Improving Application Migration to Serverless Computing Platforms: Latency Mitigation with Keep-Alive Workloads

Minh Vu[1#], Baojia Zhang[2#], Olaf David[4], George Leavesley[5], Wes Lloyd[3]

School of Engineering and Technology
University of Washington
Tacoma, Washington USA
[1]minvu, [2]bjzhang, [3]wlloyd@uw.edu
#denotes equal contribution

Object Modeling System Laboratory
Colorado State University
Fort Collins, Colorado USA
[4]odavid, [5]ghleaves@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 abstract infrastructure management including creation of virtual machines (VMs), containers, and load balancing from users. To conserve cloud server capacity and energy, cloud providers allow serverless computing infrastructure to go COLD, deprovisioning hosting infrastructure when demand falls, freeing capacity to be harnessed by others. In this paper, we present on a case study migration of the Precipitation Runoff Modeling System (PRMS), a Java-based environmental modeling application to the AWS Lambda serverless platform. We investigate performance and cost implications of memory reservation size, and evaluate scaling performance for increasing concurrent workloads. We then investigate the use of Keep-Alive workloads to preserve serverless infrastructure to minimize cold starts and ensure fast performance after idle periods for up to 100 concurrent client requests. We show how Keep-Alive workloads can be generated using cloud-based scheduled event triggers, enabling minimization of costs, to provide VM-like performance for applications hosted on serverless platforms for a fraction of the cost.**

Keywords—Resource Management and Performance; Serverless Computing; Function-as-a-Service; Application Migration;

I. Introduction

Serverless computing recently has emerged as a compelling new approach for hosting applications in the cloud [1] [2] [3]. Serverless computing platforms provide Function(s)-as-a-Service (FaaS) by automatically managing compute infrastructure to host individual callable functions on-demand. Functions are deployed as independent code modules to provide "microservice" building blocks for new cloud-native applications. Serverless platforms offer reduced hosting costs, high availability, fault tolerance, and dynamic elasticity with automatic management of compute infrastructure by integrating support for these features directly into the platforms [4].

In FaaS, application containers hosting code plus dependent libraries are created and managed by cloud providers to provide granular compute infrastructure for each microservice [5]. Cloud providers are responsible for creating, destroying, and load balancing requests across container instances. Users are billed based on the total number of service calls and their associated runtime vs. memory utilization to the nearest tenth of a second. Serverless platforms have arisen to support highly scalable, event-driven applications consisting of short-running, stateless functions triggered by events generated from middleware, sensors, microservices, or users [6]. Early use cases have included multimedia processing, IoT data aggregation, stream processing, chatbots, short batch jobs/scheduled tasks, REST APIs, mobile backends, and continuous integration pipelines [7]. Given the many advantages of serverless computing platforms, there is considerable motivation for their adoption for a broader variety of use cases.

A. Application Migration to Serverless Computing Platforms

Application migration to serverless computing platforms involves transfer of legacy application code to run as one or more FaaS functions. Depending on application size, it is possible to migrate entire applications with minimal refactoring and recomposition. This can be of interest when resource limitations, complexity, or required developer effort make it infeasible to refactor applications [8]. Monolithic deployments provide a starting point to explore tradeoffs of serverless application hosting before committing substantial effort into refactoring. Monolithic deployments are viable when legacy applications fit within platform code size constraints set by cloud providers inclusive of source code and libraries. Serverless platforms also cap the maximum execution time for individual function calls to approximately five minutes, though worm functions provide a potential workaround [9].

B. Serverless Infrastructure Freeze/Thaw Cycle

To save server capacity, cloud providers automatically deprecate serverless infrastructure after periods of inactivity [10]. Recycling of infrastructure on serverless platforms is known as the **freeze/thaw cycle** [11]. On AWS Lambda, after approximately 45-minutes of inactivity, subsequent calls to an endpoint reveal no trace of original function containers or their host VMs [12]. Consequently, future calls force initialization of new server infrastructure adding latency to service response times. We detail how the variable state of infrastructure results in performance variation for hosted services compared to traditional hosting with Infrastructure-as-a-Service (IaaS) platforms in section V. D.

