Uncovering The Impact of Bursty Workloads on System Performance in Serverless Computing

Li Wang, Shiyue Hou, Yiming Xie, Ningfang Mi
Department of Electrical and Computer Engineering, Northeastern University, Boston, USA

2024 International Symposium on Networks, Computers and Communications (ISNCC) | 979-8-3503-6491-0/24/$31.00 ©2024 IEEE | DOI: 10.1109/ISNCC62547.2024.10759023

***Abstract*—Serverless computing has emerged as a rapidly evolving paradigm within cloud services. Understanding the diverse arrival patterns of bursty serverless workload and their impact on performance and cost within serverless platforms is paramount. To the best of our knowledge, our study represents the first comprehensive analysis of observed bursty serverless workload characteristics, uncovering both anticipated and unexpected findings that illuminate the intricate interactions among workload characteristics, serverless platforms, and the associated performance-cost trade-offs. Through this analysis, we aim to furnish data-driven insights to serverless cloud providers, system administrators, and application developers, offering guidance on navigating the performance-cost trade-offs and potential pitfalls with various bursty workload arrival patterns in the realm of serverless computing.**

I. Introduction

Motivation. Understanding bursty serverless workload characteristics and arrival patterns and developing a robust performance evaluation methodology are both crucial and time-sensitive to effectively address the challenges present in heterogeneous serverless environments. Serverless applications are known for their short-lived and stateless nature. Application developers prioritize cost-efficiency and liberation from complex system configurations, resource management, and the intricacies of developing models for efficient scheduling and load-balancing algorithms. Cloud providers face significant challenges in aiming to offer compatible serverless computing services: determining the timing window and approach for keeping containers warm to mitigate the famous cold-start issues, while aligning with the applications workload characteristics, as it can be costly to maintain containers continuously active; identifying the key metrics essential for evaluating performance in serverless computing environments; striking a balance between resource provisioning and optimizing performance to achieve desired operational efficiency.

Numerous endeavors [1], [2], [3], [4] [5], [6], [7], [8] are underway to uncover the characteristics of serverless workloads and to establish effective performance evaluation methodologies to tackle the aforementioned challenges. However, these existing works encounter several limitations. Firstly, there lack of comprehensive insights into bursty serverless workload arrival patterns. The intricacies of bursty serverless workload arrival patterns remain largely elusive, as much of the available data is tailored for specific purposes, such as testing proposed scheduling algorithms [1], [2], [3]. Secondly, there is a notable absence of dedicated bursty workload generators designed to produce tailored serverless workloads emulating real-world distributions, hindering the ability to accurately replicate real-world scenarios. Thirdly, existing evaluations primarily focus on simplistic performance metrics such as latency for cold-start, overlooking comprehensive analyses of resource utilization, platform configurations, and end-to-end latency [4] [5]. Fourthly, existing serverless benchmark suites [5], [6], [7], [8] often feature simple serverless applications, failing to capture the complexity and diversity of bursty workload arrival patterns and characteristics and conduct limited emulations to simulate various levels of burstiness in serverless computing.

Contributions. To the best of our knowledge, this is the first experimental study that characterizes various customized bursty serverless workload arrival patterns and offers a comprehensive evaluation of system configurations, performance, resource utilization, and aligning with various bursty serverless workload distributions. In particular, we build a tailored architecture that addresses the challenges of creating specific serverless workloads with burstiness and evaluating performance using various metrics. We conduct an experimental study on the open-source OpenWhisk [9] serverless platform to quantify and characterize various bursty serverless workload arrival patterns with different resource configurations. All the experiments are carried out using OpenWhisk which is installed on Ubuntu 18.04.2 LTS server, equipped with 24 CPU cores.

Our study reveals that bursty serverless workloads with varying arrival patterns show significant differences in performance and resource utilization on the serverless platform. These variations depend on system resource configurations, including CPU and memory usage, end-to-end latency, and container-related settings. Specifically, these settings encompass the maximum number of containers per invoker worker node, the query limit for a collection of list operations against OpenWhisk, and aspects of the container life cycle, such as idle duration, pause grace period, and keep-alive duration for individual containers. One of the surprising revelations of our study is that if there are no resource constraints, bursty workloads lead to fewer cold starts, shorter average function execution times, and reduced CPU and memory allocations in serverless computing environments.

