

# Performance Optimization for Edge-Cloud Serverless Platforms via Dynamic Task Placement

Anirban Das, Shigeru Imai, Stacy Patterson

Dept. of Computer Science, Rensselaer Polytechnic Institute

dasa2@rpi.edu, shigeru.imai1@gmail.com, sep@cs.rpi.edu

Mike P. Wittie

Gianforte School of Computing, Montana State University

mike.wittie@montana.edu

**Abstract**—We present a framework for performance optimization in serverless edge-cloud platforms using dynamic task placement. We focus on applications for smart edge devices, for example, smart cameras or speakers, that need to perform processing tasks on input data in real to near-real time. Our framework allows the user to specify cost and latency requirements for each application task, and for each input, it determines whether to execute the task on the edge device or in the cloud. Further, for cloud executions, the framework identifies the container resource configuration needed to satisfy the performance goals. We have evaluated our framework in simulation using measurements collected from serverless applications in AWS Lambda and AWS Greengrass. In addition, we have implemented a prototype of our framework that runs in these same platforms. In experiments with our prototype, our models can predict average end-to-end latency with less than 6% error, and we obtain almost three orders of magnitude reduction in end-to-end latency compared to edge-only execution.

## I. INTRODUCTION

In the past several years, there has been increased usage of intelligent applications on end-user devices, including voice-activated virtual assistants, smart-home security cameras, augmented reality in mobile phones, and so on. In many of these applications, the bulk of data processing is performed in a cloud data center. The increase in the number of such applications, as well as in the number of devices, has led to a surge in the amount of data generated at the periphery of communication networks [23]. This surge presents challenges to the cloud-based computational approach, as applications must compete for limited network resources [28]. The situation is further complicated by the fact that many intelligent applications are latency sensitive, and thus require near real-time access to computational resources.

Edge computing proposes to address these challenges by leveraging the computational power in close proximity to data sources, sometimes in the end user devices themselves. These *edge devices* can perform data processing like filtering, aggregation, or inference. The results, typically much smaller in size, may then be forwarded to the cloud for further processing and decision-making. This approach reduces both data processing latency and bandwidth usage [26], [28].

To achieve this latency reduction requires that edge devices execute compute tasks in near-real time. Depending on the type of application or workload, this may not be possible on resource-constrained edge devices. In such cases, it may be necessary to offload the computation to a higher-resourced

compute node in the cloud. The problem is then to determine which tasks should be executed on the edge device and which should be executed in the cloud, so as to meet developer-specified performance criteria such as latency or cost.

We study this task placement problem in the context of serverless computing, more specifically, the Function-as-a-Service paradigm, an increasingly popular model for both cloud and edge platforms [21], [4], [8]. In this paradigm, the developer writes stateless functions that can be triggered by various events. Each function executes in its own container. The developer specifies the container resource allocations, and the containers in the cloud are orchestrated and provisioned by the cloud provider. The function performance depends on the input and application characteristics, the network transfer from the edge device to the cloud, the container resources, and the time to store the function results. Several cloud platforms also provide frameworks for function execution on edge devices [1], [22].

We propose a framework that dynamically determines where to execute serverless functions so as to optimize developer-specified performance criteria. Our framework targets intelligent applications that consist of a single serverless function that executes a data processing task on an input, for example, image recognition on a single frame (image) from a camera. Our framework processes a sequence of inputs, and for each input, it dispatches the task to a function in the appropriate container, from among the edge container and a set of containers in the cloud with various resource allocations.

Our framework addresses two optimization problems: (1) minimize latency subject to a cost constraint and (2) minimize cost subject to a latency constraint, where the latency is measured in terms of the time from the ingestion of the input to the storage of the results in the cloud. To perform this dynamic task placement, our framework predicts the application latency and cost for the various container configurations and input characteristics. Thus, a key contribution of our work is the development of accurate, data-driven performance models for each component in the edge and cloud execution pipelines. These models encompass network transfer time, container startup time, function execution duration, and storage latency.

The specific contributions of this work are as follows: (1) we present application-specific performance models for serverless applications in an edge-cloud computing platform - these models are trained and evaluated using data collected

from applications running in the Amazon Web Services (AWS) serverless environment; (2) we propose a dynamic task placement framework, using these models, that optimizes for developer-specified metrics such as latency or cost; (3) we provide extensive evaluations of our framework using real-world data from applications running in AWS; and (4) we present a prototype implementation of our framework and its evaluation results. In our experiments in AWS, our prototype predicts end-to-end latency with less than 6% error. In this study, we use AWS Greengrass for edge computing and AWS Lambda for the the cloud platform, but our framework can be generalized to other serverless platforms.

The rest of the paper is organized as follows. Sec. II provides details about the system architecture and our benchmark serverless applications. In Sec. III, we describe our framework architecture and objectives. Sec. IV presents the performance models and gives details of their training and evaluation. In Sec. V, we give the details our framework implementation, and in Sec. VI, we present evaluation results from both data-driven realistic simulations and a live prototype. In Sec. VII we discuss related work, and we conclude in Sec. VIII.

## II. ARCHITECTURE AND APPLICATIONS

A serverless edge-cloud architecture consists of an edge device with access to input data. The device is connected via a network to a cloud data center that has both compute and storage capability. Industry frameworks support two approaches or *pipelines* to execute an application in this architecture, a *cloud pipeline*, where the data processing task executes in the cloud data center and an *edge pipeline*, where the task executes on the edge device itself. We briefly describe each pipeline below, as well as the factors that impact its end-to-end latency and execution cost. We then describe the benchmark applications that we use to validate our models and framework.

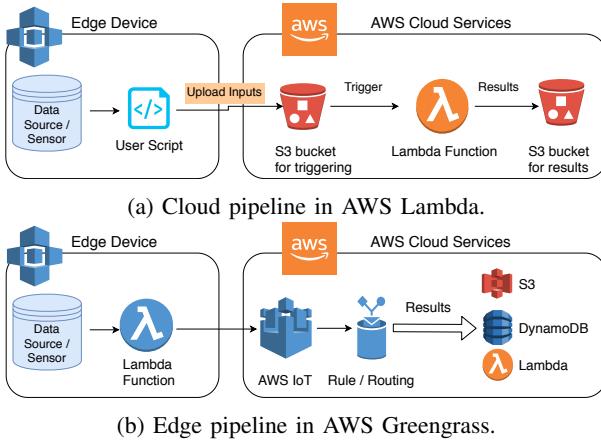


Fig. 1: AWS serverless application pipelines.

### A. Serverless Computing Pipelines

1) *Cloud Pipeline*: In the cloud pipeline, the edge device uploads the input data to a cloud storage service. This upload triggers the execution of the stateless function that performs the data processing task. The function writes the result of this

task to cloud storage. An example of this pipeline for AWS Lambda is shown in Fig. 1a. Here, the edge device uploads input data to an AWS S3 bucket, triggering an AWS lambda function that stores the result in another S3 bucket.