C. Application Migration Case Study

This paper reports on the migration of the Precipitation Runoff Modeling System (PRMS), an environmental modeling application, to the AWS Lambda serverless computing platform. We perform a monolithic deployment of PRMS to run as a single function to investigate performance, scalability, and cost implications for hosting on the AWS Lambda serverless platform. The compressed code size of Java-based PRMS is 18MB, approximately 36% of Lambda's platform constraint of 50MB, making PRMS an ideal candidate to study for migration. Unlike typical FaaS microservices which are likely to have small code sizes (e.g. < 100 KB), PRMS, as a larger application, experiences considerably more infrastructure initialization overhead. Our goal was to quantify this overhead and seek ways to amortize it. Lambda was chosen because of native support for Java. Google Cloud Functions initially has supported code deployments only in Node.js. Azure functions version 1.x did not initially support Java, with support added in version 2.x. Extending our evaluation to include Azure functions 2.x and IBM Cloud Functions remains as future work.

D. Preserving Serverless Infrastructure

When hosting web services using VMs provided by IaaS clouds, a key issue is scalability. Elastic load balancing schemes are often devised to respond to current and/or future service demand to adjust the provisioned number of VMs. Provisioning VMs is notoriously slow particularly when VMs require additional initialization beyond OS boot up. With serverless computing platforms, infrastructure scaling and load balancing are automatic, and developers have no ability to control the creation and/or retention of infrastructure. The recommended workaround is to configure one or more clients to automatically trigger serverless functions at regular intervals to preserve infrastructure to mitigate cold start latency [13]. We name these clients, **Keep-Alive** clients, and their sole purpose is not to execute the service, but to ping FaaS infrastructure to prevent deprecation after periods of inactivity. In this paper, we investigate the use of Keep-Alive clients to mitigate cold start latency for PRMS. We additionally compare the cost and ability of alternative Keep-Alive client implementations.

E. Research Questions

RQ-1: (*Performance*) What are the performance implications of leveraging serverless computing infrastructure for application migration? How does memory reservation size when coupled to CPU power impact performance?

Platforms such as AWS Lambda and Google Cloud Functions allow users to reserve memory for individual function deployments. Memory reservation size for these platforms is coupled to CPU processor power. Google provides specific CPU clock frequencies for alternate memory reservation sizes [14], while Amazon reports that when memory is doubled, CPU power, network bandwidth, and disk I/O throughput is roughly doubled in the same manner as EC2 VMs [15] [16]. We investigate the impact of memory reservation size on service performance with our PRMS case study.

RQ-2: (*Scalability*) For application migration what performance implications result from scaling the number of concurrent clients? How is scaling affected when infrastructure is allowed to go cold?

Scalability on serverless computing platforms is impacted by the state of serverless infrastructure. Cloud providers deploy functions to containers hosted on preinitialized VMs to alleviate launch latency. Serverless infrastructure states include: *VM-cold*, *Container-cold*, and *warm* [12]. We investigate scaling performance relative to infrastructure state for PRMS.

RQ-3: (*Cost*) For hosting large parallel service workloads, how does memory reservation size, when coupled to CPU power, impact hosting costs?

Serverless platforms embody the cost vs. performance tradeoff by coupling memory reservation size to CPU power for function deployments. We leverage PRMS to investigate this tradeoff space as intuition is insufficient to infer the best configurations.

RQ-4: (*Persisting Infrastructure*) How effective are automatic triggers at retaining serverless infrastructure to reduce performance latency from the serverless freeze/thaw cycle?

We investigate the use of Keep-Alive clients to prevent idle infrastructure from being deprecated to sustain *warm* performance for up to 24 hours. Leveraging PRMS, we persist 100 containers and compare the performance and cost of using alternative clients to generate Keep-Alive workloads.

F. Contributions

The primary contributions of this paper include:

1. A case study on application migration for the Java-based PRMS to a serverless platform. We contrast deployment implications of memory reservation size vs. cost, performance, and scalability.
2. An investigation of Keep-Alive clients to persist serverless infrastructure to reduce performance latency resulting from the freeze/thaw cycle.
3. Identification of infrastructure management trends and the extent of performance variation on the AWS Lambda serverless platform.

