Edge-Adaptable Serverless Acceleration for Machine Learning IoT Applications

Michael Zhang, Chandra Krintz, and Rich Wolski
Dept. of Computer Science
Univ. of California, Santa Barbara
{lebo, ckrintz, rich}@cs.ucsb.edu

Abstract

Serverless computing is an emerging event-driven programming model that accelerates the development and deployment of scalable web services on cloud computing systems. Though widely integrated with the public cloud, serverless computing use is nascent for edge-based, IoT deployments.

In this work, we present STOIC (Serverless TeleOperable HybrId Cloud), an IoT application deployment and offloading system that extends the serverless model in three ways. First, STOIC adopts a dynamic feedback control mechanism to precisely predict latency and dispatch workloads uniformly across edge and cloud systems using a distributed serverless framework. Second, STOIC leverages hardware acceleration (e.g. GPU resources) for serverless function execution when available from the underlying cloud system. Third, STOIC can be configured in multiple ways to overcome deployment variability associated with public cloud use. We overview the design and implementation of STOIC and empirically evaluate it using real-world machine learning applications and multi-tier IoT deployments (edge and cloud). Specifically, we show that STOIC can be used for *training* image processing workloads (for object recognition) – once thought too resource-intensive for edge deployments. We find that STOIC reduces overall execution time (response latency) and achieves placement accuracy that ranges from 92% to 97%.

Keywords— serverless, IoT, scheduling, cloud functions, GPU

1 Introduction

Serverless computing (also known as Functions-as-a-Service (FaaS)) [1, 2, 3] is a popular cloud service for hosting and automatically scaling applications. Originally designed for web services [4, 5], serverless computing defines a simple, event-driven programming model and cloud platform with which developers write simple, short-lived functions that are invoked by the platform in response to specific system-wide events (e.g. storage updates, notifications, messages received, changes in state, custom events, etc.). Serverless platforms automatically configure and provision isolated execution environments (typically via Linux containers) on-demand and users pay only for the resources their functions use during execution. Given its success to date, public cloud providers and open source communities have released multiple serverless platforms with similar functionality [1, 3, 6, 7, 8, 9].

Moreover, serverless computing has been extended to work at the "edge" of the network to reduce the response latency and bandwidth associated with public cloud use by data-driven applications (e.g. those that target the Internet of Things (IoT)) [10, 11, 12]. Doing so is challenging, however, because computing and storage resources are scarce at the edge relative to resource-rich public and private clouds. Moreover, public/private clouds may offer specialized hardware (e.g. GPUs) that can significantly speed up machine learning applications, which is not commonly available in resource-restricted edge clouds.

In this paper, we investigate the use of serverless computing across the edge and public cloud deployments (hybrid cloud settings). We develop a scheduling system, called the Serverless TeleOperable Hybrid Cloud (STOIC), which automatically places and deploys functions across these systems aiming to reduce the total execution time latency (versus using either system in isolation). We specifically target image-based, object recognition using Tensorflow (for training and inference) in this work.

STOIC automatically places serverless functions at the edge (without GPUs) or in public cloud instances (equipped with 1+ GPUs) using predicted latency. We use the system to perform online training and inference for batches of images from motion-triggered, camera traps that capture images of wildlife in remote locations, with intermittent Internet connectivity.

STOIC has two placement scenarios: the first places functions only at the runtime with the least predicted latency, whereas the second places functions concurrently at both edge and public cloud, but then terminates public cloud execution if/when it determines that the edge will finish sooner. The former scenario is called *Selector* mode. The latter scenario, called *Duplicator* mode, is useful when the cloud and/or network performance used for deployment is intermittent or highly variable, or when executing at the edge incurs no cost or other penalty – to ensure that progress is made. Our results show that STOIC speeds up the total response time of the application by 3.3x versus a baseline scenario. In selector

mode, STOIC achieves a placement accuracy of 92% relative to the optimal placement. In duplicator mode, STOIC accuracy is 95% for 2 GPUs and 97% (versus optimal) for 1 GPU cloud deployments over a 24-hour period.

In summary, with this paper, we make the following contributions.

- We design and implement a serverless framework that spans heterogeneous edge and cloud systems, serving IoT requests, and leveraging GPU acceleration;
- We investigate feedback control mechanism and various analytical methodologies to precisely model the unstable edge and public cloud environments; and
- We empirically evaluate the efficacy of using this extended serverless model for machine learning applications and IoT deployments.

In the following sections, we first discuss the related work (Section 2). We then present the design and implementation of STOIC (Section 3), following by our experimental methodology and empirical evaluation of the system and application workloads, using a distributed serverless deployment (Section 4). Finally, we present our conclusions and future work plans (Section 5).

2 Related Work

We have explored an initial design and scheduler for STOIC in [13]. The work herein extends this early work with a new scheduling system and consideration of both individual and concurrent edge-cloud placements.

