A Survey of Cost Optimization in Serverless Cloud Computing

Zhe Li a, Yusong Tan b, Bao Li c, Jianfeng Zhang d, Xiaochuan Wang e

College of Computer, National University of Defense Technology Changsha, China

a lizhe18a@nudt.edu.cn, b tanyusong@kylinos.cn, c baoli@nudt.edu.cn, d zhangjianfeng@kylinos.cn, e wangxiaochuan@kylinos.cn

Abstract. Recently serverless cloud computing which was proposed by Amazon in 2015 is getting more favored by developers because of its charging model of pay-by-usage, applicability for fine-grained services and transparency of the servers to developers. Although a lot of study has proved that the performance and cost of serverless platform is more ideal than the traditional cloud computing service, the new metering model of serverless which depends on execution time, usage count and memory footprint is still unfamiliar to most developers and researchers. In this paper, we firstly make an introduction to the new metering model and the advantages of serverless over traditional cloud services. Then according to the cost affecting factors, we make a comprehensive survey of some constraints that mainly impact the cost of serverless and related methods to reduce cost in three levels—function level, container level and cloud platform level. Based on the analysis, we also propose some directions worthy of in-depth study in the future.

Keywords: cloud computing, serverless, cost optimization, FaaS

1. Introduction

In the early days of cloud computing, all kinds of resources at different levels (IaaS, PaaS and SaaS) can be charged on demand as a service [1]. The introduction to cloud services has undoubtedly promoted the migration of traditional applications to clouds in order to serve more users. However, unlimited computing resources are accompanied with the complexity to developers of managing distributed environments which include instance selection, auto scaling, fault tolerance, monitoring, logging and so on [2,3]. Meanwhile, with the development of virtualization technology and the emergence of the new container technologies such as docker, micro-service is getting favored by people[34]. It seems that there is an increasing need for a fine-grained cloud service with simple operations.

2. Definition

To keep developers away from tedious server management, Amazon proposed a new fine-grained cloud computing paradigm called AWS Lambda[4] which introduced a new conception termed "serverless" at the same time. After that, then Microsoft released Azure Functions [5], Google released Google Cloud Functions [6] and IBM released OpenWhisk [7].
Fig 1 shows a typical architecture of serverless computing. As we can see an application will be implemented by an amount of functions. A serverless computing platform always has an interface to upload codes and functions on the platform are triggered by a variety of events after deployment finished. Each function can maintain multiple versions of code at the same time and often has more than one instance while working. To maintain the states of functions, it is often backed with an object storage service.

![Fig.1 The Architecture of the serverless](image)

Serverless could be defined FaaS (Function as a service) and BaaS (Backend as a service) where FaaS represents the cloud offerings like AWS Lambda while BaaS usually represents some specific serverless cloud service like databases [2]. It must scale automatically with no need for explicit provision and be billed based on usage. This paper only discusses the core of serverless—FaaS which is a serverless platform where the unit of computation is executed in response to events such as HTTP requests. From the perspective of cloud computing, FaaS can be regarded as a special service pattern overlapped with PaaS and SaaS [8]. As we know, in an IaaS model, you have the most control over both the code and the infrastructures, but it also means you will have the most responsibility to manage all the servers you own. Developers in a PaaS model are unaware of the infrastructures, they still need to face the complex distributed environments. On the other hand, developers don't need to manage the servers but only care about their business in a SaaS model. The greatest advantage of FaaS model is that developers can develop functions of any business without managing any servers.

2.1. Background

For startups and some small or medium-sized IT companies, the most two considerable problems about IT infrastructure are hosting cost and computing power [11]. Due to its billing model and infinite resources, serverless platform is better for solving events-driven problems especially for high concurrency tasks. Lots of surveys in this area refer to the applicable scenarios of the serverless platform, mainly including event-driven web applications, big data processing and stream media processing [2,8,9,12]. In academia and industry, some research work has been done to compare the traditional cloud computing platform with serverless platform from different aspects especially in their performance and price on these particular scenarios.

