# Serverless Computing: An Investigation of Factors Influencing Microservice Performance

Wes Lloyd[1], Shruti Ramesh[4], Swetha Chinthalapati[2], Lan Ly[3], Shrideep Pallickara[5]

Institute of Technology
University of Washington
Tacoma, Washington USA
[1]wlloyd, [2]swethach, [3]lanly@uw.edu

[4]Microsoft
Redmond, Washington USA
sr3155@columbia.edu

[5]Department of Computer Science
Colorado State University
Fort Collins, Colorado USA
shrideep@cs.colostate.edu

***Abstract*— Serverless computing platforms provide function(s)-as-a-Service (FaaS) to end users while promising reduced hosting costs, high availability, fault tolerance, and dynamic elasticity for hosting individual functions known as microservices. Serverless Computing environments, unlike Infrastructure-as-a-Service (IaaS) cloud platforms, abstract infrastructure management including creation of virtual machines (VMs), operating system containers, and request load balancing from users. To conserve cloud server capacity and energy, cloud providers allow hosting infrastructure to go COLD, deprovisioning containers when service demand is low freeing infrastructure to be harnessed by others. In this paper, we present results from our comprehensive investigation into the factors which influence microservice performance afforded by serverless computing. We examine hosting implications related to infrastructure *elasticity*, *load balancing*, *provisioning variation*, *infrastructure retention*, and *memory reservation* size. We identify four states of serverless infrastructure including: *provider cold*, *VM cold*, *container cold*, and *warm* and demonstrate how microservice performance varies up to 15x based on these states.**

***Keywords Resource Management and Performance; Serverless Computing; Function-as-a-Service; Provisioning Variation;***

## I. INTRODUCTION

Serverless computing recently has emerged as a compelling approach for hosting applications in the cloud [1] [2] [3]. While Infrastructure-as-a-Service (IaaS) clouds provide users with access to voluminous cloud resources, resource elasticity is managed at the virtual machine level, often resulting in over-provisioning of resources leading to increased hosting costs, or under-provisioning leading to poor application performance. Serverless computing platforms provide Function(s)-as-a-Service (FaaS) by hosting individual callable functions. These platforms promise reduced hosting costs, high availability, fault tolerance, and dynamic elasticity through automatic provisioning and management of compute infrastructure [4].

Serverless computing platforms integrate support for scalability, availability, fault tolerance capabilities directly as features of the framework. Early adoption of serverless computing has focused on deployment of lightweight stateless services for image processing, static processing routines, speech processing, and event handlers for Internet-of-Things devices [5]. The promised benefits, however, makes the platform very compelling for hosting ***any*** application. If serverless computing delivers on its promises, it has the potential to fundamentally transform how we build and deploy software on the cloud, driving a paradigm shift rivaling a scale not seen since the advent of cloud computing itself!

Fundamentally different than application hosting with IaaS or Platform-as-a-Service (PaaS) clouds, with serverless computing, applications are decomposed and deployed as code modules. Each cloud provider restricts the maximum size of code (e.g. 64 to 256MB) and runtime (e.g. 5 minutes) of functions. Serverless platform functions are often used to host RESTful web services. When RESTful web services have a small code size they can be referred to as microservices. Throughout this paper we refer to our code deployments as mircoservices because they are small RESTful web services, but the results of our work are applicable to ***any*** code asset deployed to a serverless computing platform. We do not focus on defining the difference between a RESTful web service and a microservice and leave this debate open.

Serverless environments leverage operating system containers such as Docker to deploy and scale microservices [6]. Granular code deployment harnessing containers enables incremental, rapid scaling of server infrastructure surpassing the elasticity afforded by dynamically scaling virtual machines (VMs). Cloud providers can load balance many small container placements across servers helping to minimize idle server capacity better than with VM placements [7]. Cloud providers are responsible for creating, destroying, and load balancing requests across container pools. Given their small size and footprint, containers can be aggregated and reprovisioned more rapidly than bulky VMs. To conserve server real estate and energy, cloud providers allow infrastructure to go COLD, deprovisioning containers when service demand is low freeing infrastructure to be harnessed by others. These efficiencies hold promise for better server utilization leading to workload consolidation and energy savings.

In this paper, we present results of our investigation focused on identifying factors that influence performance of microservices deployed to serverless computing platforms. Our primary goal for this study has been to identify factors influencing microservice performance to inform practitioners regarding the nuances of serverless computing infrastructure to enable better application deployments. We investigate microservice performance implications related to: infrastructure elasticity, load balancing, provisioning variation, infrastructure retention, and memory reservation size.

0-7695-6371-6/18/31.00 ©2018 IEEE
DOI 10.1109/IC2E.2018.00039

### A. Research Questions

To support our investigation of factors influencing microservice performance for serverless computing platforms, we investigate the following research questions:

**RQ-1:** (*Elasticity*) What are the performance implications for leveraging elastic serverless computing infrastructure for microservice hosting? How is response time impacted for COLD vs WARM service requests?

COLD service requests are sent by clients to microservice hosting platforms where the service hosting infrastructure must be provisioned to respond to these requests. Four types of function invocations exist relating to infrastructure warm up for serverless computing infrastructure. These include: (1-*provider cold*) the very first service invocation for a given microservice code release made to the cloud provider, (2-*VM cold*) the very first service invocation made to a virtual machine (VM) hosting one or more containers hosting microservice code, (3-*container cold*) the very first service invocation made to an operating system container hosting microservice code, and (4-*warm*) a repeated invocation to a preexisting container hosting microservice code.

**RQ-2:** (*Load Balancing*) How does load balancing vary for hosting microservices in serverless computing? How do computational requirements of service requests impact load balancing, and ultimately microservice performance?

Serverless computing platforms automatically load balance service requests across hosting infrastructure. Cloud providers typically leverage round robin or load balancing based on CPU load to distribute incoming resource requests [8]. For serverless computing, we are interested in understanding how the computational requirements of individual microservice requests impact load balancing and ultimately performance.

**RQ-3:** (*Provisioning Variation*) What microservice performance implications result from provisioning variation of container infrastructure?

Provisioning variation refers to random deployment variation of virtual infrastructure across the physical hardware of cloud datacenters [9] [10]. Provisioning variation results from the use of load balancing algorithms, which attempt to place VMs and containers evenly across available infrastructure. From a user's point-of-view, however, resource placement may seem random as resource assignments are made in conjunction with requests from other users resulting in a greater spread of a user infrastructure compared with private server deployments. Consolidating many containers to a single host VM leverages image caching to reduce cold launch latency, but may lead to increased resource contention when many simultaneous requests are hosted on the same VMs impacting performance [38]. We are interested in understanding the microservice performance implications for provisioning variation introduced by the cloud provider.

**RQ-4:** (*Infrastructure Retention*) How long is microservice infrastructure retained based on utilization, and what are the performance implications?

Serverless computing frameworks automatically manage VMs and operating system containers to host microservice code. Once a VM participates in hosting a microservice, the Docker container image can be cached enabling additional container instances to be created more rapidly. Containers preserved in a warm state can rapidly service incoming requests, but retaining infrastructure indefinitely is not feasible as cloud providers must share server infrastructure amongst all cloud users. We are interested in quantifying how infrastructure is deprecated to understand implications for performance as well as derive keep alive workloads to prevent microservices with strict SLAs from experiencing longer latencies.