A significant body of work [14, 15, 16] has explored low-latency geo-distributed data analytics and mobile-cloud offloading – which we take as inspiration for the STOIC design. One relevant approach is federated learning [17], by which a comprehensive model is trained across heterogeneous edge devices or servers without exchanging local data samples. Federated learning aims to address the security and networking concerns by keeping the datasets local at devices, whereas STOIC intelligently offloads jobs across multiple tiers of cloud infrastructure to further reduce latency.

In addition, STOIC targets IoT systems and leverages serverless computing and GPUs. As such, other related work includes recent advances in machine learning infrastructure, serverless computing, GPU accelerators, and container-based orchestration services. [18] and [19] conduct a comprehensive survey on serverless computing including challenges and research opportunities. We share the same viewpoint that the use of the serverless execution model will grow for online training and inference applications. [20] provides a prototype for a deep learning model serving in a serverless platform. [21] provides another use case for accelerating serverless functions by GPU virtualization in data centers. Unique in our work, STOIC extends an existing serverless framework to support GPU

	DECENTER	**HCL-BaFog**	**STOIC**
Node Selection	FoQoSAM	MultiChain	Dynamic Feedback Loop
Orchestration	Kubernetes	Docker Swarm	Kubeless
Quality of Service	latency/throughput/availability	latency/availability	latency/availability
Trust Mgmt	Smart Contracts	Blockchain	Nautilus
Application	Video Streaming	Sensor Data Sharing	Image Recognition

Table 1: The comparison table of DECENTER, HCL-BaFog and STOIC.

acceleration and distributed function placement across the edge and public clouds. [22] evaluates several serverless frameworks that use Kubernetes to manage and orchestrate use of Linux containers. STOIC also integrates Kubernetes for container orchestration, which is lightweight, flexible, and developer-friendly. We concur that Kubernetes is a promising deployment infrastructure for serverless computing.

Another relevant domain of related work is edge-to-cloud infrastructure enabling IoT device applications. [23] compares the processing time of face recognition between the edge device and smartphones. It concludes that edge devices perform comparably faster and scales better as the number of images increases. We agree with this conclusion, and as such, we design STOIC to offload image processing workloads to both edge clouds and public clouds. [24] proposes a distributed deep neural network that allows fast and localized inference at the edge device using truncated layers of a neural network. [25] defines edge cloud offloading as a Markov decision process (MDP) whose objective is to minimize the average processing time per job. Based on this setting, it provides an approximate solution to MDP with a one-step policy iteration. Similar to this approach, [26] proposes a Global Cluster Manager for orchestrating network-intensive programs within Software-Defined Data Centers (SDDCs) targeting high Quality of Service (QoS) and, further, [27] classifies available cloud deployment options by a stochastic Markov model, namely Formal QoS Assurances Method (FoQoSAM), to optimize the automated offloading process. Due to its practical utility, such a method can guarantee that QoS requirements are satisfied. [28] proposes a fog computing platform (DECENTER) and a trust management architecture based on Smart Contracts. Related to this work, [29] develops an architecture (HCL-BaFog) by the blockchain functionality to share sensor data. Table 1 summarizes the properties of DECENTER, HCL-BaFog, and STOIC. These works are complementary to STOIC and we are considering how to incorporate them into the system as part of future work.

Also complimentary to STOIC, are tracing, testing, repair, and profiling tools (which STOIC can leverage) for serverless systems. Multiple works track causal dependencies across distributed serverless deployments for use in optimization, placement, and data repair [30, 31, 32, 33]. FaaSProfiler [34] integrates testing and profiling within a FaaS platform. [35] proposes a security solution that applies reinforcement learning (RL) to provide secure offloading to the edge nodes to prevent jamming attacks. These related

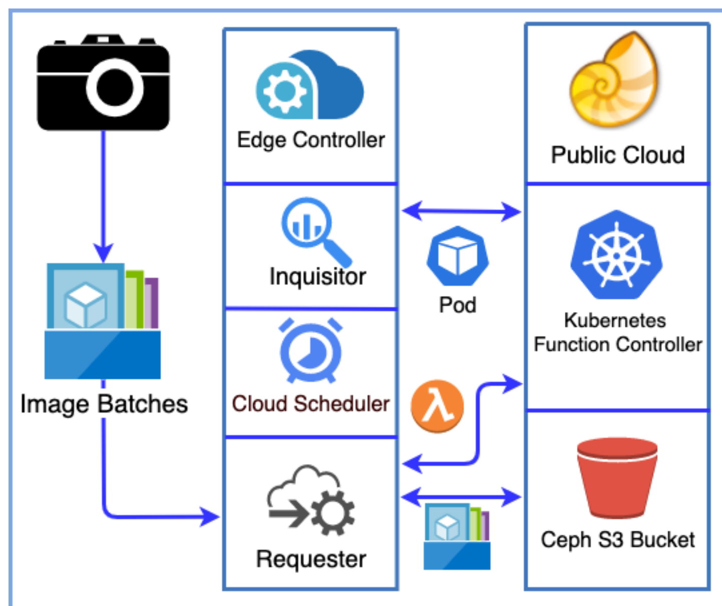

Figure 1: The STOIC Architecture

systems can be combined with STOIC to provide a robust serverless ecosystem for distributed IoT devices.