II. Background

A. *Serverless Workloads Arrival Patterns*

Serverless workloads are renowned for their transient nature and stateless characteristics. Serverless applications can be invoked either on-demand or be scheduled following specific inter-arrival time (IAT) distributions. The main statistics to represent a specific arrival pattern (i.e., IAT) include the mean (μ), standard deviation (STD, σ), cumulative distributed function (CDF), and auto-correlation function (ACF).

Burstiness. A bursty workload pattern is characterized by alternating periods of intense activity followed by relative calm. These patterns exhibit abrupt spikes in workload arrival, followed by intervals of reduced activity. Burstiness can be quantified using ACF [10], a widely used tool in time series analysis for identifying bursty workload arrival patterns. (1) ACF=1 represents a perfect positive correlation between the time series and its lagged counterpart. (2) ACF=-1 denotes a perfect negative correlation. (3) ACF=0 indicates there is no correlation between the time series.

B. *Cold Start*

In serverless computing, a cold start [1] refers to the initial delay that occurs when a cloud provider must initialize a new instance of a serverless function in response to an incoming request. When a serverless function is invoked, the cloud provider needs to allocate resources, such as a container or virtual machine, to run the function. If no instance of the function is currently running, the provider must allocate resources and initialize the environment.

A cold start can occur during the first invocation of an application, when existing function instances are insufficient to handle incoming requests, or after a predefined period of container inactivity. Cold starts are an inherent aspect of serverless computing, affecting the performance of serverless applications, especially those with stringent latency requirements. Various strategies can help mitigate the frequency and impact of cold starts. Some cloud providers offer options to keep a specified number of function instances always warm by adjusting the keep-alive window, idle duration, and pause grace period of containers.

C. *OpenWhisk*

OpenWhisk is built up on Nginx, Kafka, CouchDB, and Docker containers [9]. The process flow of an action invocation in OpenWhisk is shown in Fig. 1. Commands issued via the wsk command-line interface (CLI) [11] are translated into HTTP requests directed towards the system. (1) These requests are first intercepted by Nginx, a server responsible for forwarding HTTP requests to the controller. The controller, developed in Scala [12], hosts a REST API interface, facilitating user interactions. Upon receipt of a request, the controller initiates an HTTP POST request to invoke the desired action. (2) Authentication and authorization processes are then completed with the assistance of CouchDB, following which the controller retrieves the eligible action from the CouchDB repository. Within the controller, a load balancer assesses the availability and health status of all invokers within the OpenWhisk system. (3) Communication between the controller and invokers is exclusively orchestrated through the Kafka message queuing system. (4) Upon receiving a message from the controller that contains the action to invoke and associated parameters, Kafka assigns an ActivationId to the action and dispatches the message to an available invoker. (5) The invoker, one of the core components of OpenWhisk, executes actions within self-encapsulated and secure environments, facilitated by Docker containers. Each action is executed within its designated container, which is subsequently dismantled upon completion. The results of the invocation of an action are then saved into the activations database in CouchDB. Users can use the ActivationId to retrieve invocation records stored within the CouchDB repository.

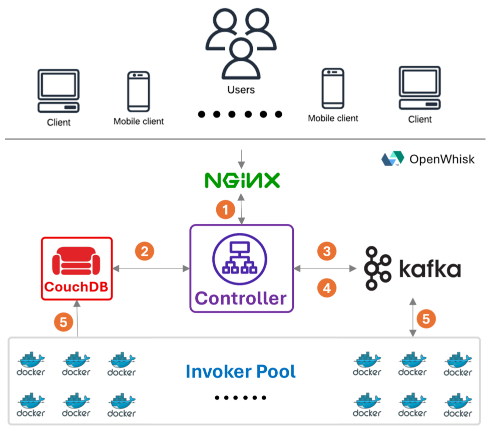


Fig. 1. The process flow of action invocation in OpenWhisk

III. Experimental Methodology

A. *Architecture Overview*

In this work, we build a tailored architecture to generate bursty serverless workloads, handle serverless invocations, and evaluate performance. Specifically, our architecture consists of a workload generator, a request initiator, and a performance analyzer.