Adzic and Chatley compared the cost and performance between Serverless platform and traditional cloud platform through two cases of MindMup and Yubl [13]. The number of MindMup users increased by 50% in 2017 with the cost reduced by 50%. In addition, they don't need to pay for any idle resources and the updates are dozens of times faster than before. Villamizar made a cost comparison of running web applications in the cloud using monolithic, micro-service, and AWS Lambda architectures [14]. The results showed that the cost of using the serverless platform was more than 60% lower than the other two, while the response time was always maintained at a low level. In addition, Ivan did the similar work to investigate different deployment environments for Web APIs in 2019 [15], pointing out that serverless performed better than traditional cloud services when the number of requests become larger.
2.2. Motivation
The problem of cost optimization has always been a hot topic in cloud computing. Considering cost in serverless, we view this problem with three levels—function level, container level and cloud platform level. Reviewing the release procedure of a PaaS or IaaS application, the cost problem almost spans the development lifecycle. Usually we will choose a public cloud platform with a high performance-price ratio at first. There is no doubt that these service providers have different cloud ecosystems which determine different applicable scenarios and different cloud application costs. Then we will design the application's distributed architecture and build a special runtime environment on the platform (for FaaS applications, the running environment is containers). A rational distributed architecture can minimize the cost and maximize the usage of resources, while a bad model can only produce nonessential waste. Finally we will develop in business according to the requirement specification, accompanied by the development cost and deployment cost. As a special cloud service, the above three steps still be considerable at cost saving and in this paper we will analysis the cost on serverless platform from top to bottom. In this paper, we do a comprehensive survey on how to make serverless applications run more economically at the above three levels.

The rest of this paper is structured as follows. In section II to section IV, we will discuss the factors that affect the cost of using the serverless platform and the related solutions at three different levels - function level, container level and cloud platform level. In section V, we make a brief conclusion and a prospect of the main research direction in the future.

3. Cost Optimization in Function-level
In the development of serverless computing, developers should always pay more attention to performance and limits, programming languages, composability and deployment[9]. Performance and limits means the maximum memory and CPU resources available and composability represents different function compositions, both of which can be attributed to the design strategy of functions. In addition, programming languages have different efficiency, which not only means execution efficiency but also means different environment initialization directly affecting the duration time.

3.1. Function Design Strategy
In serverless computing, an application must be split into multiple functions deployed on the platform. Obviously, the design strategy of the application will affect the number of functions and the executions times. Elgamal Tarek took the face recognition application in edge computing as an example, proposing a function aggregation and migration method that could effectively save the cost of serverless computing[16].

In Elgamal's research, they considered state transition, AWS Greengrass service and different memory size. State transition is a special execution factor that occurs when one function calls another. When building an application, dividing the application into multiple functions does increase scalability and usability, but it creates more data dependencies. Besides, AWS Greengrass is charged as a service for connecting edge and cloud in his experiment. In the model of this research, the cost and latency can be optimized by balancing the calculation speed with the transmission rate and the associated price. As for memory, they test different memory sizes to find the optimal solution.

In the model of that paper, functions are abstracted into the structure of a directed connected graph, where a node represents a function and an edge represents a call between two functions. Considering the architectural model of edge computing, the decomposed function can be selectively run on Edge or Cloud. In addition, all executions start with one function and end with another function which means you must follow the execution order because of requests starting from edge. This can be exemplified by the case of face recognition. The face-detection must be performed in front of checking-face-duplicate which means if the previous is executed in cloud, the latter cannot be executed on the edge, for which it will violate the data propagation direction. Thus, the price model can be depicted as [16]:

$$P(G_f, X_{1,...,n}) = \sum_{i=1}^{n} X_i \cdot r \cdot e_{(i,C)} \cdot m_{(i,C)} \cdot P_{(m_{i,C})} + r \cdot (n + 1) \cdot p_s + P_E$$ (1)
and the total execution time $T$ of the workflow is given by [16]:

$$T(G_f, X_{i=1,...,n}) = t_n(G_f, X_{i=1,...,n})t_0$$

(2)

where the notation is given by table I.

After abstracting it into a constrained shortest path problem which is NP-hard, the author then used the LARAC [35] algorithm to find an approximate optimal solution which means saving as much cost as possible with little performance degradation. The result show it can reduce the cost by 37% with only 5% increase in latency. On the other hand, this algorithm is much faster than the Bruceforce in the time to obtain the placement especially when the number of functions is large.

This method could do well in balancing the cost and performance in edge computing which means in exchange for a little performance, we can get impressive cost-savings. However, there still exists something to be further researched in the work of generalization. Furthermore, the use of other services like Greengrass in the same cloud ecosystem should also be a consideration to find an optimized strategy.

