Performance of Java in Function-as-a-Service Computing

A. Dowd
Department of Electrical and Computer Engineering
University of Texas at Austin
Austin, USA
dowda@utexas.edu

Qinzhe Wu, Lizy K. John
Department of Electrical and Computer Engineering
University of Texas at Austin
Austin, USA
qw2699@utexas.edu,ljohn@ece.utexas.edu
0000-0002-7988-1431,0000-0002-8747-5214

***Abstract*—One of the newest forms of serverless computing is Function-as-a-Service (FaaS). FaaS provides a framework to execute modular pieces of code in response to events (e.g., clicking a link in a web application). The FaaS platform takes care of provisioning and managing servers, allowing the developers to focus on their business logic. Additionally, all resource management is event-driven, and developers are only charged for the execution time of their functions. Despite so many apparent benefits, there are some concerns regarding the performance of FaaS. Past work has shown that cold starts typically have a negative effect on response latency (e.g., the initialization could add more than $10\times$ execution time to short Python FaaS functions). However, the magnitude of the slowdown is subject to varying from language to language. This paper investigates how containerization and cold starts impact the performance of Java FaaS functions, and compares with the findings from the prior Python study.**

We find that containerization overhead slows Java FaaS functions from native execution by $4.42\times$ on average (geometrical mean), ranging from $1.69\times$ up to $15.43\times$. Comparing with Python in warm containers, Java has more overhead on three of the functions, but faster on the other functions (up to $27.08\times$ faster). The container initialization time for Java is consistently less than half that of Python. However, Java has the additional overhead due to Java Virtual Machine (JVM) warmup which contributes varying amount of latency to the execution depending on the Java function properties. Overall, Java has about $2.60\times$ ($2.65\times$) speedup across seven FaaS functions over Python in cold (warm) start scenarios, respectively.

***Index Terms*—Serverless Computing, Function-as-a-Service, JVM, OpenWhisk**

I. Introduction

Over the past several years, interest in serverless computing has rapidly increased due to its flexibility and low cost. Despite the name, serverless computing does not actually fully remove the servers, it instead adds a layer of abstraction to isolate the servers from the developers to some extent. Eyk et al. [1] defines serverless computing as “a form of cloud computing which allows users to run event-driven and granularly billed applications, without having to address the operational logic”. In this case, the user of the serverless platform might be the developer of a web application, and the operational logic could refer to bringing up the server, spinning up virtual machines, managing memory, and so on. In recent years, there have been many services such as Platform-as-a-Service (PaaS [2]–[4]), Function-as-a-Service (FaaS [5]–[8]), and Software-as-a-Service (SaaS [9], [10]), and they all partially fit the aforementioned definition. The primary factor that differentiates the three is the level of developer control. PaaS allows a developer to rent hardware resources and is billed by the hour. In this case, the developer has full control over the allocated resources and is responsible for managing them effectively. If servers sit idle due to a workload decrease, the developer will still be charged. On the other end of the spectrum is SaaS. SaaS is event-driven and granularly billed but does not accept customized code. Instead, the developer can run service code selected from the preset. FaaS sits somewhere between PaaS and SaaS (as shown in Figure 1). The infrastructure is shared, but the application code is customizable. The developer is able to register modular pieces of code with the FaaS provider and set up triggers which will execute the code in response to an event, such as a user clicking a button in a web application. The term serverless is most commonly used to refer to FaaS [11], which is the model we will focus on in this paper.

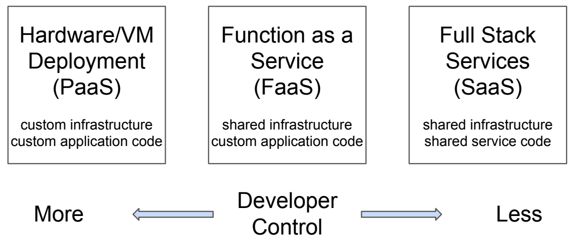

Hardware/VM Deployment (PaaS)
custom infrastructure custom application code
Function as a Service (FaaS)
shared infrastructure custom application code
Full Stack Services (SaaS)
shared infrastructure shared service code
More
Developer Control
Less

Fig. 1: Level of developer control in serverless computing [11].