II. Background and Related Work

Commercially provided serverless computing platforms provide dynamic scalable infrastructure on-demand to host microservice applications [17][18][19][20]. Fundamentally different than application hosting with IaaS or Platform-as-a-Service (PaaS) clouds, serverless platforms enable native cloud applications to be built by composing together separate microservices. One new challenge involves tracking application state and workflow, identified as the *composition-as-function* problem by Baldini [6]. Eivy and Weinman identify that serverless computing moves the cloud computing cost model from pay-for-allocation to pay-for-use, as IaaS clouds focused on billing for reserved resources that may often be idle [21]. Eivy noted that the best cloud infrastructure to host 1,000,000 service requests depends greatly on how requests are distributed. If requests are distributed evenly, then IaaS cloud may be less expensive. If demand is bursty in nature, FaaS is likely the most economical choice. In [22], Eyk et al. identified the need to identify trade-off points of FaaS and IaaS platforms for application deployment. They noted that complex pricing models of serverless platforms make determining the most cost-effective deployments more challenging, resulting in a need to automate cost evaluation to support informed application deployment decisions.

Jonas et al. evaluate the use of serverless computing with four diverse HPC use cases including: calculating π, facial recognition, password cracking, and precipitation forecasting [54]. For password cracking, the authors devise a map-reduce approach called function futures that operates similar to PyWren. PyWren enables existing Python code to be run at massive scale on AWS Lambda [23]. Jonas et al. identify FaaSification as the process of converting legacy code to FaaS functions. To enable their precipitation forecasting use case they introduced worm functions to work around execution time limits of individual function calls. Worm functions track function execution time and call a new FaaS function to transfer the computation to a new instance shortly before timeout. They offer a tool known as Snafu to abstract deployment of functions to multiple platforms: AWS Lambda, IBM Cloud Functions, and Google Cloud Functions. Jonas's use cases consisted entirely of Python applications. In this paper, we contribute a new study on the migration of the Java-based PRMS scientific application.

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 consequence of optimal design [24]. The use of containers generates infrastructure management overhead as platforms must constantly shuffle containers to and from host VMs to share platform infrastructure for many users. Oakes et al. developed an approach to reduce initialization overhead for larger FaaS functions by introducing a package caching mechanism to speed function deployment known as "Pipsqueak". Their approach reduces package sizes by deploying functions to containers with predeployed Python libraries [5]. By leveraging predeployed libraries, FaaS function package sizes can be smaller enabling deployment to be more agile. Oakes built and verified their approach within the OpenLambda open source serverless framework developed to support research on serverless management schemes [2]. Abad, Boza, and Eyk further leveraged Pipsqueak by offering an improved scheduler with higher package cache hit rates by consolidating function deployments to infrastructure sharing the same packages [25].

Eyk et al. also identified performance challenges for serverless computing including "Reducing FaaS overhead" in [22]. Infrastructure provisioning overhead was identified as the dominant overhead on serverless platforms. Eyk notes that provisioning overhead, the time spent to create containers and VMs for first use by serverless platforms, requires from seconds to minutes. Eyk suggested amortizing this overhead by avoiding cold deployments for every request by reusing infrastructure to achieve hot starts whenever possible. Albuquerque et al. suggested that cold start initialization latency could be avoided through the use of an external "heartbeat" routine to keep serverless resources permanently active [29]. They did not report building such routines, or evaluate their effectiveness for sustaining serverless infrastructure. In this paper, we evaluate Keep-Alive workloads for sustaining high performance concurrent serverless workloads with our PRMS use case.

III. Experimental Resources

To investigate our research questions, we harnessed the AWS Lambda serverless computing platform, a compute-bound experimental service from [12], and PRMS [26] deployed as a monolithic service.

AWS Lambda, introduced in 2014, deploys and runs code in container like environments on 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 for microservices supporting by default up to 1,000 concurrent requests. As of fall 2018, functions are provided access to 2 hyperthreads scaled relative to memory backed by the Intel(R) Xeon(R) E5-2666 v3 @ 2.90GHz CPU. Amazon reports that for every doubling of memory, CPU power, network bandwidth, and disk I/O throughput is roughly doubled in the same manner as EC2 VMs [15] [16]. 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.

Lambda Experimental Service To support experiments and devise our Keep-Alive approach, we harnessed our Lambda compute-bound experimental "calcs" service from [12]. The service can be run to generate an artificial CPU load by performing random math calculations (multiplication and division). To vary the degree of memory stress, calculations are performed using operands stored in separate large arrays of configurable size on the heap. Array indexes are selected randomly for each calculation to induce memory page faults in contrast to sequential array traversal. The experimental service can also be invoked to simply sleep for a fixed duration in milliseconds without inducing a CPU load.