**RQ-5:** (*Memory Reservations*) What performance implications result from microservice memory reservation size? How do memory reservations impact container placement?

Serverless computing platforms abstract most infrastructure management configuration from end users. Platforms such as AWS Lambda and Google Cloud Functions allow users to specify a memory reservation size. Users are then billed for each function invocation based on memory utilization to the nearest tenth of a second. For example, Lambda functions can reserve from 128MB to 3008MB, while Google Cloud Functions can reserve from 128MB to 2048MB. Azure functions allows users to create function apps. Function apps share hosting infrastructure and memory for one or more user functions. Azure function app hosts are limited 1536MB maximum memory. Users do not reserve memory for individual functions and are billed only for memory used in 128MB increments. One advantage to Azure's model is that users do not have to understand the memory requirements of their functions. They simply deploy their code, and infrastructure is automatically provisioned for functions up to the 1536MB limit. In contrast, users deploying microservices to Lambda or Google Cloud Functions must specify a memory reservation size for function deployment. These reservations are applied to Docker containers created to host user functions. Containers are created to host individual function deployments, and user functions may or may not share resources of underlying VMs.

### B. Contributions

This paper reports on our investigations of performance implications for microservice hosting on serverless computing platforms. This study analyzes performance implications related to infrastructure elasticity, service request load balancing, provision variation of hosting infrastructure, infrastructure retention, and implications of the size of memory reservations. While originally envisioned for hosting light-weight event based code, benefits of serverless computing including autonomous high availability, elasticity, and fault tolerance makes the platform very compelling for broad use. A key contribution of this study is a comprehensive profiling of the performance implications of the autonomic infrastructure management provided by serverless computing platforms. We believe our study is the first to investigate many of these performance implications in depth.

The primary contributions of this paper include:

1. Identification, and performance analysis of the four states of serverless computing for microservice hosting: *provider cold*, *VM cold*, *container cold*, and *warm*.
2. Performance, elasticity, and load balancing analysis across infrastructure provided by AWS Lambda and

Azure functions, including the use of standard deviation to quantify fairness of load balancing.

3. Analysis of the performance implications of provisioning variation of containers deployed across host VMs.
4. Performance analysis of infrastructure retention and combined memory and CPU capacity reservations provided by AWS Lambda for microservices hosting.

## II. Background and Related Work

### A. Motivation for Microservices Architecture

A microservices application architecture provides a means to develop software applications as a suite of small services [4]. Decomposing functionality of historically large, coupled, and monolithic applications into compositions of microservices enables developer agility supporting DevOps software development processes. Microservices have a small codebase, are easy to deploy and subsequently scale. Applications which compose multiple microservices as a mashup offer resilience as portions of an application can be revised while maintaining availability of the application at large.

Aderaldo et al. note that there is a lack of repeatable empirical research on the design, development, and evaluation of microservices applications [11]. They provide a set of requirements towards the development of a microservices benchmark application equivalent to TPC-W, the webserver and database benchmark. Kecskemeti et al. offer the ENTICE approach to decompose monolithic services into microservices [12]. ENTICE, however, focuses primarily on generation of dynamic VM images with requisite software libraries to support decomposition as the work does not apply to serverless environments directly. Hassan and Bahsoon identify the importance to balance design tradeoffs in microservices architecture and propose the use of a self-adaptive feedback control loop to generate potential application deployments that trade off criteria such as size, number of microservices, and satisfaction of non-functional requirements [13]. Granchelli et al. are able to decompose microservice application architecture to generate an architectural model of the system given a GitHub repository and a web container endpoint using MicroART [14]. And Frey et al. apply a genetic algorithm to reduce the search space of potential deployment configurations for traditional VM-based cloud applications to rapidly identify optimal configurations [15]. These efforts have not specifically considered microservices application deployment to serverless computing platforms. We are unaware of prior research efforts that specifically assess performance implications of microservice deployment to serverless computing platforms.

### B. Serverless Computing Frameworks

Commercially provided serverless computing platforms provide dynamic scalable infrastructure on-demand to host microservice applications [16][17][18][19]. Research into best practices for monolithic application decomposition for deployment as microservices, however, has not yet considered deployment to serverless computing environments leaving cost and performance tradeoffs unexplored [4] [20]. Recently Sill noted in his IEEE Cloud Computing magazine column that serverless computing's adoption of deploying services to containers is more of a coincidence than a direct consequence of optimal design [6]. McGrath and Brenner recently presented a serverless computing framework that runs atop of the Microsoft Azure cloud and Windows containers [21]. They contribute metrics to evaluate performance of serverless platforms including examining scaling and container instance expiration trends while showing their framework achieves greater throughput than available commercial frameworks at most concurrency levels.

### C. Improving Serverless Application Deployments

Efforts to improve elasticity of cloud computing infrastructure for application hosting almost exclusively focuses on IaaS clouds. Considerable work has focused on evaluating performance modeling and machine learning techniques to support dynamic scaling of VMs [22] [23] [24] [37] or to evaluate the efficacy of linear regression, neural networks, and support vector machines to predict performance and resource demands in the future for threshold based auto scaling. Qu et al. provide high-availability using low cost spot VMs [25].

Our investigations help inform our understanding of the factors that influence microservice performance afforded by serverless computing platforms. These understandings will help guide practitioners towards making better deployment decisions while establishing best practices.

## III. Serverless Computing Platforms

To investigate research questions described in section 1, we harness two commercial serverless computing platforms: AWS Lambda, and Microsoft Azure [16][18].

***AWS Lambda***, introduced in 2014, harnesses containers atop of the AWS Linux operating system based on Redhat Linux. Presently, Lambda officially supports hosting microservices written in Node.js, Python, Java, and C#. Lambda's billing model provides 1 million function invocations a month for free, while each subsequent 1 million requests costs approximately 20 cents ($.20 USD). Functions can use up to 400,000 GB-seconds a month for free, after which additional memory utilization costs approximately 6 cents ($.06 USD) for each 1 GB of memory reserved per hour. Functions can individually reserve from 128MB to 3008MB of memory. Lambda automatically hosts and scales infrastructure to provide microservices supporting by default up to 1,000 concurrent requests. As of fall 2017, containers are provided with 2 hyperthreads backed by the Intel Intel(R) Xeon(R) CPU E5-2666 v3 @ 2.90GHz. Lambda allocates CPU power and other resources proportional to memory. For example, reserving 256MB of memory allocates approximately twice as much CPU power to a Lambda function as requesting 128MB of memory, and half as much CPU power as choosing 512MB of memory. Docker containers support specification of CPU-period (the completely fair scheduler-CFS interval), CPU-quota (a CFS CPU quota), and CPU shares (the share of CPU resources provided when constrained) [26]. The precise mappings between Lambda memory and CPU configuration is not documented. Each container has 512 MB of disk space and can support up to 250MB of deployed code provided in compressed format up to 50MB. Microservices execution time is limited to a maximum of 5 minutes.

***Azure Functions***, is a serverless computing platform provided by Microsoft that allows users to run small pieces of

code or 'functions' in the cloud. Azure Functions is derived from, and built on the Azure App Service, and is a natural extension of WebJobs with additional features such as dynamic scaling [28]. Azure Functions automatically creates a function app for the user. This app hosts one or more individual functions managed by the Azure App Service. Function apps hosted by the Azure App Service run using low-privileged worker processes using a random application identity [28]. The runtime for Functions is the underlying WebJobs SDK host that performs all operations related to running code including listening for events, as well as gathering and relaying data [29].