3 STOIC

To leverage hardware acceleration and distributed (multi-cloud) scheduling within a serverless architecture, we have developed STOIC, a framework for distributing and executing analytics applications across multi-tier IoT (sensing-edge-cloud) settings. Specifically, STOIC optimizes the end-to-end process of packaging, transferring, scheduling, executing, and result retrieval for machine learning applications in these settings.

Figure 1 shows the distributed components of STOIC. At the edge, STOIC gathers application input data, determines whether the lower application latency will be achieved by processing the data on the edge or in the cloud, and then actuates the application's computation (with the necessary data) using the "best" choice. The public cloud component manages whatever cloud resources are needed to receive the data from the edge, trigger the computation, and return the results to the edge. The edge and cloud systems mirror each other, running Kubernetes [36, 37] overlaid with kubeless [38], to provide a uniform infrastructure for the framework.

Our system design is motivated by a need to classify wildlife images in a location

where it is possible to site a relatively powerful edge system but where network connectivity is poor. In this paper, we report on the use of STOIC for processing images from multiple, motion-triggered camera traps (sensors) deployed to a wildlife reserve currently used to study ecological land use.

3.1 Edge Controller

The STOIC edge controller is a server that runs in an out-building at the reserve. It communicates wirelessly with the sensors and triggers analysis and computation upon their arrival. The edge controller is connected to a research facility (which has full Internet connectivity) via a microwave link. When a camera trap detects motion, it takes photos and persists the images in flash storage buffer, where human experts would label images for training tasks. Periodically, sensors transfer saved photos to the edge controller. During a transfer cycle, the edge controller compresses and packages all images generated and transfers the package to the public cloud, if/when necessary. STOIC runs on the edge controller and its executions are triggered by the arrival of image batches.

As an intermediate computational tier between the sensors and the public cloud, the edge controller can be placed anywhere, preferably near the edge devices, to lower the response latency for the data processing and analytics applications. It consists of three major components:

- The **cloud scheduler** predicts the total latency based on historical measurements for each available runtime.
- The **requester** takes as input the runtime and cloud predicted by the scheduler to have the least latency. The requester stores the image package in an object storage service running in this cloud. It then triggers a serverless function (running in a Kubernetes pod) via a RESTful HTTP request to process the images.
- The **inquisitor** monitors public cloud deployment time. To enable this, it periodically in the background deploys each runtime (using Kubernetes pods [39]) and records the deployment times in a database. No task/process is executed in this process (the runtime is simply deployed and taken down). We use the inquisitor to establish the historical time series for predicting the deployment latency of remote runtimes.

The edge cloud that we use in this study is deployed at a research reserve and is connected via the Internet. It consists of a cluster of three Intel NUCs [40] (6i7KYK), each with two Intel Core i7-6770HQ 4-core processors (6M Cache, 2.60 GHz) and 32GB of DDR4-2133+ RAM connected via two channels. The cluster is managed using the Eucalyptus cloud system [41], which mirrors the Amazon Web Services (AWS) interfaces for Elastic Compute Cloud (EC2) to host Linux virtual machine (VM) instances and Simple

Storage Service (S3) to provide object storage. The STOIC edge runtime uses Kubernetes and kubeless for serverless function execution and S3 (i.e. walrus) for object storage on the edge cloud.

3.2 Public/Private Cloud

To investigate the use of the serverless architecture with hardware acceleration, we employ a shared, multi-university, GPU cloud, called Nautilus [42], as our remote cloud system. Nautilus is an Internet-connected, HyperCluster research platform developed by researchers at UC San Diego, the National Science Foundation, the Department of Energy, and multiple, participating universities globally. Nautilus is designed for running data and computationally intensive applications. It uses Kubernetes [37] to manage and scale containerized applications. It also uses Rook [43] to integrate Ceph [44] for object storage. As of May 2020, Nautilus consists of 176 computing nodes across the US and a total of 543 GPUs in the cluster. All nodes are connected via a multi-campus network. In this study, we consider Nautilus a public cloud that enables us to leverage hardware acceleration (GPUs) in the serverless architecture. The STOIC cloud/GPU runtimes use Kubernetes and kubeless for serverless function execution and Ceph for object storage on the public cloud.

A major challenge that we face with such deployments is hardware heterogeneity and performance variability. On Nautilus, we have observed 44 different types of CPU (e.g. Intel Xeon, AMD EPYC, among others) and 9 GPU types (e.g. Nvidia 1080Ti, K40, etc.). Both CPUs and GPUs of different types have different performance characteristics. Moreover, the object storage service is run on dedicated nodes that are distributed globally.

