1 request per FaaS deployment to AWS API Gateway.

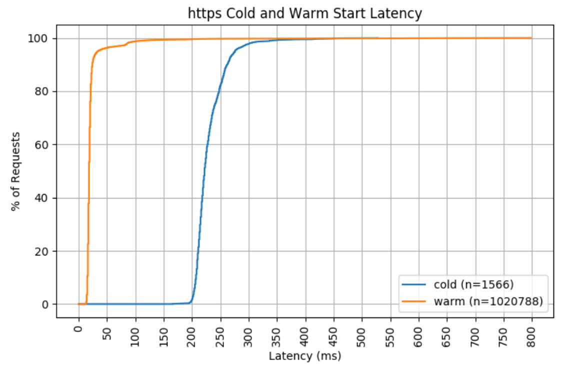

Figure 3: CDFs of cold vs warm start latency when sending many requests to AWS API Gateway.

a few lines within the infrastructure deployer and, if the trigger interface is non-standard, the trigger requester.

5.1 HTTPS Trigger Results

5.1.1 Cold Start. As shown in Figure 2, our simple HTTPS cold start benchmarks tend to follow similar patterns across all memory sizes tested, although 128 MB had a longer cold start than those of the other memory sizes. This may be evidence that 128 MB FaaS configuration is given lower allocation priority compared to other sizes. Based on our benchmark alone, however, we cannot form a strong hypothesis regarding why this would be the case. Performance is fairly stable up to the 99th percentiles, although we found that higher percentiles could have tails longer than 1000 ms in very rare cases. The median cold start time was 221 ms, and standard deviation was 47.8 ms; 95% of the cold starts complete within 100 ms of the first cold start completion.

5.1.2 Cold vs. Warm Start. Figure 3 displays both the HTTPS warm start and cold start CDFs on the same graph, including data from all FaaS memory sizes. We restrict the range for the x-axis to be 0 ms to 800 ms in order to focus on bulk of the latency data; though

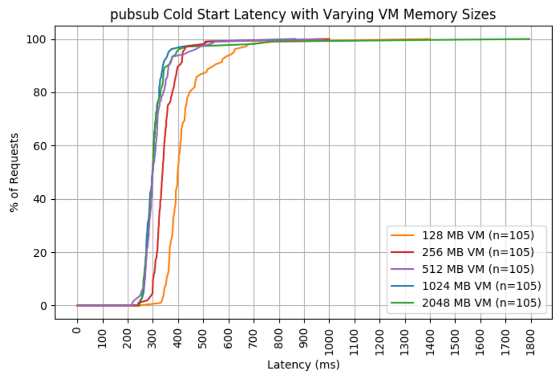

Figure 4: CDF of cold start latency when sending 1 request per FaaS deployment to AWS Simple Queuing Service.

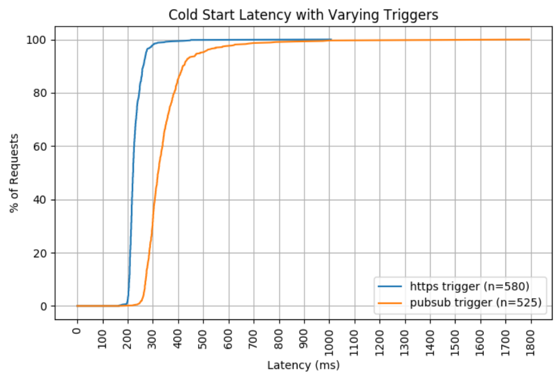

Figure 5: CDF of cold start latencies, comparing the https and pub/sub triggers. Each CDF is an aggregate of all test runs with varying FaaS instance memory sizes.

the tail is quite long, reaching over 5000 ms for warm start latency. Overall, cold start and warm start follow the same shape. Cold start lags behind warm start, as expected, by approximately 200 ms.