Presently, Functions supports hosting of code in C#, F# and Node.js. Support for languages such as Python, PHP and Bash are currently under development. Functions can be run using two plans, i.e. the Consumption Plan and the App Service Plan. The type of plan decides how the function/app will scale and what resources are available to the host [30]. The Consumption Plan automatically decides the resources required to run the app, provides dynamic scaling, and bills only according to resources consumed. Each app is limited to 1.5 GB total memory, and functions are limited to 10 minutes execution time [31]. In the App Service Plan, function apps are run on dedicated VMs, similar to Web Apps. For the App Service Plan the functions host is always running and it is possible to manually scale out by adding new VMs, or by enabling auto scaling.

## IV. Experimental Setup

To support experimentation, we developed and deployed AWS Lambda and Azure functions microservices aimed at introspecting platform performance and infrastructure management. We describe the microservices below.

### A. Lambda Experimental Microservice

For Lambda, we developed a compute-bound microservice that performs random math calculations where operations avoid using stack variables with operands stored in large arrays on the heap. To vary the degree of memory stress we make the array size a service parameter. For each calculation, operands are referenced from a random location in the array. When the array size is larger, this referencing scheme induces additional page faults resulting in memory stress. When the array size is small, the behavior of the function is primarily bound only by random multiplication and division calculations. While maintaining the same number of calculations, we observe that larger array sizes result in microservice failure on Lambda function deployments with low memory/CPU allocations confirming the efficacy of our ability to introduce memory stress.

TABLE I. Summary of Calculations Service Configurations

| Stress Level | # of Calculations | Operand Array Size | Function Calls |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 2 | 2,000 | 100 | 20 |
| 3 | 20,000 | 1,000 | 20 |
| 4 | 200,000 | 10,000 | 20 |
| 5 | 500,000 | 25,000 | 20 |
| 6 | 2,000,000 | 100,000 | 20 |
| 7 | 10,000,000 | 10,000 | 1,000 |
| 8 | 10,000,000 | 100 | 100,000 |
| 9 | 6,000,000 | 20 | 300,000 |

To support our experiments, we defined 9 stress levels for our calculations services each introducing increasing stress and requiring longer execution time. These stress levels are described as calculations service configurations in table I. They reflect the different function parameterizations used in our experiments. Stress level 1 is used to simply evaluate the microservice round trip time to execute a calculation free service call. Stress levels 2 to 9 introduce 2,000 to 10,000,000 random math operations. Operand array size reflects memory stress. Three arrays are used to store two multiplication operands and one division operand. The number of function calls reflects stress from call stack activity.

Our Lambda testing leveraged the AWS API Gateway and all of our functions invocations were synchronous. The API Gateway imposes an unalterable 30-second limit on synchronous service requests. We performed our Lambda tests using bash scripts under Ubuntu 16.04 hosted by a c4.2xlarge 8-vCPU EC2 instance with "High" networking performance. We pinned our EC2 instances and Lambda functions to run using a default VPC in the us-east-1e availability zone. The default configuration for Lambda function deployments is to span multiple sub-regions for redundancy and fault tolerance. To eliminate potential performance variability from randomly communicating across different sub-regions, we elected to fix our deployment to one sub-region. Please note that Amazon sub-regions are floating. Each user experiences different mappings to avoid having users accidentally provision too many cloud resources in the first sub-region, us-east-1a. High networking performance is generally considered to be approximately ~1 Gbps. We harnessed GNU parallel to perform all requests in parallel and utilized the command-line curl REST client to instrument all HTTP-POST JSON service requests. Given the small JSON request/response payloads of our microservice, we observed little stress on our c4.2xlarge client instance while performing up to 100 concurrent Lambda service invocations.

Our calculations microservice provided a compute-bound workload to exercise AWS Lambda. Additionally, we augmented our microservice to introspect the Lambda execution environment.

***Container Identification***: Each Lambda function is hosted in a Docker container with a local 512MB filesystem. When a service request is received by a container we check for the presence of a temporary file. If the temporary file is not present, we create it and stamp the container with a universally unique identifier (UUID) by writing to this temp file. When the container is retained for subsequent service requests, the UUID serves to identify new vs. recycled containers. In the service response, we report the unique UUID and whether the request generated a new container.

***Host Identification:*** Docker containers provide access to the /proc filesystem of the Linux host. We determined the hosts run Amazon Linux VMs. By inspecting **`/proc/cpuinfo`** we identify that Docker containers leverage the Intel Xeon E5-2666 v3 @ 2.90GHz CPU. This is the same CPU as used by c4/m4/r4 EC2 instances, Amazon's 4th generation of VMs. We also identified that each container has access to two vCPUs. We are interested in knowing how many Docker containers run on each

AWS Lambda host VM. We leverage "btime" in **`/proc/stat`** to identify the boot time of the VM in seconds since the epoch, January 1, 1970. Using this technique, we are able to identify when the Docker containers used to host a Lambda function "***appear" to leverage the same host (VM).*** While this technique does not guarantee that we've found host affinity, statistically it is highly reliable.

Let's consider the probability that two VMs boot and initialize their "btime" variable at the same exact same second. To establish probability, we consider the infrastructure used to host 132 trials of 100-concurrent requests for the 12 memory reservation levels described in table II. For this workload, we observed a range of boot times across the 91 unique VMs that participated in hosting these Lambda service requests. We measured the uptime for VMs to establish their estimated lifetime. **For these 91 VMs, we observed VM uptime to range from a minimum of 17 minutes and 45 seconds, to a maximum of 3 hours and 26 minutes.** The average uptime was approximately 2 hours and 8 minutes. The VM uptime here spans a range of 11,302 seconds. If we consider the probability of a VM receiving a given boot time in this 11,302 second range to be 1/11,302, then the probability of any given boot time is just $P(boot\_time_A) = 0.00008848$. The probability of two VMs having the same boot_time will be $P(boot\_time_A \cap boot\_time_B) = 7.8*10^{-9}$. While it is unlikely that VMs are launched entirely randomly by Lambda, actual probability will likely be higher as VMs are inevitably launched in groups. We deem our approach as suitable to identify host affinity and its efficacy is verified through successful use throughout all of our experiments.

***Client Testing:*** Our clients process service outputs to determine the number of service requests processed by each container and host. We capture the uptime, and number of new containers generated. We also calculate the standard deviation of service requests distributed to both containers and hosts. When the standard deviation of this distribution is zero, it identifies perfectly even balancing of requests across infrastructure akin to round-robin. As the standard deviation of load balancing increases, this indicates discontinuity. For example, if 4 hosts participate in 100 service requests, and the distribution of requests is: 10, 5, 2, and 83, then the standard deviation is 38.8. This uneven distribution of requests will stress the 4th node unevenly, potentially leading to performance degradation. We capture standard deviation in our test scripts to identify fairness of service request load balancing. Our testing also captures average service execution time for all requests, and the quantity of requests serviced by individual containers and hosts. Using this approach, we can clearly see when new hosts (VMs) and containers join the infrastructure pool for a microservice.