In commercial serverless platforms, each function instance runs in its own container. Further, some platforms allow developers to configure the container resources, e.g., memory and CPU, during application deployment. This resource configuration, in turn, impacts the function performance when executed in that container. For the purposes of this work, we adopt the container configuration options offered by AWS. The developer specifies the memory allocation for the container, ranging from 256 MB to 3008 MB. AWS assigns CPU power to the container proportionally to the allocated memory.

To scale serverless applications, serverless platforms create containers as needed. When a function is triggered, if a container is available (not currently executing a function), the function executes in the existing container. Otherwise, a new container is created for that function execution. If a container is idle for some extended period, it is destroyed. If a new container needs to be created before executing a function, this is called a *cold start*. A cold start imposes a non-trivial time penalty to initialize the container and load any libraries before the function execution begins. If a function is assigned to an existing container, this is a *warm start*, and the startup time can be up to an order of magnitude lower than a cold start.

a) *Pipeline performance*: We measure the performance of the cloud pipeline in terms of the *end-to-end latency* for processing a single input. This measurement begins when the input is ingested by the application on the edge device and ends when the result is saved in cloud storage (the second S3 bucket in Fig. 1a). The latency, for an input  $k$  and memory configuration  $m$ , consists of the following components:

- $upld(k)$ : The time to transfer the input from the edge device to cloud storage. This consists of the network transfer time and any write overhead to the first S3 bucket.
- $start(m)$ : The time to start the cloud container for function execution. The time depends on the container memory  $m$ , but not the input  $k$  and varies for a cold start or warm start.
- $comp(k, m)$ : The compute time in the function.
- $store(k)$ : The time to save the output to cloud storage.

The end-to-end latency is therefore:

$$T_c(k) = upld(k) + start(m) + comp(k, m) + store(k). \quad (1)$$

b) *Pipeline cost*: In serverless cloud platforms, the cost is typically based on the function execution time. In this work, we use the AWS pricing model. AWS bills users for the duration for which code executes on AWS systems, rounded up to the nearest 100 ms. The price is proportional to the amount of memory allocated to the container at  $\$1.667 \times 10^{-6}$  per GB-s execution [2]. Further, there is a fixed charge of \$0.20 per 1 M lambda function requests. To determine function execution cost from execution time  $comp(k, m)$ , we round the execution time to the nearest ms and then apply the AWS pricing model. We limit our study to the function execution cost, as this is the most challenging to model and predict.

2) *Edge Pipeline*: One can also use an edge pipeline, where data processing is performed within a function on the edge device itself. Upon function completion, only the results, usually much smaller in size, are sent to the cloud for storage. Similar to the cloud platform, the developer has a facility to constrain the memory limit of the lambda function, but here the upper limit is dictated by the available resource in the edge device. Thus, we assume a single memory configuration for the edge device container.

In Fig. 1b, we show an example of an edge pipeline using the AWS Greengrass edge computing framework. Similar pipelines can be created in other edge computing platforms, e.g. Azure IoT Edge. In Greengrass, a lambda function executes inside the Greengrass run-time on an edge device, and the function results are subsequently sent to the AWS IoT Core service in the cloud. The IoT Core service forwards the results to a developer-specified endpoint, for example, S3 or DynamoDB.

Greengrass offers two execution models for lambda functions, a stateless function and a ‘long-lived’ function [1]. The stateless model is similar to the model used in the AWS Lambda platform, in that multiple functions may execute on an edge device in parallel. In the long-lived model, the function runs continuously on the edge device; it can write to and read from device storage and this storage persists for the lifetime of the function. We use the long-lived function model for several reasons. First, we consider resource-constrained edge devices that may not have the power to execute functions in parallel, depending on the application. Second, co-location of multiple functions contending for limited hardware resources may cause unpredictable behavior. This unpredictability limits the ability to optimize task placement.

a) *Pipeline performance*: We measure the performance of the edge pipeline in terms of the end-to-end latency for processing a single input. The latency for an input  $k$  consists of the following components:

- $comp(k)$ : The compute time on the edge device.
- $iotup(k)$ : The time to send the results from the edge device to the cloud IoT Core, including the network transfer time and the framework-induced overhead.
- $store(k)$ : The delay between when the results are received in the cloud IoT Core service and when they are available in the cloud storage.

The end-to-end latency for the edge pipeline is:

$$T_e(k) = comp(k) + iotup(k) + store(k). \quad (2)$$

b) *Pipeline cost*: For the edge pipeline, we again consider the AWS pricing model. Lambda function execution inside AWS Greengrass is free, but there exists a fixed yearly device registration fee of \$1.49 - \$2.05 based on the region. The cost is fixed per active edge device per month and is independent of the number of function executions. Thus, we consider the amortized function execution cost at the edge to be zero. Our cost analysis excludes storage and network costs, as processed data sent by edge devices is small and costs are easy to predict with respect to request volume.

## B. Applications

We implement three representative applications, motivated by real-world use cases. All are implemented in Python.

**Image Resizing (IR):** An image file is taken as input, and the function reduces the dimensions of the image to a  $128 \times 128$  pixel thumbnail and sends the thumbnail to the cloud. This reduction can be done to save bandwidth or storage cost, or to regularize the image for use in a deep learning application. This application mimics a scenario where a traffic camera takes a stream of pictures to identify traffic congestion. For the input workload, we use a set of images from the Image of Groups [12] database.

**Face Detection (FD):** Here, the application, given an input image, finds the number of faces present. The application mimics a smart camera that detects faces in a captured frame, for either security purposes, for example. For face detection, we use the `dlib` [18] library, and for simplicity, we store only the number of faces detected in the frame. We use images from the Images of Groups database for our input workload.

**Speech-To-Text (STT):** An audio file containing speech is provided as input, and the application transcribes the speech into text. This emulates the functionality of a smart speaker where the user issues commands that are translated to text and then used as input in a search or activity. For the transcription, we use CMU’s `pocketsphinx` [6] library. For our input workload, we use audio files from the Tatoeba Corpus [29].

We note that different applications may have different input rates from the data source. While input commands to a smart speaker may be sparse, a traffic camera may produce a fixed number of images per minute. To simulate this behavior, the applications ingest input files from a local directory on the edge device at a fixed rate. For IR and FD, we implement a faster input rate of four files or images per second, and for STT, we use slower rate of one audio file every ten seconds.

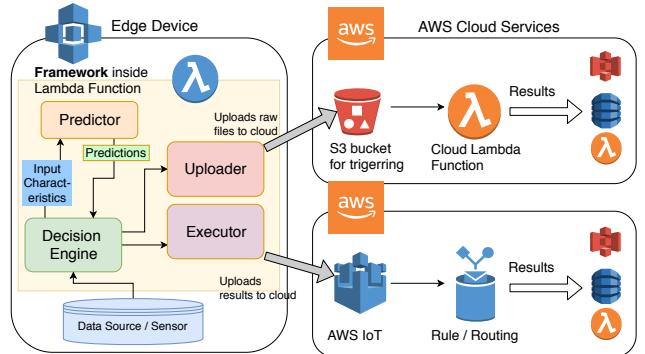


Fig. 2: Dynamic task placement framework.