5.2 SQS Trigger Results

5.2.1 Cold Start. For our simple cold start test for our pub/sub AWS SQS trigger, we see a similar CDF shape across all memory sizes (Figure 4). Again, we see that the 128 MB FaaS instances had a longer cold start compared to those of the other memory sizes, supporting our hypothesis that certain FaaS instance memory sizes are given lower priorities than others (though we also see the 256 MB size having a worse cold start, which we did not see with HTTPS). Besides the 128 MB size, performance is fairly stable up to the 90th percentile, after which we begin to see longer tails; the longest cold start test took 1794 ms. The 128 MB size has an even worse performance dropoff after the 80th percentile (1 in 10 cold starts will take over 500 ms). The median cold start time was 319 ms, and standard deviation was 120.6 ms, much greater than that of https cold starts.

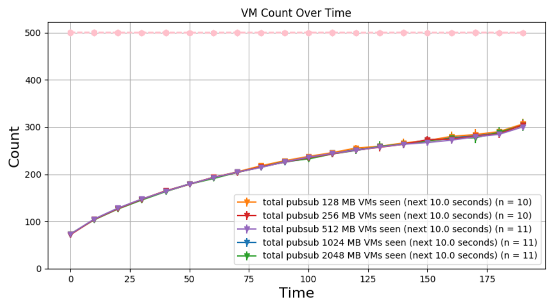

Figure 6: Graph of number of unique VMs (concurrent FaaS instances) seen in each 10 second period as 500 messages are sent every 10 seconds to AWS SQS (10k messages per trial). Tests are run with varying memory sizes. Error bars (unnoticeable due to their very small size) are shown at the 5th and 95th percentiles.

5.2.2 HTTPS vs. Pub/Sub Cold Start. Figure 5 compares the CDFs for the pub/sub and https triggers (aggregating across all memory sizes for both). While the distribution of the CDFs follow the same pattern, the pub/sub trigger is uniformly slower than the https trigger. The pub/sub trigger takes longer to start up initially and has a much longer tail. The former is expected, as pub/sub is an asynchronous trigger; instead of reacting immediately to the trigger as with synchronous triggers, the FaaS scheduler has to continuously poll a stream of events before invoking new instances.

5.2.3 Pub/Sub Scaling Under Load. Figure 6 compares the number of FaaS instances that spin up to handle pub/sub messages that are sent at 50 requests per second. The messages further instruct the FaaS instance to sleep for 5 seconds, to produce an execution time that should dominate FaaS system overhead and provide a reliable runtime number for evaluating the concurrency formula provided by AWS. Here, we see a fairly uniform increase of FaaS instances over time, and this trend does not vary much at all even when memory size of the FaaS instances increase. The error bars are also very small; there is almost no variance across trials. Given the near uniformity across different memory sizes and trials, it is possible that Amazon has a deterministic formula for number of FaaS instances to invoke that depends on the presence of and or the number of requests that need to be handled. However, it does not appear that their formula for number of concurrent instances[21] is being followed here (at 50 trigger events per second and 5 seconds of execution, that should be 250 concurrent FaaS instances, though this number is not provisioned until 100 seconds in and is surpassed a little while later).

5.2.4 Pub/Sub Latency Under Load. Figure 7 compares the latency, with 5th and 95th percentile bars, incurred over time as FaaS instances spin up to respond to 50 requests per second sent by the pub/sub trigger. As expected, we see latency increase over time, although we begin to see a slight decrease after approximately 125 seconds, when approximately 250 unique FaaS instances respond to requests. Importantly, we see the distance between the 5th and 95th

