2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)

EdgeBench: Benchmarking Edge Computing Platforms

Anirban Das, Stacy Patterson, Mike P. Wittie

***Abstract*— The emerging trend of edge computing has led several cloud providers to release their own platforms for performing computation at the 'edge' of the network. We compare two such platforms, Amazon AWS Greengrass and Microsoft Azure IoT Edge, using a new benchmark comprising a suite of performance metrics. We also compare the performance of the edge frameworks to cloud-only implementations available in their respective cloud ecosystems. Amazon AWS Greengrass and Azure IoT Edge use different underlying technologies, edge Lambda functions vs. containers, and so we also elaborate on platform features available to developers. Our study shows that both of these edge platforms provide comparable performance, which nevertheless differs in important ways for key types of workloads used in edge applications. Finally, we discuss several current issues and challenges we faced in deploying these platforms.**

I. INTRODUCTION

As the Internet of Things (IoT) is becoming mainstream, the number of connected devices is growing at an exponential rate [1]. In this paradigm, IoT devices, which are often geographically distributed at the edge of the network, will generate a massive quantity of data. Transmitting, storing, and processing this huge amount of data in the cloud is expected to lead to high bandwidth usage and prohibitive costs [2]. Further, many applications that run at the edge of the network, such as autonomous vehicles and augmented reality, have real-time requirements that are difficult to meet with relatively distant cloud datacenters [3].

Edge computing has the potential to mitigate these cost and performance bottlenecks. This computing paradigm enables applications to leverage compute nodes in close proximity to data sources to perform data processing, such as aggregation, filtering, and classification, before forwarding the results to other nodes and cloud servers [2], [4]. For example, instead of sending an image to the cloud to perform facial recognition, and edge device may simply report whether the image contains the particular signature. This approach reduces bandwidth usage and can speed up application response.

To make application development in the edge computing model easier, several cloud providers, have put forward their own edge computing platforms. These platforms provide the ability to deploy and orchestrate applications, such as machine learning models, on edge devices in the form of stateless serverless functions or user code in containers. The resource provisioning and runtime is provided by the edge platforms. Such serverless functions or user scripts in containers then act on raw data, and depending on the configuration, send results to the cloud and additionally also make them available in other cloud services.

Since these platforms use different paradigms and technologies, it is important to compare them to understand their tradeoffs and to select the best platform for a given use case. Criteria of interest include the platform architecture, programmability, performance, and cost. To quantify these differences consistently requires benchmarks of common uses cases. Further, it requires standardized collection of metrics, such as end-to-end latency, device compute time, device resource utilization, bandwidth usage, and cost. To make an informed choice between edge and cloud platforms, the benchmarks and metrics must also allow fair comparison across different types of deployments.

We present EdgeBench [1] – an open-source benchmark suite for serverless edge computing platforms. EdgeBench features three key applications: a speech/audio-to-text decoder, an image recognition machine learning model, and a scalar value generator emulating a sensor. Each application processes a bank of input data on an edge device and sends results to cloud storage. We target EdgeBench for two of the most popular edge computing platforms currently available, AWS Greengrass [5] and Microsoft Azure IoT Edge [6]. EdgeBench also provides cloud-based workload implementations. Our aim is to quantify the differences between the different edge platforms, as well as the providers' respective cloud-only alternatives. In future work, we plan to extend EdgeBench to other emerging edge platforms, such as Google's Cloud IoT Edge [7] and IBM Watson IoT Platform Edge [8], as these offerings mature. We report on initial experiments with EdgeBench using a Raspberry Pi 3B, a relatively resource-constrained device, to emulate the IoT device that sends traffic to the AWS and Azure cloud platforms. We provide a performance comparison across the different workloads and platforms.