## III. FRAMEWORK

For applications such as those in Sec. II-B, it is necessary to execute tasks in container configurations with appropriate resources to meet performance criteria. Hence, we introduce a framework that dynamically selects the cloud or the edge pipeline to execute the application tasks (functions) based on the workload characteristics and latency or cost requirements.

## A. Framework Architecture

Fig. 2 depicts our framework architecture. We consider a single edge device that ingests an input workload. The edge device runs a single lambda function. There are also  $N$  lambda functions configured in the cloud with  $N$  distinct memory configurations. Let  $\Phi = \{\lambda_{m_1}, \lambda_{m_2}, \dots, \lambda_{m_N}\}$  be the set of  $N$  cloud container configuration options, where container type  $\lambda_m$  is configured with  $m$  MB of memory.

The framework functionality resides in the lambda function on the edge. It ingests input workload from the Data Source. The data then flows to the Decision Engine. The Decision Engine first calls the Predictor; given an input, Predictor predicts the end-to-end latencies and costs for executing the application for each configuration in  $\Phi$ , as well as for executing the data processing task in the edge device. Due to the differences in cold start and warm start times, the Predictor must predict both whether an available container exists for a given cloud configuration, and what the performance of the function will be in that configuration.

The Decision Engine, then, based on the objective (Sec. III-B) and the predicted latencies and costs, either places the task at the edge or selects a  $\lambda_m$  in cloud to offload the task. If the edge is selected, the job is sent to the Executor, which contains a FIFO task queue. It executes tasks one at a time and sends the results to the cloud. We note that the Executor in Fig. 2 corresponds to the lambda function in Fig. 1b. With some abuse of terminology, we refer to the Executor as  $\lambda_{edge}$ . If a cloud configuration is selected, the Uploader uploads the input file to the corresponding S3 bucket in the cloud. The rest of the standard cloud pipeline then continues as in Fig. 1a. We configure the edge function execution to be non-blocking, so that the Decision Engine processes each input without waiting for the completion of the previous task.

## B. Optimization Objectives

Our framework provides two options for task placement policy. We detail these policies below.

### a) Cost minimization subject to deadline constraints:

The objective is to minimize the execution cost, subject to developer-specified end-to-end latency deadline  $\delta$  per task. The framework achieves this goal by, for each task  $k$ , selecting the least expensive configuration that satisfies the deadline. This formulation targets applications with strict latency requirements, for example, an application that uses a smart camera to monitor a secure site and send alerts on detection of occupants for further analysis. Note that for this problem to have a feasible solution, there must be at least one configuration that can process any input within the given deadline.

### b) Latency minimization subject to cost constraints:

Here, the objective is to minimize the end-to-end latency while keeping the cost of each task execution under some budget  $C_{max}$ . For example, a store may use a smart camera to recognize customers and text them coupons. In this case, the application providers may have a strict budget, and while low latency is desirable, it is not necessary. If we consider only a

single input  $k$ , the goal is then to select a configuration that solves the problem:

$$\begin{aligned} & \text{minimize } T(k) \\ & \text{subject to } C(k) \leq C_{max}, \end{aligned} \quad (3)$$

where  $T(k)$  and  $C(k)$  are the latency and cost for task  $k$ , respectively.

It is possible that single function execution may not use up the entire budget  $C_{max}$ , i.e., a sequence of tasks may leave budget surplus that could have been used to reduce latency. So, instead, we consider the constraint that for any sequence of  $K$  tasks, we have  $\sum_{k=1}^K C(k) \leq KC_{max}$ . We implement this constraint using the following optimization problem:

$$\begin{aligned} & \text{minimize } T(k) \\ & \text{subject to } C(k) \leq C_{max} + \alpha \times surplus(k) \end{aligned} \quad (4)$$

where  $surplus(k)$  is the accumulated sum of unused budget,  $surplus(k) = \sum_{i=1}^{k-1} (C_{max} - C(i))$ , and  $\alpha \in [0, 1]$  is a scaling factor that determines how much of the surplus can be used for task  $k$ . Since the cost of executing the task at the edge is zero, it is always possible to find a task placement that satisfies the cost constraint. Thus, the surplus is never negative.

## IV. PERFORMANCE MODELS

To solve the task placement problems described in Sec. III-B requires accurate methods for predicting the end-to-end latency  $T(k)$  for any input  $k$ . From this predicted latency, we can also compute the predicted cost  $C(k)$ . To predict the latency, we create data-driven performance models for each latency component described in Sec. II-A. Due to the heterogeneity of application characteristics, we create *application-specific* performance models, trained using sample input data. In this section, we describe our performance models and model training, and we give results on the model accuracy.

### A. Cloud Performance Model

We segregate the end-to-end latency  $T_c(k)$  for job  $k$  configuration  $\lambda_m$  into four parts, as described in Eqn. (1).

- *Upload time*: We model the upload time as a linear function of the input data size:

$$upld(k) = \theta_1 + \theta_2 \times size(k),$$

$\theta_1$  and  $\theta_2$  are determined via regression over training data.

- *Lambda startup time*: The startup time varies depending on whether it is a cold start  $start_c(m)$ , or a warm start  $start_w(m)$ . Based on training data, we observe that each of these startup times follows a normal distribution, which we model by taking the mean of the training data.
- *Compute time*: We observe that the compute time is a non-linear function of  $size(k)$  and the container memory configuration  $m$ . After experimenting with several regression methods, we identified Gradient Boosted Regression Trees [11] to be the most accurate.
- *Storage time*: The sizes of the function outputs are both small and very similar across applications. Further, because

AWS S3 quantizes the file availability timestamp to seconds, we are only able to measure the storage time with coarse granularity. Thus, our training data exhibits no correlation between the input and the storage time. We, therefore, model the storage time as a quantized normal random variable, and we model  $store(k)$  by taking the mean over the training set.

- *Container idle time:* To predict whether invoking a function will cause a warm start or a cold start, we also need to model the container idle time, i.e., how long containers stay warm in AWS infrastructure before their resources are reclaimed due to inactivity. We observe that this idle time is independent of any input or application characteristics and model this by a single value  $T_{idl}$ . We perform experiments, similar to the approach taken in [32], and use a binary search to find  $T_{idl}$ . Our findings corroborate with this previously measured value of  $T_{idl} \approx 27$  minutes.

### B. Edge Performance Model

For edge pipelines, we also model the components of Eqn. (2) separately.

- *Compute time:* We model the compute time as a linear function of the input file size:

$$comp(k) = \phi_0 + \phi_1 \times size(k).$$

We determine the parameters  $\phi_0$  and  $\phi_1$  using regression over the training data.

