Function-as-a-Service Application Service Composition: Implications for a Natural Language Processing Application

Mohammadbagher Fotouhi
School of Engineering and Technology
University of Washington
mfotouhi@uw.edu

Derek Chen
School of Engineering and Technology
University of Washington
dchen14@uw.edu

Wes J. Lloyd
School of Engineering and Technology
University of Washington
wlloyd@uw.edu

ABSTRACT

Serverless computing platforms provide Function-as-a-Service (FaaS) to end users for hosting individual functions known as microservices. In this paper, we describe the deployment of a Natural Language Processing (NLP) application using AWS Lambda. We investigate and study the performance and memory implications of two alternate service compositions. First, we evaluate a switchboard architecture, where a single Lambda deployment package aggregates all of the NLP application functions together into a single package. Second, we consider a service isolation architecture where each NLP function is deployed as a separate FaaS function decomposing the application to run across separate runtime containers. We compared the average runtime and processing throughput of these compositions using different pre-trained network weights to initialize our neural networks to perform inference. Additionally, we varied the workload dataset sizes to evaluate implications of inferencing throughput for our NLP application deployed to a FaaS platform. We found our switchboard composition, that shares FaaS runtime containers for all application tasks, produced a 14.75% runtime performance improvement, and also a 17.3% improvement in NLP processing throughput (samples/second). These results demonstrate the potential for careful application service compositions to provide notable performance improvements and ultimately cost savings for application deployments to serverless FaaS platforms.

CCS CONCEPTS

• **Computer systems organizations** → Cloud computing;
• **Software and its engineering** → Software design tradeoffs;

KEYWORDS

Serverless, Cloud Computing, FaaS, Software architecture, Natural Language Processing

ACM Reference format:

Mohammadbagher Fotouhi, Derek Chen and Wes Lloyd. 2019. Function-as-a-Service Application Service Composition: Implications for a Natural Language Processing Application. In *Proceedings of ACM WoSC'2019 conference (WoSC'19). ACM, Davis, CA, USA, 6 pages.*

1 Introduction

Function-as-a-Service (FaaS) platforms have recently emerged as a new cloud computing delivery model that provides a compelling approach for hosting applications bringing us closer to the idea of instantaneous scalability [2][3][4]. As a leading provider of FaaS, AWS Lambda has grown in popularity based on its ability to automatically and seamlessly execute code based on user events while managing related resources [1][7]. Similarly, deep learning models have risen to the forefront of computer science by achieving state-of-the-art performance for many machine-learning problems, while experiencing exponential growth in the field of natural language processing (NLP). Recently, serverless platforms have been studied for their potential to host machine learning inferencing services [12] [17], and even to train neural networks [13].

In this paper, we investigate microservice composition for hosting an NLP application on the AWS Lambda FaaS platforms. We describe our application implementation in Python, and subsequent deployment to AWS Lambda. We leverage our project as a case study to investigate service composition where the goal is to contrast the performance and memory implications of alternate microservice compositions for our specific application and datasets. Our NLP application has six separate services that perform a series of operations on chat dialogues.

Traditional software engineering best practices encourage developers to minimize coupling while maximizing cohesion among classes or modules of a system. As we enter the era of serverless software, where code composition directly impacts creation and maintenance of ephemeral server infrastructure impacting underlying performance and hosting costs, traditional best practices require reevaluation. The evaluation problem is compounded by the complexity of determining optimal software compositions. Bell's number represents the number of partitions of a set (k) consisting of (n) members [14]. If considering a serverless application given a set of (n) microservices, then the total number of possible microservice compositions is Bell's number (k). Bell's number grows rapidly for (n) microservices (n=3, 4, 5, 6, 7, 8) to produce k compositions (k=5, 15, 52, 203, 877, 4140). This complexity leads developers to frequently make ad hoc deployment decisions due to the difficultly in inferring performance implications given an explosive number of potential service compositions requiring profiling. In these scenarios, brute force testing rapidly becomes intractable from both a time and cost perspective.

