The Serverless Application Analytics Framework:
Enabling Design Trade-off Evaluation for Serverless Software

Robert Cordingly, Hanfei Yu, Varik Hoang, Zohreh Sadeghi, David Foster, David Perez, Rashad Hatchett, Wes Lloyd
School of Engineering and Technology
University of Washington
Tacoma WA USA
rcording, hanfeiyu, varikmp, zsadeghi, davidf94, daperez, rhatch26, wlloyd@uw.edu

ABSTRACT

To help better understand factors that impact performance on Function-as-a-Service (FaaS) platforms we have developed the Serverless Application Analytics Framework (SAAF). SAAF provides a reusable framework supporting multiple programming languages that developers can integrate into a function's package for deployment to multiple commercial and open source FaaS platforms. SAAF improves the observability of FaaS function deployments by collecting forty-eight distinct metrics to enable developers to profile CPU and memory utilization, monitor infrastructure state, and observe platform scalability. In this paper, we describe SAAF in detail and introduce supporting tools highlighting important features and how to use them. Our client application, FaaS Runner, provides a tool to orchestrate workloads and automate the process of conducting experiments across FaaS platforms. We provide a case study demonstrating the integration of SAAF into an existing open source image processing pipeline built for AWS Lambda. Using FaaS Runner, we automate experiments and acquire metrics from SAAF to profile each function of the pipeline to evaluate performance implications. Finally, we summarize contributions using our tools to evaluate implications of different programming languages for serverless data processing, and to build performance models to predict runtime for serverless workloads.

CCS CONCEPTS

• **Computer systems organization** → Cloud computing;

KEYWORDS

Serverless Computing, Frameworks, Function-as-a-Service, Performance Evaluation, Programming Languages

ACM Reference format:

Robert Cordingly, Hanfei Yu, Varik Hoang, Zohreh Sadeghi, David Foster, David Perez, Rashad Hatchett, Wes Lloyd. 2020. The Serverless Application Analytics Framework: Enabling Design Trade-off Evaluation of Serverless Software Designs In *Proceedings of 6th International Workshop on Serverless Computing (WoSC6) 2020. ACM, TU Delft, The Netherlands, 6 pages.*

1 Introduction

In recent years Function-as-a-service (FaaS) platforms have arisen offering many desirable features for applications deployed to the cloud. FaaS platforms offer high availability, fault tolerance, automatic scaling, while billing developers only for the runtime of functions. As runtime is the primary factor driving hosting costs, it is important to profile and optimize serverless application performance.

Commercial FaaS platforms exhibit additional challenges when profiling applications. For example, limited deployment package size, no access root to the operating system, and the absence of a package manager makes installing and using existing profiling tools more difficult. Observability of infrastructure is another challenge given the serverless nature of FaaS platforms. Hardware details and performance metrics are abstracted or entirely hidden from the user. Finally, every FaaS platform is different. Each platform supports different languages, supported by different backend implementations, and some even use proprietary operating systems.

To aid in understanding performance implications of FaaS platforms, we developed the Serverless Application Analytics Framework (SAAF) [1]. SAAF is deployed in the package of a function and is invoked in the function by adding a few lines of code. SAAF supports functions written in Java, Python, Go, Node.js, and Bash on AWS Lambda, Google Cloud Functions, IBM Cloud Functions, Azure Functions, and OpenFaaS [2][3][4][5][6].

SAAF collects metrics from the Linux operating system and FaaS environment from function instances. These metrics can be used to determine FaaS specific information such as infrastructure state (cold vs warm), the number of function instances sharing the same host [4][7], and resource utilization. This enables accurate performance and cost characterizations of FaaS application deployments.

Alongside SAAF, we developed FaaS Runner. FaaS Runner is a client-side application that automates complex experiments on FaaS platforms. FaaS Runner supports profiling functions that incorporate SAAF, to provide further insight into a FaaS application. The SAAF project

includes many tools to aid in application development and deployment. Each language includes scripts that automatically deploy functions to all supported platforms. Section 3 details how to use our tools, how they are implemented, and the metrics they collect. Section 4 explores research and case studies where we have used SAAF and FaaS Runner.