In section 1, we described three types of COLD runs: *provider cold*, *VM cold*, and *container cold.* Each type of COLD run using Docker results in different performance overheads. *Provider cold* runs force the container image to be initially built/compiled requiring the most time. *VM cold* runs force the container image to be transferred to new hosts, while for *container cold*, images are already cached at the host and overhead is limited to container initialization time for creating new containers. Making a configuration change in Lambda is sufficient to generate *container cold* tests. Creating a new function is required to force requests to be *provider cold*, while we observed that waiting ~40+ minutes between requests is nearly infallible to generate *VM cold* runs. **Given the reliance of serverless computing infrastructure on Docker container infrastructure, the significance of container initialization overhead cannot be overlooked!** This reliance is not limited to only the AWS Lambda serverless computing platform [2] [19] [21].

### *B. Azure Functions Experimental Microservice*

For Functions, we developed an Http-Triggered Functions App, that contains a single function written in C#. The function logs the App Service Instance Id and the current worker process Id to an Azure Table [32], to provide information about the App Service Instance that services individual microservice requests. We utilized the Consumption Plan to assess automatically provided infrastructure for our Functions App. Function Apps store files on a file share in a separate storage account, thereby allowing files to be easily mounted onto new App Service instances as the app scales. We harnessed the Visual Studio Team System (VSTS) to implement performance load tests and to provide stress on our functions for the experiments [33].

We investigated COLD and WARM performance of our functions app. COLD runs measure the behavior of functions when provisioned for the very first time with newly assigned App Service Instances. We observed that once an App Service Instance was assigned to the function app, it's lifespan was at least 12 hours. Restarting the function app, or changing the code executed by the function did not assign a new app service instance. To force COLD runs, we created a new function app for each COLD run. We leveraged a URL-based load test, to perform stress tests against our function endpoint for a specified duration of time and with a given concurrency. For concurrency testing we scaled the number of requests as follows: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200. For COLD runs we tested a load duration of 2 minutes, while for WARM runs we tested 2, 5, and 10 minutes for each test.

For each microservice invocation the App Service Instance Id and Worker Process Id responding to the request were recorded in an Azure Table. We were able to capture the Service Instance Id from the environment variable **`WEBSITE_INSTANCE_ID`** set by Kudu in the Azure runtime environment [34]. We used the Power BI Desktop to view and analyze the data captured into our Azure Tables.

## V. EXPERIMENTAL RESULTS AND DISCUSSION

### *A. Docker Performance Comparison*

AWS Lambda hosts microservice code using fleets of Docker containers that are dynamically provisioned based on client demand. To compare performance of Lambda with "equivalent" infrastructure to gauge overhead of the platform, we harnessed Docker-Machine, a tool that enables remote hosts and containers to be provisioned on-the-fly [35]. Aided by Docker-Machine, we deployed our Calculations service code into our own Docker container independent of Lambda on EC2. The motivation for using Docker-machine was to emulate TCP networking overhead similar to that incurred by the AWS API

Gateway and Lambda function invocation. For both Docker-Machine and Lambda testing, we harnessed a c4.2xlarge 8 vCPU EC2 instance with "High" 1-Gigabit networking as a test client. We observed minimal load on our client while running remote tests.

TABLE II. DOCKER-MACHINE CONFIGURATIONS: MEMORY QUOTA AND MAXIMUM CPU CAPACITY

| Memory (MB) | Expected CPU% |
|---|---|
| *128* | 16.6% |
| *256* | 33.3% |
| *384* | 50.0% |
| *512* | 66.6% |
| *640* | 83.3% |
| *768* | 100.0% |
| *896* | 116.7% |
| *1024* | 133.3% |
| 1152 | 150.0% |
| *1280* | 166.60% |
| 1408 | 183.30% |
| 1536 | 200.00% |

Lambda states that every time memory is doubled, the CPU capacity is doubled [16]. Using this guideline, we calculated plausible ratios of memory and CPU resource allocations shown in Table II. To calculate these ratios, we assume that when Lambda has access to the maximum allowable memory (1536 MB, August 2017), it will have full access to 2 Intel Xeon E5-2666 v3 @ 2.90GHz vCPUs. Lambda memory limits were increased to 3008 MB in November 2017. We constrain our Docker containers by providing "--cpus" and "--memory" settings to Docker-Machine for container creation to enable our performance comparison as shown in Table II.

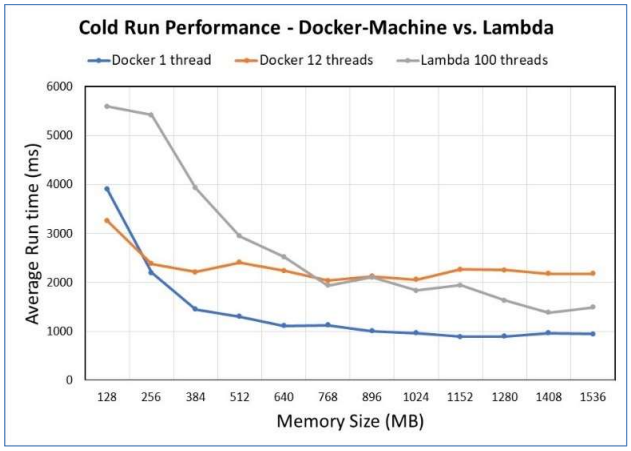


Fig. 1: Cold Performance:
Docker-Machine vs. AWS Lambda

We performed *container cold* and *container warm* test runs by hosting individual containers for 1 or 12 concurrent runs on a c4.8xlarge 36 vCPU EC2 instance. This instance type is selected because it is backed by the same CPU, the Intel Xeon E5-2666 v3 CPU, as used by the AWS Lambda platform. We evaluate Docker-Machine performance for 1 thread and 12 threads, because from our memory testing experiments (RQ-5) we observed that the average number of runs per container in Lambda across all tests was ~12.3. Our ultimate goal is to evaluate whether the serverless computing platform "overhead" is reasonable compared to an equivalent implementation using remote Docker containers to host code. Figure 1 details a COLD performance comparison, while figure 2 shows a WARM performance comparison. We observe that Lambda's WARM performance is quite good given the comparison to our Docker-Machine analog, while *container cold* performance could be improved. When hosting 12-threads per host, Docker-Machine outperformed Lambda for COLD runs when containers reserved 640MB or less. Lambda performance excelled beyond Docker-Machine performance for 12-thread COLD tests at 768MB and above. For WARM runs Lambda clearly outperforms out Docker-Machine analog for all tests. For memory configurations of 512MB and above, however, the performance slowdown averaged around a somewhat manageable ~41%. We posit that our Docker-Machine analog could match Lambda performance if we slightly reduce the number of containers per host. *Docker 12 threads* always underperforms *Docker 1 thread* due to CPU context switching.

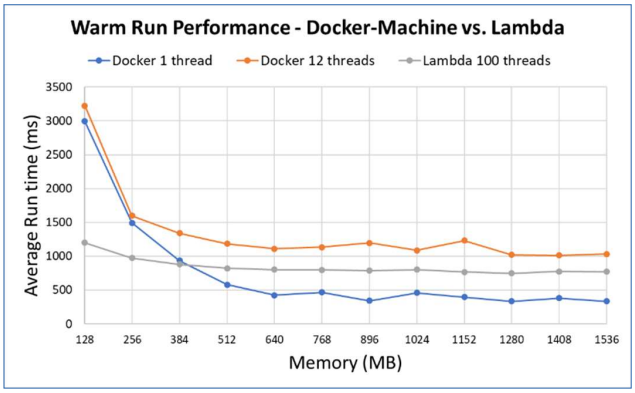


Fig. 2: Warm Performance:
Docker-Machine vs. AWS Lambda

### B. *Elasticity (RQ-1)*

To investigate RQ-1, we evaluated COLD performance of our Calculation service at stress level #4 on AWS Lambda with a 128MB deployment. We performed from 1 to 100 concurrent requests. We tested 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 60, 70, 80, 90, and 100 concurrent requests with a 10-second sleep between requests. By making an "advanced" configuration change to our Lambda function through the UI or CLI we were able to force runs to be *container cold*. Configuration changes such as modifying the container memory size or function timeout were sufficient to force containers to be deprecated forcing the platform to create new containers. Figure 3 shows *container cold* performance with an increasing number of concurrent requests.