1.1 Background

Natural language processing (NLP) is an area of research that explores how computers can be used to understand and manipulate natural language text or speech to perform useful tasks. One particular task within NLP is dialogue modeling, which is usually divided into three components. Consider a scenario where a user is talking to an agent. The agent involved in conversation takes in the raw text as input, and then executes the following three workflow phases:

- Intent Tracking: determines what the user wants. For example, if the user has a question about a subject, or a request to perform a task.
- Policy Management: takes in the user intent, and based on the policy, determines what the chosen agent action will be. For example, given a user intent of a question, the action chosen may be to give an answer, or to ask for clarification about the question.
- Text Generation: Generate the actual text, or retrieve a template to return to the user. For example, if the chosen agent action is to "answer" a question about phone numbers, the text returned is "The number is 571-599-8447."

All three phases include an initialization step where the raw inputs are vectorized into a format recognizable by the network, and an inference step where the network processes the inputs to produce a prediction. In total for dialogue modeling there are 3 network phases with 2 steps (initialization and inferencing) which we map into 6 FaaS microservices.

1.2 Contributions

Leveraging our NLP use case, we investigate the impact of factors surrounding the performance of dialogue modeling implemented using microservices deployed on the AWS Lambda FaaS platform.

We research the effects of FaaS function ***memory reservation size*** on application performance. We explore the implications of the performance to memory size coupling on AWS Lambda. Current best practices for FaaS service composition suggest composing applications as fine-grained sets of fully decomposed lightweight microservices.

We investigate the impact of ***service composition*** on application performance derived from Function-as-a-Service platforms. We explore trade-offs between a fully aggregated, and fully disaggregated service composition for our NLP use case.

We investigate how the adjustment of ***neural network weights*** affects performance of our NLP application deployed on AWS Lambda. Our investigation leverages experiments designed to measure the trade-offs between different configurations.

To save server capacity, cloud providers automatically deprecate serverless infrastructure after periods of inactivity [15]. The recycling of ephemeral server infrastructure on serverless platforms occurs during the ***freeze/thaw lifecycle*** [16]. On AWS Lambda, after approximately 45-minutes of inactivity, original function containers and their host VMs are disassociated from the functions causing FaaS endpoints to go cold (freeze). Consequently, future function calls force initialization (thaw) of new server infrastructure, adding latency to service response times. In this paper, we investigate how the service composition of our NLP application creates or mitigates performance overhead from the freeze/thaw lifecycle of serverless infrastructure.

2 Related Work

With the advent of FaaS platforms, developers and scientists alike have been drawn to leverage their simplicity and elasticity to enable rapid scaling of compute resources to improve application performance. In [9], Lloyd et al. investigated the memory vs. performance tradeoff for migrating a Java-based water supply forecasting modeling application to AWS Lambda. They found hosting costs could be cut in half at 512 MB vs 3008 MB by sacrificing runtime. In [10], Kijak et al. adapted the Deadline-Budget Workflow Scheduling (DBWS) algorithm to support scheduling of scientific workflows on FaaS platforms. They evaluated DBWS for FaaS by scheduling a small-scale 0.25-degree Montage workflow with 43 tasks while experimenting with memory reservation sizes and noted challenges scheduling with heterogeneous resources.