2 Related Work

In this section we will review and discuss related work centered around FaaS performance analysis and other FaaS frameworks compared with SAAF.

2.1 FaaS Performance Analysis

J. Kuhlenkamp and S. Werner expressed the need for accurate FaaS performance profiling and benchmarking tools [8]. While using existing benchmarks can be useful to profile performance, FaaS platforms provide unique challenges and problems that must be addressed. Many Function-as-a-Service platforms demonstrate unique challenges such as the existence of heterogenous CPUs [9], the freeze-thaw lifecycle [10][11], latency variation between runtime languages [12], and performance scaling [7][11]. These features lead to one of the most fundamental issues with FaaS platforms: applications have unpredictable hosting costs. Several efforts have provided methods of modeling FaaS performance with the intent to better understand the cost of an application [13][14][15].

2.2 FaaS Frameworks and Tools

Kuhlenkamp and Klems offer a cost-tracing framework known as Costradamus to improve observability and traceability of function hosting costs by automatically aggregating log files [16]. Their framework, however, was not focused on performance analysis. Roland et al. leveraged proxy functions between the client and target FaaS function to support performance profiling [17]. Proxy functions are deployed to the same FaaS platform as the target function. The client calls the proxy function which then calls the target to collect metrics. This man-in-the-middle approach, referred to as Proxy Cloud Functions (PCFs) has benefits and drawbacks compared to SAAF. For example, PCFs do not need to modify the deployment package of functions being benchmarked. A significant drawback of PCFs is that they result in double billing as the proxy functions must wait for a response during synchronous function invocations. The scope of metrics that can be collected is also limited. PCFs can only observe metrics such as total execution time, latency, and throughput metrics. FaaS Runner calculates these metrics.

Another FaaS tool proposed by Shahrad et al. known as FaaSProfiler provides an alternate approach to FaaS profiling [18]. Instead of leveraging proxy functions or implementing profiling inside the function, FaaSProfiler provides an external tool that directly communicates with the FaaS platform itself by leveraging platform specific support. FaaSProfiler integrates with the open source OpenWhisk FaaS platform [19]. Kuntsevich et al. also designed a similar distributed analysis and benchmarking framework that integrates with Apache OpenWhisk [20].

These approaches provide observability to monitor server-side aspects of FaaS platforms from the perspective of a cloud provider to offer visibility that neither PCFs nor SAAF provide. Direct access methods provide deep insight into function and platform behavior which are obfuscated as a result of platforms being ‘serverless’. A notable drawback is tight coupling to the OpenWhisk FaaS platform. Commercial FaaS platforms such as AWS Lambda, do not provide API’s to directly monitor the hardware running functions and are not supported. This drawback limits the FaaSProfiler from being used on publicly hosted FaaS platforms.

3 Framework Design

The Serverless Application Analytics Framework (SAAF) is designed to be included inside the deployment package of FaaS functions [1]. Unlike frameworks that leverage proxy functions or that are deployed directly on the host hardware of a FaaS platform, SAAF is included in source code of the function to instrument data collection from the perspective of the function. This design allows SAAF to profile performance of software deployments to any commercial FaaS platform while enabling introspection of the infrastructure used by each platform.

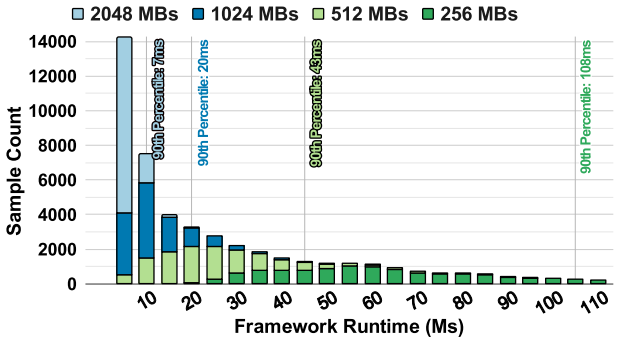

Figure 1: SAAF profiling overhead (ms) at different memory settings on AWS Lambda

Supporting SAAF, we have developed the FaaS Runner, a client-side application used in conjunction with SAAF to automate profiling experiments. FaaS Runner compiles experimental results into reports to aggregate data for quick analysis. FaaS Runner combines performance, resource utilization, and configuration metrics from many

concurrent sessions enabling observations not possible when profiling individual FaaS functions calls.