- *IoT Core upload time:* As previously mentioned, the size of function results are small and similar across inputs and applications. Thus, we attribute any variability in recorded times to framework and network overhead. We model this upload time as a normal random variable, and we estimate  $iotup(k)$  by the mean over the time measurements of the training data.
- *Storage time:* We adopt a similar approach to the storage time for cloud pipeline, and estimate  $store(k)$  using the mean over the measurements from the training data.

### C. Model Training and Evaluation

To generate the measurements for training and evaluation, we execute the applications described in Sec. II-B using the pipelines shown in Fig. 1a and Fig. 1b. For all experiments, for the cloud pipelines, we use 19 AWS Lambda function memory configurations between 640 MB and 3008 MB. For the edge device, we use a Raspberry Pi 3B running Greengrass core version 1.7.0. The edge device container is provisioned with 512 MB RAM and set to run indefinitely. The image and audio file directory and the directory for storing metrics are mounted into the Greengrass execution environment as ‘Local Resources’. The edge device is connected to the internet via a wireless router using the 2.4 GHz spectrum. A dedicated Stratum 1 NTP time server, TM2000A [30] is used to synchronize the time of the Raspberry Pi to AWS servers.

- 1) *Cloud pipeline data collection:* To train the components of the cloud latency model, we first collect measurements of the  $upld(k)$ ,  $start_c(m)$ ,  $start_w(m)$ ,  $comp(k, m)$ , and  $store(k)$  by running the pipeline using only warm starts. We

ensure there is an available container by first executing the function on a dummy input. We then upload each input file to the specified S3 bucket. This upload triggers the execution of the lambda function. We wait for a time interval in between uploads to ensure the previous function execution has completed. For each input, we measure each mentioned time component following a method similar to our previous work [8].

To measure the cold start time  $start_c(m)$  for separate values of  $m$ , we use a method similar to that in [19]. For each container configuration, we measure 100 cold start latencies. The cold start latency does not appear to be correlated to the container memory size for the three applications.

TABLE I: Mean latencies (ms) used for training examples.

	Cloud Pipeline			Edge Pipeline	
	Warm Start	Cold Start	Store	IoT Upload	Store
IR	162	741	549	n/a	579
FD	163	1500	584	25	583
STT	145	1404	533	27	579

2) *Edge pipeline data collection:* For each input workload, we measure the components in Eqn. (2):  $comp(k)$ ,  $iotup(k)$ ,  $store(k)$ . We set up the edge device running AWS Greengrass to ingest input files from a directory. Results are then sent to AWS IoT on the cloud, where a ‘Rule’ redirects the results to an S3 bucket. We note that for the IR application, we directly transmit the resized image to S3 due to Greengrass’ limitations on data upload type. Hence, we measure the storage time  $store(k)$  as the time between when a file is sent from the edge device to when it is available in S3, and we do not include  $iotup(k)$  in the end-to-end latency.

3) *Model Training and Evaluation:* For the IR and FD applications, we collect measurements for each configuration for 1400 input images. For the STT application, we collect measurements for 3400 input auto files for each configuration. We use the common 80:20 train:test split to train our models. For the regression techniques, we use a grid search for model tuning and chose the best performing estimator by using 3-fold cross-validation. In the FD and IR applications, we quantify  $size$  by the total number of pixels in the image, as image manipulation depends on the matrix of the image pixel values. In STT,  $size$  is measured in bytes. We use the gradient boosting regressor in the scikit-learn package for regression analysis of  $comp(k, m)$  and  $upld(k)$  respectively, for each application and ridge regression for modeling  $comp(k)$ .

Table II shows the MAPE of the end-to-end latency over their respective test sets for the edge pipelines and cloud pipelines with warm start. The error is less than 16% for most of the applications. This suggests our models can predict the end-to-end latency relatively accurately. The one exception is the IR cloud pipeline, with MAPE of 25.3%.

In general, more variability in performance leads to higher MAPE values. Fig. 3 depicts the end-to-end latency predicted by our model for the cloud pipeline for the FD and STT applications, with a 1536 MB warm start configuration. The figure also shows the measurements from the test data. We can see that even for a single lambda function configuration and

TABLE II: Mean Absolute Percentage Error in the end-to-end latency of the cloud and edge pipelines.

	IR	FD	STT
Cloud	25.38	13.24	14.56
Edge	2.15	3.78	15.70

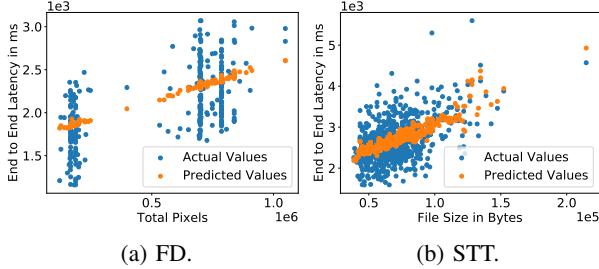


Fig. 3: Performance of end-to-end model on test data set for 1536 MB cloud lambda memory configuration (warm starts).

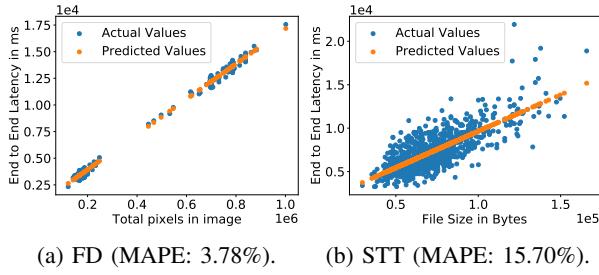


Fig. 4: Performance of edge end-to-end model on test dataset for edge pipelines.

similar input file sizes, there is a notable variance in end-to-end latency, which presents challenges for generating accurate predictions. Fig. 4 depicts the end-to-end latency prediction and the test data measurements for FD and STT in the edge pipelines. We note that the edge pipelines exhibit far less variation, and thus it is possible to predict the performance of these pipelines more accurately.

## V. FRAMEWORK IMPLEMENTATION

In this section, we elaborate on implementation details of the framework, specifically, the Predictor and the Decision Engine, and how they use the prediction models in Sec. IV to solve the optimization problems.