Yan et al. describe the FaaS architecture for an NLP use case, a chatbot deployed to the OpenWhisk FaaS platform in [11]. They described how the serverless model helped improve the extensibility of their chatbot while describing their approach to instrument their microservice workflow with six abilities: location-based weather reports, jokes, date, reminders, and a simple music tutor. Ishakian et al. in [12] evaluated the performance of network inferencing with large pre-trained MxNet neural network models deployed on AWS Lambda. They measured acceptable performance for inferencing with warm serverless infrastructure, but noted significant overhead for inferencing with cold function calls. Bhattacharjee et al. present a distributed and scalable system that forecasts demand and formulates an optimization problem to minimize the total cost of hosting deep learning inferencing services [17]. Their approach provides serverless inferencing by automatically managing container infrastructure across cloud VMs in response to demand. Feng et al. explored the use of FaaS platforms for training large and small neural networks studying challenges involving the use of ephemeral stateless FaaS infrastructure with cold start latency in [13]. They found that FaaS platforms could support hyperparameter tuning for small neural networks as these workloads fit within constraints and could leverage the elasticity provided. Feng also proposed changes to the runtime design of FaaS platforms to better enable future support of deep learning. In this paper, we investigate the utility of FaaS platforms for dialogue modeling as an NLP use case, while studying the performance implications of service composition.

3 Comparison Studies

Each of the three processing stages of our NLP application (intent tracking, policy management, and text generation) include an initialization step, and an inference step that requires significant processing time. Overall our application consists of six total steps decomposed into six functions as described in table 1.

Function ID	Title	Description
F1	Initialize Intent Tracker	Text preprocessing and create sentence embedding
F2	Run Intent Tracker	Load the weights and predict user intent
F3	Initialize Policy Manager	Create action embedding
F4	Run Policy Manager	Load the weights and predict agent action
F5	Initialize Text Generator	Create the generated output embeddings
F6	Run Text Generator	Load the weights and create final text output

Table 1. AWS Lambda Inference Functions

We implement these six inference steps as separate AWS Lambda functions. We then performed two experiments:

1. We varied the type of service composition used to initialize and run the network.
2. We varied the size of the workload being placed on the network.

3.1 Experimental Approach

A neural network can be viewed as a series of non-linear transformations governed by the weights of the network where each layer performs an affine transformation as shown in equation (1).

$$Y = X^T W + b \qquad (1)$$

We anticipate a large time associated with the loading of the weights. Concurrently, there will also be a variable component depending on the size of the inputs, namely the X_0 matrix containing the number of chats to process.

Since there are three processes, there will be exactly three networks to initialize. We are not training a model here, but loading the network weights of a pre-trained model at each initialization step to then run the inferences for each stage of the dialogue pipeline. We note that the weights of the Text Generator (F5/F6) are noticeably smaller by design. The size and complexity of the networks differ dramatically, and we believe initializing the Intent Tracker (F1/F2) will require noticeably more time than initializing the Text Generator.

3.2 Dataset

As input data we leveraged the Multi-Domain Wizard-of-Oz 2.0 dataset (MultiWOZ_2.0), a fully labeled collection of human-human written conversations spanning over multiple domains and topics [6]. MultiWOZ 2.0 provided an existing dataset to provide a set of sample workloads. Client requests were passed in as a JSON file that the inference service parsed.

The MultiWOZ 2.0 dataset describes the communication between a user trying to find a restaurant, and an agent recommending a place to eat. Each sample of this dataset will be a sentence regarding the user asking an agent about restaurants. For example: "I would like to eat some Chinese food on the north side." We can easily partition the data to fit the size we want, which is where the variable workload comes from. Note the explicit goal of a neural network based dialogue agent is to support arbitrary natural language requests within the given domain (i.e. restaurant reservations), so if there is any need, an exponentially large number of new inputs can be easily generated.

3.3 Design Tradeoffs

We investigate and study the performance and memory implications of two different types of service compositions: a switchboard architecture, and full service isolation.