SAAF is built specifically to profile software deployments to FaaS platforms and to help evaluate implications of serverless software designs to ultimately improve function performance and cost. A key design consideration of SAAF is to minimize profiling overhead as a component of the application hosting cost. To quantify SAAF profiling overhead, we collected all metrics on AWS Lambda from an empty "hello world" function containing only the SAAF library. For 90% of function executions at 256 MBs, SAAF's runtime overhead was less than 108 milliseconds where 100 milliseconds is the smallest billable time interval on AWS Lambda. Figure 1 depicts profiling overhead for each memory setting.

3.1 Supported Platforms Languages

SAAF provides support to profile functions created with Python, Node.js, Java, Go, and AWS Lambda custom runtimes using Bash. Each version is written natively in their respective language to offer the best performance, minimize dependencies, and to make using SAAF as easy as possible. Programmers include the SAAF library and a few lines of code to enable profiling. Complete documentation and example functions in each language are available on the SAAF Repository [1]. Table 1 describes which languages are supported on each platform by SAAF.

Platform	Python	Node.js	Java	Go	Bash
AWS Lambda	✓	✓	✓	✓	✓
Google Cloud Functions	✓	✓	✗	✗	✗
IBM Cloud Functions	✓	✓	✓	✗	✗
Azure Functions	✓	✓	✗	✗	✗
OpenFaaS	✓	✗	✗	✗	✗

Table 1: Currently Supported Platforms and Languages

SAAF includes scripts to help streamline the process of deploying functions to each platform. The included example Python project can be built and deployed automatically to all supported platforms without requiring any code changes. This structure provides the ability to create multi-platform functions within a single code base.

3.2 Collecting Analytics with SAAF

SAAF collects metrics from the Linux **/proc** filesystem and appends them onto the JSON payload returned by the function instance. Attributes collected include Linux Time Accounting metrics such as CPU idle, user, kernel, and I/O wait time, wall-clock runtime, and memory usage [21]. As SAAF is dependent on Linux, SAAF does not support profiling functions deployed to Azure Functions using Windows. To identify infrastructure state, SAAF stamps function instances with a unique ID and uses the existence of the ID to identify if the environment is new (cold) or recycled (warm) [22].

To control profiling verbosity, and to optimize performance, programmers can specify which attributes SAAF collects. After including the SAAF package and initializing the Inspector object, the attributes collected are defined by which functions the programmer calls. CPU, memory, function instance, Linux and platform profiling functions offer granular and customizable profiling. Profiling functions and important metrics are defined in Table 2.

Leveraging SAAF to profile a function requires adding a couple lines of code. Advanced profiling activities may require additional coding. Profiling is enabled within FaaS functions through modifications in five sections:

1. **Initialization:** Initialize the SAAF Inspector object at the start of the FaaS function.
2. **Inspection:** Call initial SAAF inspect functions such as inspectAll(), inspectCPU(), etc. to collect base values for metrics.
3. **Workload:** Implement function, this is where the implementation of the function should be.
4. **Inspect Deltas:** After function code is complete, call SAAF inspect delta functions, e.g. inspectAllDeltas(), inspectCPUDeltas(), to calculate resource utilization.
5. **Finalize:** Obtain SAAF output by calling the finish() function. Return this object or append to an existing return object. If the function is asynchronous save this object to external data storage for future retrieval.

3.3 Running Experiments with FaaS Runner

FaaS Runner provides a client-side application that is used to define and execute experiments on FaaS Platforms that works in conjunction with SAAF. Depending on the number of concurrent client requests desired for the experiment, FaaS Runner can either execute using a local computer, or leverage a cloud-based virtual machine with many virtual CPU cores.