Precipitation Runoff Modeling System, (PRMS) was deployed as a monolithic Lambda function to provide a proof-of-concept case study to investigate legacy application migration to a serverless computing platform. We leveraged a Java based implementation of the 2008 version of the Precipitation-Runoff Modelling System (PRMS) [26]. PRMS is a deterministic, distributed-parameter model developed to evaluate the impact of various combinations of precipitation, climate, and land use on stream flow and general basin hydrology. The Java based version of PRMS, implemented using the Object Modelling System (OMS) 3.0 component-based modelling framework [27], was deployed to Lambda. This version of PRMS consists of approximately ~11,000 lines of code and compiles to a compressed and uncompressed Jar file size of 18MB and 67 MB respectively.

IV. Experimental Setup

To support our experiments, we deployed our experimental "calcs" service and the PRMS application as Lambda functions. Bash client scripts harnessed the AWS command line interface (CLI) to invoke functions synchronously. Techniques from [12] were used to characterize serverless infrastructure automatically provisioned by the cloud provider. These techniques allowed us to identify the number of unique containers and VMs used to host our workloads, and also to observe load balancing of service requests.

We executed bash client scripts on Ubuntu 16.04 c4.2xlarge 8-vCPU and c4.4xlarge 36-vCPU EC2 instances with "High" (1 Gbps) and "10 Gigabit" networking performance. We pinned EC2 instances and Lambda functions to run using the default virtual private cloud (VPC) in the us-east-1e availability zone. Availability zone assignments are relative to individual user accounts on AWS. Users experience different zone mappings to balance resource provisioning across all cloud users. We deployed PRMS to a single availability zone to eliminate performance variation from deployments spanning multiple zones. Client VMs were created in the same availability zone to minimize network latency between EC2 and Lambda. We leveraged the GNU parallel library to facilitate parallel concurrent workloads. We modeled runoff for the East

Fork of the Carson River near Gardnerville, USGS station 10309000, a basin area of ~356 mi^2. PRMS input datafiles were 118KB in CSV format and preloaded to an S3 bucket. JSON model inputs sent to PRMS were minimal in size and included only file pointers to data in S3.

For PRMS Keep-Alive workloads, a single c4.8xlarge 36 vCPU VM was used to submit 100 Lambda requests using concurrent threads. The c4.8xlarge VM was fast enough to force AWS Lambda to provision separate containers for each request even when PRMS was allowed up to 3008MB of memory. Using a c4.2xlarge 8 vCPU client VM, Lambda only provisioned separate containers when the function memory reservation was 896 MB or less. With higher Lambda memory allocations, service performance increased and some requests completed before the c4.2xlarge could submit 100 requests. We also leveraged AWS CloudWatch events to generate PRMS Keep-Alive workloads [28]. CloudWatch events provide a general-purpose event stream where rules can be configured to respond to events by performing actions such as calling an AWS Lambda function. Scheduled events can be configured to automatically trigger Lambda functions on a regular basis similar to Linux cron jobs. Presently there is no cost to generate scheduled events on AWS, eliminating the cost of renting a VM as a Keep-Alive client. Default account limits enable up to 100 CloudWatch rules to be configured with 5 targets each to generate up to 500 Lambda calls at scheduled intervals.

V. Experimental Results and Discussion

A. *RQ-1: Performance vs. Memory*

To investigate the impact of memory reservation size on PRMS performance we scaled from 256MB to 3008MB. 256MB was the minimum memory required by PRMS. We performed 100 concurrent PRMS model runs using a c4.2xlarge and c4.8xlarge EC2 instance as a client. To ensure performance measurement of only preinitialized infrastructure, we first submitted 3 separate sets of 100 concurrent requests to thoroughly warm infrastructure, and captured performance data for subsequent sets. Our scripts verified all infrastructure was warm. Figure 1 depicts performance speedups. We observed a 4.3x and a 10.1x performance speed-up using our c4.2xlarge and c4.8xlarge clients respectively when scaling from min to max memory- an order of magnitude performance improvement for PRMS. For high memory configurations, the c4.2xlarge client with only 8 vCPUs became the bottleneck, not Lambda, for performing 100 concurrent requests. In this case, Lambda completed requests faster than a c4.2xlarge could generate them. This bottleneck is shown by the decreasing number of containers in figure 2 beyond 896 MB. Figure 2 also depicts the number of VMs for our PRMS Lambda function growing linearly with memory until a jump at 1792MB. When increasing memory for PRMS, the total memory for all PRMS containers on a single VM host appears to grow to ~4 GB until the ratio of VMs doubles from 42 to 85 preventing this limit from being surpassed.