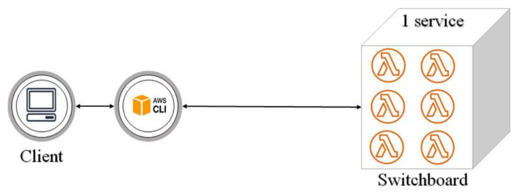

Figure 1: Switchboard Architecture - Asynchronous Flow Control

In the **switchboard composition**, the client initiates the pipeline by calling our fully composed NLP Lambda function that aggregates together our six microservices using a single deployment package. A "switchboard" routine intercepts the call, and routes the client request internally. The advantage here is that when warming cloud infrastructure to run F1, infrastructure for F2, F3, F4, F5, and F6 are also warmed subverting the freeze/thaw lifecycle for the remaining 5-steps of our pipeline. Additionally, this composition provides benefits for hosting requests from concurrent users, as rather than requiring separate runtime containers for each function (F1-F6), all pre-warmed runtime containers can perform any function (F1-F6). **This increases the likelihood of FaaS runtime container reuse while minimizing iterations of the freeze-thaw lifecycle.** Figure 1 depicts our pipeline, where first the intent tracker weights are initialized to run the intent tracker, and then the result is passed on. The weights for the policy manager are then loaded, and the policy manager is run to find a suitable policy. After policy identification, the text generator weights are loaded, and the text generator is run to generate the final results for retrieval by the client.

In contrast, our **service isolation composition** fully decomposes the six inference steps as independent microservices which are then deployed independently. The cloud provider must then provision separate runtime containers to host each microservice. Infrastructure for each service experiences the freeze/thaw serverless lifecycle [8][9]. Ultimately, for both compositions, we are interested in measuring the average runtime and throughput for processing varying dataset sizes.

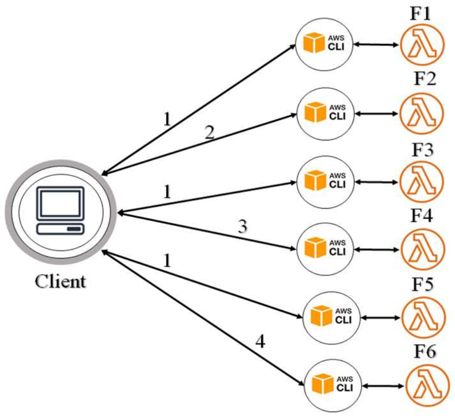

Figure 2: Full-Service Isolation Architecture - Synchronous Client-side Flow Control

In the service isolation architecture, each phase is decomposed as two services: one initialization service, and one inference service, enabling the initialization phases to run in parallel. This composition has the effect of distributing infrastructure used to host the functions across six separate FaaS function runtime environments providing better resource isolation and load balancing when initialization phases run in parallel for concurrent workloads. However, as the number of concurrent queries increases, our switchboard architecture gains the upper hand as the added cost of initializing significantly more runtime containers for service isolation becomes a bottleneck. Here, the switchboard architecture recycles more serverless infrastructure minimizing the cold start initialization time. With the switchboard architecture, every pre-warmed runtime container can perform ***any*** task in our pipeline increasing the chance for infrastructure reuse and retention before infrastructure is automatically reclaimed due to inactivity [8][9].

3.4 Application Implementation

Our neural network model is uploaded onto AWS S3. We used the AWS CLI to submit service requests to the Lambda functions. Our Lambda functions collectively load our NLP model from Amazon S3, provision it, and execute the model locally to generate results. Concretely, provisioning the data and weights from S3 for each of the three networks can cause a relatively large delay.

All software dependencies such as NumPy and Scikit-Learn were identified for inclusion in Lambda FaaS function deployment packages. Dependencies were first identified using AWS EC2's Python Cloud9 IDE, and later downloaded to a local IDE for further development. All dependencies were packaged as a zip file and uploaded for final deployment. Composing all dependencies into a single zip file that fit within the maximum file size constraints of AWS Lambda was critical to deploying our NLP inferencing pipeline.

4 Experimental Results

We performed a series of experiments to investigate the effects of memory reservation size while varying neural network weights to evaluate implications on application performance for our two different service compositions. Additionally, we also varied the size of the input data to test 3, 10, 30, 100, 300, and 1000 samples. Each test was performed 10 times, and we report average values for metrics (e.g. runtime, memory utilization).

4.1 Runtime Performance

We composed all six of our functions together into the switchboard architecture for deployment, and then varied the size of input data to perform a series of experiments to measure runtime. Here, all six initialization and inference calls are performed by the same runtime container on AWS Lambda.