This heterogeneity impacts application execution time (which STOIC attempts to predict) in three significant ways. First, different CPU clock rates affect the transfer of datasets from the main memory to GPU memory. Second, there is significant latency and performance variability between runtimes and the storage service (which hold the datasets and models). Third, the multi-tenancy of nodes (common in public cloud settings) allows other jobs to share computational resources with our applications of interest at runtime.

These three factors negatively make it difficult for users to determine which runtime to use (to reduce application turn-around time) and when to execute locally (avoiding public cloud use altogether). With STOIC, we address these challenges via a novel scheduling system that adapts to this variability. In our results, we ensure reproducibility (avoiding network performance variability) by confining nodes and GPUs (still heterogeneous) to a single Nautilus region.

3.3 Runtime Scenarios

To schedule machine learning tasks across hybrid cloud deployments, we define four runtime scenarios: **(A)** *Edge* - A VM instance on the edge cloud with AVX2 [45] support; **(B)** *CPU* - A Kubernetes pod on Nautilus containing a single CPU with AVX2 support [45]; **(C)** *GPU1* - A Kubernetes pod on Nautilus containing a single GPU; **(D)** *GPU2* - A Kubernetes pod on Nautilus containing two GPUs. STOIC considers each of these deployment options as part of its scheduling decisions. Users can parameterize STOIC with their choice of deployment or allow STOIC to automatically schedule their applications.

3.4 Execution Time Estimation

As depicted in Figure 1, the STOIC's edge controller listens for image batches from the remote camera traps and makes machine learning job requests. After a preset period (parameterizable but currently set to an hour), STOIC estimates total response time (T_s) of a requested batch, based on 4 different runtime scenarios. The total response time (T_s) includes data transfer time (T_t), runtime deployment time (T_d), and the corresponding processing time (T_p). We define total response time (T_s) as $T_s = T_t + T_d + T_p$.

3.4.1 Transfer time (T_t)

T_t measures the time spent in transmitting a compressed batch of images from the edge controller to edge cloud and public cloud. We calculate transfer time as $T_t = F_b/B_c$ where F_b represents the file size of batch and B_c represents the bandwidth at the moment provided by a bandwidth monitor at the edge controller.

3.4.2 Runtime deployment time (T_d)

T_d measures the time Nautilus uses to deploy requested kubeless function. Since the scarcity of computation, it is common that multi-GPU runtime takes longer to deploy than single-GPU and CPU runtimes. Note that, for *edge* runtime, the deployment time zeroes out since STOIC executes the task locally in the edge cloud.

Because Nautilus is a shared cloud system, we observe significant variation in deployment time on Nautilus for different times of the day. To accurately predict deployment time, we analyze deployment times as a time series using three methods: (1) autoregression modeling, (2) average sliding window, and (3) median sliding window. Autoregression [46] is a time series modeling technique based on the auto-correlation between previous time steps and the following ones. The average sliding window is the moving average [47] scanning through the time series by a fixed-length window. Similarly, the median sliding window captures the moving median cross the time series. All window

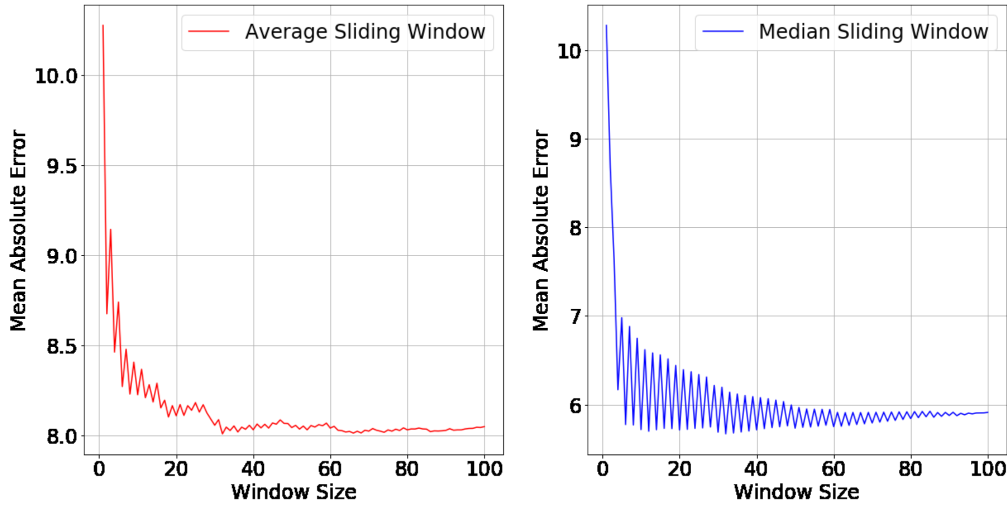

Figure 2: The Mean Absolute Error (MAE) of deployment time for the GPU1 runtime. The x-axis is the window (history) size. The left subplot is MAE when STOIC uses the average sliding window, the right subplot is MAE when STOIC uses the median sliding window.