EdgeBench complements previous work on benchmarking serverless cloud computing platforms. Malawski et al. have developed two CPU-intensive benchmark suites and evaluated them on the different industry providers [9]. The recent work by McGrath and Brenner presents a comparison of the overhead of various platforms, measured using a custom-developed tool [10]. The work by Back and Andrikopoulos presents a performance study of industry serverless cloud

Anirban Das and Stacy Patterson are with the Department of Computer Science, Rensselaer Polytechnic Institute, Troy, New York 12180, USA. dasa2@rpi.edu, sep@cs.rpi.edu

Mike P. Wittie is with the Gianforte School of Computing at Montana State University, Bozeman, Montana 59715, USA. mwittie@cs.montana.edu

This work was funded in part by NSF awards CNS-1527287, CNS-1553340, CNS-1555591, and CNS-1527097

[1] https://github.com/akaanirban/edgebench

978-1-7281-0359-4/18/$31.00 ©2018 IEEE
DOI 10.1109/UCC-Companion.2018.00053

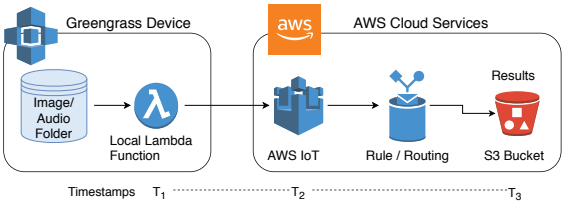


Fig. 1: Amazon Greengrass Architecture.

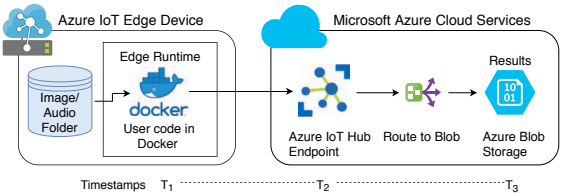


Fig. 2: Azure IOT Edge Architecture.

platforms using compute/memory constrained workloads, with a focus on cost [11]. Finally, Deese presents a study of the performance of K-means clustering using AWS Lambda functions in the cloud [12]. To the best of our knowledge, our work is the first that benchmarks industry platforms that use the serverless paradigm for edge computing.

The rest of the paper is organized as follows. Sec. II provides details about the architectures of AWS Greengrass and Azure IoT Edge. Sec.III describes the EdgeBench workloads and metrics. In Sec. IV, we describe the experimental setup and results of the benchmark study and discuss some observations, and finally, we conclude in Sec. V.

II. SYSTEM ARCHITECTURES

On an abstract level, both platforms, AWS Greengrass (henceforth, Greengrass) and Azure IoT Edge (henceforth, Azure Edge) share a common general architecture. There is an edge device that runs user code in the platform's runtime system. This user code can access local volumes or local devices, such as sensors and cameras, performs computations, and sends messages to the cloud. In both platforms, the cloud has a high throughput message ingestion service, the IoT Hub. The cloud ingests the messages from the edge devices and sends them to configurable destinations, such as AWS S3 or Azure blob for storage using 'Rule' (for AWS) or a 'Route' (for Azure).

Below, we highlight some of the details of each platform:

A. AWS Greengrass

The Greengrass pipeline is shown in Fig. 1. Greengrass edge devices run the Greengrass core software. The core software allows users to run Lambda functions locally on the edge devices and manage, modify or update them through the AWS console website or deployment API. Developers can constrain the maximum memory usage of the local Lambda functions. The Greengrass core software also takes care of the authentication, authorization, and secure message routing, through the MQTT protocol [13], between the devices, Lambda functions, and the cloud.

Greengrass core Lambda runtime currently supports code deployment in Python 2.7, Node.JS 6.10, Java 8, C, C++, and any language that supports importing C libraries. Code running inside Lambda functions can also access all other AWS services, such as Amazon S3 or DynamoDB, using the standard AWS SDKs. Once the AWS IoT Hub receives a message in the cloud, a 'Rule' can be defined to trigger one of 15 actions (as of now), including invoking Lambda functions that run in the cloud or saving data in S3 or DynamoDB. If the 'Rule' declares to save messages in S3 storage, the hub does so by creating one blob file for each message in the specified S3 bucket, as soon as the message is processed by the hub.