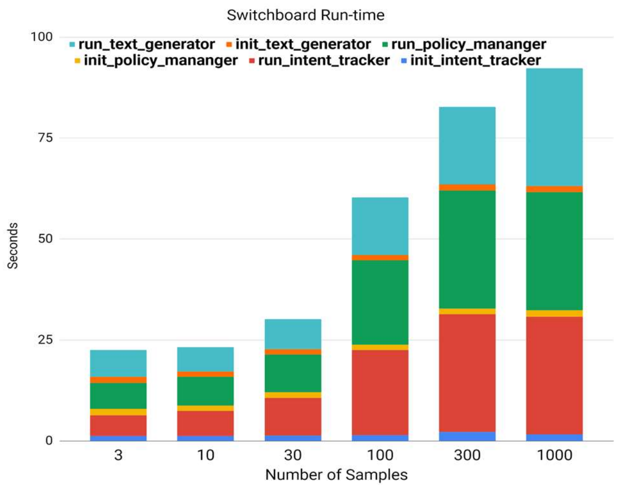

Figure 3. Switchboard Architecture Runtime Performance of FaaS Functions with Increasing Data Sizes

As seen in Figure 3, running the inferences (running the neural network) is the performance bottleneck, and also increasing the data size increases the runtime. Performance ranged from 22.46 sec to process 3 samples, to 92.31 second to process 1,000 samples while throughput (samples/sec) increased ~81x. The Coefficient of Variation (CV), defined as the standard deviation divided by the mean, provides a normalized comparison of performance variance across test configurations. The coefficient of variation (CV) averaged 10.3% when processing 3, 10, and 30 samples, and just 6.7% for processing 100, 300, and 1,000 samples. The Intent tracker initialization time was slower than other initialization phases because it includes the Lambda function cold start initialization time. The other initialization phases are shorted because the application was already in the warm-state when they execute.

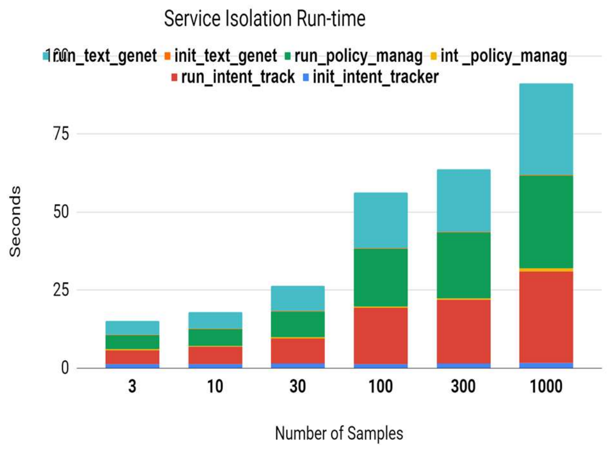

Figure 4. Service Isolation Architecture Runtime Performance of FaaS Functions with Increasing Data Sizes

Our service isolation architecture divides the workload across six different runtime containers on AWS Lambda. As seen in Figure 4, as the data is loaded in parallel in the pipeline for the initialization phases, all have the same performance. In all cases, the performance bottleneck is running the network, not initialization, and the runtime increases with larger input data sizes. Performance ranged from 14.19 seconds to process 3 samples, to 108.29 seconds to process 1,000 samples while throughput (samples/sec) increased ~43x. The coefficient of variation (CV) averaged 12.6% when processing 3, 10, and 30 samples, and just 5.6% for processing 100, 300, and 1,000 samples.

From these observations, one challenge for deploying deep learning NLP is the Lambda memory limitations for processing large sample datasets.

