Device-driven On-demand Deployment of Serverless Computing Functions

Trang Quang and Yang Peng
University of Washington Bothell, Bothell, WA 98011
{quangt3, yangpeng}@uw.edu

***Abstract*—Due to its merit of elasticity and scalability, serverless computing has been accepted as a fundamental technology block when building the edge-cloud back end of IoT systems. Though existing edge computing frameworks such as AWS Greengrass and Azure IoT Edge have enabled the deployment of serverless computing functions to edge, users must manually start the deployment process in cloud. Such a user-driven push-model approach demands heavy user involvement, which makes it less efficient and sometimes slow to react to dynamic changes in IoT environments. Considering these limitations, we propose a novel scheme that enables device-driven on-demand deployment of serverless computing functions to edge. This solution can allow a large number of IoT devices to utilize edge computing resources in a more responsive and scalable manner. Extensive evaluations have been conducted in AWS, and the obtained performance results on both end-to-end and step-wise operation latency demonstrated that the proposed scheme could successfully and efficiently achieve the desired on-demand deployment with minimal overhead in various scenarios.**

I. Introduction

Recent growth in edge computing brings the power of cloud computing to network edges, where advanced data analytics such as machine learning inference can be performed, and intelligent decisions can be returned to nearby IoT devices quickly. When using edge and cloud computing technologies, modular computing jobs (e.g., stream IoT data processing and machine learning inference) can be programmed as serverless computing functions such as AWS Lambda [1] and Azure Functions [2], and execute in lightweight containers on edge or in cloud without special configurations.

Existing edge computing frameworks such as AWS Greengrass [3] and Azure IoT Edge [4] allows deploying serverless computing functions from cloud to edge, but such a deployment process is usually initiated via user's manual operations in cloud. Although this user-driven push-model approach permits full control over cloud and edge resources, it demands heavy user involvement, which makes it inefficient and sometimes slow to react to large-scale, dynamic changes in IoT environments. For many emerging IoT applications, in-time deployment of the needed computing logic – serverless computing functions to a target edge is highly desirable. This requirement is particularly important with the trend of moving AI from cloud to edge.

For example, people who carry cognitive assistance devices [5] may need a nearby edge to help process a large amount of multimedia data and generate safety-critical guidance. The decision-making edge must be equipped with various machine learning models to handle incoming data that varies over time, weather, and other environmental conditions. When numerous devices are using the same edge, the edge shall also execute personalized models. However, the limitation of storage space may not allow having all needed functions pre-loaded in anticipation of a large number of heterogeneous devices, their generated requests, and dynamic operating conditions. In the case of autonomous devices such as surveillance drones, they typically continue moving over a large area, thus possibly traversing through multiple edges. It is difficult to accurately predict which edge these drones are going to interact and seek computing facilitation. Hence, it is hard to schedule and quickly deploy suitable computing tasks to the needed edge by an administrator in a push manner.

Considering the limitation of the existing user-driven push-model approach for deployment of serverless computing functions, we propose a novel scheme that enables device-driven on-demand deployment of serverless computing functions. Besides improving the deployment process, this scheme also allows IoT devices to specify where (on edge or in cloud) to execute a requested function for efficient utilization of edge and cloud resources given different computing requirements. Therefore, such an innovative method can facilitate IoT devices to operate in various situations with the assistance of pervasive computing power. To achieve a responsive, scalable and robust solution, we need to tackle the challenges on state synchronization between edge and cloud, automatic configuration of communication channels, and efficient use of edge/cloud resources during the process of development.

The proposed scheme has been implemented and evaluated in AWS, and the performance results demonstrated that it could successfully provide the proposed device-driven on-demand deployment of serverless computing functions and remote invocation of desired functions with minimal overhead.

II. Design and Implementation

In this section, we present the design and implementation details of the proposed device-driven on-demand deployment scheme.

A. System Overview

The system includes three major components.

- End IoT devices. An IoT device generates data but has limited computing power to process the data. Therefore, some of the generated data must be processed on a connected edge or in cloud. There is no direct communication path between an IoT device and the cloud.
- Edge devices. An edge device connects IoT devices and the cloud, and multiple IoT devices can communicate with the same edge. Various serverless computing functions run in containers on edge to perform requested

data-processing jobs, and a manager program on edge schedules these functions.