B. Microsoft Azure IoT Edge

Azure Edge uses lightweight virtualization, specifically, Docker compatible containers, to deploy computation on edge devices. The Docker containers run as 'edge modules' in the Azure Edge platform, as shown in Fig. 2. The modules can contain Azure Functions, user code, and libraries. As of now, the platform supports five languages: C#, C, Node.JS ver $>$ 0.4.x.x, Python (both 2.7 and 3.6), and Java 7+. It is also possible to deploy Streaming Analytics and Azure ML models directly in the containers. The former is a managed service from Azure for doing analytics on streaming data and the latter are the models developed in Azure's machine learning service. Modules can be deployed, updated, and modified via the Azure IoT Edge Cloud web interface or the Azure command line interface.

The runtime system on a single edge device consists of an `edgeAgent` module and an `edgeHub` module. The `edgeAgent` takes care of provisioning and monitoring user deployed modules. The `edgeHub` takes care of the connection between the modules and the cloud and also maintains security and authentication. The `edgeHub` supports edge-to-cloud connections using the MQTT and AMQP protocols The edgeAgent sends messages to the cloud-hosted Azure IoT Hub, as shown in Fig. 2. Messages are then routed from the IoT Hub to a user-specified IoT Hub Endpoint, such as Azure Blob Storage. For the Blob Storage Endpoint, the IoT Hub batches the incoming messages and writes multiple results in a single blob file. If user selects Blob Storage endpoint, batching is the only option. The batching window can be configured by either time window, the smallest being 60 s, or by chunk size, the smallest being 10 MB.

III. EDGEBENCH

In this section, we describe EdgeBench benchmark suite and the performance metrics. We also summarize the cloud-based implementations of the benchmark applications.

A. Benchmark Applications and Workloads

We selected three canonical applications: a speech/audio-to-text application; an image recognition application; and a scalar value generator that emulates a sensor, e.g. a temperature sensor. With the popularity hike in use of a myriad of smart speakers, such as Amazon Echo and Google Home, has made audio and speech decoding and translation

very relevant. Similarly, with the proliferation of smart cameras and autonomous vehicles, image processing and image classification has become very common. Currently, these applications are often executed in the cloud. Hence, it is interesting to investigate performance of such applications in an edge computing setting. The scalar benchmark, however, is an example of an extremely lightweight workload; it allows us to measure the performance of each framework when the computation and data volume at the edge are negligible.

All benchmark codes are written in Python. In all three pipelines, the edge devices send data in messages to the IoT Hub We use a either a 'Rule' or 'Route' to push each message payload in the cloud to an AWS S3 bucket or an Azure Blob, respectively.

- **Audio/Speech to Text Translation (Audio Pipeline)** Here, the edge application reads audio files from a local directory, decodes them to get the translated text, and then sends the text to the cloud. Each audio file is processed one at a time. For our experiments, we use a mobile version of Carnegie Mellon University's Sphinx speech recognition system, called PocketShpinx [14]. We use the default acoustic model provided with the package. For the audio workload, we use 104 samples contributed by user '**rhys_mcg**' in Tatoeba Corpus [15], a free collaborative online database of example sentences. We have converted the audio files into 16khz, 16 bit, mono 'wav' file format to comply with the requirements of PocketSphinx. The average realtime length of files in the dataset is about 2.4 s.
- **Image Recognition (Image Pipeline):** The serverless function performs an image recognition task; specifically, given an image as input, the function recognize the objects present in the image and generates class labels for these objects. In both platforms, the edge application reads an image from a directory. It then uses OpenCV [16] to resize the image to standard ($224 \times 224 \times 3$) size. Finally, the application uses the open-source, deep learning framework MXNet [17] to recognize and classify the objects in that image. The results are sent to the cloud. This is repeated for all images in the directory. For the classifier architecture, we chose a pretrained classifier, Squeezenet [18] because its small model size ($\approx$ 5 MB) and low compute footprint are suitable for resource-constrained edge devices. For the input workload, we select 500 images from the ILSVRC2012 image dataset [19]. The input is stored on a local directory on the device.
- **Scalar Sensor (Scalar Pipeline):** The application is a simple sensor emulator. The serverless function generates random scalar values at a user-specified frequency. At a user-specified interval, e.g., 1s, the set of values generated in that interval is sent to the cloud in a single message. The cloud side of the pipeline then simply stores these values in the specified storage location.