FaaS experiments are not only for experimental research, but they also help to provide analytics to compare and contrast alternate serverless software designs. Understanding FaaS performance allows developers to make educated design decisions (e.g. optimal function composition, memory setting, language selection). Without proper insight, developers are left to make ad hoc design decisions that will directly impact the performance and cost of a serverless application.

FaaS Runner has the ability to run multiple types of experiments. From basic single function executions, to complex multi-function pipelines, many options are provided to customize the execution and orchestration of

experiments. FaaS Runner can execute functions sequentially, in parallel, synchronously, or asynchronously across all of SAAF's supported platforms and through HTTP requests. Experiments are defined either using command line arguments, or through JSON configuration files. Additional details on FaaS Runner configuration options are described in section 3.5. The results of each function invocation are persisted and when all functions are complete the ReportGenerator is automatically invoked to compile and analyze results.

FaaS Runner's ReportGenerator organizes results to provide user friendly reports in CSV format. The ReportGenerator calculates metrics to aggregate data spanning multiple output files. For example, latency can be calculated by measuring the round-trip time of function calls from the client's perspective, and by then subtracting runtime reported by SAAF. The average latency is then calculated by aggregating results over a batch of function calls. Function tenancy, which is the number of functions hosted by the same cloud-based virtual infrastructure, can be determined by comparing the number of function calls sharing the same vmID attribute where chronological time of execution overlaps. The ReportGenerator can be used to regenerate reports over archived data, or to compile reports over data generated by other experiment clients besides the FaaS Runner. Table 2 provides a list of key metrics provided by the FaaS Runner and ReportGenerator. In total 48 distinct metrics are collected by SAAF and FaaS Runner.

Attribute	Function	Description
newcontainer	inspectContainer	Whether container is new or if it has been reused
vmuptime	inspectContainer	Time of host boot in seconds since Jan 1, 1970
cpuType	inspectCPU	Model name of the CPU
cpuUsrΔ	inspectCPU	Time spent executing in user mode
cpuKrnΔ	inspectCPU	Time spent executing processes in kernel mode
cpuIdleΔ	inspectCPU	Time spent idle
cpuIowaitΔ	inspectCPU	Time spent waiting for I/O to complete
cpuIrqΔ	inspectCPU	Time spent servicing interrupts
cpuSoftIrqΔ	inspectCPU	Time spent servicing software interrupts
vmcpustealΔ	inspectCPU	Cycles waiting for hypervisor serving other vCPU
totalMemory	inspectMemory	Total kBs memory allocated to the instance
pageFaultsΔ	inspectMemory	Total page faults of the instance since boot
containerID	inspectPlatform	Platform specific function instance identifier
vmID	inspectPlatform	Platform specific virtual machine identifier
functionMemory	inspectPlatform	Configured memory setting on the FaaS platform
runtime	finish	Runtime of the function from start to finish
saafRuntimeΔ	inspectAll	Time to calculate all initial metrics of SAAF
userRuntime	inspectAllΔ	Time in ms between initial inspection and deltas
X_avg/sum/list	FaaS Runner	Average/sum/list any attribute
roundTripTime	FaaS Runner	Time between request and response.
latency	FaaS Runner	Total runtime subtracted from the roundTripTime
runtimeOverlap	FaaS Runner	Number of concurrent function instances
tenants	FaaS Runner	Number of tenants a function host may have

Table 2: Key attributes collected by SAAF or FaaS Runner. Δ indicates initial and delta versions are provided.

3.5 FaaS Runner: Configuration and Usage

FaaS Runner provides multiple options to configure function execution and report generation for FaaS experiments. Configuration options can be specified in files or through command line arguments. FaaS Runner leverages two types of configuration files: function files and experiment files. Function files provide required information to access FaaS function endpoints, and experiment files define the operations and inputs to an experiment.