FaaS has a couple of advantages over the traditional application development model. Resources are allocated by the service provider as needed, and developers are only charged for the execution time of their functions making it a highly scalable and cost-effective solution for enterprise application development [12]. This model is especially attractive due to the recent shift of application architectures to containers and microservices [11]. While Villamizar et al. [13] have shown that there are clear financial benefits to using FaaS, it should be noted that there are several limitations of FaaS. For example, functions must be stateless and short. Most providers limit the execution time to 10-15 minutes [4], [5], [14].

Few studies [15]–[18] have been conducted, attempting to understand the primary factors affecting FaaS performance. The earlier research [15], [16] focus more on the platform level. For instance, Manner et al. [16] and Jackson et al. [15] reveals that different FaaS providers optimize their platforms/runtimes for different programming languages. In contrast, Shahrad et al. [17] deployed an open-source FaaS framework, Apache OpenWhisk [14], to a self-owned machine and performed server-level profiling. Among a few hardware implications that Shahrad et al. found, the containerization slowdown of Python FaaS functions could go up to 20×, and the cold start could add initialization time 10× more than the actual function execution time. Those findings are based on the experiments with FaaS functions written in Python (and some NodeJS), so naturally it raises the questions about Java, another very popular language in cloud computing. As pointed by the prior study (more details in Section II), programming language might change the FaaS performance significantly. Here are the primary questions the paper wants to answer:

1) What is the containerization overhead for Java FaaS functions?
2) What is the impact of a container cold start for functions written in Java, and whether Java Virtual Machine (JVM) plays a role in this scenario?
3) How does Java performance on OpenWhisk compare with Python?

In order to find the answers for those questions, we design a set of experiments using the same facilities from Shahrad et al. [17], further investigating the overhead due to containerization and cold starts for functions written in Java. Specifically, we measure and break down the response latency of various Java functions into container initialization plus function execution time, expecting to see a significant slowdown over the native execution, especially for cold start scenarios; we repeat the same experiments for Python implementation for comparison, anticipating a better performance from Java due to the pre-compiled bytecode; we scale the Java FaaS function data size in both cold and warm start scenarios, looking for a constant overhead from JVM warmup [19].

This paper makes the following contributions:

1) We measure the response latency of Java functions running on an open-source FaaS framework. For those measured Java FaaS functions, we determine the containerization overhead (over native execution) causes 4.42× and 12.44× slowdown on average for cold and warm starts, respectively.
2) From the analysis and comparison with Python implementation of the same FaaS functions, we show Java has shorter (about 1/3) container initialization time than Python and the overall performance is better on most functions (on average 2.60× faster).
3) We also reveal cold starts would incur an additional overhead, JVM warmup, for Java FaaS functions.

For the rest of the paper, we will give a brief introduction on the related work (§ II), and describe our experimental methodology in details (§ III), and present the results collected from each test (§ IV), then discuss the future work (§ V) and conclude the implications from the results (§ VI).

II. Related Work

As shown in several studies, the primary factors which have affect FaaS performance are cold start overhead, programming language and FaaS provider, and containerization overhead.

FaaS Cold Start: For security reasons, a new container must be started for each new function run on a particular machine. Containers may be reused, but only for exactly the same function registered by the same developer. Idle containers are shut down after a short grace period for better resource utilization. This means that if functions are triggered infrequently, the containers will have to be unpaused or restarted every time the function is run. Past work [15], [16], [18] has shown that cold starts typically have a negative effect on response latency, but the magnitude of the slowdown varies depending on provider and language and container image versions [20]–[22]. All providers and languages exhibit a cold start latency of at least 300ms, but in some cases the latency can be as long as 24s [16]. This could be up to 10× the execution time of extremely short functions [17].

Programming Language: Several studies [15], [16] has shown the effect that programming languages have on FaaS functions vary widely. Manner et al. [16] tested functions written in Java and JavaScript on AWS Lambda and Microsoft Azure. They found that in both cases Java incurred a larger cold start overhead, but performed much better than JavaScript in a warm container. They suggested that the larger cold start overhead was due to the virtual machine warmup time required by Java. On the other side, they also noticed that Java outperforming JavaScript in a warm container, as that Javascript is an interpreted language while Java uses pre-compiled bytecode. Principally, bytecode could save runtime since some of the work of translating high level code, is done prior to execution.