$[16]$

| Tab.1 Notation |
|----------------|
| $n$: Total number of functions |
| $r$: Total number of executions of a workflow |
| $G_f$: Input function graph |
| $f_i$: Function $i$ in graph $G_f$ |
| $X_i$: Placement variable (1: $f_i$ on cloud, 0: $f_i$ on edge) |
| $t_i$: Completion time of function $i$ |
| $e_{i,c}$: Execution time of function $i$ on the cloud |
| $e_{i,e}$: Execution time of function $i$ on the edge |
| $m_{i,c}$: Memory allocated to function $i$ |
| $p_c$: Price of connecting on edge device to AWS cloud |
| $p_s$: Price of one state transition |

3.2. Language Selection Strategy

Once the design is finished, the rest of the implementation is simple. But how to choose a suitable programming language in a serverless platform? To solve this problem, Jackson and Clynch did a research on AWS Lambada and Azure Functions which are two serverless platforms with the highest popularity [17]. To clear up the impact of language runtime on the performance and cost, they made an experiment with a small use case and compared all the programming languages on the these platforms, which are also representative on other platforms [18].

All tests were performed in batches under cold start [15] (when a function is called for the first time, it would have a very high delay to initialize the container and the language runtime environment) and warm start conditions, the test framework invoked the function through a single POST operation in the API And the result is given by table II.

Though the unit price of different languages on the same platform is with no difference, we still see the difference of the execution time which directly determines the total cost. In the experiment of cold start, it is obviously that C# spent far more time than others. Its cost is 5-10 times higher. In the case of warm start, since all the languages spent less than 100ms which is the unit billing duration unit, the cost result is not given. On the other hand, the average execution time of C# in the case of cold start on Azure is 16.84ms and in the case of warm start it even dropped to 0.93ms. In addition, NodeJS also performed well. This may be related to the internal containers on Azure which are currently based on windows container technology while the Linux-based containers in use on AWS. This would be introduced in detail in section IV. Python is obviously more worth considering if you want to use AWS Lambda and due to the complexity of Java virtual machine, it needs more time to initialize the environment than others.
This article shows the difference in performance and cost of different programming languages on a serverless platform. But what's not enough about the experiment is that the test case is relatively simple (just a web request). The duration time could almost be represented by the container initialization time. In order to better illustrate the efficiency of the runtime, we need to make some specific functional comparison according to some specific business. [17]

| Language | Runtime (ms) | Warm Start | Cold Start | Average Cost Per Million |
|----------|--------------|------------|------------|--------------------------|
|          |              | Average Execution time | Average Execution time | Average Billed Duration |                          |
| C#. Net  | 6.32         | 2500.09    | 2600       | 5.61775                  |
| Golang   | 19.21        | 8.97       | 100        | 0. 408375                |
| JAVA 8   | 11.13        | 391.91     | 400        | 1.0335                   |
| NodeJS   | 11.46        | 23.67      | 100        | 0. 408375                |
| Python   | 6.13         | 2.67       | 100        | 0. 408375                |

4. Cost OPTIMIZATION in Container-Level

Building a serverless platform is always with two key bottlenecks which are communication latency and initialization time [2,19]. Before an instance of a function running, the most important step is the initialization of the container to provide runtime environment and temporary storage for the function. This startup time is almost the most difficult bottleneck in the implementation of a serverless platform [9,20]. It means users need to spend an extra time waiting until the environment finished if there is no available instance. After the container initialization finished. It will take some time to load packages if the libraries are too large, which is also a quite burden on duration.

Another bottleneck of a serverless application lies in communication if not considering algorithm design and hardware performance. Due to the feature of fine granularity in serverless, communication between functions will increase, and each call is charged. Both of the two kinds of overheads will affect the platform cost through the crucial factor-duration time.

4.1. Initialization Overhead

In the immature period of serverless, researchers proposed some models to implement an entire serverless system following AWS Lambada but the improvement is not obvious [21,22]. With the widespread of serverless, some researchers have realized that cold start is a key problem in FaaS which could take much time at the first time of the runtime environment initialization. The initialization procedure of an instance could be viewed as two steps: container initialization and libraries initialization. The former is to provide a work space for the working process and the latter may depends on the size and number of the dependencies.

Container Initialization A simple way to prewarm the container is to call a function every so often. In the experiment of SCAR [27], the request processing is set to be blocked until the first call is finished which means there is a warm container, and the result shows great time reducing.