- Cloud. The cloud can only communicate with edge nodes directly. The source code of all available serverless computing functions is stored in cloud for deployment.

Fig. 1 illustrates the system's architecture, which aims to leverage the state-of-the-art deployment strategies offered by representative cloud service providers such as AWS and Azure. In this architecture, IoT devices can initiate a new function deployment by sending a request function to some edge node. Upon receiving this request, the edge must first determine if it contains the demanded function and then reaches out to the cloud for a new deployment if none could be found on edge. This design assumes that an IoT device has sufficient information to decide its next course of data processing jobs, hence triggers the deployment of missing processing logic (functions) or invokes them at edge or cloud.

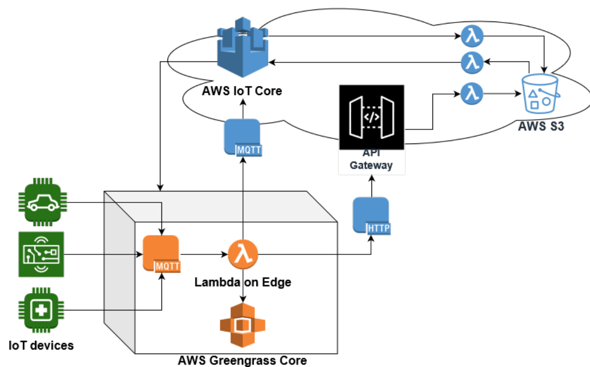

Fig. 1. Overview of system architecture

B. Function Management in Cloud

All functions for on-demand deployment or remote invocation must be stored and registered in cloud at first. To register a function, this function's source code package must be uploaded to a source code repository in cloud (e.g., an S3 bucket). An automated system process is designed to create (for the first time) or update the corresponding function entry. At the same time, metadata about the created/updated function is also generated and saved in cloud for later queries of this function. A system function (to be presented in Section II-C3) can scan the metadata file to determine a function's availability based on its name and version.

C. Function Deployment to Edge

1) Deployment Processes: When an IoT device requests for a function to be deployed to an edge node, the requested function may or may not exist on that edge. Fig. 2 shows the communication sequence between the IoT device, edge, and cloud when a requested function does not exist on edge at the point of request. In this case, the function's availability needs to be checked in cloud, which maintains the latest registry information of all available functions. If the cloud cannot identify the function, a failure message is sent to the requesting device, and the communication ends. Otherwise, the function's definition, together with required invocation methods (e.g., MQTT topics), will be added to the affected edge group's definition. A deployment then can be triggered with the updated definition information.

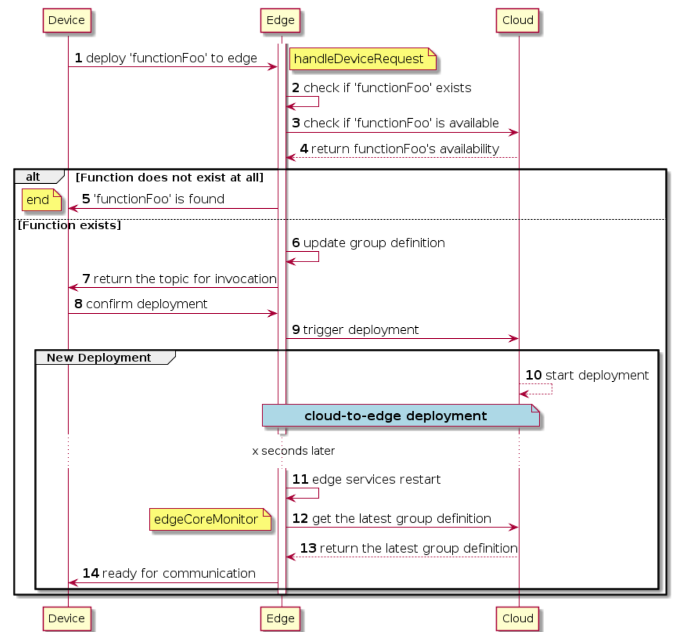

Fig. 2. Deployment workflow for a function unavailable on edge

Fig. 3 illustrates the deployment workflow when a function already exists on edge. In this case, the edge already has the information of the requested function (i.e., its configuration and invocation method), which is recorded in the edge group's definition file stored on edge locally. The invocation method to trigger the requested function can then be returned to the IoT device directly, and regular communication between the requesting IoT device and edge can proceed afterward to use the interested function.