[21] `(invocations per second) * (average execution duration in seconds)`

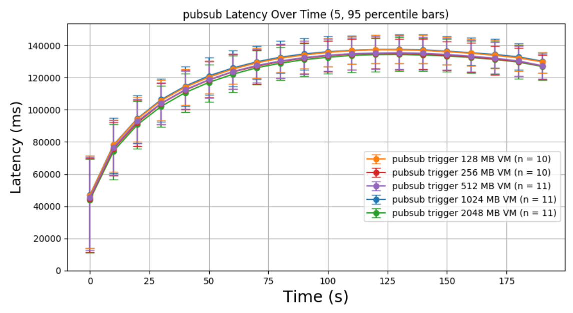

Figure 7: Latency over time as 500 requests are sent every 10 seconds to AWS SQS. Tests are run with varying memory sizes. Error bars are shown at the 5th and 95th percentiles.

percentiles decrease over time. We hypothesize this is because as the number of unique FaaS instances handling requests increases, the rate at which enqueued requests are handled also increases resulting in overall lower response latency.

6 FUTURE WORK

This section outlines shortcomings in our approach and potential avenues for future work to address them.

Firstly, our benchmarking approach includes the time required for packets to traverse the network to reach the serverless function's trigger endpoint. It is possible that this network time varies across trials and requests, adding an unknown variable that could contribute to unobserved errors (regardless of the testing we performed as discussed in 4.3.3). It is also possible that the packets travel across the same path, so sending many requests at once from the same source to the same destination could lead to congestion along the network or contribute to head-of-line blocking.

Secondly, we were unable to test scalability for our HTTPS trigger. We attempted to measure latency and FaaS instance count over time, configuring our requester to send 50 requests every second and letting functions sleep for 5 seconds. However, because HTTPS is a closed test, it cannot invoke at a fast enough rate because of the sleep delay. In order to hit the same rate as the pub/sub trigger, we would need double the maximum number of TCP connections we were able to open on our test VMs. Further, we anticipate that this benchmark would show the HTTPS trigger immediately invoking concurrent FaaS instances up to the concurrency limit, since the invocations are synchronous. Because of these expectations, we did not prioritize solving this limitations and instead we leave it to future research to overcome. One potential approach would be leveraging the approach we used for asynchronous trials with callback requests sent to the local listener (this would allow us to terminate the connection to API Gateway from the requester as soon as the initial HTTPS request was sent, but before a response was returned). Another approach would be to leverage HTTP pipelining further, since many HTTP requests could be sent over a single TLS connection. The HTTP pipelining implementation we leveraged did not support this in addition to request limits per second, however, it should be possible to augment it.[22]

Finally, it would be useful to set up our benchmarking suite to automatically run and present results. Serverless computing is a growing field with platforms that are still evolving. An automated benchmark across multiple platforms would help track developments in the FaaS ecosystem as a whole. Moreover, it would allow developers and researchers to have insight into noteworthy, but currently unnoted, changes in FaaS offerings; as well as help project trends about where the field is headed for the future.

7 CONCLUSION

FaaS providers offer little transparency about performance. As such, users are unable to compare different platforms, the same platform across different updates, and different triggers within platforms to each other. We propose several microbenchmarks that address the core tradeoff and promises with serverless: cold start latency, warm start latency, unique FaaS instance counts over time, and latency over time for asynchronous triggers. We hope that users will be able to use our benchmarks to select the optimal triggers and platforms for their FaaS use case, if they determine that the overhead of FaaS is acceptable at all. We also hope that researchers and industry members will leverage our microbenchmarks as they explore more macro- and micro-benchmarking suites in the future.

ACKNOWLEDGMENTS

We'd like to thank Professor Eddie Kohler and Professor Yu Minlan, the instructors for CS260r and CS245 respectively. We appreciate their guidance in our literature review on creating a useful benchmark and their help in determining what metrics were of value in regards to a serverless platform. Also, thanks to Professor Kohler's LaTeX Usage Notes[23], which, though we are certain to have failed in some manner, were helpful in improving this paper's aesthetic readability (content readability is another matter entirely).

Also, thank you to the computers that handled our 2,573,146 FaaS invocations and 29.3 GB-days of computation.