Modeling	Runtime	Optimal Window Size	Minimum MAE
AutoReg	CPU	15	8.977
AutoReg	GPU1	15	9.605
AutoReg	GPU2	15	17.918
Avg. SW	CPU	33	7.714
Avg. SW	GPU1	31	8.006
Avg. SW	GPU2	91	16.52
Med. SW	CPU	13	**5.96**
Med. SW	GPU1	31	**5.668**
Med. SW	GPU2	27	**14.48**

Table 2: Mean Absolute Error of three time series modeling methods for runtime deployment time: auto-regression (AutoReg), average sliding window (Avg. SW), and median sliding window (Med. SW). The median sliding window achieves the lowest minimum MAE at optimal window size (that with the lease MAE) for all three runtimes.

sizes used for three modeling processes are optimized based on historical data of deployment time (T_d) in January 2020. We then compare the minimum Mean Absolute Error (MAE) from each to select the best modeling methodology.

In this example, we consider a time series of 1244 data points for each runtime. Figure 2 shows representative analytics for GPU1 deployment time, in which MAE oscillates as window size varies. We observe that the median sliding window reaches a lower minimum MAE than the average sliding window at optimal window size. As listed in Table 2, all three runtimes achieve the lowest minimum MAE using the median sliding window. Therefore, STOIC adopts this methodology for deployment time prediction.

The inquisitor measures and records deployment time for each public cloud runtime every minute (called the inquisitor period). After the inquisitor records 10 new measurements (called the calibration period), the scheduler recomputes the window size over the previous 100 measurements that result in the minimum MAE. It then uses this minimum MAE window size to estimate of deployment time when jobs arrive. The inquisitor period, calibration period, and maximum window size are all modifiable.

3.4.3 Processing time (T_p)

T_p is the execution time of a specific machine learning task and the target of task scheduling across the hybrid cloud. STOIC formulates a linear regression on execution time histories of STOIC jobs, and uses it to predict processing time relative to input (image batch) size. Specifically, we use Bayesian Ridge Regression [48] due to its robustness to ill-posed problems (relative to ordinary least squares regression [49]). STOIC queries the database for the most recent processing time data (e.g. 10 data points) for each regression. This ensures that the parameters of the regression line reflect the current runtime performance.

As part of our investigations into this approach, we have found that this approach is highly susceptible to outliers. The root cause of these outliers is sporadic congestion and maintenance (for nodes, networking, etc.) of the public cloud. Deviating significantly from the average, outliers skew the regression line and overestimate the runtime latency for extended periods (due to the windowing approach). We thus augment regression using a random sample consensus (RANSAC) technique [50], which iteratively removes outliers from the regression. The algorithm 1 illustrates our RANSAC approach in STOIC.

3.4.4 Adaptability

To verify that STOIC's estimation of execution time captures the actual latency of the public cloud, we execute the application 50 times with 150-image batch using the GPU1 runtime. Depicted in Figure 3, we observe that actual total latency varies significantly and predicted total latency has a non-negligible difference from the actual total latency at the

Algorithm 1: Random Sample Consensus

Data: (1) Observation set of Process time T_p
(2) Bayesian Ridge Regression model M
(3) Minimum sample size n
(4) Residual threshold t
(5) Maximum iteration k
(6) Required inlier size d
(7) Minimum Root Mean Square Error e
Result: A set of parameters that best fits the data

```
1  while iterations ≤ k do
2  |   curr_sample := n data points from observation;
3  |   curr_model := M regressed on curr_sample;
4  |   fit_data := empty set;
5  |   for every data point p in curr_sample do
6  |   |   if error of p ≤ t then
7  |   |   |   p → fit_data;
8  |   end
9  |   if fit_data size ≥ d then
10 |   |   curr_error := average error in fit_data;
11 |   |   if curr_error ¡ e then
12 |   |   |   Update M and e
13 |   else
14 |   |   Increment iteration
15 |   end
16 end
17 return M
```

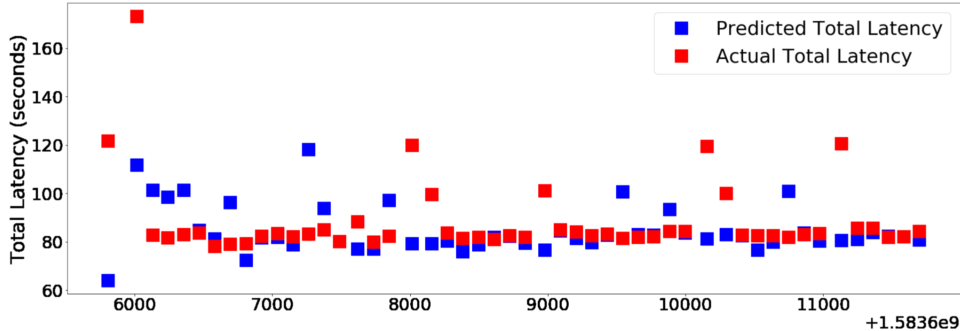

Figure 3: The comparison of predicted and actual total latency on 50 GPU1 benchmark executions with 150-image batch size. The x-axis is the epoch time and the y-axis is the total latency.