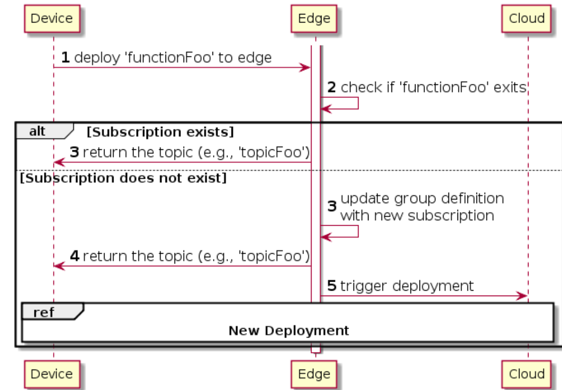

Fig. 3. Deployment workflow for a function existing on edge

2) Edge Group Management: To effectively utilize existing function deployment services provided by AWS Greengrass or Azure IoT Edge, an edge must be aware of its own group's configuration. For each edge group, which includes an edge node (edge core) and several connecting IoT devices, a definition file such as `group.json` for AWS and `deployment.json` for Azure exists in cloud and is used by the corresponding cloud deployment services to manage the deployment process. In the definition file, the list of functions running on edge and how IoT devices, edge, and cloud services communicate are specified. Among the configuration sections

in the definition file, *Function* and *Subscription* are the most important ones for the proposed deployment scheme.

In our design, an edge obtains the needed information about its edge group's definition from cloud after each successful deployment. When a deployment is requested, the edge needs to formulate an updated group definition to include the function under request and other related configuration as clear messages (e.g., steps 6 and 9 in Fig. 2) and send them to cloud. A system function in cloud will handle this request message afterward. Any changes to an edge group's definition, for example, adding a new function or a new topic subscription, results in a new *definition version* of the edge group. A deployment job will be built based on such a version number to help the deployment engine keep track of the correct group configuration effectively.

3) Supporting System Functions for Deployment: For enabling the proposed deployment scheme, seven supporting system functions (implemented as serverless computing functions as well) are developed and deployed to edge and cloud. Along with these functions, event routing is also configured to help trigger corresponding communication between cloud services and these functions correctly. Table I summarizes the locations and triggering events of these functions.

TABLE I
SYSTEM FUNCTIONS SUPPORTING THE DEPLOYMENT PROCESS

location	function name	triggering event
cloud	`createFunction`	S3 put
cloud	`validateFunction`	HTTP request
cloud	`handleDeploymentRequest`	MQTT topic match
cloud	`performDeployment`	S3 put
cloud	`getGroupDefinition`	HTTP request
edge	`edgeCoreMonitor`	MQTT topic match
edge	`handleDeviceRequest`	MQTT topic match

The following five system functions run in cloud.

- `createFunction`. This function is responsible for creating and updating all serverless computing functions accessible to IoT devices. It is triggered by an S3 put event when a function's zip package is saved into a specific bucket. This zip package must contain a configuration file that describes special settings for a function to run in edge or cloud environment properly.
- `validateFunction`. This function checks if a requested function is accessible to an IoT device and executable in a specified location. If the requested function exists and the execution condition is satisfied, `validateFunction` returns a set of detailed configuration information that the edge can use for future communication and execution.
- `handleDeploymentRequest`. This function listens for edge messages on a specific MQTT topic. An edge device requests for deployment by publishing its updated group definition to this topic and `handleDeploymentRequest` writes the updates as a JSON file into an S3 bucket for deployment activities. Such a store-first-action-later strategy is to guarantee the consistency and correctness of the deployment process.
- `performDeployment`. This function is triggered by the S3 put event that is generated by the execution of `handleDeploymentRequest`. A deployment process targeting the edge group specified in the corresponding JSON file will be initiated using cloud service provider's deployment engine.
- `getGroupDefinition`. This function returns an edge group's configuration with all its definitions for a specific group and version.