Lambda claims that performance doubles for every doubling of memory. Figure 3 compares performance gains for PRMS when increasing memory reservation size in Lambda vs. expected linear performance gains based on performance measurements at 256 MB. Lambda provided better than linear performance gains for memory reservations sizes less than 1024MB, but failed to keep pace beyond as linear performance gains would be slightly greater than measured Lambda performance.

B. *RQ-2 Scalability Performance*

We tested the scalability performance of PRMS deployed to Lambda by scaling stepwise from 1 to 100 concurrent requests at 512MB and 1664MB. By scaling slowly, Lambda could provision VMs individually and slowly fill them with containers. At 512MB, new VMs were added for every 6 requests. Once added, additional runs executed faster on the same VM as a PRMS container image was likely cached. At 1664MB, new VMs were added for every 2 requests except after 80 requests when new VMs processed 6 requests reducing performance. The resulting performance is shown in Figure 4. We then tested COLD scaling performance by scaling from 1 to 100 in steps of 10, while waiting 45 minutes between subsequent calls. We observed no scaling performance benefit with long delays between sets of concurrent requests shown in Figure 5 as the platform did not retain infrastructure.

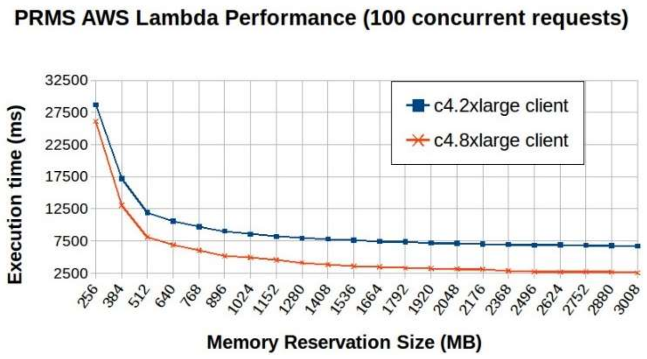

Fig. 1: PRMS Performance vs. Memory Reservation Size

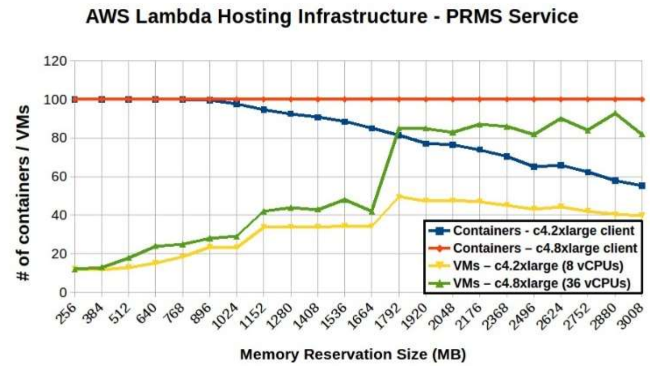

Fig. 2: PRMS Infrastructure vs. Memory Reservation Size

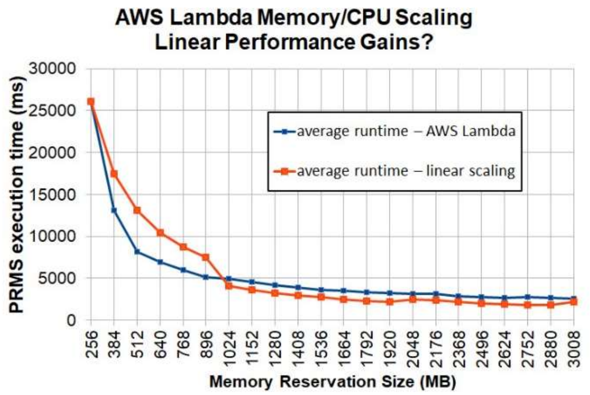

Fig. 3: PRMS Performance Gain vs. Memory: Linear Performance vs. Lambda

C. RQ-3 Cost

Based on our performance results relative to memory reservation size we estimated the cost to complete 1,000,000 PRMS model runs as shown in Figure 6. We assumed the use of a client to generate 1,000 concurrent Lambda requests. The least expensive memory size for 1,000,000 runs was 512MB with execution requiring ~2.26 hours for a total cost of $66.20. At 3008MB, runs could be completed in just ~.71 hours at a total cost of $124.92. This result demonstrates the importance of profiling performance for determining the best memory reservation size for an application. Depending on application CPU requirements, reducing CPU power too far via memory reservation can increase hosting costs.