Platform: It is also clear that different platforms provide runtimes that are tuned for different languages. Jackson et al. [15] also tested the effect of language runtime on AWS Lambda and Microsoft Azure. They found that on AWS Lambda Python outperformed all other languages, including Java, on warm starts. Python is usually regarded as an interpreted language (i.e., high-level language is not translated into machine readable instructions until the line is executed), so the result is an exception to the principle mentioned above. They also found that C# .NET performed best on Microsoft Azure, but performed badly, particularly in cold start scenarios on AWS. NodeJS exhibited the exact opposite behavior, performing well on AWS and poorly on Azure. These differences are not surprising as providers are motivated to improve the performance of the most popular languages for their platforms. However, it is extremely important for developers to understand how their choice of language and platform may impact function performance.

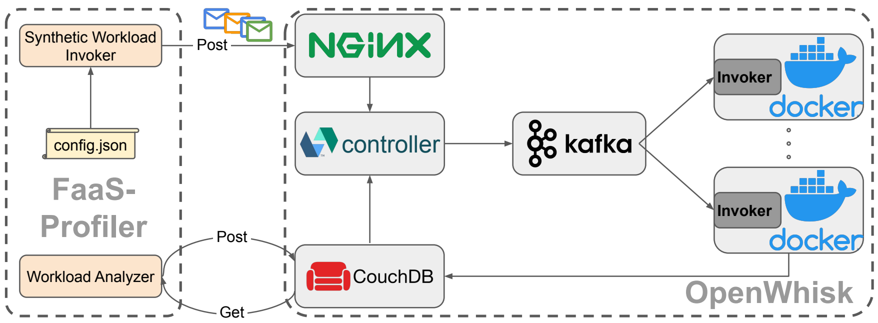


Fig. 2: OpenWhisk-FaaSProfiler architecture [17]

Containerization Overhead: On commercial FaaS platforms, it is difficult to figure out the total slowdown caused by FaaS when compared to native execution. Shahrad et al. [17] developed a methodology that uses open-source FaaS framework, OpenWhisk [14], and a tightly-integrated profiling tool, FaaSProfiler, to study the containerization overhead (more details of OpenWhisk and FaaSProfiler in Section III-A). Shahrad et al. found that for functions written in Python, there is a significant slowdown (up to 12×), and predicted functions in other programming languages should suffer the containerization overhead at the same order of magnitude.

III. Methodology

A. OpenWhisk and FaaSProfiler

Figure 2 illustrates the architecture of OpenWhisk [14] (the dashed box on the right) and FaaSProfiler [17] (the left dash-line box). Just like all other FaaS design, OpenWhisk [14] executes function code provided by a developer in an isolated environment, a docker container [23] for this case. Because different FaaS functions are registered as the actions for different events, a database, CouchDB [24], is used for storing those mapping information. Whenever the frontend, NGINX [25] for OpenWhisk, receives a request, the OpenWhisk controller queries CouchDB to find the corresponding FaaS function, then the pair of request and FaaS function is queued into Kafka [26], which manages the resources and schedule the actual execution of the FaaS function. Finally, a new docker container is started, if necessary, and the invoker runs the function. The result is returned to the database and the requester application. Most academical experiments to date have to reverse engineer the commercial FaaS system. This makes it difficult to take reliable measurements due to the lack of control of the entire system. The FaaSProfiler is developed to study the server-level behaviors of FaaS functions [17] with the open-source FaaS framework OpenWhisk. As shown in Figure 2, there are a few components in FaaSProfiler: a Synthetic Workload Invoker takes a configuration written in JSON in order to create the requests in a desired way (e.g., specific Query-Per-Second, request length etc.); and a Workload Analyzer module is hooked with the CouchDB in OpenWhisk system to retrieve the first-hand execution data (e.g., request queuing time, container initialization time, function execution time and so on).

Based on the OpenWhisk and FaaSProfiler publicly available on GitHub [27], [28], we setup OpenWhisk-FaaSProfiler facility as described above, and follow the instructions on the GitHub pages to adjust the configurations properly (listed in Table I). The blade server where we deploy this setup has fairly modern specifications (Table II). As the system has only a single worker node, the range of request arrival rate that the system is capable to handle is relatively narrow, so we adjust the configuration carefully to ensure the system is in a balanced state [17]. It allows us to focus on the containerization overhead and cold start effect, without worrying about other latency factors (e.g., Kafka queuing latency).