4.2 Memory Utilization

AWS Lambda pricing is based on memory usage. The Free Tier includes 1 million free requests per month and 400,000 GB-seconds of compute time. Pricing above the Free Tier usage is based on memory reservation time, and the total number of function calls, where time is rounded up to the nearest 100-millisecond interval. To evaluate performance, we completed a series of experiments to determine the memory required to run our application for both service compositions while changing the input data size. Figure 5 and Figure 6 depict the memory utilization of the switchboard architecture and service isolation architectures. For both, increasing the number of input samples increased the required memory to run our neural networks. Benchmarking actual application memory utilization helps identify the minimum required memory reservation size to successfully execute the functions on AWS Lambda without error.

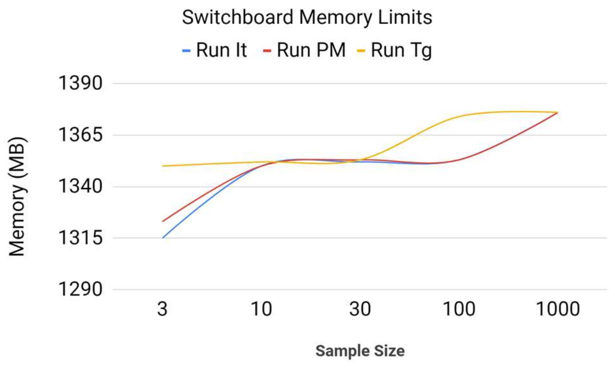

Figure 5. Switchboard Architecture Memory Utilization

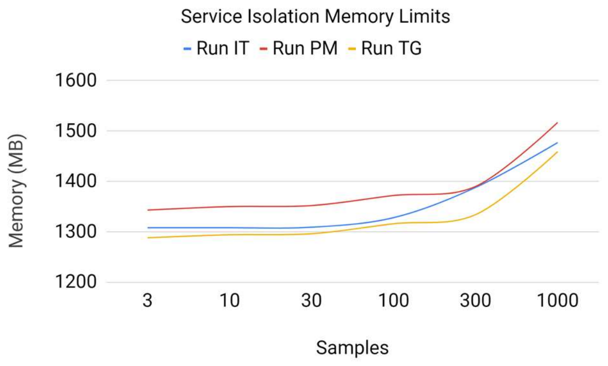

Figure 6. Service Isolation Architecture Memory Utilization

Given less than 100 samples, our two service compositions exhibited different behavior with respect to required memory. Beyond 100 samples, the memory requirement of both compositions increased steadily with the input data size.

4.3 Performance Comparison

We depict the runtime performance comparison of our service compositions in Figure 7. The service isolation architecture is shown to perform more efficiently than the switchboard when the size of the input data is relatively small, but as the input data size grows, the switchboard outperforms the service isolation architecture. The service isolation architecture runtime performance normalized to switchboard performance was 63.2%, 73%, 84%, 91.5%, 94.6%, and 117.3% for 3, 10, 30, 100, 300, and 1,000 sample tests respectively.

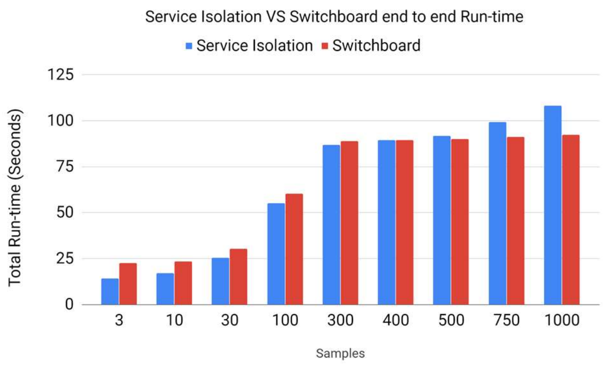

Figure 7. Runtime Performance of Service Isolation and Switchboard Architectures

5 Conclusion

In this paper, we have described the runtime performance and memory limitations of an NLP (Natural Language Processing) application using AWS Lambda using two different service compositions: a switchboard architecture, and a service isolation architecture. By examining memory utilization of networking inferencing, we determined that the main challenge of deploying an NLP application over AWS Lambda was not library size, but runtime memory limitations. We have shown that memory limits increase steadily while increasing the input data size. When comparing runtime performance, we observed that the switchboard architecture minimized cold starts, and performed more efficiently over larger input dataset sizes. With 1,000 samples, the end to end runtime of our NLP pipeline of our switchboard architecture was 14.75% faster than our service isolation architecture. This improvement produced a 17.3% increase in throughput. In contrast, when inferencing just 3 samples, the service isolation architecture was faster (36.96%), while yielding higher throughput (58%).