Fig. 4 shows the complete flow involving the developed supporting functions and AWS Greengrass to trigger a function deployment to edge. This process starts from the point where an updated group definition is sent to cloud (step 1) and completes when it signals AWS Greengrass of a new deployment for the newly created group version (step 4). The actual deployment of function files to edge is controlled and managed by AWS Greengrass's deployment engine.

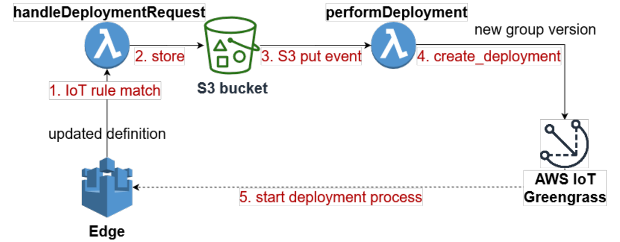

Fig. 4. A deployment flow using AWS Greengrass

In addition to system functions running in cloud, the following two system functions run on edge to support the device-driven on-demand deployment scheme.

- `edgeCoreMonitor`. This is a long-lived function [3], so that a single container can be created and maintained throughout a group version's life-cycle. Its main purpose is to monitor the latest configuration of an edge group and store a local copy of it on edge. Besides, it is responsible for broadcasting at startup to notify all waiting IoT devices that the edge is ready for communication after a new deployment.
- `handleDeviceRequest`. This is an on-demand function [3], multiple container instances of which may be created in response to IoT devices' concurrent requests. This function is the first step for handling deployment and invocation requests from IoT devices.

D. Remote Function Execution

Besides function deployment, our design also enables remote execution of a function, either on edge or in cloud, after its successful deployment.

1) Remote Function Execution on Edge: In the current design, functions on edge can be invoked either asynchronously or synchronously. The asynchronous invocation works for requests that do not need return values, while the synchronous invocation returns the results to requesting IoT devices.

To enable asynchronous invocations, a dedicated subscription (i.e., an MQTT topic, a publisher, and a subscriber) for the requested function must exist per the requirement of either AWS or Azure. This requires updating an edge's group configuration to include the dedicated topic for each newly added function (subscriber) and requesting device (publisher). Differently, a system function `edgeRemoteExecution` acting as a proxy can support generic invocations of other functions on edge synchronously. Compared to asynchronous

invocations, synchronous invocations only need a common topic linked to the proxy function. Instead of updating group configuration for every dedicated topic, this common topic can be set up at an early stage when an IoT device joins the edge group. Therefore, it only incurs a one-time update of an edge group's definition file. Nonetheless, the asynchronous invocation has the benefit of being less resource-demanding compared to the synchronous case, since it only triggers one dedicated function. Besides, it is more suitable to execute long-latency data-intensive processing functions in a decoupled manner and the scenarios where devices may not care about the result.

Fig. 5 presents a closer look at the different handling methods to support the asynchronous and synchronous invocation of a requested function on edge. At the first stage (step 1 – 3), topics for both synchronous and asynchronous invocations are returned to a requesting device if the requested function already exists on edge. Later, when a device invokes the function, the synchronous invocation will return execution results to the requesting device, while the asynchronous invocation will discard the execution results if there are any.