For this slowly scaling workload the cloud provider creates the initial infrastructure and slowly scales up. For cold service

execution, from 1 to 100 requests, performance degraded by a factor of 3.8x. From 10 requests to 100, performance degrades by a factor of 2.6x. Cold service hosting required one Docker container for every request, and these containers leveraged from 1 to 7 host VMs as shown in figure 4. For this experiment VMs hosted an average of 11.3 service requests.

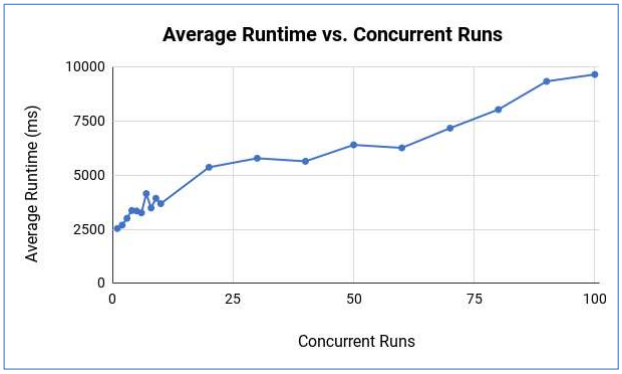


Fig. 3: AWS Lambda *Container cold* Calculation Service Scaling Performance

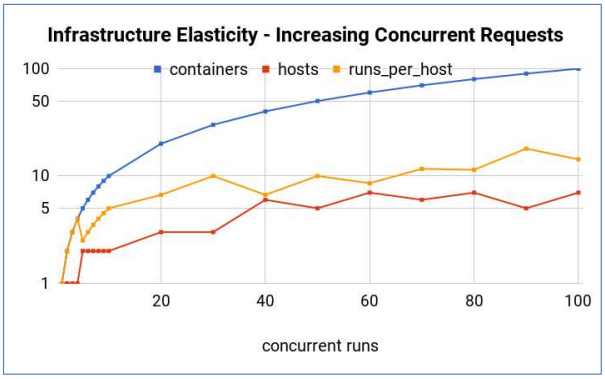


Fig. 4: AWS Lambda - Calculation Service Cold Scaling Test: Infrastructure Elasticity and Load Balancing

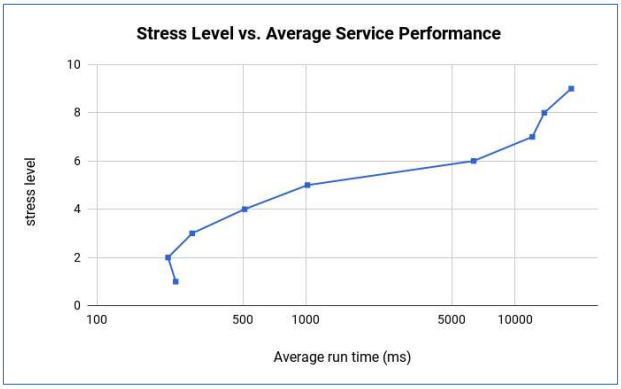


Fig. 5: AWS Lambda: Stress Level vs. Calculation Service Performance

### C. *Load Balancing (RQ-2)*

To investigate load balancing of *warm* service requests, we executed 10 sets of 100 concurrent requests using each of the calculation service stress levels described in table I. Each level reflects an increasing amount of required CPU time. Services requests were hosted using 128 MB containers on AWS Lambda with 10 seconds sleep between test sets. From level 1 to level 9, average service execution time increased by a factor of ~ 78x. Performance is shown in figure 5 where the x-axis has been plotted using a log scale.

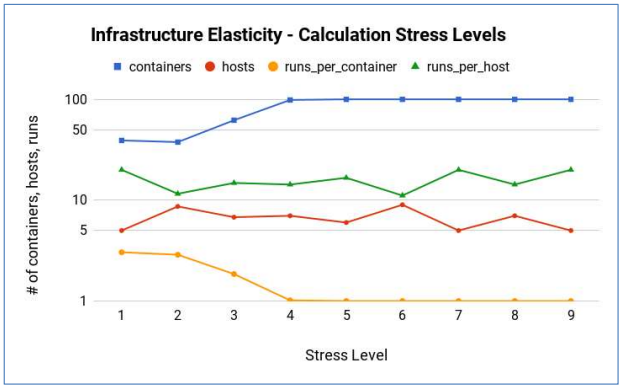


Fig. 6: AWS Lambda: Infrastructure Elasticity and Load Balancing vs. Microservice CPU Stress

Figure 6 shows infrastructure utilized by AWS Lambda to host 100 concurrent requests for our Calculation Service. Stress level 1 only required an average of 39 containers with each completing an average of ~3 requests. Each stress level required an increasing number of containers for hosting, with stress levels 5 and above requiring one container for each microservice request. The number of hosts (VMs) used to execute our service requests did not appear to vary based on the stress level of our service with 5 to 9 hosts being used. Load balancing of service requests across containers improved to a standard deviation of 0 for stress levels 5 and above, while the standard deviation of load balancing only improved slightly for service request distribution across VM hosts.

### D. *Provisioning Variation (RQ-3)*

To evaluate the impact of container placement across host VMs on service performance we captured data from our experiment examining elasticity for RQ-1. We computed a linear regression and found there is a very strong relationship ($R^2$=.9886) between the number of containers per host VM and the resulting COLD service performance. The data used for this regression is from stress level #4 service requests performed on 128 MB Docker containers. Figure 7 shows our regression plot.

This stress of container initialization on the host is significant. **When microservice infrastructure is initialized placing too many containers on the same host directly correlates with poor performance.** We hypothesize that container initialization requires substantial network and disk I/O contributing to poor performance.

Next, we next tested 3,000 concurrent WARM requests at stress level #9 while configuring our Lambda service to use 512

MB of memory. Due to client limitations, sending 3,000 requests required approximately ~1.6 minutes. This workload harnessed 218 containers across 47 host VMs where each container processed an average of 13.76 runs, and each VM an average of 63.83 runs. Linear regressions were, between containers per host (VM) and (1) average WARM service execution time ($R^2$=0.53), and (2) the number of requests per host ($R^2$=.063). These relationships, though not as strong as for COLD service execution with 128 MB containers, are still clear. We observed the maximum number of containers per host drop from ~26 for 128MB containers, to just ~12 containers for 512MB containers. Increasing container memory from 128MB to 512MB provided a performance improvement of 6.5x for our Calculations Service at Stress Level 9. Amazon suggests we should observe a 4x increase in CPU performance as performance is doubled twice from 1

28-to-256MB and 256-to-512MB. The additional performance improve could be attributed to fewer containers per host.