B. Benchmarks and Experiments

We use four microbenchmarks (*base64*, *http*, *json*, *primes*) from the FaaSProfiler repository [28], and five more benchmarks from the the Java Microbenchmark Harness (JMH) repository [29]. The four functions coming with FaaSProfiler are written in Python, so we implement the corresponding Java version by ourselves, and also write the corresponding Python version implementations for the five JMH benchmarks. Descriptions of each of the functions are given in Table III along with a data size definition. One of our experiments scales the data size as we will describe later.

We run each function on OpenWhisk in both cold and warm start scenarios, achieved by shutting down the services and run right after a warmup invocation of the same function, respectively. FaaSProfiler records the container initialization time and function execution time separately, so we can double check that the warm start runs have zero initialization time while the cold start runs have. In order to understand the containerization overhead, we also measure the same functions running natively using the Java Microbenchmark Harness (JMH) [29]. JMH ensures that the benchmarks are run in a warm environment and that no dead code is optimized out. We chose to collect the native execution data in a warm environment because this is more realistic to the traditional programming model, which constructs and executes one monolithic application. In contrast, FaaS never guarantee a warm environment as containers are spun up and shut down frequently.

TABLE I: Configurations of OpenWhisk-FaaSProfiler.

Parameter	Description	Value Set
	OpenWhisk	
invocationPerMinute	The maximum number of action invocations allowed per minute	60000
concurrentInvocations	The maximum number of invokers allowed to run concurrently	30000
firesPerMinute	The allowed trigger firings per minute	60000
sequenceMaxLength	The maximum length of a sequence action	50000
	FaaSProfiler	
test_name	Name of tests to be run (can choose anything)	
random_seed	Ensures run to run consistency	100
blocking_cli	True/false determines whether blocking CLI calls are used	false
test_duration_in_seconds	Total duration of test in seconds, measurement stop after this time	15-90
instances	Set of functions to run during measurement period	
application	Name of function (should be the exact name registered)	
distribution	Distribution of function invocations	Uniform
rate	Number of function invocations per second	1-30
activity_window	Range of time in seconds during which function invocations should occur	[5,-5]
perf_monitoring	Set of scripts to be run during or after test	
runtime_script	Monitoring script run during test	default
post_script	Optional post processing script	null

TABLE II: System Specifications

Processors	12× X86_64 cores @ 2.2 GHz, 2 hyperthreads/core
Cache	32KB private I-Cache, 32KB private D-Cache, 256KB private L2, 30 MB shared L3
Memory	56GB DDR4-2400
NIC	PCIe 2.1 5GT/s GbE NIC
kernel version	`Linux 4.4.0-138-generic`

TABLE III: Microbenchmarks

Name	Description	Data Size (n)
base64	Encodes and decodes a string	Length of the string
http	Performs API call to retrieve current time	Number of API invocations
json	Reads a JSON object from a file, averages values in common fields	Length of the JSON object
primes	Finds the number of primes between 1 and n	Upper bound on range of numbers for prime search
bigDec	Creates array of BigDecimals and compares all elements to element 0	Number of compare operations performed
bigInt	Creates array of BigIntegers and multiplies each element together	Number of elements in array
arrCpy	Copies an array of bytes to an empty array (deep copy)	Length of byte array
intMax	Rinds maximum value in an array of integers	Length of integer array
fileRW	Writes to tmp file and then reads data back	Number of bytes written to file

For FaaS function, it is clear one of the cold start overheads comes from the container initialization. Java FaaS function running in a cold start scenario have additional overhead added to the execution time. One of the possible sources for the extra overhead is Java Virtual Machine (JVM) warmup, so we also performed a data scaling experiment by varying the data size as defined in Table III. If the difference in the execution times (cold vs. warm start) is due to the JVM warming up, it should be consistent for the same function across data sizes [19].

To make comparison between Python and Java, we measure the container initialization time as well as the total response latency (warm and cold starts) for the Python version of all aforementioned FaaS functions (§ III-B) except *fileRW*, which cannot reach a balanced state on a single worker server.