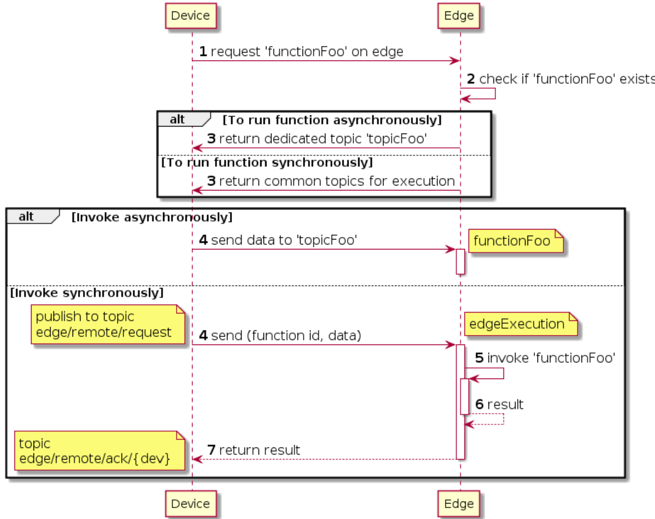

Fig. 5. Invoking a remote function on edge

2) Remote Function Execution in Cloud: When a device requests a function to utilize specific resources in cloud, a communication channel between the device and cloud shall be established. With the assumption that IoT devices would have a better connection with nearby edge nodes, a proxy function runs on edge to forward requests and responses between IoT devices and requested functions in cloud. For the requirement of receiving results from the invoked function, we choose to use HTTP as the triggering method of remote functions in cloud. The reason for selecting HTTP over MQTT is twofold. First, the message exchanging via MQTT requires mandatory two-way subscriptions between the proxy function on edge (i.e., `edgeRemoteExecution`) and the functions in cloud. Besides, the proxy function in cloud must include the additional logic of publishing results back to the response topic on edge when using MQTT. Second, the connection between cloud and edge is more stable than that between edge and device, and MQTT's benefit on tolerance of disconnectivity between communicating entities is less valuable in this case.

3) Supporting System Functions for Remote Function Execution: Two supporting system functions are respectively deployed to edge and cloud as proxy functions to enable the remote execution of functions.

- `edgeRemoteExecution`. This function is configured as an on-demand function running on edge. Upon receiving IoT device's invocation requests through MQTT, it triggers synchronous execution of a requested function, either on edge or in cloud.
- `cloudRemoteExecution`. This function resides in cloud and is triggered by HTTP request events, after which it invokes the requested function synchronously.

III. Performance Evaluation

The proposed scheme has been implemented and evaluated in AWS. Since the design is to support IoT devices' real-time computing needs, we primarily evaluated the proposed scheme's performance on latency. All the results shown in this section were obtained over 10 experiment trials, and the unit is in seconds in all the tables shown in this section.

A. Experiment Setup

Both edge and IoT devices were set up to run on EC2 instances (*t2.micro*) under the availability zone US West (Oregon). This setup is to minimize the network latency between device, edge, and cloud services when evaluating the actual latency overhead introduced by the proposed scheme. The edge operated on AWS Greengrass Core runtime version 1.7.1, and its Lambda functions were created with Core SDK version 1.3.0. IoT devices ran in AWS IoT SDK version 1.3.1 for communicating with the AWS Greengrass Core device.

B. Evaluation Results for Supporting System Functions

We first evaluated the processing latency of each system function running in cloud, and Table II shows the obtained performance results. The maximum and minimum latency values reflect the performance with and without cold-start effects, respectively.

TABLE II
Processing Time of System Functions in Cloud

function name	min	max	avg.	std.
`createFunction`	2.044	2.563	2.189	0.124
`validateFunction`	0.056	0.636	0.328	0.140
`handleDeploymentRequest`	0.226	0.523	0.355	0.058
`performDeployment`	0.675	1.841	1.408	0.241
`getGroupDefinition`	0.468	1.346	0.698	0.176

From Table II, we can find that the processing latency introduced by three of the supporting system functions is within 1 s. However, `createFunction` needs to read and write S3 buckets and register a function using AWS Lambda's framework. As reported in literature [6], the interaction latency with serverless storage typically takes a longer time. Though `createFunction` may take over 2 s to complete, it is only triggered when a user uploads a new function. The latency of `performDeployment` is primarily introduced by AWS's deployment engine, whose performance will be further revealed in the next section.

C. Evaluation Results for The End-to-End Process of Function Deployment

When measuring the latency of the end-to-end deployment process, the following two typical scenarios are considered.