The workload generator is developed to tailor serverless workload arrival patterns by manipulating the key workload statistics. To generate workloads with burstiness, the process involves the following steps. (1) Input parameters to the KPC-ToolBox[1] include IAT statistics (e.g., μ, σ, ACF) and the number of invocations. (2) KPC-ToolBox then generates a Modulated Markovian Arrival Process (MMAP) that closely matches the desired trace characteristics. (3) Using the generated MAP and additional inputs such as a random seed and the required number of data points, the BMAP-Trace module finally generates the corresponding arrival trace.

The request initiator is designed to efficiently distribute requests to a serverless computing platform, OpenWhisk, accommodating diverse distributions and execution criteria across various scenarios. The inputs to the request initiator include the generated IAT trace, the set of OpenWhisk actions, and the specified emulation duration. Subsequently, the request initiator initiates the invocations of uploaded actions. Clients

[1]KPC-Toolbox [13] is a tool to accurately fit trace data into Markovian Arrival Processes (MAPs), which are used to fit real-world workloads with time-dynamic characteristics.

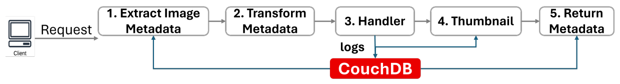


Fig. 2. The serverless application - image processing sequence

can retrieve activation details using the associated activation ID.

The performance analyzer is engineered to evaluate system performance and behavior by leveraging an array of performance metrics. The performance metrics of evaluation include average end-to-end latency (L), average CPU utilization (U_{cpu}), average memory utilization (U_{mem}), average memory allocation (M), and cold start ratio (R_C). Here, the cold start ratio is defined as the ratio of the number of cold start invocations N_C to the total number of invocations at the function level N_F.

B. *Serverless Platform*

In this work, we adopt OpenWhisk to conduct the performance analysis for serverless computing under bursty workloads. Specifically, our experiments involve executing different numbers of concurrent invocations (up to 500) and serverless arrival patterns with different IAT distributions (e.g., Gaussian, MMPP), and varying mean arrival rates and degrees of burstiness. Our experiments are also performed under both resource-constrained and non-resource-constrained OpenWhisk system configurations by changing the number of containers, container idle duration, pause grace period, keep-alive duration, etc. The serverless application used in our experiments is an image-processing sequence (see Fig. 2) from Serverlessbench [5]. We use an input image with a size of 2MB.

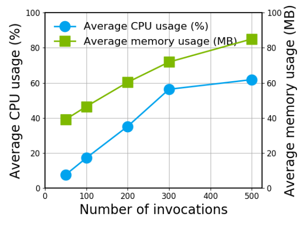
IV. Result Analysis

In this section, we analyze the performance results at both the host and the OpenWhisk levels in terms of different concurrency, bursty serverless workload arrival patterns, and OpenWhisk container configurations.

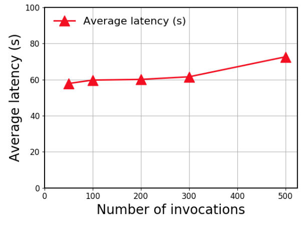
A. *Sensitivity Analysis on Concurrency*

We first conduct sensitivity analysis on different levels of concurrency by varying the number of concurrently running invocations (e.g., from 50 to 500). In this set of experiments, we use the default configurations in OpenWhisk and set the average IAT as 10 seconds. Fig. 3 shows (a) the resource utilization and (b) the average end-to-end latency under different concurrency levels.

Our observations indicate that as the number of concurrent invocations (N_A) increases, there is a corresponding increase in both resource utilization (i.e., U_{cpu} and M), and latency L. This trend highlights a degradation in system performance and an increase in resource utilization with high concurrency levels. We also observe that memory usage increases linearly while CPU utilization and latency reach an inflection point at a concurrency level of 300, indicating resource saturation. At the concurrency level of 500, the average latency (L) shows a significant increase. This happens because the system's resources become more saturated, resulting in longer processing times for each request.