After an experiment is defined, FaaS Runner automates key tasks such as dynamically adjusting platform memory settings and supplying functions with different payloads. Experiments execute autonomously to completion with no required user interaction. FaaS Runner experiment definitions are portable as different client computers can be used to perform the experiments by reusing experiment configuration files. Experiments are launched by running the FaaS Runner application through the command line.

4 SAAF Case Studies

In this section we detail three serverless case studies enabled by SAAF: (1) an image processing pipeline, (2) a data processing pipeline with implementations in four different programming languages to contrast performance implications [23], and (3) random compute-bound workloads used to develop and refine serverless performance models using Linux time accounting principles [13].

4.1 Image Processing Pipeline

To demonstrate the efficacy of SAAF and FaaS Runner, we deployed an existing image processing pipeline available from the AWS Serverless Application Repository which uses Node.js [24]. The pipeline consists of five serverless functions to perform resize, rotate, blur, sepia filtering, and image compression. These functions were deployed on AWS Lambda using the maximum memory size of 3GBs. We added SAAF to each function and created an experiment to execute the entire pipeline with FaaS Runner. Integrating SAAF into these functions required no code changes or knowledge regarding the image processing algorithms as each function's source code was simply augmented by adding SAAF method invocations at the start and end of each function. Once configured, we used FaaS Runner to orchestrate an experiment to process 100 identical images concurrently. We leveraged SAAF to obtain CPU metrics to observe the CPU profile of each function as shown in Figure 2. For this image processing pipeline, functions had similar CPU profiles, where ~46-52% of the total time the CPU was

idle, ~42-48% of the time the CPU executed code in user mode, and ~5-7% of the time the CPU executed kernel mode instructions. The runtime varied between functions: blur was the slowest at 53 seconds, and resize was the fastest at 15 seconds as shown in Figure 2.

Without a deep understanding of each function it is difficult to infer why a function may perform poorly on a FaaS Platform. Using metrics from SAAF, we can gain insight into a function's performance. For example, the Resize, Rotate, Sepia, and Compress functions all had 26,000 to 28,000 page faults per second compared to Blur which had greater than 40,000 page faults per second.

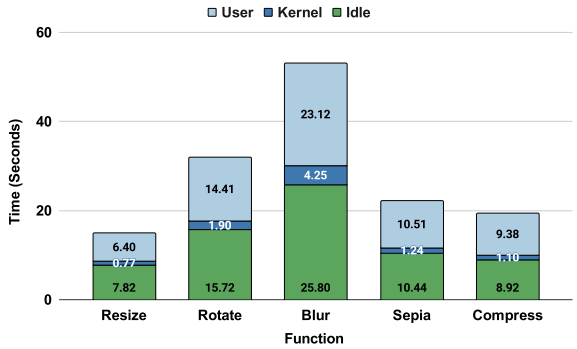

Figure 2: Workload profiling with FaaS Runner and SAAF on a five-function image processing pipeline.

Alongside memory performance, FaaS platforms have the potential of exhibiting resource contention as multiple functions execute concurrently. Using FaaS Runner's runtimeOverlap metric, we observed that the average number of concurrent function instances varied for each function. Resize, Sepia, and Compress had on average 80.4, 80.1, and 69.3 concurrent instances over their runtime duration. Since we processed 100 images, the maximum possible function instances would have been 100. Rotate and Blur exhibited more concurrency where an average of 91.2 (Rotate) and 89.8 (Blur) functions executed concurrently. This observation shows that these functions ran closer together due to either FaaS platform scheduling or chain-of-execution timing when compared to the Resize, Sepia, and Compress functions. This example highlights obersevations made by combining SAAF and FaaS Runner.

4.2 Programming Language Comparison

In our paper: "*Implications of Programming Language Selection for Serverless Data Processing Pipelines"* [23], we developed four identical Transform-Load-Query pipelines in Java, Python, Go, and Node.js and compared the performance of each language on AWS Lambda. Using SAAF we profiled each language using Linux Time Accounting metrics.

By leveraging SAAF, combined with FaaS Runner, we are able to create experiments to investigate the serverless freeze thaw lifecycle [11]. SAAF is able to characterize infrastructure state allowing us to observe the performance impact of running on cold versus warm infrastructure.