IV. Results

A. Native vs. OpenWhisk Execution

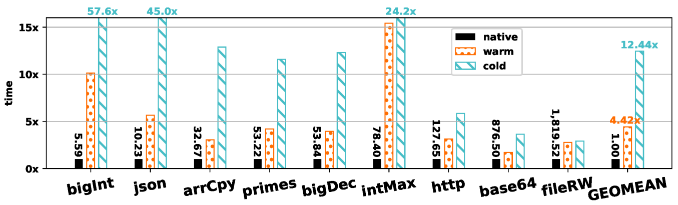
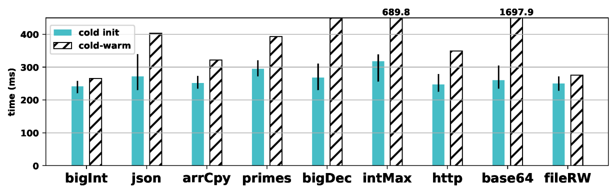


Fig. 3: OpenWhisk response latency normalized by native execution time for Java functions. Benchmarks are sorted by the native execution time (the numbers above the black solid bars are the absolute execution time in milliseconds).

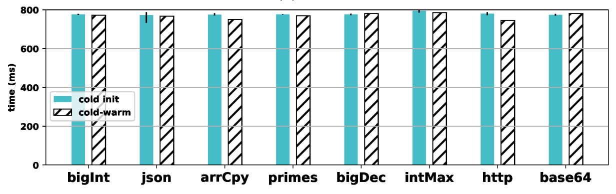
Function execution on OpenWhisk is consistently slower than native execution in both cold and warm start scenarios. Figure 3 shows the response latency of each function on OpenWhisk normalized by its latency in native environment. The end-to-end latency for functions in warm containers is $1.6\times$ to $15\times$ longer than the native execution. Most functions experience a $3\times$ to $24\times$ longer response latency (over native latency) during cold starts. However, extremely short functions such as *bigInt* and *json* suffer from much more slowdowns (up to $57.64\times$). This is partially due to that the overhead caused by container start up and initialization is relatively constant (as we will show in Section IV-B). Across all 9 benchmarks, the geometric average (GEOMEAN in Figure 3) of slowdown is $4.42\times$ for warm start, while cold start is $12.44\times$.

B. Cold Start

As shown in Figure 4a, container initialization time (the cyan solid bars) remains about the same (266 ms) across all Java FaaS functions. This is expected because the time to start a container should not depend on which function it will execute. Surprisingly, Figure 4a also indicates that container initialization time is not the only factor contributing to the

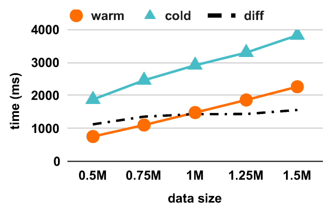


(a) Java

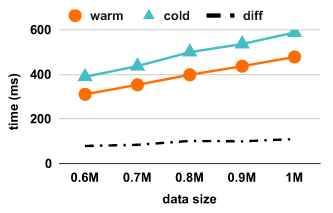


(b) Python

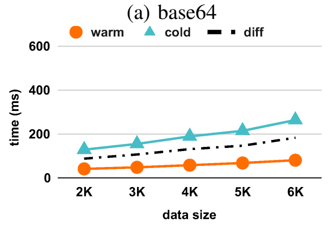
Fig. 4: Java function cold start container initialization time, which explains part of the overhead cold start has over warm start. Python function container initialization time is about the latency difference between cold start and warm start.



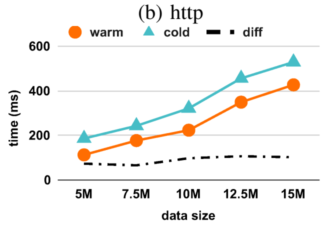
(a) base64



(b) http



(c) json



(d) primes

Fig. 5: Java function cold start and warm start execution times scale as the data size increases. The difference between cold start and warm start execution times remains relatively stable.

longer cold start response latency (compared with the warm start latency). The differences (the hatched bars) between the latency on cold and warm starts are more or less (but still statistically) greater than the container initialization time.