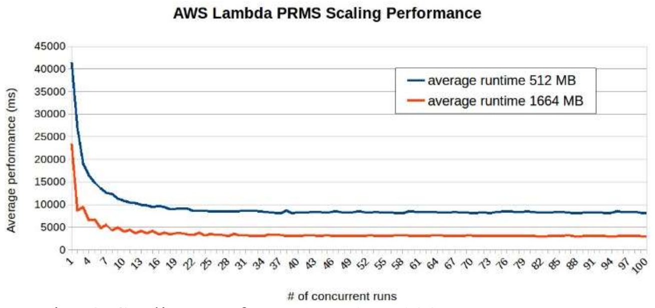

Fig. 4: Scaling Performance: 1 to 100 concurrent requests

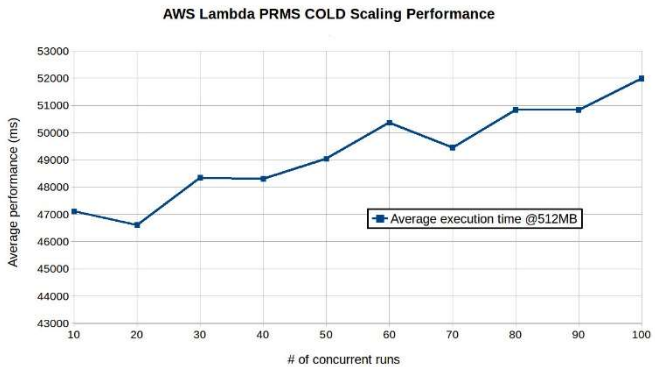

Fig. 5: PRMS Cold Scaling Performance

D. RQ-4 Keep-Alive Infrastructure Preservation

We first leveraged the c4.8xlarge ec2 instance as a client to generate a Keep-Alive workload for PRMS. Our objective was to preserve 100 containers for a 24-hour period to negate the serverless freeze/thaw cycle for a concurrent workload. We analyzed how infrastructure was created, retained, and replaced for 24-hour periods using our experimental "calcs" service at 192MB, 256MB, 384MB, and 512MB. In Figure 7, we depict the time from initial infrastructure creation until replacement begins ranging from 4.75 to 7.75 hours. When replacement started, all infrastructure was slowly replaced over a period of ~2 hours. After this time no original infrastructure (VMs or containers) could be detected. New infrastructure "generations" produced performance variation ranging from -14.7% to +19.4% of average performance for our "calcs" service. Every 6-8 hours when infrastructure was automatically replaced, performance was found to vary by up to 34%. Addressing performance variation from infrastructure provisioning variation represents an open problem in serverless computing. The performance variation was more pronounced with smaller memory reservation sizes from an average of 9% at 192MB to only 3.6% at 512MB. Figure 7 shows a negative correlation between service demand and retention. When we generated more service requests per hour, Lambed initiated replacement of infrastructure sooner (p=.001).