### A. Predictor

Recall that the job of the Predictor is to predict the cost and end-to-end latency for a given input for every container configuration. Given the end-to-end latency prediction models of executions with cold and warm starts and the container idle time model, to predict the end-to-end latency, the Predictor must determine whether a function execution will lead to a cold start or a warm start. Since AWS does not expose any API for obtaining this information during execution, the Predictor maintains an offline data structure, the active container information list (CIL) that estimates which container configurations are warm in the AWS cloud. For each  $\lambda_m$ , the CIL stores a list of active containers, and for each container,

### Algorithm 1 Minimize latency subject to cost constraint.

```

1: function MINLATENCY (Inputs,  $\Phi$ ,  $C_{max}$ ,  $\alpha$ )
2:   surplus = 0
3:   for input  $k$  do
4:     ( $times$ ,  $costs$ ) := PREDICTOR.PREDICT(input  $k$ )
5:      $\mathcal{M} := \{\lambda_j \mid \lambda_j \in \Phi \cup \{\lambda_{edge}\} \text{ and}$ 
6:        $costs(\lambda_j) \leq C_{max} + \alpha \times surplus\}$ 
7:     config  $\leftarrow \lambda_j \in \mathcal{M}$  with minimum latency
8:     Use config for function execution
9:     surplus +=  $C_{max} - costs(config)$ 
10:    PREDICTOR.UPDATECIL(times(config), cost(config))
11:   end for
12: end function

```

it keeps track of (1) whether the container is ‘idle’ or ‘busy’ executing a function, (2) the completion time of latest function executed within that container, and (3) the estimated time the container will be destroyed, obtained using (2) and  $T_{idl}$ .

The Predictor exposes two methods to the Decision Engine: predict, that takes the input file as input and returns predictions of the end-to-end latencies and costs for  $\lambda_m \in \Phi$  and for  $\lambda_{edge}$ . To generate these predictions, for each  $\lambda_m \in \Phi$ , the Predictor queries the CIL to determine if there exists an ‘idle’ container for that  $\lambda_m$ . If so, the Predictor generates the end-to-end latency using the model in Sec. IV-A with a warm start time. In case there are no containers or all containers are ‘busy’ for  $\lambda_m$ , then the Predictor generates the cold start end-to-end latency. The Predictor uses the model in Sec. IV-B to predict the end-to-end latency for  $\lambda_{edge}$ . The costs for each configuration are then computed as described in Sec. II-A.

Once the Decision Engine has selected the configuration for the input, it invokes the updateCIL method on the Predictor, passing in the chosen configuration. The Predictor then updates the CIL, adding a new container if the lambda function execution results in a cold start, and updating the container status and function completion time, based on the estimated  $comp(k, m)$  or  $comp(k)$ . If there are multiple ‘idle’ containers for the selected configuration, we assume the function is assigned to the one with the most recent function completion time. This assumption is based on our empirical observations of AWS Lambda. In each call of updateCIL, the Predictor also checks for and removes dead containers from the CIL, based on estimated container lifetime information.

### B. Decision Engine

The algorithm used by the Decision Engine for minimizing latency is described in Alg. 1. The algorithm for minimizing cost is similar. The *Decision Engine* obtains predictions from the *Predictor*. In both algorithms, for each input, the framework first uses Predictor.Predict to find the end-to-end latencies and costs for edge and cloud lambda functions. If the objective is to minimize cost, the Decision Engine firsts create the list of configurations  $\mathcal{M}$  that satisfies the latency constraint  $\delta$ . For each cloud configuration, the Decision Engine checks whether the predicted latency from the Predictor is less than  $\delta$ , and if so, adds the configuration to  $\mathcal{M}$ . For  $\lambda_{edge}$  the Decision Engine checks whether the predicted latency (from the Predictor) plus the predicted time in the Executor’s FIFO

queue (based on the predicted latency for any earlier tasks in the queue as well as any executing task) is less than  $\delta$ . If so,  $\lambda_{edge}$  is added to  $\mathcal{M}$ . The Decision Engine then selects the configuration with the minimum predicted cost from  $\mathcal{M}$ . If  $\mathcal{M} = \emptyset$ , there is no configuration that satisfies the deadline, so to save cost, the task is added to the Executor queue.

If the objective is to minimize latency, the Decision Engine first creates the set of configurations  $\mathcal{M}$  that satisfies the cost constraint, from the list of predictions returned by the Predictor. It then selects the configuration with minimum predicted end-to-end latency from this set, where latencies are determined as described in the previous paragraph (lines 5–7 of Alg. 1). The Decision Engine then updates the surplus based on the predicted cost. At the end of both algorithms, the Decision Engine calls `Predictor.updateCIL` to update the CIL with container information.

## VI. EXPERIMENTS

We first present an evaluation of our framework using simulation-based experiments, with measurements collected in AWS Lambda and Greengrass. We then show results from a live experiment with our framework prototype.

### A. Simulation-Based Experiments

For our experiments, we consider the same set of 19 cloud container configurations and the same edge configuration as used for the model training in Sec. IV.

We first collect warm latency measurements for a new set of input data for each application in each container configurations, both cloud and edge, using the process described in Sec. IV-C. We use 600 input files for each application. Since it difficult to collect a large number of cold start latency measurements, we instead simulate the cold start time by randomly selecting samples from the best-fit distribution on cold-start values from our training data. Similarly, we simulate  $T_{idl}$  by randomly selecting samples from a normal distribution fitted on our observed measurements of container lifetime.

We implement an event-driven simulation framework, which contains complete implementations of the Predictor and the Decision Engine. The Predictor uses the trained models described in Sec. IV-C3. We feed input into the framework at intervals generated with a Poisson process, with arrival interval rate of four files per second for IR and FD and one file every ten seconds for STT. The Decision Engine selects a configuration based on the predicted end-to-end latency and cost. We then simulate execution using the actual end-to-end latency and actual costs from the measured data.

We initially perform all experiments using the training data to identify *configuration sets*. We observe that with a candidate set of all possible configurations, only a few configurations are ever selected. We thus create sets that contain only the configurations the framework selected for the training data. Every configuration set contains  $\lambda_{edge}$  by default; we only state the elements of  $\lambda_m$  explicitly for brevity.

We present results of our simulations for both optimization problems for the three applications.

TABLE III: Simulation: minimizing cost subject to deadline constraint. All configuration sets also include  $\lambda_{edge}$ .

(a) IR:  $\delta = 2.7\text{s}$ , Avg. actual end-to-end latency  $\approx 1.37\text{s}$ .

Configuration Set	Total Actual Cost (\$)	Cost Prediction Error %	% Deadlines Violated	Average Violation (ms)
<b>640,1024,1152</b>	0.00155841	8.54	0.83	1.38
640,1024,1408	0.00156019	5.88	1	1.73
640,896,1152,1280	0.00156681	8.57	1.17	3.12
640,768,1152	0.00157790	9.68	0.83	5.67

(b) FD:  $\delta = 4.5\text{s}$ , Avg. actual end-to-end latency  $\approx 2.43\text{s}$ .

Configuration Set	Total Actual Cost (\$)	Cost Prediction Error %	% Deadlines Violated	Average Violation (ms)
<b>1280,1408,1664</b>	0.01470774	0.26	0.33	3.7
1152,1408,1664	0.01475062	0.49	0.33	3.27
1152,1536,1792	0.01483715	2.85	0.5	1.72
1280,1408,1536,1792	0.01483860	3.38	0.67	4.25

(c) STT:  $\delta = 5.5\text{s}$ , Avg. actual end-to-end latency  $\approx 3.35\text{s}$ .