- Case 1. The requested function is available on edge.
- Case 2. The requested function is not yet on edge.

In case 1, the latency includes the device-edge communication latency through MQTT and the invocation latency of `handleDeviceRequest` on edge. Table III shows the evaluation results with and without the effect of cold start. From the result, we can tell that supporting function `handleDeviceRequest`, as the most frequently used function in the system, only introduces less than 10 ms delay when it is invoked in warm-start (i.e., frequent usage).

TABLE III
FUNCTION DEPLOYMENT LATENCY FOR CASE 1

case	# cold-start	min	max	avg.	std.
1	1	0.386	1.040	0.684	0.292
1	0	0.007	0.007	0.007	0.0001

In case 2, the deployment process can be further divided into two stages. In stage-I (step 1–7 in Fig. 2), an IoT device's request triggers `handleDeviceRequest` on edge via MQTT, which further triggers `validateFunction` in cloud via HTTP. In stage-II (step 8–14 in Fig. 2), `handleDeviceRequest` on edge is triggered again when an IoT device confirms its request, and the actual deployment process happens afterward. Table IV shows the evaluation results for these two stages.

TABLE IV
FUNCTION DEPLOYMENT LATENCY FOR CASE 2

case	stage	# cold-start	min	max	avg.	std.
2	I	2	1.316	2.006	1.671	0.226
2	I	1	0.765	1.017	0.886	0.091
2	I	0	0.073	0.270	0.183	0.086
2	II	0	30.409	37.708	32.588	2.400

It is easy to identify the significant latency in stage-II; thus, we conducted a further evaluation and comprehensive analysis of this stage to understand what contributed to the considerable latency value. In the evaluation, we excluded the network latency between device and edge and the processing latency of `handleDeviceRequest` on edge, which is minimal according to the results shown in Table III.

Based on our analysis, the operation in the rest of stage-II can be divided into the following three steps.

- Pre-deployment. This process starts from the time when edge notifies cloud of a new deployment (invoking `handleDeploymentRequest` and `performDeployment`) to the time when AWS Greengrass executes the deployment (calling AWS Greengrass's `create_deployment` API). This process takes about 6.78 s on average, out of which the processing latency of `handleDeploymentRequest` and `performDeployment` is about 1.76 s together.
- Actual-deployment. In this step, edge receives the function files from cloud, configures communication channels on edge, and restarts edge manager and all residing serverless computing functions. This process, which heavily relies on AWS Greengrass's design, takes about 24.21 s on average.
- Post-deployment. After the actual-deployment process, edge retrieves the latest group definition via invoking `getGroupDefinition`, and it takes less than 1 s.

It is thus clear that AWS Greengrass contributes the most to the overall deployment latency (about 30 s on average). Still, our design only introduces minimal additional processing time (less than 3 s on average). Based on this result, it is imperative to consider combining multiple deployment requests within a reasonable time window such that the edge downtime can be reduced, and so does the overall deployment latency.

D. Evaluation Results for The End-to-End Process of Function Invocation

When evaluating the end-to-end latency for invoking functions remotely, we implemented two test functions `edgeTest` and `cloudTest`, which share the same simple logic. This way, the processing latency caused by code logic can be minimal. The following two scenarios are considered.

- Case 3. IoT device invokes a function on edge.
- Case 4. IoT device invokes a function in cloud.

In case 3, the end-to-end latency includes both communication latency through MQTT and the invocation latency of `edgeRemoteExecution` and `edgeTest` on edge. In case 4, the end-to-end latency includes communication latency through MQTT, the invocation latency of `edgeRemoteExecution`, communication latency through HTTP, invocation latency of `cloudRemoteExecution` and `cloudTest` in cloud. From the evaluation results shown in Table V, we can tell that the usage of a proxy function, on edge and in cloud, introduces insignificant additional latency.