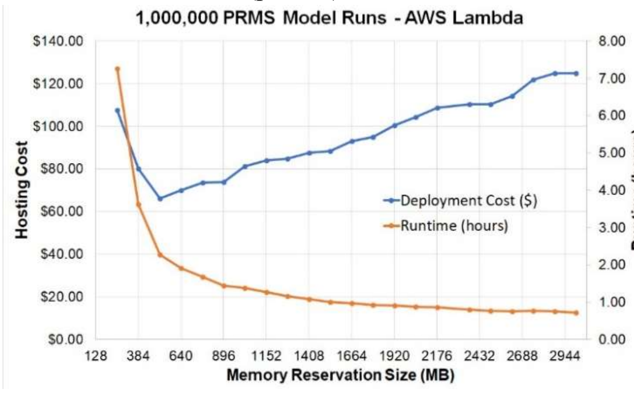

Fig. 6: Cost and Runtime vs. Memory Reservation 1,000,000 PRMS Model Runs

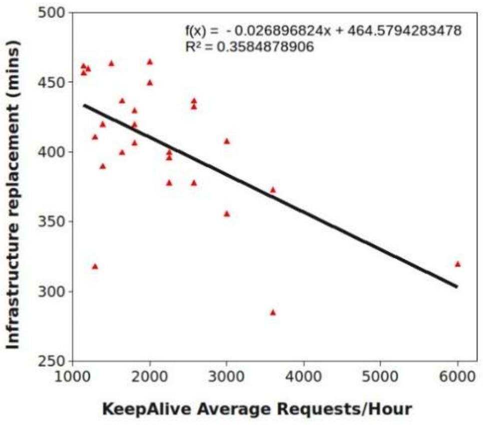

Fig. 7: AWS Lambda- Time to Infrastructure Replacement

VM-client: To Keep-Alive 100 containers for PRMS we first generated a periodic workload using a c4.8xlarge ec2 instance. We added an alternate parameterization to PRMS to perform a fixed number of "calcs" operations as in [12], to ensure 100 concurrent requests overlapped in time but with a shorter runtime than executing PRMS to save costs. Later, we refactored to sleep only as busy operations were not required for infrastructure preservation. We generated 100 Keep-Alive requests for 24-hours at 3, 4, and 5-minute intervals. While running, a separate c4.8xlarge VM invoked 100 PRMS model runs at 45-minute intervals to measure performance.

CloudWatch client: We next configured 20 CloudWatch rules and 5 targets to generate 100 PRMS requests every 5 minutes but found that CloudWatch performance limitations required a sleep duration of 49 seconds to Keep-Alive all containers. To work around this performance limitation, we configured 100 CloudWatch rules with 5 targets each to produce 500 calls that slept 5-seconds each at 5-minute intervals to preserve PRMS containers. The ***5-target x 100-rules x 5sec*** CloudWatch Keep-Alive configuration resulted in less total Lambda execution time and lower costs than the ***1-target x 100-rules x 49sec*** configuration.

Table I provides an analysis of Keep-Alive performance. These tests used a 512MB memory reservation size for PRMS.

Due to the non-deterministic nature of Keep-Alive request scheduling relative to PRMS client activity, over 24-hour periods we observed a ~10% slowdown compared to Lambda WARM performance. Our speedup, however, was ~400% faster than Lambda COLD performance. Table II summarizes annual costs assuming no free-tier resources for hosting Keep-Alive infrastructure for 100 concurrent PRMS requests. Without free tier resources, 100,000 PRMS runs cost ~$9.50. Annual hosting costs to support 100 concurrent PRMS model runs with Lambda + CloudWatch Keep-Alive were 17.6x and 5.5x *less expensive* than hosting PRMS with on demand or spot EC2 c4 instances respectively. Serverless infrastructure (e.g. Lambda + CloudWatch Keep-Alive) can provide a less expensive alternative for highly available and responsive application hosting compared to IaaS cloud.

TABLE I. PRMS KEEP-ALIVE CLIENT ANALYSIS

Keep-Alive client type:	c4 VM 5 min	c4 VM 4 min	c4 VM 3 min	CloudWatch 5 min	CloudWatch 4 min
PRMS perf avg(ms)	11,305	10,971	10,052	11,136	13,465
Slowdown vs. WARM	13.3%	10%	0.7%	11.6%	35%
Speedup vs.COLD	403%	415%	453%	409%	338%
Average new containers/test	2.4	2.8	0.4	5.4	14.7
Total new containers	77	90	12	141	250
Test duration (hours)	24	24	24	18	12
Keep-Alive client cost/hour	$12.24	$12.24	$12.24	$0	$0
Keep-Alive runtime avg(ms)	4,492	4,407	4,463	~5,000	~5,000
Keep-Alive calls/set	100	100	100	500	500
Memory (GB-sec/hour)	2695	3305	4463	15,600	19,500

TABLE II. KEEP-ALIVE ANNUAL COST - 100 USERS

PRMS Host Infrastructure	Total	Savings
Lambda + EC2 Keep-Alive 3min	$4,494.76	892%
Lambda + EC2 Keep-Alive 4min	$4,487.71	893%
Lambda + EC2 Keep-Alive 5min	$4,484.00	894%
Lambda + CloudWatch Keep-Alive 5min	$2,278.06	1759%
Lambda + CloudWatch Keep-Alive 4min	$2,847.57	1407%
Spot c4 ec2 instances	$12,579.84	319%
On Demand c4 ec2 instances	$40,077.00	baseline

VI. CONCLUSIONS

In this paper, we detailed how memory reservation size impacts performance of our PRMS application up to 10x on AWS Lambda (RQ-1). We identified that stepwise scaling of client load results in minimal performance loss as infrastructure is gradually added (RQ-2). In settings where CPU power is coupled to memory size, the most economical configuration is likely not the platform minimum or maximum, and applications will likely require profiling to establish the best configurations (RQ-3). And finally, leveraging Keep-Alive workloads to retain hosting infrastructure can reduce freeze/thaw infrastructure latency improving performance while enabling ~18x cost savings versus hosting with dedicated VMs (RQ-4).