REFERENCES

[1] [n. d.]. AWS re:Invent 2016: Coca-Cola: Running Serverless Applications with Enterprise Requirements. https://www.youtube.com/watch?time_continue=8&v=yErmil00DYs
[2] [n. d.]. Bustle Case Study - AWS. https://aws.amazon.com/solutions/case-studies/bustle/
[3] [n. d.]. Chapter 17. Configuring NTP Using the chrony Suite. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/system_administrators_guide/ch-configuring_ntp_using_the_chrony_suite publisher: Red Hat.
[4] [n. d.]. chrony âĂŞ Comparison of NTP implementations. https://chrony.tuxfamily.org/comparison.html
[5] [n. d.]. How Reuters Replaced WebSockets with Amazon Cognito and SQS. https://serverless.com/blog/how-reuters-replaced-websockets-with-amazon-cognito-and-sqs
[6] [n. d.]. HTTPS |Node.js v8.16.0 Documentation. https://nodejs.org/docs/latest-v8.x/api/https.html
[7] [n. d.]. iRobot Case Study - AWS. https://aws.amazon.com/solutions/case-studies/irobot/
[8] [n. d.]. ntp.org: Home of the Network Time Protocol. http://www.ntp.org/
[9] [n. d.]. Serverless - The Serverless Application Framework powered by AWS Lambda, API Gateway, and more. https://serverless.com/

[22] We already augmented it (extremely minimally) to disable Nagle's algorithm on TCP sockets.
[23] https://read.seas.harvard.edu/ kohler/latex.html

[10] [n. d.]. Time Synchronization. https://help.ubuntu.com/lts/serverguide/NTP.html.en publisher: Canonical.

[11] [n. d.]. Understanding Scaling Behavior. https://docs.aws.amazon.com/lambda/latest/dg/scaling.html

[12] [n. d.]. Zappa - Serverless Python Web Services. https://www.zappa.io/

[13] 2016. AWS re:Invent 2016: Serverless Computing Patterns at Expedia. https://www.slideshare.net/AmazonWebServices/aws-reinvent-2016-serverless-computing-patterns-at-expedia-svr306

[14] 2019. AWS Lambda - Pricing. https://aws.amazon.com/lambda/pricing/

[15] 2019. BinaryAlert: Serverless, Real-time & Retroactive Malware Detection. https://github.com/airbnb/binaryalert original-date: 2017-07-12T21:27:13Z.

[16] 2019. Combine serverless with artillery and you get serverless-artillery for instant, cheap, and easy performance testing at scale. https://github.com/Nordstrom/serverless-artillery original-date: 2016-08-12T21:17:28Z.

[17] 2019. enmasseio/timesync: Time synchronization between peers. https://github.com/enmasseio/timesync original-date: 2015-01-26T13:32:41Z.

[18] 2019. "Hello, Retail!" is an open-source, mobile-first, 100% serverless functional proof-of-concept showcasing a complete event sourcing approach applied to the retail platform space. https://github.com/Nordstrom/hello-retail original-date: 2016-12-07T19:30:13Z.

[19] 2019. MoonMail: Shoot billions of emails using SES & Lambda. https://github.com/MoonMail/MoonMail original-date: 2016-03-31T15:07:00Z.

[20] 2019. Netflix/bless: an SSH Certificate Authority that runs as a AWS Lambda function. https://github.com/Netflix/bless original-date: 2016-05-18T22:19:30Z.

[21] 2019. prisma-archive/chromeless: Chrome automation made simple. https://github.com/prisma-archive/chromeless original-date: 2017-06-01T16:11:11Z.

[22] Akadia. [n. d.]. Time Synchronization with NTP. https://www.akadia.com/services/ntp_synchronize.html

[23] Yan Cui. 2017. How does language, memory and package size affect cold starts of AWS Lambda? *A Cloud Guru* (jun 2017). https://read.acloud.guru/does-coding-language-memory-or-package-size-affect-a15e26d12c76

[24] 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.