(a) U_{cpu} and M



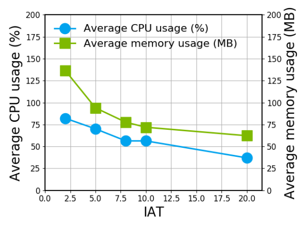
(b) L

Fig. 3. Average resource utilization and latency: concurrency, IAT=10s

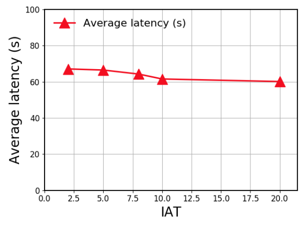
Summary and Implications. The performance degradation observed with increasing concurrency is primarily due to resource contention and saturation. Increased concurrency can lead to frequent context switching between processes or threads, adding overhead and further contributing to increased latency and reduced overall efficiency.

B. *Sensitivity Analysis on IATs*

We conduct experiments with various mean IATs (e.g., 20s, 10s, 8s, 5s, and 2s) but fix the concurrent invocation number (N_A) as 300. The arrival process is drawn from a Gaussian distribution. From Fig. 4, we observe that the workload intensifies when IAT decreases from 20s to 2s. Consequently, there are corresponding increases in resource utilization and end-to-end latency. More frequent requests lead to higher CPU and memory usage as the system needs to handle multiple invocations simultaneously. This can quickly exhaust available resources, especially in a serverless environment where resources are allocated dynamically. Furthermore, when IAT is decreased from 10s to 2s, there is a significant increasing trend in resource utilization, see in Fig. 4 (a). This indicates that the resource saturation point is reached. As a result, we can also an increase in end-to-end latency as shown in Fig. 4 (b). Each request now takes longer to process due to the high demand for computational resources and the overhead associated with managing multiple concurrent invocations.



(a) U_{cpu} and M



(b) L

Fig. 4. Performance with various IATs, concurrency=300.

Fig. 5 further depicts the variance of latency for two different IATs (i.e., 2s and 10s) to investigate the impact of workload intensity on latency distributions. In this figure, we also present the results for three groups of invocations, i.e., the first, middle, and last ten invocations. The variance in latency distribution shows notable changes across different IAT values. More frequent requests (i.e., smaller IATs) lead to higher variance and larger median latency. In addition, the first ten invocations show slightly higher latency compared to the middle and last ten invocations, particularly at lower IATs. This could be due to the initial load on the system. The middle and last ten groups tend to have similar latency distributions.

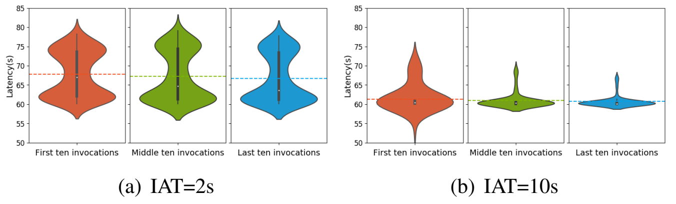


Fig. 5. Variance of latency with different IATs, concurrency=300.

Summary and Implications. As workload intensifies, dynamic resource allocation must be optimized to handle increased load and prevent resource saturation. To maintain optimal performance, IATs should be monitored to adjust configurations.

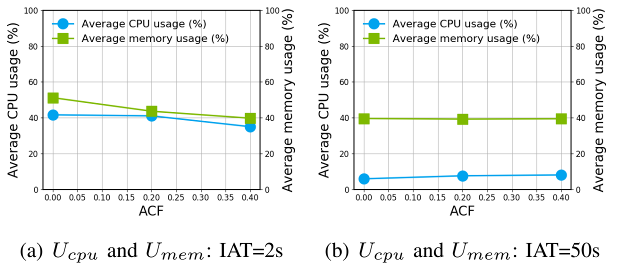
C. Sensitivity Analysis on Burstiness (ACF)

We conduct experiments with various degrees of burstiness by changing ACF at lag 1 of (0.0, 0.2, 0.4) under both light workloads (e.g., IAT=50s) and heavy workloads (e.g., IAT=2s). We note that to thoroughly understand the impact of burstiness on performance, we set a non-constraint resources environment in OpenWhisk by increasing the number of containers in an invoker pool by 5X and the size of user memory in a container by 1024X.