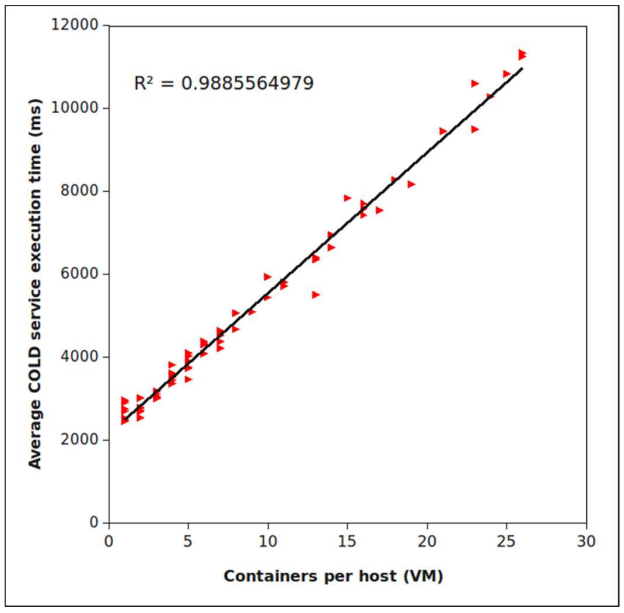


Fig. 7: AWS Lambda - Linear Regression: Container Placement vs. *Container Cold* Performance – Stress Level 4

### *E. Infrastructure Retention (RQ-4)*

We investigated how serverless computing infrastructure is retained over time to host microservices. Given our observations of poor performance for COLD service execution time seen while investigating RQ-1 and RQ-3 we already know how important infrastructure retention will be for service performance. For RQ-4 we sought to quantify precisely ***how much*** infrastructure (hosts and containers) is retained, and for ***how long***. We also investigated the microservice performance implications of infrastructure retention. To test container retention, we executed sets of 100 concurrent Calculation Service requests at Stress Level 4 on Lambda configured to use 128 MB Docker containers. Warming the service created 100 containers across 5 hosts. We then sent subsequent sets of 100 concurrent requests interspersed with .166, 1, 5, 10, 15, 20, 25, 30, 35, 40, and 45-minute idle periods. We captured the number of new and recycled containers and hosts (VMs) involved in the test sets.

When testing after 40 minutes of inactivity, no containers or VMs were recycled and all infrastructure was reinitialized from the *VM cold* state. For Stress Level 4 warm service execution time required only an average of 1.005 seconds. **When all infrastructure is *VM cold*, average service execution time increased over 15x to 15.573 seconds!** Service execution time increased as follows: 10 min ~ 2x, 15 min ~ 5x, 20 min ~ 6.9x, 25 min ~ 9.3x, 30 min ~ 13.4x, 35 min ~ 14.1x. After just 10 minutes of idle time 41.8% of the containers had to be recreated. An open question is, does infrastructure retention vary throughout the day based on system load? **Cloud providers should consider opportunistically retaining infrastructure for longer periods when there is idle capacity to help offset the performance costs of container initialization.**

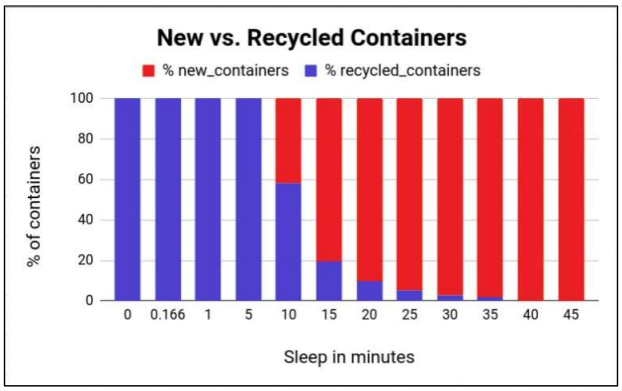


Fig. 8: AWS Lambda: Microservice Performance and (%) Recycled Containers for Long Duration Retention Tests

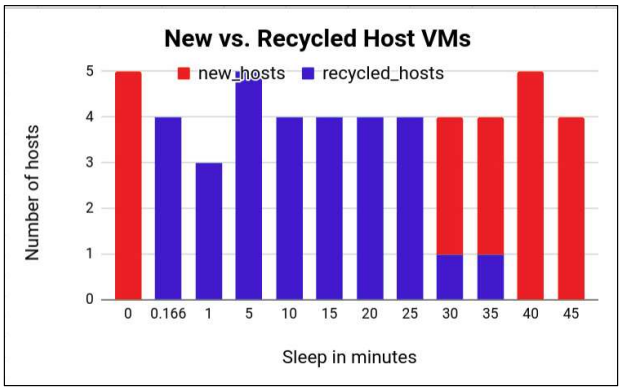


Fig. 9: AWS Lambda: New vs. Recycled Host VMs for Long Duration Retention Tests

For RQ-2, we observed that Stress Level 5 or higher is sufficient to involve all containers of 128MB size in service sets of concurrent service requests. If infrastructure is fully retained within 5 minutes, then executing 8,640 service sets a month at 5-minute intervals should help retain infrastructure. Complicating this service "warming" is a recent report that host VMs are recycled every ~4 hours [36]. Avoiding microservice performance degradation from infrastructure deprecation may require redundant service endpoints.

### F. *Memory Reservation (RQ-5)*

Lambda supports reserving memory from 128MB to 3008MB in 64MB increments. Additionally, CPU capacity is scaled proportional to memory with capacity doubling when memory is doubled presumably as we estimate in Table II. To investigate RQ-5, we are interested in examining how these capacity adjustments impact container density on VM hosts and service performance. We tested the following memory increments in MB: 128, 256, 384, 512, 640, 768, 896, 1024, 1152, 1280, 1408, and 1536 using our Calculation Service at Stress Level 4. We warmed infrastructure first, and then performed 10 sets of 100 concurrent runs for each memory configuration. We paused for 10 seconds between every set and performed 10 batches for a total of 1,200 sets. Graphs presented here represent averages across the batch of tests. Performance vs. memory reservation size is shown in figure 10, while figure 11 shows memory reservation size vs. active hosts (VMs).

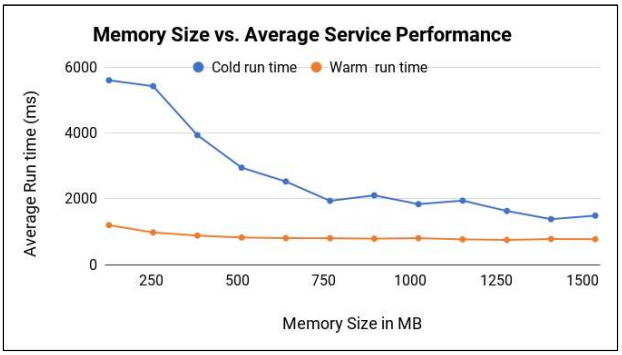


Fig. 10: AWS Lambda: Memory Reservation Size vs. Service Performance

We observed a ~4x performance boost for average COLD service execution time when increasing the function's memory reservation from 128MB to 1536MB. This is in contrast to the expected increase of 12x, if each time memory is doubled, performance doubles. **For WARM service execution time, we observed only a 1.55x performance improvement.** Memory configurations of 512MB achieve this improvement. Reserving (and paying for) memory beyond 512MB was not helpful to improve our WARM service performance. We posit that for our simple calculation service much of the execution time is actually overhead, and adding additional memory and CPU power is not sufficient to increasing the speed of Lambda framework overhead. Our results demonstrate the importance to profile microservice functions to determine an optimal memory reservation. Ad hoc tuning may be insufficient to guess an optimal memory and CPU performance setting. Users with minimal cost constraints or for hosting microservices with very light load, may opt to simply allocate the maximum memory to provide optimal COLD service performance.