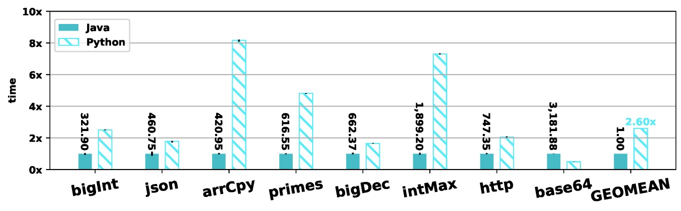
We can see in Figure 4b that Python containers has very stable initialization time (about 778 ms), nearly triple of Java container initialization time. Python containers take longer time than the Java ones to initialize, because OpenWhisk launches Java container and Python container based on different docker images, and the Python image has much more layers (preparing commands/actions) than the Java image. Unlike Java FaaS function, Python version has a much simpler answer for the cold start overhead. The difference of response latency between cold start and warm start for Python FaaS functions, perfectly match to the container initialization time, meaning the container initialization is the only cold start overhead for Python FaaS.

Based on the observations from Figure 4, we know there must be some overhead in addition to the container initialization in the cold start scenario that Java suffers but Python avoids. It is likely Java Virtual Machine (JVM) warmup that contributes more latency to the Java FaaS function execution, after the container is up.

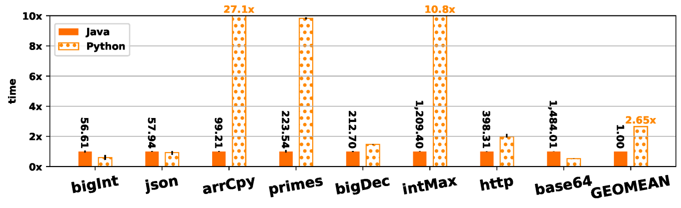
C. Data Scaling

As revealed in prior work [19] that loading Java classes and interpreting bytecodes that JVM has not seen before are the main sources to JVM warmup overhead. Therefore, JVM warmup overhead should not scale with the input size. Based on this theory, we scale the data size and monitor how the difference between cold starts and warm starts changes. Figure 5 visualizes the results. For *base64*, *http*, and *primes*, the cold start and warm start difference remains almost constant as the overall response latency rise steeply. It is a bit different for *json* (Figure 5c), where the gap between cold start and warm start enlarges gradually. This is because *json* function is relatively short and the function invocation rate is fairly high. A small increase in the response latency of *json* could lead to the chained effect, that more requests are queued up and the requests arrive later accumulate latency. In other word, the server drifts from the balanced state to over-invoked state similar to what was observed in Shahrad et al. [17].

D. Java vs. Python OpenWhisk Performance



(a) cold start



(b) warm start

Fig. 6: Java and Python OpenWhisk functions performance comparison. The end-to-end execution time of Python functions are normalized to the time of Java functions, whose absolute values are labeled above the solid bars.

Figure 6 shows a comparison of Java vs. Python functions in cold (Figure 6a) and warm (Figure 6b) execution environments. Both plots show total response latency (i.e., the sum of function execution time and initialization time if cold start). Due to the longer initialization time that Python has (§ IV-B), the Java function is faster in all except *base64* for the cold start scenario. For the warm start scenario, Java is still faster on average, but there is more variation between functions. On average (geometric mean), **Java function implementations**

are $2.60\times$ faster than Python in cold start scenarios and $2.65\times$ faster than Python in warm start scenarios.

V. Future Work

We have shown that function length affects containerization overhead, and that typically longer functions perform better (§ IV-A). Further study could try to determine the optimal function length for any language or FaaS architecture. Similar optimization on memory consumption and invocation rate should also be done. Most FaaS research to date has been focused on the performance of very short functions. Understanding the FaaS overhead for larger applications would be extremely helpful in determining the range of use cases where FaaS can provide a cost or performance benefit.

VI. Conclusions

This study provides three key takeaways for the FaaS developers. *First, function execution time should be significantly longer than the container initialization time in order to keep FaaS overheads low.* In general, short functions have a higher containerization overhead than long ones, and especially in cold start scenarios. Writing functions with longer execution time (breaking application code into larger pieces) can amortize the constant container initialization time.

Second, *pay special attention to cold start overheads for functions which are not invoked frequently enough.* The cold start overheads of these functions will not be amortized enough. The cost of these cold starts varies depending on language. The results presented here show that the container initialization time for Python functions is higher than that of Java. However, Java functions typically incur an additional slowdown (from JVM) in the function execution phase on cold starts, while Python functions have consistent execution times regardless cold or warm containers. Pre-warming and other strategies could be explored to reduce these overheads.