Further, we used FaaS Runner to conduct experiments to investigate increasing the number of concurrent function invocations, and also to investigate function performance across a variety of memory reservation sizes. These three experiments allowed us to evaluate the performance implications of data processing pipeline implementations in each language over a variety of configuration scenarios.

FaaS Runner allowed us to easily perform new experiments as new versions of our data processing pipeline in different languages were introduced by simply changing function configuration files. As some experiments were resource intensive, we deployed and executed FaaS Runner using both Amazon EC2 virtual machines and local client computers. For experiments that test latency, it is crucial to use a virtual machine in the same subnet as a FaaS function to minimize network overhead.

We found that no single language performed the best in all of our experiments and that a hybrid pipeline combining functions written in both Go and Java offered the best performance. Node.js had the slowest performance, resulting in an application costing 94% more than the hybrid version. Go exhibited the least cold-start latency of any language. For scalability Go, Python, and Java performed similarly (8-19% increase in runtime between 1 and 50 requests), while Node.js was impacted more heavily (35% increase). Finally, all languages scaled performance similarly as we increased the reserved memory up to 1536 MBs. Due to the single-threaded nature of our pipeline, memory allocations greater than 1536 MB offered no performance improvements while incurring increased costs.

4.3 Serverless Performance Modeling

In our paper: "*Predicting Performance and Cost of Serverless Computing Functions with SAAF*" [13], we evaluated regression modeling combined with Linux time accounting principles to predict runtime of compute-bound FaaS functions. In particular, this paper focused on identifying factors that contribute to FaaS performance variance to enable building accurate performance models.

In Fall 2019 our experiments identified that AWS Lambda and IBM Cloud Functions used multiple different CPU types to implement FaaS function instances, a phenomenon known as CPU heterogeneity. Using SAAF, we categorized CPU types and determined function tenancy. We found that each CPU type offered varying

performance, and function tenancy had a large impact on performance on IBM Cloud Functions.

We evaluated our Linux time accounting approach to performance modeling for runtime prediction of FaaS function deployments. In our approach we build regression models for key CPU timing metrics (e.g. CPU user mode time, CPU idle time) and then apply Linux time accounting to derive runtime predictions. We evaluated our approach for FaaS function deployments to alternate CPU types, with different memory sizes, and to different public FaaS platforms. We performed experiments using increasingly complex function workloads where each subsequent workload introduced additional random behavior and performance variance. To collect sufficient data for each function configuration, we performed over 65,000 function invocations on AWS Lambda and IBM Cloud Functions. We found that model error correlated roughly with performance variance when modeling functions with increasingly variable performance outcomes. By closely observing function tenancy, we found a significant difference between how AWS and IBM execute functions with respect to memory management.

While pricing models between AWS Lambda and IBM Cloud Functions appear similar on the surface, we found that differences in platform implementation produced significant price differences based on the number of concurrent function calls. By varying memory settings, we saw performance scale on AWS Lambda while runtime remained constant on IBM for sequential function invocations. IBM appears to not restrict the CPU share of individual function instances resulting in competition for available resources of the host. The available memory of servers on IBM appears to limit the maximum number of co-located function tenants. In contrast, AWS restricts the CPU share for each function instance so that performance remains fairly constant regardless of the number of co-resident function executions occurring on the host. This observation was made possible by SAAF, and provides a significant example of FaaS price obfuscation. The same workload on IBM can cost anywhere from $8.89 to $113.97 depending on the tenancy of function executions across host VMs for concurrent client requests.

5 Conclusions

SAAF is a serverless computing framework that provides insight into the performance and infrastructure of microservices deployed to a variety of FaaS platforms in multiple languages. SAAF is easily integrated into new and existing functions deployed to many commercial FaaS platforms. When used with FaaS Runner, SAAF provides an invaluable tool for scientists and practitioners to automate execution of experiments and aggregate results to help evaluate performance tradeoffs of microservice composition and alternate serverless software architectures.