Fig. 6 shows the the CPU and memory utilization and Fig. 7 shows the latency and the cold start ratio under different ACFs. These results indicate that for the workload with IAT=2s, U_{cpu} decreases slightly as the ACF increases from 0.0 to 0.4, starting around 50% and dropping to about 35%; while U_{mem} decreases linearly as ACF increases. A similar trend is observed in the latency, where L decreases significantly from 23s to 4s when ACF increases from 0.0 to 0.4. We interpret this by observing that the cold start ratio R_C decreases significantly as ACF increases to 0.4. This means that high ACF leads to a high locality of invocations and thus increases the hit rate of the invoke instances, reducing the cold start. However, when the workload is very light, e.g., IAT=50s, this effect of ACFs becomes negligible.

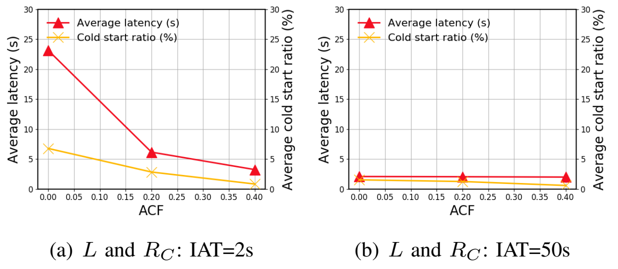
High ACF values often correlate with bursty traffic patterns, which results in periods of intense resource usage. Fig. 8 shows (a) the CPU utilization across time on two CPU cores used in our experiments and (b) the corresponding arrival rates (i.e., the number of invocations) across time. We can observe that the patterns of resource usage and arrival rates align very well. This is because the system must handle large volumes of requests within short time frames, leading to higher CPU and memory consumption. With more bursts of activities, resources can become saturated quickly, causing contention as multiple functions compete for the same resources.

However, as discussed above, with abundant resources, both heavy (IAT=2s) and light (IAT=50s) traffic can benefit from burstiness due to locality and container proxy mechanisms, such as keep-alive windows and idle duration settings. These mechanisms allow frequently accessed resources to remain available through warm containers, which helps improve system performance and resource utilization. By keeping containers warm, the system can respond more quickly to incoming requests, reducing the latency typically associated with cold starts and enhancing overall efficiency during periods of high demand.



(a) U_{cpu} and U_{mem}: IAT=2s (b) U_{cpu} and U_{mem}: IAT=50s

Fig. 6. Average resource usage with various ACFs.



(a) L and R_C: IAT=2s (b) L and R_C: IAT=50s

Fig. 7. Average latency and cold start ratio with various ACFs.

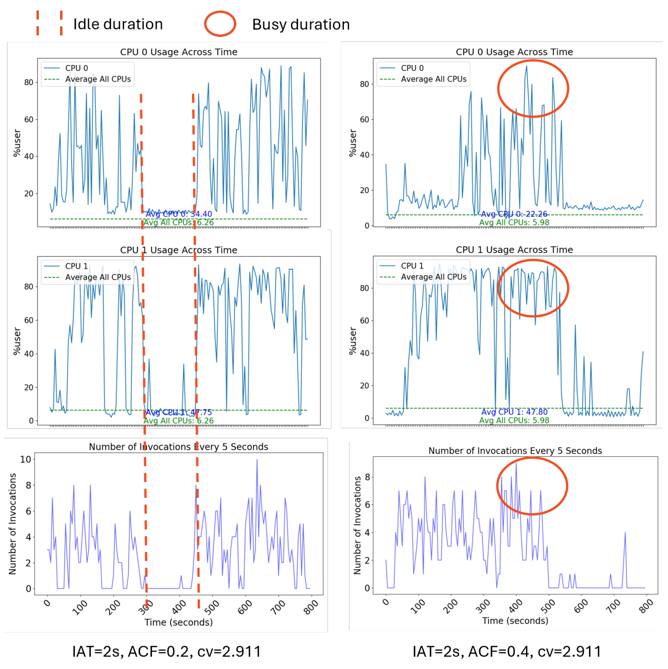


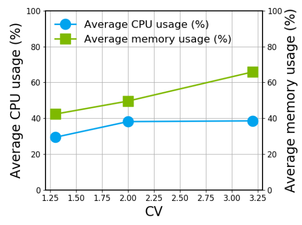
Fig. 8. CPU usage and workload arrival patterns with ACF=0.2 and IAT=2s.