B. Metrics

In all pipelines of both platforms, each message receives three UTC timestamps during the pipeline execution, as shown in Fig. 1 and 2, from which we calculate delay

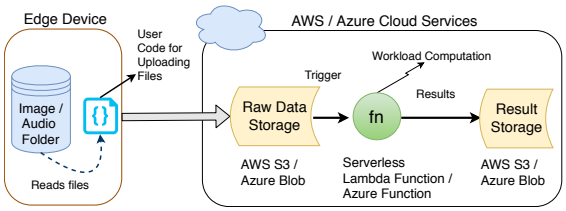


Fig. 3: Schematic of a cloud only pipeline using AWS/Azure

metrics. Here, for valid calculations on these metrics, we need the time of both the edge device and cloud to be independently synced with accurate clocks. The user code in the edge device adds the T_1 timestamp before sending the result message from the edge device. Timestamp T_2 is added automatically by the platform when the message is en-queued in the IoT Hub, and finally, T_3 is the creation timestamp of the blob file in which the message is stored after it is routed out of the IoT Hub.

We capture the following metrics to study performance:

- **Compute time:** This is the total time required for processing one image or audio or to generate the scalar values in the Raspberry Pi and is denoted by C_{edge}.
- **Time-in-flight:** This is the time taken for a message to reach the IoT Hub after it is sent from the edge device. It is given by $T_2 - T_1$.
- **End-to-end latency:** This is the difference between the time when the input is ingested at the edge device and the time when the final results are available in the storage. This value is given by $C_{edge} + (T_3 - T_1)$.
- **Payload size**: This is the size of the message sent from the edge device without the framework overhead.
- **CPU and memory utilization**: As the platforms use different architectures, i.e., Lambda vs Docker, it is interesting to look at the memory and CPU usage patterns. We measure the average CPU and memory utilization on the edge device over the execution of a given benchmark. For AWS, we use the `top` command in Linux and for Azure we use the `docker stats` command to obtain the resource utilization while the applications are running.

Our applications log compute time, the payload size, and resource utilization locally on the edge device, while T_1 is added to the header. Therefore, T_1, T_2 and T_3 are retrieved from the message meta-data in the cloud.

C. Cloud-Only Pipelines

We implement cloud-only versions of the three benchmark workloads described in Sec. III-A for both the Amazon and Microsoft Azure cloud platforms. The pipelines use the serverless architecture, as shown in Fig. 3, to process device data. All code is written in Python.

For the image classification and speech-to-text benchmarks, we upload either image or audio file from the edge device to a S3 bucket using `boto3` library. For the scalar pipeline in Amazon AWS, we generate and upload the sensor values as JSON blob files in S3. Lambda functions are triggered by the creation of the new blob file in the bucket. The Lambda function reads the value from the blob file and

simply stores it in another S3 bucket. The upload of a file triggers the Lambda function which, in turn, either performs the image recognition or the audio-to-text conversion. After the computation, the results are stored as blobs in another S3 bucket. In the Azure implementations, we use Azure Functions, which are similar to Lambda functions. The Azure functions are triggered by the upload of an audio or image file or a scalar value JSON file in Azure blob storage. After the computation, the results are stored in a different blob. The majority of the code in cloud and edge pipelines are the same, excepting changes for handling input/output and receiving and handling the events due to different API specifications.