Configuration Set	Total Actual Cost (\$)	Cost Prediction Error %	% Deadlines Violated	Average Violation (ms)
<b>768,1152,1280,1664</b>	0.019970506	2.49	6.17	49.6
640,768,1280,1664,1792	0.020009885	1.75	7.83	71.94
640,768,896,1280,1664	0.020022751	1.91	7.67	66.49
640,896,1152,1664	0.020223292	3.33	6	58.40

TABLE IV: Simulation: minimizing latency subject to cost constraint. All configuration sets also include  $\lambda_{edge}$ .

(a) IR:  $C_{max} = \$5.33442 \times 10^{-6}$ ,  $\alpha = 0.02$ .

Configurations	Avg. Actual Time/Task (s)	Latency Prediction Error %	% Constraints Violated	% Budget Used
<b>1408,1664,2944</b>	1.30	9.72	2.33	84.8
1536,1664,2048,2944	1.314	7.90	2.17	88.6
1280,1536,1664,2944	1.315	7.99	2.17	88.7
1280,1408,1536,2944	1.329	10.73	1.83	84.8

(b) FD:  $C_{max} = \$2.96997 \times 10^{-5}$ ,  $\alpha = 0.02$ .

Configurations	Avg. Actual Time/Task (s)	Latency Prediction Error %	% Constraints Violated	% Budget Used
<b>1536,1664,2048</b>	2.1218	0.34	2.5	90.8
1664,1920,2048	2.122	1.14	2	92.3
1280,1664,2048	2.126	0.3	2.17	90.7
1536,1664,1920	2.151	1.22	1.33	90.3

(c) STT:  $C_{max} = \$3.0747 \times 10^{-5}$ ,  $\alpha = 0.03$ .

Configurations	Avg. Actual Time/Task (s)	Latency Prediction Error %	% Constraints Violated	% Budget Used
<b>1152,1280,1664</b>	3.492	0.47	15.5	99.4
1664	3.494	0.86	13.33	99.2
1024,1280,1664	3.504	0.50	14	99.3
1024,1152,1280,1664	3.561	0.85	15.17	99.3

1) *Cost Minimization*: We first evaluate our solution for cost minimization subject to a per-function-execution deadline. We select the deadline  $\delta$  for each application based on the training data, ensuring that each configuration set contains a feasible configuration for every input in the training set.

In Table III, we present the performance of different configuration sets in increasing order of total actual cost for each application, along with the percentage error between the actual and predicted total cost. The total costs are computed over all 600 inputs. We measure cost prediction error % as the absolute percentage error between the total actual cost and total predicted cost. We also show the percentage of inputs where actual end-to-end latency violated the deadline.

We observe that the configuration sets {640MB, 1024MB, 1152MB}, {1280MB, 1408MB, 1664MB}, and {768MB,

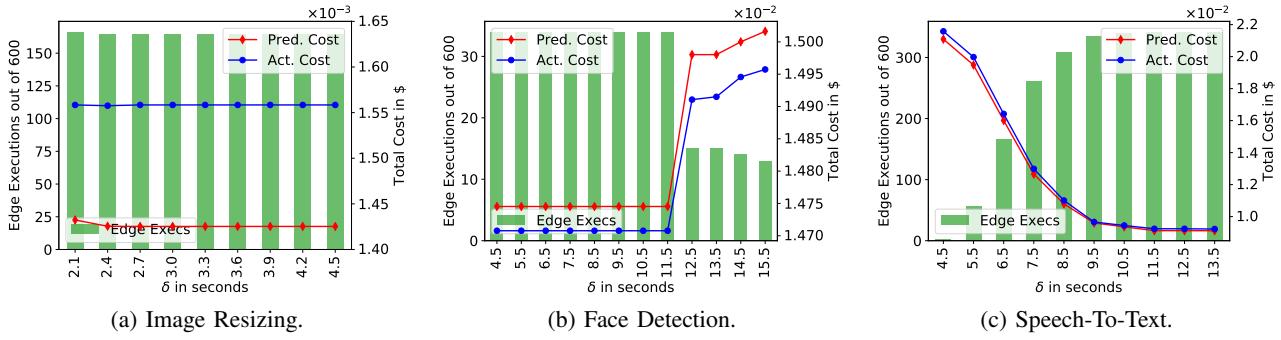


Fig. 5: Total execution cost (right Y axis) in \$ vs.  $\delta$  (in seconds) for best performing configuration of different applications in minimizing total cost. The bar chart (left Y axis) represents number of edge executions out of 600.

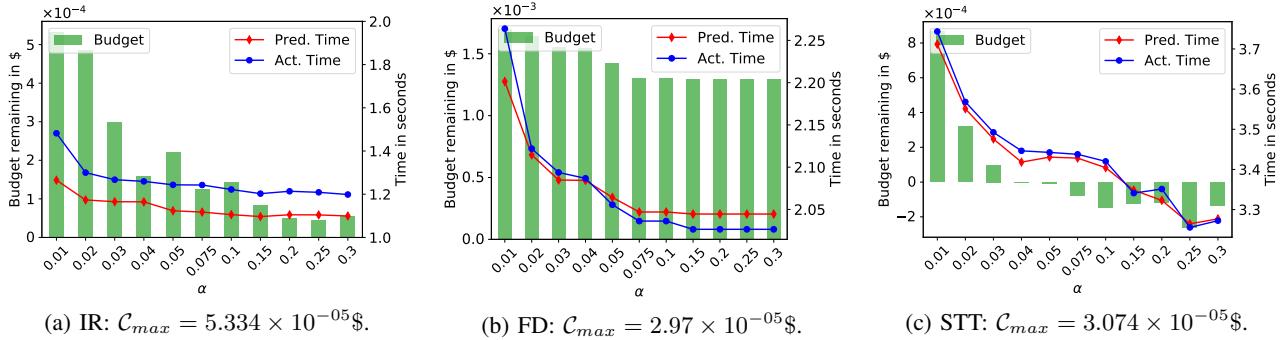


Fig. 6: Average end-to-end latency (right Y axis) vs.  $\alpha$  for best performing configuration of different applications in minimizing end-to-end latency. The bar chart (left Y axis) represents total budget \$ remaining at the end of execution.

1152MB, 1280MB, 1664MB} achieve the smallest total actual cost in the IR, FD, and STT applications, respectively. We also observe that a smaller cost prediction error leads to better performance in terms of the total cost. In general, smaller function execution times are prone to higher cost prediction error. AWS quantizes billed amount in multiples of 100ms, e.g., 98 ms compute time would be rounded to 100ms, whereas a 101ms compute time will be rounded to 200ms, and so a small error in the execution time prediction may result in a larger error in cost prediction when the magnitude of execution time is low. We also see that fewer deadline violations are correlated with better performance in terms of minimizing total cost.