Interesting behavior is seen in figure 11 regarding the number of host VMs. WARM and COLD 128MB deployments are shown to leverage a large number of hosts. Immediately, the number of hosts plummets at 256MB particularly for WARM runs. The unusual use of additional VMs for 128MB was limited to our investigation of RQ-5. When we executed similar tests at Stress Level 4 with 100 concurrent requests for other experiments only 5 to 7 host VMs were used. We determined that the use of additional hosts appears to be opportunistic in nature. These hosts were present from previous 1536MB tests and were reused for the subsequent 128MB test. While we typically observed around ~26 requests per host for WARM Service Level 4 requests against 128 MB containers, for this experiment the average number of requests per hosts was just 1.8! COLD requests per host dropped from approximately 20 to just 4.8. This *"host hangover"* effect was replicated every time (10 times) in repeated experimental runs. This effect helped to cut cold initialization average service execution time in half!

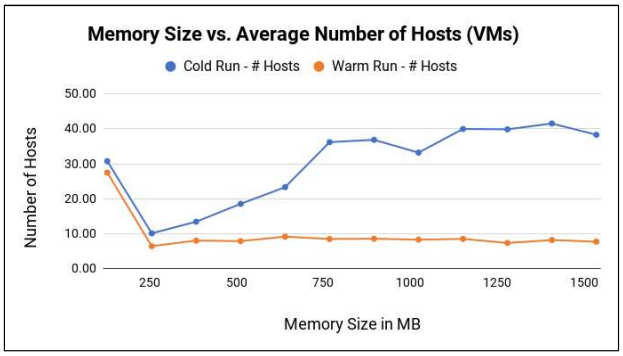


Fig. 11: AWS Lambda: Memory Reservation Size vs. Number of Hosts

### G. *Microsoft Azure Functions Elasticity (RQ-1)*

To investigate elasticity of infrastructure provided by the Azure Functions platform we performed scaling tests to measure service performance and infrastructure scalability for COLD service tests. We evaluated infrastructure scaling by performing a two-minute scaling load test. We increased concurrency every six seconds using the steps: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, and 200. Average execution time of our Azure function barely increased from 1 to 50 concurrent requests. Beyond 50 concurrent requests, average service execution time increased rapidly as shown in Figure 12. Cold service hosting utilized one worker process for each request while leveraging from 1 to 4 host VMs to host app service instances.

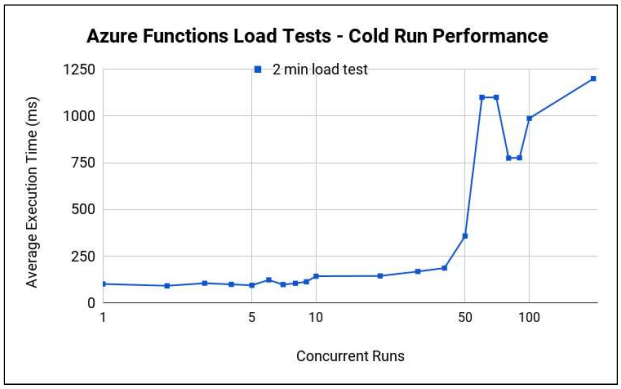


Fig 12: Azure Functions: Average Execution Time vs. Number of Concurrent Runs

Next, we investigated the number of App Service instances involved in hosting our microservice scaling workloads for WARM infrastructure. For this test, we did not create a new function app before each run, but reused our existing app to preserve previous infrastructure. For this test, we generated a continuous scaling load for 2, 5, and 10 minutes with scaling steps at ~ 6, 15, and 30 seconds respectively. We observed the number of App Service Instances that participate in hosting our scaling workloads as shown in Figure 13. We observed that for our 5-minute test, ~10% more App Instances were used, and for our 10-minute scaling test, ~90% additional App Instances were used. Given additional time, Azure Functions created additional App Instances as additional VMs were available ultimately enabling better microservice performance.

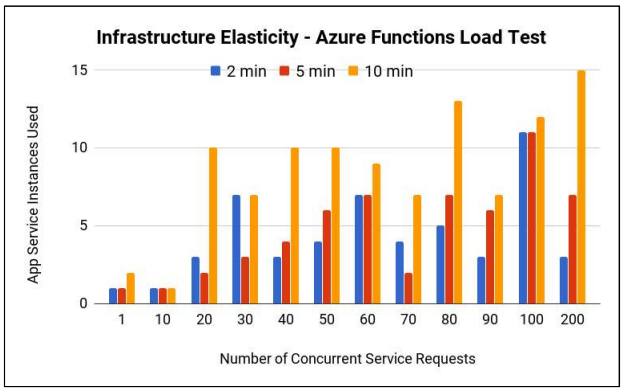


Fig 13: Azure Functions: Number of App Service Instances vs. Number of Concurrent Runs

## VI. Conclusions

In this study, we report our investigations on the performance implications of microservice hosting using Serverless Computing Infrastructure. Specifically, we profile microservice hosting and analyze implications of infrastructure *elasticity*, *load balancing*, *provisioning variation*, *infrastructure retention*, and *memory reservations*. For *elasticity* (RQ-1), **we found that extra infrastructure is provisioned to compensate for initialization overhead of COLD service requests.** Docker container initialization overhead significantly burdens serverless computing platforms, especially for *VM cold* initialization. Future service requests against WARM infrastructure do not always reuse extraneous infrastructure created in response to COLD initialization stress, with higher reuse rates corresponding with higher stress levels of microservice requests. With respect to *load balancing* of requests against serverless infrastructure (RQ-2), we observed well balanced distribution across containers and host VMs for COLD service invocations and for WARM service invocation at higher calculation stress levels. For low stress WARM service invocations, load distribution was uneven across hosts. This uneven use of infrastructure may lead to early deprecation if client workloads do not utilize all nodes.

Our investigations on provisioning variation (RQ-3) found that when too many container initialization requests go to individual host VMs, COLD service performance degrades up to 4.6x times. Once a VM participates in microservice hosting and the container image is cached, there is a tendency to stack containers at the host. We observed up to 26 collocated containers with a memory reservation size of 128MB, and ~12 containers at higher memory reservation sizes. Regarding infrastructure retention (RQ-4), we identified four unique states of serverless computing infrastructure: *provider cold*, *VM cold*, *container cold*, and *warm*. After 10 minutes, we observed that containers were deprecated first, followed by VMs, producing service performance degradation approaching 15x after 40 minutes of inactivity. **After 40 minutes all original hosting infrastructure for the microservice, containers and VMs, are no longer used.** We observed an average uptime of VMs participating in Lambda microservice hosting to be 2 hours and 8 minutes.

Regarding memory reservation sizes (RQ-5), we discovered that the coupling of memory and CPU power by Lambda significantly constrained microservice performance for low memory reservation sizes. To compensate Lambda allocates and retains as many as 4x more containers to host microservice workloads when memory reservation size is small. With WARM infrastructure, we observed performance improvements when increasing memory reservation size until reaching 512 to 640MB. Beyond this we observed diminishing returns as adding additional memory (and CPU power) did not significantly improve microservice performance. **Determining the optimal memory reservation size for microservices hosting requires benchmarking behavior on the platform.** Platform users without cost constraints may consider using the highest available memory reservation size 1536MB to ensure optimal COLD and WARM service performance.