Note that in our benchmarks, Azure functions run on Windows while Lambda functions run on Amazon Linux. Linux is available on a preview basis for Azure Functions, but it supports only JavaScript and .Net runtime as of now. Moreover, even in Windows in Azure Functions, the support of Python is in the experimental phase. We manually installed MXNet, OpenCV, PocketSphinx and other necessary libraries from the KUDU console in Azure Python runtime to use the libraries in Azure Functions.In AWS, the dependencies are packaged along with the Lambda function.

IV. EXPERIMENTS

A. Experimental Setup

We run the set of benchmarks using a standard Raspberry Pi 3B model as the edge device. The edge device is connected to the internet via a wireless router using 2.4 Ghz spectrum. We have used a dedicated Stratum 1 NTP time server, TM2000A [20] with accuracy $\approx$ 50 μs to synchronize the time of the Raspberry Pi. Also, AWS and Azure are known to use highly precise clocks to accurately sync their services. We use AWS and Azure cloud services in the US East region, both in Virginia. We used `ping` to find the round trip latency from our institution's server to virtual machines in both Azure and AWS. We are unable to measure this from the edge device, due to security restrictions on `ping`. The average delay for AWS is 9.5 ms and for Azure is 11.36 ms. Assuming the extra delay within the institution network is same for both Greengrass and Azure Edge, we observe latencies to both the cloud platforms are close.

We use Greengrass core version 1.5.0 and Azure IoT Hub Device client 1.4.0. In the experiments with Greengrass, each Lambda function is provisioned with 256 MB RAM and made 'long lived', i.e., it will run indefinitely. However, this option is absent in Azure Edge. In Greengrass, the image, audio and local statistics directories for storing metric values are mounted into the execution environment as 'Local Resources'. In Azure Edge, the same directories are directly mounted as volumes in the Docker container. In Azure Edge, we use the geo-redundant storage option RA-GRS for blob storage in the cloud. AWS S3 replicates data automatically across at least 3 availability zones.

We also measure performance of the cloud-only pipelines in AWS and Azure. For AWS, the memory of the Lambda functions is set to 3008 MB. In AWS, CPU allocation is proportional to the memory, hence, this configuration has the highest memory and CPU performance. As Lambda CPU is not configurable, to keep the setups comparable, we select the Consumption Host Plan for Azure functions that auto-scales Azure functions based on system load. In all cloud pipelines, we wait for a period of 10 to 15 s between uploading subsequent image/audio files to avoid congesting the system. Uploading too many image/audio files very quickly resulted in many functions being triggered concurrently and out of order. This results in reordering of results and some missed images, making it difficult to find the end-to-end latency. The input data sizes are shown in Table I.

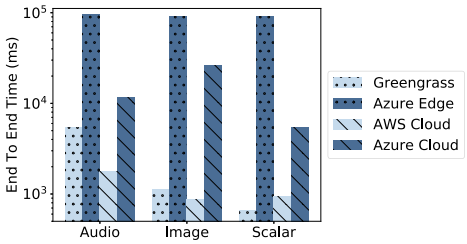


Fig. 4: Average end-to-end latency in the edge and cloud-only pipelines for all benchmark applications.

B. Results

1) End to End Latency: We give the average end-to-end latency for each of the various benchmark configurations in Fig. 4. We observe that in all three applications, across both the cloud and edge pipelines, Azure Edge has the largest end-to-end latency. This is because the Azure IoT service batches the messages from the edge device in the IoT Hub in the cloud, and it writes the results from the multiple messages in a batch in a single blob file. We also found that when the batching interval is 60 s, the average time spend in hub is $\approx$ 90 s, while for a 90 s batching interval, the average time in the hub is $\approx$ 93-94 s. It appears that messages are held back in the IoT hub in Azure for some time interval before being written to the blob file, and this time does not coincide with the batching interval. If this were not the case, given that the messages are received by the IoT Hub approximately uniformly across a batching interval, the average time a message spends in hub should have been roughly equal to half the batching interval. This extra delay, adding to latency, exists irrespective of the blob storage type used.