TABLE V
FUNCTION INVOCATION LATENCY

case	# cold-start	min	max	avg.	std.
3	2	1.409	1.587	1.526	0.052
3	0	0.011	0.0162	0.012	0.001
4	3	0.774	0.926	0.854	0.050
4	0	0.055	0.150	0.074	0.028

E. Summary of Performance Evaluation

The evaluation results demonstrated that the proposed design introduces minimal delay to the end-to-end function deployment or invocation. However, the utilized AWS services such as S3, Lambda Framework, and deployment engine contributes a major portion of the overall delay. It is thus necessary to explore alternative solutions to further improve the performance of the on-demand deployment process.

IV. RELATED WORK

Recently, serverless computing has drawn considerable attention within both industrial and research communities. Among many applications that can be supported by serverless computing, IoT has been identified as a suitable use case in survey literature on serverless framework [7] and fog computing [8]. Because serverless computing depends on a cloud provider's internal controls, such as load balancing, many research studies have explored various directions aiming to reduce the overall latency when using serverless computing. Among them, there are container or process migration schemes [9], [10], middleware to support orchestration of computing

resources [11], automated management of resource pool and optimal placement of serverless functions [12].

Identified as one of the system level challenges in serverless computing [13], the cold-start problem poses a significant risk in the performance of these serverless systems in latency-sensitive applications. The most straightforward approach to address this problem is keeping the container warm or ready when the event is dispatched for the next invocation. This is illustrated as frequent function pinging suggested [14], in a notable open-source serverless project Zappa [15] or a pool of warm containers as proposed by [16]. There have been many researchers attempting to quantify/evaluate the performance of serverless functions in cold-start vs. warm-start execution [14].

Specifically, to mobile devices and offloading issues, many studies performed an extensive evaluation of stringent latency-constraint applications following the Mobile Edge Computing model [17], [18]. Among these, Cicconetti et al. [19] took an approach similar to our project in the sense of using devices other than edge and cloud to coordinate the serverless function execution. However, their approach emphasized efficient request routing via networking devices operating in different network topologies, while our project focuses on supporting IoT devices' real-time needs in general. Their similarity to our project is the recognition of the limitation when a traditional locally centralized entity could not optimally/responsively dispatch (or deploy) tasks to a set of executors. However, they used networking devices as main drivers of the orchestration process where fast-changing load and network conditions can be gauged/detected on participating edge nodes. Pinto et al. [20] proposed a similar solution where a proxy package, which was distributed at the fog layer (a local network of IoT devices), intercepted requests from IoT devices to determine the best route to forward those requests. The difference is in their design of using the proxy package to decide on where functions should be executed, as compared to our approach of allowing IoT devices to specify the location.

V. Conclusion and Future Work

In this paper, we presented the design and implementation of a device-driven on-demand scheme to deploy serverless computing functions to edge. Compared to the current user-driven push-model deployment method offered by major cloud service providers, this new approach avoids the user's heavy involvement in the deployment process and enables edge and IoT devices to react to dynamic environment changes promptly. It is anticipated that various emerging IoT systems, such as personal cognitive assistance and autonomous drones, can benefit from this scheme to utilize pervasive computing power in real-time needs. Extensive evaluations have been conducted in AWS, and the performance results for both end-to-end and step-wise operation latency demonstrated that the proposed scheme could successfully and efficiently enable the desired on-demand deployment and invocation with minimal overhead in various scenarios.

The current implementation is fully based on the AWS technology stack, which suffers the vendor lock-in problem. In the future, Azure IoT Edge and other Azure technologies can be integrated into the on-demand deployment scheme. An IoT device thus can not only select what functions to deploy and run, but also decide which provider's resources to use for both performance and data security considerations. Furthermore, we will also investigate strategies that can aggregate multiple deployment requests arriving at the same time window and thus save the deployment latency on average. Another important direction for future work is to avoid the service interruption when deploying new functions, and we plan to develop new schemes for swapping services between edges and/or clouds.

References

[1] AWS, "What Is AWS Lambda? - AWS Lambda," https://docs.aws.amazon.com/lambda/latest/dg/welcome.html, (Accessed: 10-15-2019).

[2] Micorsoft, "Azure functions documentation," https://docs.microsoft.com/en-us/azure/azure-functions, (Accessed: 10-15-2019).

[3] AWS, "What Is AWS Greengrass? - AWS Greengrass," https://docs.aws.amazon.com/greengrass/latest/developerguide/what-is-gg.html, (Accessed: 10-15-2019).