	Deployment T_d	**Processing** T_p	**Total** T_s
First Half	42.7%	11.2%	15.8%
Second Half	29.2%	5.3%	9.2%

Table 3: The percentage mean absolute error (PMAE) of deployment, processing, and total latency. PMAE is a latency-normalized metric and calculated as MAE divided by mean latency, which indicates the residual in a measured period. The decline of three latency metrics in the second half demonstrates the adaptability of STOIC.

beginning of the experiment. However, over time, as STOIC learns the various latencies of the system, the difference is significantly reduced. In Table 3, we report the percentage mean absolute error (PMAE), which we compute as the MAE divided by mean latency. The decrease in all three PMAE values in the second half of the execution trace also show STOIC's adaptability.

3.5 Workload Generation

To drive our empirical evaluation in faster-than-real time, we construct a workload generator from image batch histories (traces) collected by our camera traps. We consider the set of images that occur together within an hour (i.e. due to motion events) a batch. Our camera trap trace, starting on July 13th, 2013 and ending Jan. 15th, 2017, comes from a fixed camera located at a watering hole in a remote area of our research reserve. The trace contains images of bear, deer, coyote, puma, and birds as well as wind-triggered empty images and other animals.

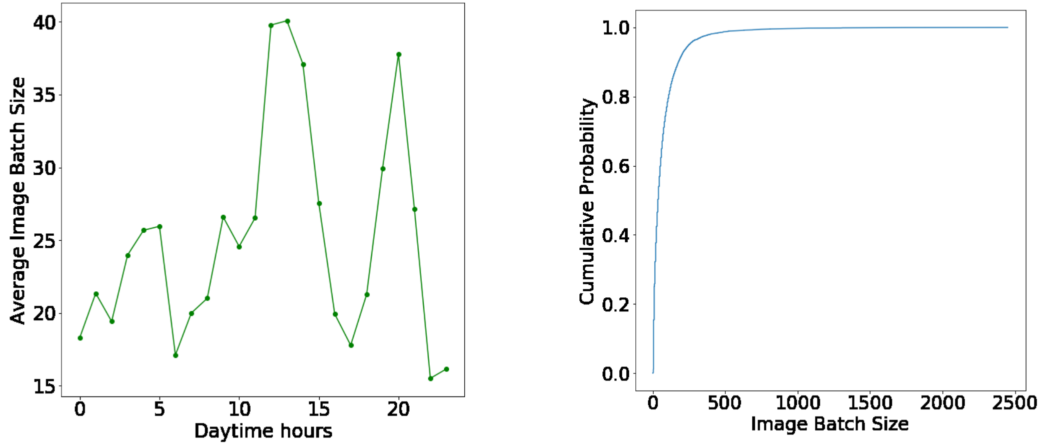

Figure 4: Wildlife Hourly Activity Level (left graph) and its Conditional Empirical Cumulative Distribution Function (right graph). The left graph demonstrates the mean activity level of wildlife throughout the daytime. Based on the curve, 1 PM and 8 PM are two peak hours of animal activities. The right graph shows the empirical CDF, which STOIC randomly samples for image batches to drive our faster-than-real time empirical evaluation of the system.

After excluding camera maintenance periods (gaps), we extract 1136 effective days (27264 hours) of data. The maximum size of hourly image batch is 2450, whereas the minimum size is unsurprisingly zero, which constitutes 18139 hours out of 27264 hours (66.53%). On average, an hourly image batch size contains 25 images. The left graph in Figure 4 illustrates the wildlife hourly activity level based on the image batch size. We infer from the curve that 1 PM and 8 PM are two peak hours of animal activity.

Specifically, we construct a conditional empirical cumulative distribution function (ECDF) based on the probability definition of $Pr(x < K|x > 0)$, where x is the image batch size and K is the cutoff value. This conditional ECDF effectively represents the trajectory of the animal activity level and makes the evaluation empirical. The right graph in Figure 4 plots the conditional ECDF. The x-axis is the image batch size ranging from zero to 2450, whereas the y-axis is the cumulative probability. The STOIC workload generator draws image batch sizes by randomly sampling this ECDF. Using this process, we are able to evaluate and conclude by replaying the image stream from the camera traps in fast-than-real time for the purposes of comparative evaluation.

3.6 Implementation

We implement STOIC using Golang [51]. Golang provides high-performance execution (vs scripting languages) and a user-friendly interface [52] to Kubernetes and database technologies. STOIC currently supports machine learning applications developed using the TensorFlow framework [53] and can be easily extended to permit other machine learning libraries.