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. et al., 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] Oakes, E. et al., Pipsqueak: Lean Lambdas with large libraries. In Proc. of the 2017 IEEE 37th Int. Conf. on Distributed Computing Systems Workshops (ICDCSW 2017), Atlanta, GA, USA, June 2017, pp. 395-400.

[6] Baldini I. et al., The serverless trilemma: function composition for serverless computing. In Proc. of the 2017 ACM SIGPLAN Int. Symp. on New Ideas, New Paradigms, and Reflections on Programming and Software, Oct. 2017, pp. 89-10.

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

[8] Kumanov D., Hung L., Lloyd W., Yeung K., Serverless computing provides on-demand high performance computing for biomedical research. arXiv preprint, arXiv:1807.11659, July 2018.

[9] Spillner J., Mateos C., Monge D., Faaster, better, cheaper: The prospect of serverless scientific computing and HPC. In Proc. of the Latin American HPC Conference, Sept 2017, pp. 154-168. Springer, Cham.

[10] Adzic G., Chatley R., Serverless computing: economic and architectural impact. In Proc. of the 11th Mtg on Foundations of Software Engr Aug 2017, pp. 884-889.

[11] Pérez A., Moltó G., Caballer M., Calatrava A., Serverless computing for container-based architectures. Future Generation Computer Systems. 2018 June;83:50-9.

[12] Lloyd W., Ramesh S., Chinthalapati S., Ly L., Pallickara S., Serverless computing: An investigation of factors influencing microservice performance. In Proceedings of the 2018 IEEE International Conference on Cloud Engineering (IC2E), April 2018, pp. 159-169.

[13] Performance- How to keep the desired amount of AWS Lambda function containers warm, https:// stackoverflow.com/questions/ 51210445/how-to-keep-desired-amount-of-aws-lambda-function-containers-warm

[14] Pricing – Cloud Functions Documentation – Google Cloud, https://cloud.google.com/functions/pricing

[15] AWS Lambda Prod. features, https://aws.amazon.com/lambda/ features/

[16] Configuring Lambda Functions – AWS Lambda, https:// docs.aws.amazon.com/lambda/latest/dg/resource-model.html

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

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

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

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

[21] Eivy A., Weinman, J., Be Wary of the Economics of Serverless Cloud Computing. IEEE Cloud Computing. 2017 Mar;4(2):6-12.

[22] Eyk E., Iosup A., Abad C., Grohmann J., Eismann S., A SPEC RG cloud group's vision on the performance challenges of FaaS cloud architectures. 2018 ACM/SPEC Int. Conf. on Performance Engr. April 2018, pp. 21-24.

[23] Jonas E., Pu Q., Venkataraman S., Stoica I., Recht B., Occupy the cloud: Distributed computing for the 99%. In Proceedings of the ACM Symposium on Cloud Computing Sept 2017, pp. 445-451.

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

[25] Abad C., Boza E., Eyk E., Package-Aware Scheduling of FaaS Functions. In Proc. of the 2018 ACM/SPEC Int. Conf. on Performance Engineering, April 2018, pp. 101-106.

[26] Leavesley G., Markstrom S., Viger R., USGS Modular Modeling System (MMS) - Precipitation-Runoff Modeling System (PRMS). Watershed Models. 2005 Sep 28:159.

[27] David O., Ascough II J., Lloyd W., Green T., Rojas K., Leavesley G., Ahuja L., A software engineering perspective on environmental modeling framework design: The Object Modeling System. Environmental Modelling & Software. 2013 Jan 1;39:201-13.

[28] Amazon CloudWatch FAQs – Amazon Web Services (AWS), https:// aws.amazon.com/cloudwatch/faqs/

[29] Albuquerque L., Ferraz F., Oliveira R., Galdino S., Function-as-a-Service X Platform-as-a-Service: Towards a Comparative Study on FaaS and PaaS. In Proc. of the 12th Int. Conf. on Software Engineering Advances (ICSEA 2017), Oct. 2017, pp. 206-212.