Summary and Implications. The findings suggest that (1) the R_C decreases for both heavy and light workloads because of non-constraint resources; (2) more bursty workloads (higher ACF values) may lead to marginally better resource efficiency, improve latency performance under heavy loads, and reduce the frequency of cold starts to improve performance.

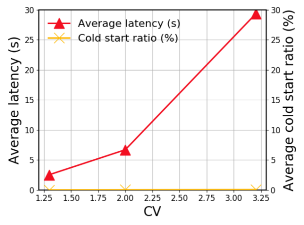
D. *Sensitivity Analysis on CV*

The coefficient of variation (CV) allows for the comparison of variance across different workloads or periods. Here, we conduct sensitivity analysis of CVs(1.3, 2.0, and 3.2) but fix IAT (2s) and ACF (0.2). Fig. 9 illustrates that both U_{cpu} and U_{mem} increase with the increase of CV but R_C does not change significantly due to low ACFs.

Both CPU and memory usage increase with higher CVs, indicating that higher variability in the arrivals of requests leads to higher resource demand. This trend suggests that the system needs to allocate more resources to handle the increased load variability effectively. Moreover, higher CVs lead to more unpredictable workloads, causing delays in processing and longer wait times for resources. As a result, we can observe the dramatic increase in L with higher CV values in Fig. 9 (b), which indicates severe degradation in system performance.



(a) U_{cpu} and U_{mem}



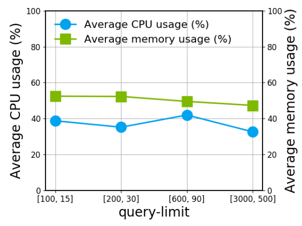
(b) L and R_C

Fig. 9. Performance of various CVs; IAT=2s, ACF=0.2.

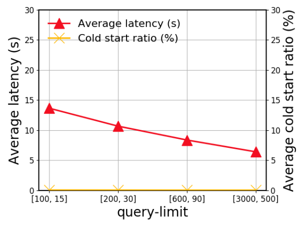
Summary and Implications. If the system is not adequately provisioned for peak demands due to high variability, it can lead to resource contention and significantly degrade performance. High CV necessitates more sophisticated dynamic scaling strategies to adjust resources in real-time based on fluctuating demand, adding complexity to system management and incurring additional overhead.

E. *Sensitivity Analysis on Container Configurations*

To better understand different configurations on containers in OpenWhisk, we first conduct experiments on various *query-limit* settings ([100, 15], [200, 30], [600, 90], and [3000, 500]), which represent the maximum and the default numbers of entities that can be requested from a collection on a list operation in OpenWhisk [9]. In this set of experiments, we consider a heavy workload with IAT=2s and ACF=0.2. Our observations from Fig. 10 indicate that the best system performance is achieved at configuration [3000, 500] with the lowest resource usage, latency, and cold start ratio.



(a) U_{cpu} and U_{mem}



(b) L and R_C

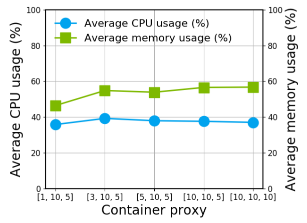
Fig. 10. Performance of various query-limit configurations.

We then continue to conduct experiments under the query-limit setting of [3000, 500] but with different container proxies to better understand how idle container duration and keep-alive duration affect system performance and resource utilization. Here, idle container duration refers to the time a container remains inactive until the system terminates it, while keep-alive duration is the time the system maintains a container in an active state. Specifically, we set four different container proxies by varying idle container duration (1, 3, 5, 10 mins) and keep-alive duration (5, 15, 60 mins). In addition, we also set different pause grace periods (i.e., the period the system maintains a container after deciding to terminate it) to 5, 10, and 60 minutes. So, the container configuration shown in the following figures is formatted as [idle-duration, pause-grace, keeping-duration].