We used different pre-trained network weights to initialize and run inference in the same fashion as neural networks, and observed how varying the network weights increases the runtime of both service compositions. Additionally, we experimented with varying the dataset sizes to evaluate our pipeline's throughput in samples/second. Ultimately, we found that the switchboard composition helped improve runtime and throughput for neural network inferencing, though inferencing remains compute intensive regardless of the microservice composition used.

ACKNOWLEDGEMENTS

This research is supported by the US National Science Foundation's Advanced Cyberinfrastructure Research Program (OAC-1849970), the NIH grant R01GM126019, and by the AWS Cloud Credits for Research program.

REFERENCES

[1] AWS Lambda – Serverless Compute – Amazon Web Services, https://aws.amazon.com/lambda/

[2] G. Eason, B. Noble, and I. N. Sneddon, On certain integrals of Lipschitz-Hankel type involving products of Bessel functions, Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.

[3] J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[4] I. S. Jacobs and C. P. Bean, Fine particles, thin films, and exchange anisotropy, in Magnetism, vol. III, G. T. Rado, and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[5] ConvNetJS: Deep Learning in your browser, https://cs.stanford.edu/people/karpathy/convnetjs/

[6] MultiWOZ Corpus – Dialog Systems Group, http://dialogue.mi.eng.cam.ac.uk/index.php/corpus/

[7] Mathew S., Varia J., Overview of Amazon Web Services, Amazon Whitepapers., Nov. 2014.

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

[9] Lloyd W., Vu M., Zhang B., David O., Leavesley G., Improving Application Migration to Serverless Computing Platforms: Latency Mitigation with Keep-Alive Workloads. In Proc. of the IEEE/ACM Int. Conference on Utility and Cloud Computing, Workshop on Serverless Computing (WOSC 2018), Dec. 2018, pp. 195-200.

[10] Kijak J., Martyna P., Pawlik M., Balis B., Malawski M., Challenges for Scheduling Scientific Workflows on Cloud Functions. In Proc. of the Int. Conf on Cloud Computing (CLOUD 2018) July 2018, pp. 460-467.

[11] Yan M., Castro P., Cheng P., Ishakian V., Building a chatbot with serverless computing. In Proc. of the 1st Int. Workshop on Mashups of Things and APIs 2016, Dec 2016, 5p.

[12] Ishakian V., Muthusamy V., Slominski A., Serving deep learning models in a serverless platform, In Proceedings of the IEEE Int. Conf. on Cloud Engineering (IC2E 2018), April 2018, pp. 257-262.

[13] Feng L., Kudva P., Da Silva D., Hu J., Exploring serverless computing for neural network training. In Proc. of the IEEE 11th Int. Conf. on Cloud Computing (CLOUD 2018) July 2018, pp. 334-341.

[14] Chen, W., Deng, E., Du, R., Stanley, R., Yan, C., Crossing and Nesting of Matching and Partitions, In Transactions of the American Mathematical Society, vol. 359, No. 4, April 2007, pp. 1555-1575.

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

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

[17] Bhattacharjee, A., Chhokra, A. Kang, A., Sun, H., Gokhale A., and Karsai, G., BARISTA: Efficient and Scalable Serverless Serving System for Deep Learning Prediction Services, in Proceedings of the 2019 IEEE International Conference on Cloud Engineering (IC2E 2019), Prague, Czech Republic, 2019, pp. 23-33.