Another method to avoid cold start is to place the available container into a pool like a thread pool [23]. The author implemented the platform in an open source framework called Knative [24] which could provide a platform to develop and manage container applications on the top of Kubernetes. As we can see in fig 2, the auto-scaler will scale up/down the number of desired pods (a group of containers which is the unit in Kubernetes) automatically. Based on the original mechanism of Knative, the author added the step of request to pool (fig 3). When a request coming, if the number of pods is not enough, it will check the pool if there are any idle pods, then the available pods will be migrated to the target to process the request. If we can't get pods from the pool, then it will build a new one. [23]
Though keeping containers warm means we can get it instantaneously, it would incur the potential security risks in the application since the last provisional data will be retained in a warm container.

One solution to the security problem is partial reuse [25]. Only reusing the part of network can we still achieve a quick container initialization with high security. After analysis, the researchers suggested that the bottleneck of cold start is the delay of network initialization. Container pausing [26] is used to pre-initialize the network environment of a pool of containers which are remained to be bound to the runtime container. To avoid memory waste, PCPM (Pause Container Pool Manager) would activate/stop pause containers which is still in the container pool. Fig 3 shows the complete lifecycle of a runtime container. When the service is triggered, a new container will be created to provide a runtime environment which would be bundled to a pause container with initialized network environment from the PCPM. After the work finished, the runtime container would die but the pauses container would be released to reuse. The results show the total time consumed is 80% less than that of the baseline and only a little slower than the prewarm case, which is acceptable to get better isolation and security.

The core of prewarming containers is reusing the idle but not dead containers, all of their experiment nearly twice as efficient as serverless computing platform of AWS or Azure. However there still exists some deficiencies that can be optimized. For example, we can optimize the container's reservation time according to users' habits, since it would occupy extra resource if in long term idle, on the contrary if the period is too short then it would give rise to cold start. [25]

**Dependencies Initialization** There's a tendency in programming language to become more abstract with a lot of large packages emerging. The strategy of AWS Lambada or Azure is to load the packages to each function instance, but Oakes E et al. raised a package awareness in his article where different instances share a common repository of libraries [30] as fig. 4 shows.
As we know, a package should be downloaded, installed and imported before it could be used. Pipsqueak was proposed as a package caching mechanism in OpenLambda [21] where a cache entry is corresponding to an initialized interpreter process with most necessary packages loaded. Considering the malicious codes, all the interpreters are isolated into different containers and all of them are sleeping before being invoked. The procedure of an interpreter process forking starts with an entry selection, then the chosen process will fork a child in the same container. After a new container initialized, the child process would be migrated to it. Besides reducing the initialization time through forking, another novelty of Pipsqueak is the strategy of the cache which takes advantage of tree management. A child node can import extra packages in addition to the packages from the parent node. To avoid memory waste, copy on write is introduced to speed up forking. Although it performed well in their experiments, it still needs to explore since some languages don't support fork. [30]

![Diagram of Library Support](image)

(a) No Support  (b) Package Awareness

**Fig.4** Library Support

### 4.2. Communication Overhead

**Interaction between Functions** SAND is a new serverless computing system aiming to provide lower latency, better resource efficiency and more elasticity [31]. It pays more attention to application isolation but with coarser granularity because the author wanted to improve the communication speed between functions in a same application.

SAND place different applications in different containers, and all functions in an application run in the same container, this greatly differs from typical serverless computing. When there is an external request, the two-level message queue in SAND will process it as fig. 5 shows. The first-level is a global bus to realize reliability with the condition that when the resources of one host are not enough it would distribute messages to the same application on another host. The second level is a local queue mainly to deal with internal calls between functions in an application, using FIFO strategy, while the order of events in the global queue is determined according to load balancing strategy. From the perspective of resource-saving, it would not adopt warm pool but through process forking.

In SAND, a function of an application is called grain and a workflow defines invoking order between functions, one application can contain many of them. Each application runs independently in an isolated "sandbox" (implemented by a container). A grain can be reused for multiple applications and isolated using sandboxes. Each grain is managed by a specific operating system process called grain worker. When a request coming, the grain worker would fork itself to create a new instance to process it. Besides the feature of lightweight and fast forking process, it could also allow the functions in the same application share the loaded libraries benefited from OS mechanisms.