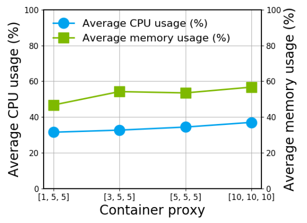
Figs. 11 and 12 show the performance results under the four different container proxies. We find that when the idle duration is 5 minutes and the keep-alive duration is set to 5, 10, or 60 minutes, R_C, U_{cpu}, and L outperform other configurations. Setting the idle duration to 5 minutes means that containers are kept ready for a moderate amount of time before being terminated. This strikes a balance between keeping containers warm to reduce cold starts and not holding onto resources for too long. Optimal utilization of CPU resources is thus achieved as the system efficiently manages active containers, avoiding the overhead of frequent cold starts. Reduced cold starts and efficient resource management contribute to lower average latency, improving overall user experience.

Meanwhile, keep-alive duration allows containers to remain available for subsequent requests, reducing the likelihood of cold starts. We then further conduct experiments to investigate keep-alive window settings (i.e., keeping-duration=3, 5, 15, 30 min) with fixed idle duration (1 min) and pause grace period (5 mins). We show the performance results of these settings and the default configuration [10, 10, 10] in Fig. 13. Our observations from Fig. 13 indicate that the configuration [1, 5, 5] obtains lower U_{mem} and L than other configurations, while the default configuration [10, 10, 10] performs best in terms of the cold start ratio (R_C). The configuration [1, 5, 5] optimizes memory usage by balancing container readiness and resource allocation, preventing excessive memory consumption. With shorter idle and keep-alive duration, this configuration minimizes latency by ensuring that containers are frequently available, thus reducing the time needed to handle incoming requests. While, the longer idle and keep-alive duration in the default configuration [10, 10, 10] reduces the frequency of cold starts, as containers are kept warm for extended periods, ensuring they are ready to handle new requests promptly.

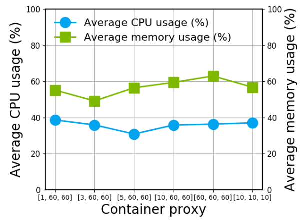
Summary and Implications. (1) With fewer cold starts due to the optimal idle and keep-alive duration, the cold start ratio is minimized, leading to improved system responsiveness. (2) By carefully configuring idle and keep-alive duration, the system maintains a balance between resource availability and utilization, resulting in superior performance metrics compared to other configurations. (3) By experimenting with different



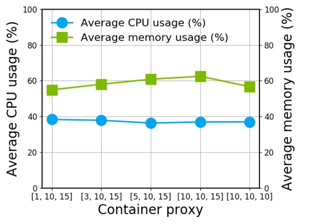
(a) U_{cpu} and U_{mem}: proxy 1



(b) U_{cpu} and U_{mem}: proxy 2

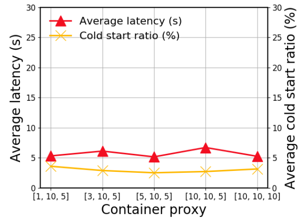


(c) U_{cpu} and U_{mem}: proxy 3

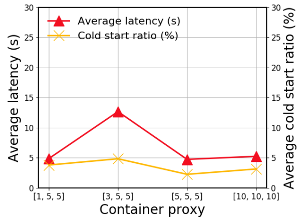


(d) U_{cpu} and U_{mem}: proxy 4

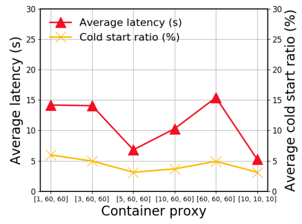
Fig. 11. Average resource usage with various container proxies.