ACKNOWLEDGMENTS

Supported by the NSF Advanced Cyberinfrastructure Research Program (OAC-1849970), NIH grant R01GM126019, and AWS Cloud Credits for Research.

REFERENCES

[1] “SAAF: Serverless Application Analytics Framework,” 2020. https://github.com/wlloyduw/SAAF.
[2] “AWS Lambda,” 2014. https://aws.amazon.com.
[3] “Google Cloud Functions.” https://cloud.google.com/functions/.
[4] “IBM Cloud Functions.” https://cloud.ibm.com/functions/.
[5] “Azure Functions - Develop Faster with Serverless Compute.” https://azure.microsoft.com/en-us/services/functions/.
[6] “OpenFaaS - Serverless Functions, Made Simple.,” 2016. https://www.openfaas.com.
[7] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, “Peeking Behind the Curtains of Serverless Platforms,” *2018 USENIX Annu. Tech. Conf. (USENIX ATC 18)*, 2018.
[8] J. Kuhlenkamp and S. Werner, “Benchmarking FaaS platforms: Call for community participation,” 2019, doi: 10.1109/UCC-Companion.2018.00055.
[9] E. Jonas *et al.*, “Cloud programming simplified: a berkeley view on serverless computing,” *arXiv Prepr. arXiv1902.03383*, 2019.
[10] A. Pérez, G. Moltó, M. Caballer, and A. Calatrava, “Serverless computing for container-based architectures,” *Futur. Gener. Comput. Syst.*, 2018, doi: 10.1016/j.future.2018.01.022.
[11] W. Lloyd, M. Vu, B. Zhang, O. David, and G. Leavesley, “Improving application migration to serverless computing platforms: Latency mitigation with keep-Alive workloads,” 2019, doi: 10.1109/UCC-Companion.2018.00056.
[12] D. Jackson and G. Clynch, “An investigation of the impact of language runtime on the performance and cost of serverless functions,” 2019, doi: 10.1109/UCC-Companion.2018.00050.
[13] R. Cordingly, W. Shu, and W. J. Lloyd, “Predicting Performance and Cost of Serverless Computing Functions with SAAF,” 2020.
[14] P. Leitner, J. Cito, and E. Stöckli, “Modelling and managing deployment costs of microservice-based cloud applications,” 2016, doi: 10.1145/2996890.2996901.
[15] T. Elgamal, A. Sandur, K. Nahrstedt, and G. Agha, “Optimizing cost of serverless computing through function fusion and placement,” 2018, doi: 10.1109/SEC.2018.00029.
[16] J. Kuhlenkamp and M. Klems, “Costradamus: A cost-tracing system for cloud-based software services,” in *International Conference on Service-Oriented Computing*, 2017, pp. 657–672.
[17] R. Pellegrini, I. Ivkic, and M. Tauber, “Function-as-a-Service Benchmarking Framework,” May 2019, doi: 10.5220/0007757304790487.
[18] M. Shahrad, J. Balkind, and D. Wentzlaff, “Architectural implications of function-as-a-service computing,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 1063–1075.
[19] “Openwhisk.” https://console.bluemix.net/docs/openwhisk/.
[20] A. Kuntsevich, P. Nasirifard, and H.-A. Jacobsen, “A Distributed Analysis and Benchmarking Framework for Apache OpenWhisk Serverless Platform,” in *Proceedings of the 19th International Middleware Conference (Posters)*, 2018, pp. 3–4.
[21] W. J. Lloyd *et al.*, “Demystifying the Clouds: Harnessing Resource Utilization Models for Cost Effective Infrastructure Alternatives,” *IEEE Trans. Cloud Comput.*, 2015, doi: 10.1109/tcc.2015.2430339.
[22] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, “Serverless computing: An investigation of factors influencing microservice performance,” 2018, doi: 10.1109/IC2E.2018.00039.
[23] R. Cordingly *et al.*, “Implications of Programming Language Selection for Serverless Data Processing Pipelines,” 2020.
[24] E. Chiu, “Serverless Galleria,” 2017. https://github.com/evanchiu/serverless-galleria.