The result shows that the communication cost is greatly reduced, however this optimization is limited by forking because not all languages support fork. In addition, sharing libraries can improve the communication speed between functions, they are still faced with potential security problem.
Global Data Accessing  

In addition to calls between functions, high frequency access to global data can also cause high overhead. From this purpose, Vodrahalli and Zhou proposed a software-defined cache to reduce communication overhead in a serverless environment [28].

The cache using C/S model is implemented as a hashtable with LRU page replacement algorithm. The interprocess communication is realized with the ZeroMQ [29] that could support a simple client-server model. In this model, functions of a same user are running on the same machine sharing a local cache. Each function is as a client that can access the local cache by putting or getting. When a function needs to get data, the first step is to retrieve it from the local cache. If not obtained, then the cache would get it from the object storage. On the other hand, global broadcast information needs only to be sent to one function in the co-location, while the rest share messages through the cache.

The results show it could increase 10x IOPS and decrease 7x to complete a broadcast instruction while maintaining transparency to users. And the cost of implementation is very low because it is a software with no need to buy extra service. Yet both performance improvements and traffic reductions depend on the availability of large amounts of reused data, only access to long-term objects can reflect the value of caching. In another experiment about numpywren [36], the effect of the cache is not so clear which can be accounted by changing local data, thus the applicability of the software cache should be further explored. [31]

5. Cost Optimization in Platform-Level

Considering the universality of public cloud services as well as the limitations and costs of building a private serverless computing environment, serverless computing platform provided by large vendors will be a good choice.

It is obvious that serverless computing has a great advantage than traditional cloud computing, yet it does not have a unified programming specification, the applicability of platforms of different vendors varies. In [17], the experiment suggested that .Net performs better on Azure than Lambda, whereas Python does the opposite, which suggests us to take our actual requirements into account to choose a cheaper and more efficient platform. In this section we would summarize the difference of four mainstream platforms: AWS Lambda, Azure Functions, Google Cloud Functions and IBM OpenWhisk.

Lynn et al. did a detailed survey on enterprise serverless computing platforms from a preliminary review [18]. The author made a detailed investigation of these various specifications from a perspective of commercialization, comparing these platforms from basic parameters to scenarios. All of these platforms could be triggered by HTTP events, some of vendors that have their own cloud ecosystems could eventialize some of their other services such as Amazon S3, Azure Storage and Google Cloud Storage. On the other hand, the scenarios generally include data processing, micro-service development, edge computing (serverless backend) etc. However, AWS Lambda is the platform with the largest number of cited high profiles users,

followed by Azure Functions and IBM OpenWhisk. What makes these platforms with similar functions differentiate with the acceptability?

Malawski et al. performed a simple CPU performance test on different platforms, including an integer test and a floating point test [37]. We observed some differences in performance and request

Fig.5 SAND’s key building blocks
processing strategies between these platforms. CPU is proportional to memory allocation in GCF and AWS Lambada but not in Azure or OpenWhisk. In addition to performance rating, AWS can achieve over 30GFlops with 3.2 GHz CPU frequency whereas GCF can only reach 18GFlops with 2.1GHz. As for CPU performance testing, Lee et al. compared concurrent function throughput, concurrency for disk intensive workload, concurrency for network intensive workload, elasticity, etc. [32]. The result showed that AWS Lambda was far ahead of other platforms in almost every index. Especially in concurrency including concurrent throughput, concurrent CPU performance, concurrent write/read speed and concurrent download speed. This advantage of several times speed can be attributed to its elasticity. This can also explain that why other platforms can sometimes perform well in the case of 1 concurrent while very bad in the case of 99 concurrent. In the testing of continuous deployment and integration, AWS Lambda and GCF adopt the new-old strategy that old handlers would still work during the code updating, which is great faster than the only new strategy like OpenWhisk. As for supported languages, except GCF the other three platforms all can support 4 kinds of languages and AWS still performs better than others. [18,32]

| Features       | AWS Lambda                  | Azure Functions              |
|----------------|-----------------------------|------------------------------|
| Triggers       | 18 triggers                 | 6 triggers                   |
| Price per Memory| $0.0000166/GB-s             | $0.0000166/GB-s              |
| Price per Execution | $0.2 per 1M               | $0.2 per 1M                  |
| Free Tier      | First 1M Exec               | First 1M Exec                |
| Maximum Memory | 3008MB                      | 1536MB                       |
| Container OS   | Linux                        | Windows NT                   |
| Execution Timeout | 5 minutes                   | 10 minutes                   |
| Code           | 50/250 MB (compressed/ uncompressed) | n/a                          |
| Language       | Node.js, Java, C#, Python   | Node.js, Java, C#, Python    |