As mentioned previously, the STOIC serverless architecture leverages kubeless [38]. As a Kubernetes-native serverless framework, kubeless uses the Custom Resource Definition (CRD) [54] to dynamically create functions as Kubernetes custom resources and launches runtimes on-demand. For specific machine learning tasks that STOIC executes, we build custom Docker images that we upload to Docker Hub [55] in advance. When the function controller receives a task request, it pulls the latest image from Docker Hub before launching the function. This deployment pipeline makes the runtime flexible and extensible for evolving applications.

To leverage the computational power of our CPU systems, we compile Tensorflow with AVX2, SSE4.2 [45], and FMA [56] instruction set support. We use this optimized version of Tensorflow on both the edge and public clouds.

To enable GPU access by serverless functions (available in the public cloud), we equip our Docker container with NVIDIA Container Toolkit [57]. This includes the NVIDIA runtime library and utilities, which link serverless functions to NVIDIA GPUs. We also install CUDA 10.0 and cuDNN 7.0 to support the machine learning libraries.

3.7 Workflow

STOIC considers two workflows upon receiving an image batch: selector mode and duplicator mode. Both are depicted in Figure 5. In selector mode, STOIC predicts the total response times (T_s) of the four deployment options: Edge, CPU, GPU1, and GPU2. It then selects the runtime with the shortest estimated response time and deploys it locally (Edge) or remotely (non-Edge). Once deployed, the pod notifies the STOIC requester at the edge which then triggers the serverless function via an HTTP request. When the task completes, the pod notifies the requester, which retrieves the results and runtime metrics from the deployment and stores them in the database for use by the scheduler.

To handle deployment failure, STOIC implements a retry mechanism using exponential back-off. Starting at 100 milliseconds, STOIC waits 2X length of time for retrying the deployment on Nautilus. After 10 failed attempts, STOIC claims timeout and returns an error.

STOIC also attempts to reduce startup time (i.e. cold starts) at both the edge and public cloud. On the edge cloud, STOIC creates a standby pod to serve the incoming request upon application invocation. On the public cloud, STOIC triggers a function with

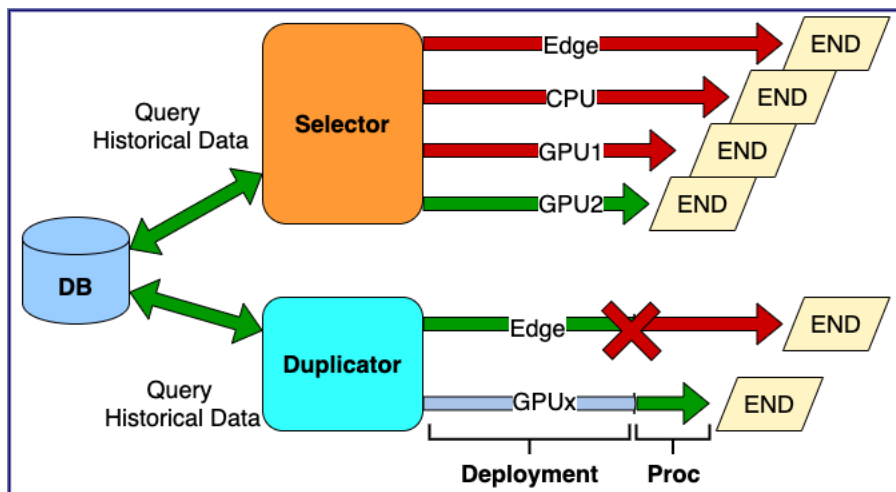

Figure 5: The selector and duplicator modes of STOIC.

a single image to retrieve and cache the base model in memory at each pod.

We observe from Table 3 that there are significant variations in the deployment time of the runtimes on the shared public cloud. To enable STOIC to adapt to this variability, we consider a second workflow called **duplicator** mode. Using this mode, when the scheduler selects a public cloud runtime (i.e. CPU, GPU1, GPU2), the requester *also* deploys the job on the edge cloud. It then terminates edge cloud execution if the remaining time at edge cloud is longer than the expected processing time (T_p) at the GPU runtime once deployment completes. This "lagging decision" mechanism reduces the variability of deployment time in the prediction. As a result, STOIC must only consider processing time, which is more accurately predicted, to deploy tasks. Note that duplicator mode is less energy-efficient because it runs tasks regardless of latency prediction and may waste cloud resources by killing the function in the middle. However, if such waste can be tolerated, significant prediction accuracy and latency reduction are possible.

In addition, the inquisitor running in the background deploys the public cloud runtimes periodically and stores the deployment time duration in the database for use in the prediction. We set a timeout (i.e. 10 minutes) to terminate this process for any unresponsive deployment. That is, the inquisitor marks the runtime unavailable (from the point of view of the requester) when the deployment hits the set timeout. The inquisitor continues to attempt deployment of this runtime periodically and makes it available to the requester once a deployment attempt is successful.