We observe that the end-to-end latency for Azure cloud pipelines is larger than both AWS edge and cloud pipelines across all applications. The majority of the latency is caused by the total time of execution of Azure function in the cloud. Though, the average time for audio to speech and image recognition in Azure cloud is 5.57 s and 1.19 s respectively, for each message, added to this is the time for loading libraries and trigger the function, which is very high for Azure python runtime. This may have been be caused by the fact that the Python runtime in Azure is still experimental and hence, not optimized. It may also be that importing the libraries takes a lot of time. We obtain the smallest end-to-end latency results using the AWS Cloud (1.79 s for

		Total Input Size (Mbytes)	Total raw Payload Size (Mbytes)	Total MBytes Transmitted in Network	
				AWS	Azure
Audio Trials = 104	Edge	8.83	0.02	0.25	0.26
	Cloud		8.83	9.06	9.09
Image Trials = 500	Edge	71.69	0.38	0.9	0.96
	Cloud		71.69	73.10	73.49
Scalar Trials = 200	Edge	0.05	0.05	0.33	0.26
	Cloud			0.47	0.38

TABLE I: Total input, payload, and actual data transmitted in edge and cloud only pipelines for the three benchmark applications along with number of trials.

Audio, 0.87 s for Image, and 0.936 s for Scalar), followed by Greengrass (5.36 s for Audio, 1.1 s for Image and 0.66 s for Scalar). It appears that image processing at the edge with Greengrass is highly feasible, as both cloud and edge end-to-end latencies are very close.

In Fig. 5a we observe, Azure takes on average from 1-18 ms longer to deliver the messages to the cloud for the edge pipelines compared to AWS. The flight times of AWS and Azure edge are very close, which suggests that as long as the user selects data centers with similar latencies, the flight time will not contribute much to the difference of end-to-end message latencies of AWS compared to Azure.

Finally, Fig. 5b shows that Azure, in general, has a higher compute time for all pipelines compared to Greengrass. The highest is the audio pipeline, with Azure Edge taking 6 s, on average, and Greengrass taking 4.77 s, on average. This difference may indicate a place where the different architectures (Lambda vs. Docker) may have made a difference. If this time is large, then it has a significant impact on end-to-end latency. We also observe that, for the audio pipeline in the edge, considering the average length of each clip is approximately 2.4 s, it may not be possible to analyze the audio in real time using a Raspberry Pi without optimizing the code. However, using a more powerful edge device would help to reduce compute time.

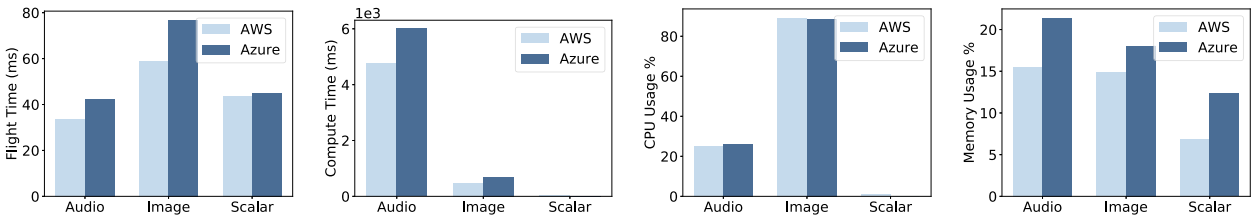
2) Bandwidth Utilization: The average payload size in both platforms for audio pipeline is 162 bytes, for image pipeline it is 752 bytes and for scalar it is 234 bytes. On comparing flight times for each edge pipeline in Fig. 5a we observe that the transmission delay (flight time) of messages between the edge and cloud is roughly proportional to the message size. A cloud-only approach requires the upload of the raw image or audio file to the cloud service. In the edge pipelines, we only the send results of the applications as text to the cloud. Hence, we observe that there is drastic reduction in the per message size in edge pipelines compared to cloud. We also used `vnstat` [21] to obtain the total bandwidth usage of the applications in the edge and cloud pipelines, shown in Table I. The results are measured with respect to 200 scalar values, 500 images, and 104 audio files, respectively. To avoid measuring the TLS handshakes and other module startup network overhead, we explicitly add a configurable delay (60 s, in this case) before the application begins processing data. We see that framework data overhead itself is negligible and comparable in both platforms. Comparing the total data transmitted during pipeline executions, we see a massive reduction of data transmission while using the edge pipelines compared to cloud. AWS sends 36 times and 81 times more data when using the cloud pipelines compared to the edge, for the audio and image applications, respectively. Azure sends 36 times and 77 times more data using the cloud pipelines compared to the edge in audio and image applications, respectively.