| Features       | Google Functions            | IBM OpenWhisk                |
|----------------|------------------------------|------------------------------|
| Triggers       | 3 triggers                   | 3 triggers                   |
| Price per Memory| $0.0000165/GB-s             | $0.000017/GB-s               |
| Price per Execution | $0.4 per 1M               | n/a                          |
| Free Tier      | First 2M Exec               | Free Exec / 40000 GB-s       |
| Maximum Memory | 2048MB                      | 512MB                        |
| Container OS   | Debian GNU/Linux 8          | Alpine Linux                 |
| Execution Timeout | 9 minutes                   | 5 minutes                    |
| Code           | 100MB (compressed) for sources, 500MB(uncompressed) for sources plus modules | 48MB                          |
| Language       | Node.js, Java, Python, swift | Node.js, Python, swift       |

6. Conclusion and Future Work
The goal of this paper is to help with better cost awareness in serverless computing when developing serverless applications or building a serverless platform. We studied related cost optimization
techniques in three layers from top to bottom, then we summarized corresponding methods after bottleneck-analysis.

The cost problem has always been a hot topic in cloud computing. Although serverless technology is just emerging recently, it has a great prospect with its high performance, convenience and low cost. At the end of the article, we suggest the following topics about serverless cost that could be further researched.

Most of related work have been proved serverless is more efficient and economical than traditional cloud services in many situations. But when dealing with some tasks with high-communication overhead [33], it does not perform well and even more expensive than PaaS. One improvement can be with a more optimized policy of function fusion and decomposition to find a way to reduce.

Though many solutions have been proposed to solve the problem of cold start, there are still many shortcomings. The warm-up strategy does not have good isolation and the process fork strategy can only be applied to some programming languages. How to combine the advantages of these two strategies is a worthwhile research direction.

After AWS Lambda, various large vendors have launched their own serverless architecture platform but there still exists no uniform programming specification. It means that once developers intend to change a platform, they need to refactor the previous project, which resulted in high additional migration costs.

Almost vendors have their own cloud ecosystem besides providing serverless computing. Although a large number of studies have shown the advantages of serverless in dealing with high concurrent tasks, considering the actual business (we may need some persistence layer or database services), we can combine the serverless computing with some other cloud services to find a scheme with the lowest cost.

Acknowledgments
This work was funded by the National Key Research and Development Program of China under grant No. 2018YFB1003602k.

References
[1] A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, and I. Stoica. Above the clouds: A berkeley view of cloud computing. Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS, 28(13): 2009, 2009.
[2] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar et al. Cloud programming simplified: a berkeley view on serverless computing. arXiv preprint arXiv:1902.03383, 2019.
[3] B. Varghese and R. Buyya. Future Generation Computer Systems (Part 3), 2018.
[4] “AWS Lambda Developer Guide”. http://docs.aws.amazon.com/lambda/latest/dg/lambda-dg.pdf, 2019.
[5] “Azure Functions Documentation”. https://docs.microsoft.com/en-us/azure/azure-functions/, 2019.
[6] “Google Cloud Functions Documentation”. https://cloud.google.com/functions/, 2019.
[7] “Getting Started with OpenWhisk”. https://console.bluemix.net/docs/openwhisk/index.html, 2019.
[8] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski. The server is dead, long live the server: Rise of serverless computing, overview of current state and future trends in research and industry. arXiv preprint arXiv: 1906.02888, 2019.
[9] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, et al. Serverless computing: Current trends and open problems. In Research Advances in Cloud Computing, pages 1–20. Springer, 2017.
[10] J. Spillner. Practical tooling for serverless computing. In Proceedings of the 10th International Conference on Utility and Cloud Computing, pages 185–186. ACM, 2017.
[11] M. Mao and M. Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud
workflows. In SC’11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–12. IEEE, 2011.

[12] M. Yan, P. Castro, P. Cheng, and V. Ishakian. Building a chatbot with serverless computing. In Proceedings of the 1st International Workshop on Mashups of Things and APIs, page 5. ACM, 2016.

[13] G. Adzic and R. Chatley. Serverless computing: economic and architectural impact. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pages 884–889. ACM, 2017.