## VII. Acknowledgements

This research is supported by a grant from the National Science Foundation's Computer Systems Research Program (CNS-1253908), the U.S. Department of Homeland Security (D15PC00279), and a Monfort Professorship at Colorado State University. Additionally, cloud computing resources were provided by the AWS Cloud Credits for Research, and a Microsoft Azure for Research award.

## References

[1] Yan M., Castro P., Cheng P., Ishakian V., Building a Chatbot with Serverless Computing. In Proceedings of the 1st International ACM Workshop on Mashups of Things and APIs, Trento, Italy, Dec 2016, 5 p.

[2] Hendrickson S., Sturdevant S., Harter T., Venkataramani V., Arpaci-Dusseau A.C., Arpaci-Dusseau R.H., Serverless computation with openlambda. In Procedings of the 8th USENIX Conference on Hot Topics in Cloud Computing (Hot Cloud '16), Denver, CO, June 2016, 7p.

[3] Baldini I., Castro P., Chang K., Cheng P., Fink S., Ishakian V., Mitchell N., Muthusamy V., Rabbah R., Slominski A., Suter P., Serverless Computing: Current Trends and Open Problems., arXiv preprint arXiv:1706.03178. June 2017, 20 p.

[4] Microservices, https://martinfowler.com/articles/microservices.html

[5] Openwhisk common use cases, https://console.bluemix.net/docs/openwhisk/openwhisk_use_cases.html#openwhisk_common_use_cases

[6] Sill A. The design and architecture of microservices. IEEE Cloud Computing. 2016 Sep;3(5):76-80.

[7] H. Liu, A Measurement Study of Server Utilization in Public Clouds, Proc. 9th IEEE International Conference on Cloud and Green Computing (CAG'11), Sydney, Australia, Dec 2011, pp.435-442.

[8] AWS Documentation: Concepts – Auto Scaling , 2013, http://docs.aws.amazon.com/AutoScaling/latest/DeveloperGuide/AS_Concepts.html

[9] M. Rehman, M. Sakr, Initial Findings for Provisioning Variation in Cloud Computing, Proc. of the IEEE 2nd Intl. Conf. on Cloud Computing Technology and Science (CloudCom '10), Indianapolis, IN, USA, Nov 30 - Dec 3, 2010, pp. 473-479.

[10] W. Lloyd et al., Performance implications of multi-tier application deployments on IaaS clouds: Towards performance modeling, Future Generation Computer Systems, v.29, n.5, 2013, pp.1254-1264.

[11] Aderaldo C., Mendonça N., Pahl C., Jamshidi P., Benchmark requirements for microservices architecture research. In Proceedings of the 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering, May 2017, pp. 8-13.

[12] Kecskemeti G., Marosi A., Kertesz A., The ENTICE approach to decompose monolithic services into microservices. In International Conference on High Performance Computing & Simulation (HPCS 2016), July 2016, pp. 591-596.

[13] Hassan S., Bahsoon R., Microservices and their design trade-offs: A self-adaptive roadmap. In 2016 IEEE International Conference on Services Computing (SCC 2016), June 2016, pp. 813-818.

[14] Granchelli G., Cardarelli M., Di Francesco P., Malavolta I., Iovino L., Di Salle A., Towards Recovering the Software Architecture of Microservice-Based Systems. In 2017 IEEE International Conference on Software Architecture Workshops (ICSAW 2017), April 2017, pp. 46-53.

[15] Frey S., Fittkau F., Hasselbring W., Search-based genetic optimization for deployment and reconfiguration of software in the cloud. In Proceedings of the 2013 International Conference on Software Engineering (ICSE 2013), May 2013, pp. 512-521.

[16] AWS Lambda – Serverless Compute, https://aws.amazon.com/lambda/

[17] OpenWhisk, https://console.bluemix.net/openwhisk/

[18] Azure Functions – Serverless Architecture, https://azure.microsoft.com/en-us/services/functions/

[19] Cloud Functions – Serverless Environments to Build and Connect Cloud Services | Google Cloud Platform, https://cloud.google.com/functions/

[20] Serverless Architectures, https://martinfowler.com/articles/serverless.html

[21] McGrath G., Brenner P., Serverless Computing: Design, Implementation, and Performance. In IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW 2017), June 2017, pp. 405-410.

[22] Islam S., Keung J., Lee K., Liu A., Empirical prediction models for adaptive resource provisioning in the cloud. Future Generation Computer Systems. 2012 Jan 31;28(1):155-62.

[23] Nikravesh A., Ajila S., Lung C., Towards an autonomic auto-scaling prediction system for cloud resource provisioning. In Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, May 2015, pp. 35-45.

[24] Nikravesh AY, Ajila SA, Lung CH. Measuring prediction sensitivity of a cloud auto-scaling system. In IEEE 38th International Computer Software and Applications Conference Workshops (COMPSACW 2014), July 2014, pp. 690-695.

[25] Qu C., Calheiros R., Buyya R., A reliable and cost-efficient auto-scaling system for web applications using heterogeneous spot instances. Journal of Network and Computer Applications. 2016 Apr 30;65:167-80.

[26] Limit a container's resources, https://docs.Docker.com/engine/admin/ resource_constraints/#configure-the-default-cfs-scheduler.

[27] Understanding AWS Lambda Coldstarts, https://www.iopipe.com/2016/09/understanding-aws-lambda-coldstarts/

[28] Choose between Flow, Logic Apps, Functions, and WebJobs, https://docs.microsoft.com/en-us/azure/azure-functions/functions-compare-logic-apps-ms-flow-webjobs

[29] Operating system functionality on Azure App Service , https://docs.microsoft.com/en-us/azure/app-service/web-sitesavailable-operating-system-functionality#file-access

[30] Making Azure Functions more "serverless", https://blogs.msdn.microsoft.com/appserviceteam/2016/11/15/making-azure-functions-more-serverless/

[31] Guidance for developing Azure Functions, https://docs.microsoft.com/en-us/azure/azure-functions/functions-reference

[32] Test your Azure web app performance under load from the Azure portal, https://docs.microsoft.com/en-us/vsts/load-test/app-service-web-app-performance-test

[33] Azure Functions hosting plans comparison , https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale

[34] Azure runtime environment, https://github.com/projectkudu/kudu/wiki /Azure-runtime-environment

[35] Docker Machine, https://docs.Docker.com/machine/.

[36] Understanding AWS Lambda Coldstarts, https://www.iopipe.com/2016/09/understanding-aws-lambda-coldstarts/

[37] W. Lloyd, S. Pallickara, O. David, M. Arabi, T. Wible, J. Ditty, and K. Rojas, "Demystifying the Clouds: Harnessing Resource Utilization Models for Cost Effective Infrastructure Alternatives," in IEEE Transactions on Cloud Computing, vol. 5, no. 4, pp. 667-680, Oct.-Dec. 1 2017.

[38] W. Lloyd, S. Pallickara, O. David, M. Arabi and K. Rojas, "Mitigating Resource Contention and Heterogeneity in Public Clouds for Scientific Modeling Services," 2017 IEEE International Conference on Cloud Engineering (IC2E), Vancouver, BC, 2017, pp. 159-166.