3) Local Resource Utilization: We study the average resource usage of the edge device (Raspberry Pi) across the edge pipelines over three separate runs of the experiments. In Azure Edge, we look at the average total CPU and memory percentage used by all of the containers running specific to Azure Edge. In Greengrass, we look at the total average CPU and memory percentage usage by all processes under the greengrass user, `ggc_user`.

We observe in Fig. 5c, 5d that the image recognition application is predominantly a CPU intensive job, with CPU utilization as high as 88-90% in both Azure and AWS. We also observe that audio-to-text is not very CPU-heavy, but it consumes more memory than the other applications. We further observe that within each application, the CPU % of Greengrass and Azure Edge are very similar, though the RAM consumed in Azure Edge is always higher. Azure Edge consumes on average about 29.5 MB to 54.5 MB more memory on the Raspberry Pi. We believe this difference would be less discernible if a more powerful edge device were used. Overall, we observe that the edge pipelines, on average, consume less than 200 MB RAM and do not saturate the CPU usage. This strengthens the case for the feasibility of running some carefully chosen computations on resource-constrained devices.

4) Infrastructure Cost: We do a rough infrastructure cost estimate of running the applications in the edge pipelines versus the cloud-only pipelines. Cost of running pipelines in both vendors are comparable and so, for simplicity, here, we look only at the image pipeline in AWS. Assume there is one traffic camera, generating one image every 10 s. Let us further assume the average image size is similar those used in our image benchmarks, i.e., 143.12 KB (In general, image size and rate would be larger in real world scenarios.) This amounts to $6 \times 60 \times 24 \times 30 = 259,200$ images per month. Also, on average, the duration for which AWS charges for Lambda function execution is 300ms in the image recognition cloud only pipeline in our study for image of that size. In the Greengrass image pipeline, the average size of a message to the cloud is 752 bytes. We assume, with headers, it would be approximately 1 KB per message. All prices are calculated in region US-East, Virginia.

For Greengrass total cost is the expense of running Greengrass plus the cost of storing results in S3 plus put requests for results in S3, which equals $0.2627 + 0.0057 + 1.29 =$ \$ 1.5584 / month. For the cloud pipeline, the cost is the expense of storing raw images and final results in S3 plus get and 2×put requests cost in S3 plus the cost of running Lambda functions , which equals $0.814 + 0.0057 + 2.69 + 4.517 =$ \$ 8.027 / month. Though this estimate appears



(a) Avg. time-in-flight/ message. (b) Avg. compute time/ message. (c) Avg. CPU utilization %. (d) Avg. RAM utilization %.

Fig. 5: Comparison of the performance metrics in Greengrass (AWS) and Azure Edge (Azure) across different pipelines.

cheap, if there are, for example, 50 road side cameras, the cost for the cloud-only pipeline escalates quickly. This rough cost estimate indicates that executing image recognition on the cloud setting is $\approx$ 5.2 x more expensive than edge, with only an extra 230 ms in average end-to-end latency. It is possible to reduce the cost of the cloud-only pipeline by using less powerful Lambda functions, however, the storage cost alone is larger than the entire edge pipeline cost. Bandwidth usage-wise, the edge pipeline sends around 253.125 MB data over the network per month, whereas uploading images to cloud requires sending 35.38 GB data/month.

C. Discussion

We observed that both platforms do not handle very high throughput messaging well yet. In this case either the messages are delayed or fail to reach the cloud from the edge device. In future we want to benchmark this throughput.