[4] Microsoft, "What is Azure IoT Edge," https://docs.microsoft.com/en-us/azure/iot-edge/about-iot-edge, (Accessed: 10-15-2019).

[5] J. Pan and J. McElhannon, "Future edge cloud and edge computing for internet of things applications," *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 439–449, Feb 2018.

[6] E. Jonas, J. Schleier-Smith, V. Sreekanti, C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. J. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud programming simplified: A berkeley view on serverless computing," *arXiv:1902.03383 [CoRR]*, 2019.

[7] H. Lee, K. Satyam, and G. Fox, "Evaluation of Production Serverless Computing Environments," in *IEEE 11th International Conference on Cloud Computing*, 2018, pp. 442–450.

[8] L. Bittencourt, R. Immich, R. Sakellariou, N. Fonseca, E. Madeira, M. Curado, L. Villas, L. DaSilva, C. Lee, and O. Rana, "The internet of things, fog and cloud continuum: Integration and challenges," *Elsevier Internet of Things*, vol. 3-4, pp. 134 – 155, 2018.

[9] M. Horii, Y. Kojima, and K. Fukuda, "Stateful process migration for edge computing applications," in *IEEE Wireless Communications and Networking Conference*, 2018, pp. 1–6.

[10] P. Karhula, J. Janak, and H. Schulzrinne, "Checkpointing and Migration of IoT Edge Functions," in *ACM 2nd International Workshop on Edge Systems, Analytics and Networking*, 2019, pp. 60–65.

[11] M. Alam, J. Rufino, J. Ferreira, S. H. Ahmed, N. Shah, and Y. Chen, "Orchestration of Microservices for IoT Using Docker and Edge Computing," *IEEE Communications Magazine*, vol. 56, no. 9, pp. 118–123, Sep. 2018.

[12] S. Nastic, T. Rausch, O. Scekic, S. Dustdar, M. Gusev, B. Koteska, M. Kostoska, B. Jakimovski, S. Ristov, and R. Prodan, "A Serverless Real-Time Data Analytics Platform for Edge Computing," *IEEE Internet Computing*, vol. 21, no. 4, pp. 64–71, 2017.

[13] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, "Serverless computing: Current trends and open problems," in *Research Advances in Cloud Computing*, S. Chaudhary, G. Somani, and R. Buyya, Eds. Springer Singapore, Dec. 2017, pp. 1–20.

[14] D. Bardsley, L. Ryan, and J. Howard, "Serverless Performance and Optimization Strategies," in *IEEE International Conference on Smart Cloud*, 2018, pp. 19–26.

[15] R. Jones, "Serverless Python. Contribute to Miserlou/Zappa development by creating an account on GitHub," https://github.com/Miserlou/Zappa, (Accessed: 10-15-2019).

[16] P.-M. Lin and A. Glikson, "Mitigating Cold Starts in Serverless Platforms: A Pool-Based Approach," *arXiv:1903.12221 [cs]*, 2019.

[17] S. Maheshwari, D. Raychaudhuri, I. Seskar, and F. Bronzino, "Scalability and Performance Evaluation of Edge Cloud Systems for Latency Constrained Applications," in *IEEE/ACM Symposium on Edge Computing*, 2018, pp. 286–299.

[18] X. Zhang, H. Huang, H. Yin, D. O. Wu, G. Min, and Z. Ma, "Resource provisioning in the edge for iot applications with multilevel services," *IEEE Internet of Things Journal*, vol. 6, no. 3, pp. 4262–4271, 2019.

[19] C. Cicconetti, M. Conti, and A. Passarella, "An Architectural Framework for Serverless Edge Computing: Design and Emulation Tools," in *IEEE International Conference on Cloud Computing Technology and Science*, 2018, pp. 48–55.

[20] D. Pinto, J. P. Dias, and H. S. Ferreira, "Dynamic Allocation of Serverless Functions in IoT Environments," in *IEEE 16th International Conference on Embedded and Ubiquitous Computing*, 2018, pp. 1–8.