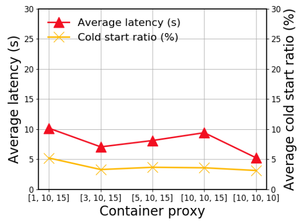
(a) L and R_C: proxy 1



(b) L and R_C: proxy 2



(c) L and R_C: proxy 3



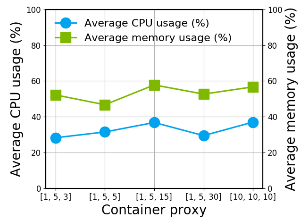
(d) L and R_C: proxy 4

Fig. 12. Average latency and cold start ratio with various container proxies.

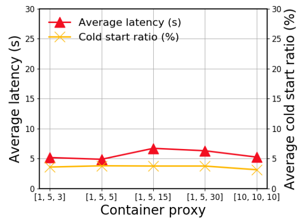
configurations, we can tailor the system's performance to specific metrics, optimizing for CPU/memory utilization, end-to-end latency, or cold start ratio. Each configuration offers unique advantages, allowing us to choose the best setup based on the desired performance criteria.

V. Conclusion and Future Work

Our comprehensive study delves into the intricacies of serverless computing, particularly focusing on understanding bursty serverless workload arrival patterns and their impact on system performance and resource utilization with various configurations on the OpenWhisk platform. This work offers practical guidance for improving serverless platform design and application development practices and advancing the understanding and optimization of serverless environments. Future research should focus on investigating the long-term impacts of various configurations on cost-efficiency and scalability or bursty workload arrival patterns, artificial intelligence-based workload prediction, as well as resource management



(a) U_{cpu} and U_{mem}



(b) L and R_C

Fig. 13. Performance of various keep-alive windows.

strategies, which will be crucial for advancing serverless computing technologies.

Acknowledgments

This work was partially supported by National Science Foundation Award CNS-2008072.

References

[1] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 205–218. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/shahrad

[2] A. Sahraei, S. Demetriou, A. Sobhgol, H. Zhang, A. Nagaraja, N. Pathak, G. Joshi, C. Souza, B. Huang, W. Cook, A. Golovei, P. Venkat, A. Mcfague, D. Skarlatos, V. Patel, R. Thind, E. Gonzalez, Y. Jin, and C. Tang, "Xfaas: Hyperscale and low cost serverless functions at meta," 10 2023, pp. 231–246.

[3] A. Wang, S. Chang, H. Tian, H. Wang, H. Yang, H. Li, R. Du, and Y. Cheng, "FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021, pp. 443–457. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/wang-ao

[4] S. Eismann, J. Scheuner, E. van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, and A. Iosup, "A review of serverless use cases and their characteristics," 2021.

[5] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, "Characterizing serverless platforms with serverlessbench," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '20. Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3419111.3421280

[6] J. Scheuner, S. Eismann, S. Talluri, E. van Eyk, C. Abad, P. Leitner, and A. Iosup, "Let's trace it: Fine-grained serverless benchmarking using synchronous and asynchronous orchestrated applications," 2022.

[7] A. Katangur and L. Zhang, Eds., *Services Computing - SCC 2021 - 18th International Conference, Held as Part of the Services Conference Federation, SCF 2021, Virtual Event, December 10-14, 2021, Proceedings*, ser. Lecture Notes in Computer Science, vol. 12995. Springer, 2022. [Online]. Available: https://doi.org/10.1007/978-3-030-96566-2

[8] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, "Sebs: A serverless benchmark suite for function-as-a-service computing," 2021.

[9] Apache OpenWhisk, https://https://openwhisk.apache.org/.

[10] NIST, https://www.itl.nist.gov/div898/handbook/eda/section3/eda35c.htm.

[11] Apache OpenWhisk Command-line Interface wsk, https://github.com/apache/openwhisk-cli.

[12] Docker, https://www.scala-lang.org/.

[13] G. Casale, E. Z. Zhang, and E. Smirni, "Kpc-toolbox: Simple yet effective trace fitting using markovian arrival processes," in *Fifth International Conference on the Quantitative Evaluaiton of Systems (QEST 2008), 14-17 September 2008, Saint-Malo, France*. IEEE Computer Society, 2008, pp. 83–92. [Online]. Available: https://doi.org/10.1109/QEST.2008.33