In Fig. 5, we plot the predicted and actual total costs versus the deadline  $\delta$  for the best configuration (in bold) in Table III for each application. For all three applications, our predicted cost closely mirrors the actual cost. We observe that for IR, the number of edge executions does not appear to be correlated with the deadline. This is because for IR, in general, the edge pipeline execution is faster than the cloud pipeline execution. We also observe that for FD, as the deadline increases, the number of edge executions decreases, and accordingly, the cost increases. This is because, with a larger deadline, tasks are assigned to the edge, causing the edge to be busy, which in turn, leads to more tasks being assigned to the cloud. STT exhibits a more expected behavior; as the deadline increases, more tasks are executed at the edge. This is in part due to the

slower input rate for STT, which increases the availability of the edge for task execution.

We observe that the absolute error between predicted and actual total cost for the best performing configurations of FD and STT are less than 3%. Also, all other configurations for FD and STT performs well, with less than 4% absolute total cost prediction error. Finally, we observe that warm start vs. cold start prediction mismatches can influence the total cost prediction error. A slower input rate reduces the chances of mis-predicting cold and warm starts; STT has 0% absolute error between estimated and actual cold and warm starts, while for FD it is 2.5%.

2) *Latency Minimization:* We next evaluate our framework in solving the problem of minimizing latency subject to a task cost constraint. We use the formulation in Eqn. (4), which allows surplus budget to be spent on subsequent tasks. For each application, we select  $C_{max}$  and  $\alpha$  from experiments on the training data set. We select  $C_{max}$  and  $\alpha$  to be small enough so that for some inputs, it is necessary to use  $\lambda_{edge}$ . Similar to the previous experiments, we select configuration sets that consist of configurations that the framework selected when processing the training data.

We measure latency prediction percentage error as the absolute percentage error between actual and predicted average end-to-end latency at the end of the simulation. Further, we measure the percentage of constraints violated as the percentage of tasks where the actual cost of execution violated the

corresponding cost constraint. The percentage of budget used is computed as the total actual cost for processing the input workload divided by the total budget for the input workload,  $C_{max} \times \text{number of inputs}$ .

In Table IV, we present the results for different configuration sets in increasing order of average end-to-end latency. The table also shows the latency prediction error, percentage of function executions that violated the cost constraints, and the percentage of total budget used. We observe that configurations  $\{1408\text{MB}, 1664\text{MB}, 2944\text{MB}\}$ ,  $\{1536\text{MB}, 1664\text{MB}, 2048\text{MB}\}$ , and  $\{1152\text{MB}, 1280\text{MB}, 1664\text{MB}\}$  achieve the minimum end-to-end latency for the IR, FD and STT applications, respectively. We also observe that these configurations have low latency prediction error (except IR due to its high variance). Further, even though the cost constraints were violated for some inputs, the total cost of execution of the entire input workload was always under the total budget.

In Fig. 6, we plot the actual and predicted average end-to-end latency for various values of  $\alpha$  with  $C_{max}$  fixed. We use the best configuration set per application, shown in bold in Table IV. We observe that the actual average latency obtained from the framework execution closely follows the predicted average latency, with less than 2% absolute error for FD and STT and less than 11% error for IR.

We observe that in all applications, with increasing  $\alpha$  the average end-to-end latency decreases. By increasing  $\alpha$ , more surplus budget is available per task, and thus, more cloud configurations can satisfy the cost constraint. These cloud configurations typically have shorter execution times. We further observe that for FD, the total remaining budget does not vary much with  $\alpha$ , and in both IR and STT, the budget remaining decreases with increased  $\alpha$ . With a smaller  $C_{max}$ , a larger value of  $\alpha$  may lead to budget violations. For example, in STT, we observe that for the best configuration  $\{1152\text{MB}, 1280\text{MB}, 1664\text{MB}\}$ , as we increase  $\alpha$  above 0.04, the total budget remaining becomes negative, which means the total actual cost went over the total budget. Also, for  $\alpha = 0$ , we observe very high average end-to-end latencies: IR = 10.5 s, FD = 452.2 s, and STT = 12.64 s. This is due to the fact many tasks are run on the edge, as the cost constraint restricts cloud executions. As a result, the waiting periods in the edge queue leads to an increase the average execution time.

### B. Live Evaluation

To demonstrate the effectiveness of our framework in a real-world application, we have implemented a prototype of our framework. We evaluate this prototype in AWS Greengrass and Lambda using the FD application, with the same 600 input files used in the simulations. The framework is configured to minimize end-to-end latency subject to a cost constraint. We use the edge configuration described in Sec. IV-C. For the cloud, we use the best configuration set from the simulations,  $\{1536\text{MB}, 1664\text{MB}, 2048\text{MB}\}$ .

We measure the accuracy between the predicted and actual latency. Further, we measure how many times the framework violates the budget and the percentage of the budget remaining

at the end of workload. We also measure the number of times we mis-predict ‘cold’ or ‘warm’ starts. We perform the experiment four times and show the average results.

TABLE V: Average results over four runs of the FD application with configuration set  $\{1536\text{MB}, 1664\text{MB}, 2048\text{MB}\}$ ,  $C_{max} = 2.96697 \times 10^{-5}$ , and  $\alpha = 0.02$ .

Avg. Actual End-To-End Latency	Latency Prediction Error	Violations of cost budget	% Budget Used	Warm-Cold Mismatches
1.71 s	5.65 %	8 / 600 = 1.33 %	86 %	5 / 600 = 0.83 %

We present the results in Table V. Our latency prediction error is 5.65%. While this is larger than the 0.34% prediction error observed in simulations, the prediction accuracy is still quite high. Also, we find that the total actual cost is under the total budget, with  $\approx 86\%$  of the total budget used. The warm start/cold start prediction error is also low, at 5 mis-predictions out of 600 inputs. These results suggest that our framework works well in practice. Finally, we note that when the same input workload is processed only using the edge pipeline, the average end-to-end latency is 2404 s due to queuing and is impractical compared to 1.71 s with cloud offload.

## VII. RELATED WORK

Various approaches for task placement and computation offloading have been proposed in the context of mobile cloud computing in recent years using static program analysis and annotations [7], [5], [27], as well as data flow graph-based dynamic partitioning [24]. In these works, tasks are offloaded from mobile devices to either VMs in the cloud or to remote servers. The approaches in [7], [5] formulate the offloading problem as an ILP, whereas [27], [24] depend on carefully designed greedy heuristics. More recently, offloading of neural network computation has been explored [16], [15]. In these works, deep neural network layers are partitioned across edge devices and cloud servers and executed collaboratively to satisfy latency, accuracy, and energy-saving objectives.

Cloud performance optimization in context of VMs has been studied extensively through VM allocation [31], VM performance characterization [3], and autoscaling [20], [25]. In contrast, we use serverless functions as the cloud offload destination. This imposes many behavioral constraints, for example, serverless is a stateless and event-based computation model, while VMs or servers are long-lived and stateful.