Third, *the same FaaS function written in different programming languages could have quite different performance.* Pre-compiled languages (e.g., Java) tends to be faster than interpreted languages (unless the platform has special optimizations). This is extremely important for developers to understand as choosing the appropriate platform/language pair can have a major impact on performance. In the case of OpenWhisk, Java performs better, even considering the additional overhead caused by JVM warmup during cold starts.

References

[1] E. van Eyk, L. Toader, S. Talluri, L. Versluis, A. Uță, and A. Iosup, "Serverless is more: From paas to present cloud computing," *IEEE Internet Computing*, vol. 22, no. 5, pp. 8–17, 2018.

[2] Amazon. (2020) Amazon Web Services. [Online]. Available: https://aws.amazon.com/

[3] Google. (2022) Google Cloud Platform. [Online]. Available: https://cloud.google.com/

[4] Microsoft. (2019) Microsoft Azure. [Online]. Available: https://azure.microsoft.com/en-us/

[5] Amazon. (2019) AWS Lambda. [Online]. Available: https://aws.amazon.com/lambda/

[6] Google. (2020) Could Functions — Google Cloud. [Online]. Available: https://cloud.google.com/functions/

[7] IBM. (2019) IBM Cloud Functions. [Online]. Available: https://cloud.ibm.com/functions/

[8] Microsoft. (2020) Azure Functions Serverless Compute: Microsoft Azure. [Online]. Available: https://azure.microsoft.com/en-us/services/functions/

[9] Dropbox. (2020) Dropbox. [Online]. Available: https://dropbox.com/

[10] Google. (2020) G Suite. [Online]. Available: https://gsuite.google.com/

[11] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, *Serverless Computing: Current Trends and Open Problems*. Singapore: Springer Singapore, 2017, pp. 1–20.

[12] I. Baldini, P. Cheng, S. J. Fink, N. Mitchell, V. Muthusamy, R. Rabbah, P. Suter, and O. Tardieu, "The serverless trilemma: Function composition for serverless computing," in *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 89–103. [Online]. Available: https://doi.org/10.1145/3133850.3133855

[13] M. Villamizar, O. Garcés, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, and M. Lang, "Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016, pp. 179–182.

[14] Apache. (2019) Apache OpenWhisk. [Online]. Available: https://openwhisk.apache.org/

[15] D. Jackson and G. Clynch, "An investigation of the impact of language runtime on the performance and cost of serverless functions," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, 2018, pp. 154–160.

[16] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, "Cold start influencing factors in function as a service," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, 2018, pp. 181–188.

[17] 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*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 1063–1075. [Online]. Available: https://doi.org/10.1145/3352460.3358296

[18] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 133–146. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/wang-liang

[19] D. Lion, A. Chiu, H. Sun, X. Zhuang, N. Grcevski, and D. Yuan, "Don't get caught in the cold, warm-up your JVM: Understand and eliminate JVM warm-up overhead in Data-Parallel systems," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 383–400. [Online]. Available: https://tinyurl.com/osdi16lion

[20] H. Lee, K. Satyam, and G. Fox, "Evaluation of production serverless computing environments," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018, pp. 442–450.

[21] H. Martins, F. Araujo, and P. R. da Cunha, "Benchmarking serverless computing platforms," *Journal of Grid Computing*, vol. 18, no. 4, pp. 691–709, Dec 2020. [Online]. Available: https://doi.org/10.1007/s10723-020-09523-1

[22] M. Elsakhawy and M. Bauer, "Performance analysis of serverless execution environments," in *2021 International Conference on Electrical, Communication, and Computer Engineering (ICECCE)*, 2021, pp. 1–6.

[23] (2020) Docker. [Online]. Available: https://www.docker.com/

[24] (2020) Apache CouchDB. [Online]. Available: https://couchdb.apache.org/

[25] (2020) NGINX. [Online]. Available: https://www.nginx.com/

[26] (2020) Apache Kafka. [Online]. Available: https://kafka.apache.org/

[27] Apache. (2020) Apache OpenWhisk Github. [Online]. Available: https://github.com/apache/openwhisk

[28] PrincetonUniversity. (2020) faas-profiler. [Online]. Available: https://github.com/PrincetonUniversity/faas-profiler

[29] OpenJDK. (2020) Code Tools: jmh. [Online]. Available: https://openjdk.java.net/projects/code-tools/jmh/