[25] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 363–376. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi

[26] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *arXiv preprint arXiv:1902.03383* (2019).

[27] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. Technical Report UCB/EECS-2019-3. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html

[28] Nima Kaviani and Michael Maximilien. 2016. CF Serverless: Attempts at a Benchmark for Serverless Computing. https://docs.google.com/document/d/1e7xTz1P9aPpb0CFZucAAI16Rzef7PWSPLN71pNDa5jg/edit#heading=h.hs3id0lz5anm

[29] T. Lynn, P. Rosati, A. Lejeune, and V. Emeakaroha. 2017. A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 162–169. https://doi.org/10.1109/CloudCom.2017.15

[30] Johannes Manner, Martin EndreÃ§, Tobias Heckel, and Guido Wirtz. 2018. Cold Start Influencing Factors in Function as a Service. https://doi.org/10.1109/UCC-Companion.2018.00054

[31] Guillermo Grau Panea. 2019. guigrpa/clocksy: Transport-agnostic client-server clock synchronization. https://github.com/guigrpa/clocksy original-date: 2016-07-13T13:45:29Z.

[32] Geir Kjetil Sandve, Anton Nekrutenko, James Taylor, and Eivind Hovig. 2013. Ten Simple Rules for Reproducible Computational Research. *PLOS Computational Biology* 9, 10 (10 2013), 1–4. https://doi.org/10.1371/journal.pcbi.1003285

[33] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. 2006. Open Versus Closed: A Cautionary Tale. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3 (NSDI'06)*. USENIX Association, Berkeley, CA, USA, 18–18. http://dl.acm.org.ezp-prod1.hul.harvard.edu/citation.cfm?id=1267680.1267698

[34] Malte Schwarzkopf, Derek G. Murray, and Steven Hand. Submitted. The Seven Deadly Sins of Cloud Computing Research. In *Presented as part of the*. USENIX. https://www.usenix.org/conference/hotcloud12/seven-deadly-sins-cloud-computing-research

[35] Airbnb Engineering & Data Science. [n. d.]. StreamAlert: real-Time data analysis and alerting. https://airbnb.io/projects/streamalert/ publisher:.

[36] Amazon Web Services. [n. d.]. AWS CloudFormation - Infrastructure as Code & AWS Resource Provisioning. https://aws.amazon.com/cloudformation/

[37] Amazon Web Services. [n. d.]. Using AWS Lambda with Other Services - AWS Lambda. https://docs.aws.amazon.com/lambda/latest/dg/lambda-services.html

[38] Amazon Web Services. 2017. Keeping Time With Amazon Time Sync Service. https://aws.amazon.com/blogs/aws/keeping-time-with-amazon-time-sync-service/

[39] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. 2018. numpywren: serverless linear algebra. *arXiv:1810.09679 [cs]* (Oct. 2018). http://arxiv.org/abs/1810.09679 arXiv: 1810.09679.

[40] Sindre Sorhus. 2019. sindresorhus/got: Simplified HTTP requests. https://github.com/sindresorhus/got original-date: 2014-03-27T22:40:49Z.

[41] Bernd Strehl. 2018. The largest benchmark of Serverless providers. *elbstack* (sep 2018).

[42] Bernd Strehl. 2019. Serverless Benchmark 2.0âĂŁâĂŤâĂŁPart I. *elbstack* (mar 2019).

[43] Erik van der Kouwe, Dennis Andriesse, Herbert Bos, Cristiano Giuffrida, and Gernot Heiser. 2018. Benchmarking Crimes: An Emerging Threat in Systems Security. *arXiv e-prints*, Article arXiv:1801.02381 (Jan 2018), arXiv:1801.02381 pages. arXiv:cs.CR/1801.02381

[44] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 133–146. https://www.usenix.org/conference/atc18/presentation/wang-liang