Several recent works have studied performance characteristics of serverless systems across different industry platforms. The authors in [21] proposed their own serverless platform and compared its execution performance with industry platforms. Extensive studies has been done on scalability of platforms, function latency, infrastructure retention, infrastructure provisioning [19], [32], [10], impact of language runtime on function performance [14], and latency of edge serverless platforms [8]. [13] uses serverless functions to handle incoming workloads for the duration it takes to allocate sufficient VMs to minimize SLA violations. The work [17] tackles the problem

of executing jobs in microservices under a SLA constraint from a platform provider’s perspective by maximizing the utilization of provider hardware. In contrast, we study methods to reduce cost or end-to-end latency from a client’s perspective in an edge-cloud system.

Finally, the authors in Costless [9] also present an algorithm that uses serverless functions for computational offloading from the edge. Their work however focuses on efficient partitioning of a chain of functions comprising an application, where some functions execute on the edge and some in the cloud, to reduce execution cost. In contrast, we focus on data-driven predictive offload decision making, characterizing the effect of warm and cold starts, and lastly, on performance maximization with multiple objectives using different types of real-world applications.

### VIII. CONCLUSION

We have presented a performance optimization framework for serverless applications in an edge-cloud platform. As part of this framework, we have developed models for accurately predicting end-to-end latencies and cost for functions running in the cloud or the edge. We provide a simulation-based evaluation of our framework on three representative applications. The best configurations achieved less than .3% absolute cost prediction error when minimizing total cost and less than .4% absolute latency prediction error when minimizing average latency. We also present results of live experiments, run in AWS, using the face detection application. Our evaluation shows that our framework can predict end-to-end latency with less than 6% error and obtain almost three orders of magnitude average end-to-end latency minimization compared to a naive edge execution. In future work, we will expand our prediction methods to explicitly incorporate the high variance sometimes observed in serverless platforms.

### ACKNOWLEDGMENT

This work is supported by the National Science Foundation under grants IIS 1900977, CNS 1553340 and CNS 1816307, and an AWS Cloud Credits for Research grant.

### REFERENCES

- [1] Amazon Web Services. AWS Greengrass Developer Guide, 2019. [Online; Accessed Dec 2019].
- [2] Amazon Web Services. AWS Lambda Pricing. <https://aws.amazon.com/lambda/pricing/>, February 2019. [Online; Accessed December 2019].
- [3] A. J. Awan, M. Brorsson, V. Vlassov, and E. Ayguade. Performance characterization of in-memory data analytics on a modern cloud server. In *Proc. Int. Conf. on Big Data and Cloud Computing*, pages 1–8, 2015.
- [4] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, et al. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*, pages 1–20. Springer, 2017.
- [5] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proc. of the 6th Conf. on Computer systems*, pages 301–314, 2011.
- [6] CMU Pocketsphinx Python. <https://github.com/cmusphinx/pocketsphinx-python>, 2019.
- [7] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. Maui: making smartphones last longer with code offload. In *Proc. 8th Int. conf. on Mobile systems, applications, and services*, pages 49–62, 2010.
- [8] A. Das, S. Patterson, and M. Wittie. Edgebench: Benchmarking edge computing platforms. In *IEEE/ACM Int. Conf. on Utility and Cloud Computing Companion*, pages 175–180, 2018.
- [9] T. Elgamal. Costless: Optimizing cost of serverless computing through function fusion and placement. In *IEEE/ACM Symp. on Edge Computing*, pages 300–312, 2018.
- [10] K. Figiela, A. Gajek, A. Zima, B. Obrok, and M. Malawski. Performance evaluation of heterogeneous cloud functions. *Concurrency and Computation: Practice and Experience*, 30(23):e4792, 2018.
- [11] J. H. Friedman. Stochastic gradient boosting. *Computational statistics & data analysis*, 38(4):367–378, 2002.
- [12] A. Gallagher and T. Chen. Understanding images of groups of people. In *Proc. CVPR*, 2009.
- [13] J. R. Gunasekaran, P. Thinakaran, M. T. Kandemir, B. Urgaonkar, G. Kesidis, and C. Das. Spock: Exploiting serverless functions for slo and cost aware resource procurement in public cloud. In *IEEE 12th Int. Conf. on Cloud Computing*, pages 199–208, 2019.
- [14] D. Jackson and G. Lynch. An investigation of the impact of language runtime on the performance and cost of serverless functions. In *Proc. IEEE/ACM Int. Conf. on Utility and Cloud Computing Companion*, pages 154–160, 2018.
- [15] H.-J. Jeong, H.-J. Lee, C. H. Shin, and S.-M. Moon. Ionn: Incremental offloading of neural network computations from mobile devices to edge servers. In *Proc. ACM Symp. on Cloud Computing*, pages 401–411, 2018.
- [16] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *ACM SIGARCH Comput. Archit. News*, 45(1):615–629, Apr. 2017.
- [17] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang. Grandslam: Guaranteeing slas for jobs in microservices execution frameworks. In *Proc. 14th EuroSys Conf. 2019*, page 34. ACM, 2019.
- [18] D. E. King. Dlib-ml: A machine learning toolkit. *Journal of Machine Learning Research*, 10:1755–1758, 2009.
- [19] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara. Serverless computing: An investigation of factors influencing microservice performance. In *IEEE Int. Conf. on Cloud Engineering*, pages 159–169, 2018.
- [20] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *J. Grid Computing*, 12(4):559–592, Dec 2014.
- [21] G. McGrath and P. R. Brenner. Serverless computing: Design, implementation, and performance. In *IEEE 37th Int. Conf. on Distributed Computing Systems Workshops*, pages 405–410, June 2017.
- [22] Microsoft Azure. Azure IoT Edge Documentation, 2019. [Online; Accessed Dec 2019].
- [23] 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 2019].
- [24] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan. Odessa: enabling interactive perception applications on mobile devices. In *Proc. 9th Int. Conf. on Mobile systems, applications, and services*, pages 43–56, 2011.
- [25] N. Roy, A. Dubey, and A. Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *Proc. IEEE 4th Int. Conf. on Cloud Computing*, pages 500–507, 2011.
- [26] M. Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, Jan 2017.
- [27] C. Shi, K. Habak, P. Pandurangan, M. Ammar, M. Naik, and E. Zegura. Cosmos: computation offloading as a service for mobile devices. In *Proc. 15th ACM Int. Symp. on Mobile ad hoc networking and computing*, pages 287–296. ACM, 2014.
- [28] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet Things J.*, 3(5):637–646, Oct 2016.
- [29] Tatoeba. <https://tatoeba.org/eng/>, 2019.
- [30] Time Machines GPS NTP+PTP Network Time Server (TM2000A). <https://timemachinescorp.com>, 2019.
- [31] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *Proc. USENIX Symp. on Networked Syst. Design and Implementation*, pages 363–378, 2016.
- [32] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. Peeking behind the curtains of serverless platforms. In *USENIX Annual Technical Conference 18*, pages 133–146, 2018.