Another important consideration is the relative ease of deploying dependencies and libraries. We can package all necessary libraries in the Docker container with a single Dockerfile in Azure Edge. On the contrary, in Greengrass Lambda functions, we need to compile external dependencies in required environment and add them in a zip file for deploying. We feel that the former is a cleaner choice for adding and managing a lot of external libraries.

In the end, from our experience, we feel both these approaches are suitable for carrying out edge computation. However, the higher end-to-end latency of Azure may be a problem for latency sensitive applications. Although Azure has richer customization options, we found development and integration to be easier in the AWS Greengrass platform.

V. CONCLUSION

We have presented EdgeBench, a benchmark suite for serverless edge computing plaforms. With this suite, we have studied two managed edge computing platforms, Greengrass and Azure Edge. Further, we have compared the performance of these platforms with that of cloud-only implementations of the same benchmarks. Our results show that the performance of Greengrass and Azure Edge are comparable, with the exception that Azure Edge exhibits higher end-to-end latency due to its batch-based processing approach. Further, our results show that for the image and scalar pipelines, the performance of Greengrass is comparable to that of the AWS cloud-only pipelines, while reducing the network bandwidth usage. These results indicate that edge computing is a promising alternative to cloud computing for CPU light workloads. In future work, we plan to extend EdgeBench to include additional applications and edge platforms.

REFERENCES

[1] P. Middleton, T. Tsai, M. Yamaji, A. Gupta, and D. Rueb, "Forecast: Internet of Things Endpoints and Associated Services, Worldwide, 2017, Gartner," 2017. [Online; Accessed August 2018].

[2] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, 2016.

[3] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young, "Mobile edge computingA key technology towards 5G," *ETSI white paper*, vol. 11, no. 11, pp. 1–16, 2015.

[4] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, 2017.

[5] Amazon Web Services, "AWS Greengrass Developer Guide," 2018. [Online; Accessed July 2018].

[6] Microsoft Azure, "Azure IoT Edge Documentation," 2018. [Online; Accessed July 2018].

[7] "Cloud IoT Edge." `https://cloud.google.com/iot-edge/`, 2018. [Online; Accessed August 2018].

[8] "Watson IoT Platform Edge overview (Preview)." `https://console.bluemix.net/docs/services/IoT/edge/WIoTP_edge.html`, 2018. [Online; Accessed August 2018].

[9] M. Malawski, K. Figiela, A. Gajek, and A. Zima, "Benchmarking heterogeneous cloud functions," in *European Conf. on Parallel Processing*, pp. 415–426, Springer, 2017.

[10] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *IEEE 37th Int. Conf. Distributed Computing Systems Workshops (ICDCSW)*, pp. 405–410, June 2017.

[11] T. Back and V. Andrikopoulos, "Using a microbenchmark to compare function as a service solutions," in *European Conference on Service-Oriented and Cloud Computing*, pp. 146–160, Springer, 2018.

[12] A. Deese, "Implementation of unsupervised k-means clustering algorithm within amazon web services lambda," in *Proc. IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing*, pp. 626–632, 2018.

[13] "Message Queuing Telemetry Transport." `http://mqtt.org/`, 2018. [Online; Accessed July 2018].

[14] "CMU Pocketsphinx Python." `https://github.com/cmusphinx/pocketsphinx-python`, 2018.

[15] "Tatoeba." `https://tatoeba.org/eng/`, 2018.

[16] "OpenCV." `https://github.com/opencv/opencv`, 2018.

[17] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," *CoRR*, vol. abs/1512.01274, 2015.

[18] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size," *CoRR*, vol. abs/1602.07360, 2016.

[19] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *Int. J. Comput. Vis.*, vol. 115, no. 3, pp. 211–252, 2015.

[20] "Time Machines GPS NTP+PTP Network Time Server (TM2000A)," 2018. [Online; Accessed July 2018].

[21] "vnStat." `https://github.com/vergoh/vnstat`, 2018.