To bootstrap the system, STOIC executes two representative tasks for an application for each runtime in both the edge and public cloud. It uses these data points as a basis for its processing time estimation by linear regression. STOIC performs this bootstrapping

each time a new version of the application is uploaded by the developer.

4 Evaluation

In this section, we empirically evaluate STOIC's performance on image processing tasks. We implement the application as a serverless function for STOIC to schedule and execute.

In each experiment, STOIC determines which resource to use for function execution (among a small set of feasible choices). We then run the function on *all* resources and compare the choice made by STOIC to the best (shortest duration) execution across all possible choices.

4.1 Experimental Setup

The image processing application that we use as a benchmark classifies animal images from a wildlife monitoring system called "Where's The Bear" (WTB) [58]. "Where's The Bear" is an end-to-end distributed data acquisition and analytics system that automatically analyzes camera trap images collected by cameras sited at the Sedgwick Natural Reserve [59] in Santa Barbara County, California. Our deployment includes an edge cloud located near the cameras where it acquires the image data. The edge cloud is connected via a slow (microwave) link to a private cloud located at a research facility located approximately 50 miles from the site. In this work, we explore using the Nautilus distributed GPU cloud [42] as the public cloud, in conjunction with the edge cloud to optimize image classification on a convolutional neural network (CNN) [60] implemented by Tensorflow and Scikit-learn [61].

In total, there are five classes that we consider: Bird, Fox, Rodent, Human, and Empty, by which we label images for training tasks and evaluate model by inference. Since class size is unbalanced due to the frequency of animal occurrences, we up-sample minority classes (e.g. fox) using the Keras ImageDataGenerator [62]. Doing so ensures that the classification model is not biased. We resize every image in the image dataset to 1920×1080, and for each class, the dataset contains 251 images used to train the CNN model. Once model training is complete, the application stores this model in hdf5 format in object storage at both edge cloud and Nautilus.

As described previously, STOIC moves images from the wildlife refuge to the public cloud in batches. To better harness the multiple GPU runtime of the public cloud, the application spawns a process (worker) for each GPU and adds all images in a batch to a shared asynchronous queue. Upon the execution, workers remove images (one at a time) from the shared queue until it is exhausted. This mechanism ensures multiple GPU runtimes evenly divide the workloads among GPUs and achieve quasi-linear acceleration at the application level, where the perfect linear speed-up is unattainable because of model

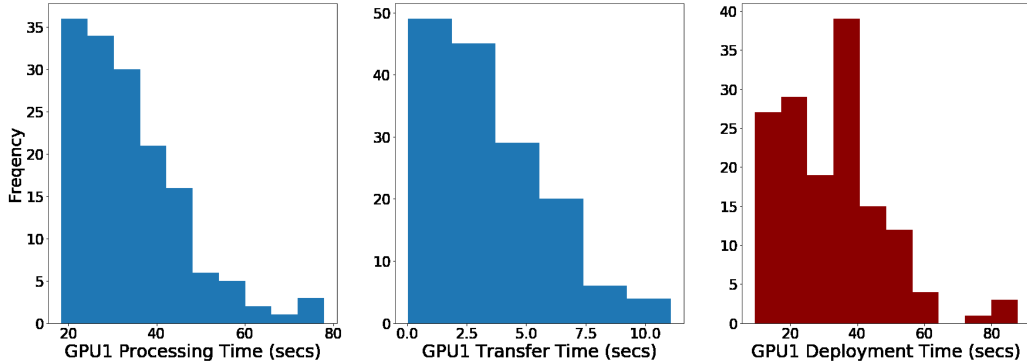

Figure 6: The distribution of three components in total response time (T_s) of 150 executions on GPU1 runtime: Processing time (T_p), Deployment time (T_d), and Transfer time (T_t). The x-axis represents the time range, while the y-axis is the frequency of executions. The deployment time, which is depicted in the red histogram, is volatile and error-prone to prediction.

loading and memory transfer overhead [63].

To drive this experiment, we use the workload generator described in Section 3.5 to facilitate faster-than-real time evaluation of STOIC. The generator uses an image series and their inter-arrival patterns from a camera trap image corpus ranging from 2013 to 2017. Figure 6 shows example histograms for processing time, transfer time, and deployment time on Nautilus for GPU1 runtime using 150 batches drawn from the workload generator. On the x-axis, we show the elapsed time for processing time, transfer time, and deployment time respectively. Note that processing time and transfer time are relatively stable compared to deployment time.

4.2 Selector Evaluation

We first evaluate STOIC selector mode for a 24-hour period consisting of 162 image batches, the sizes of which are drawn randomly from the workload generator. Each batch is executed on the edge cloud, on the Nautilus CPU, on one Nautilus GPU, and two Nautilus GPUs. Over the test period, the STOIC Selector chooses the fastest (lowest total response time) from among these four options 149 times out of the 162 runs or 92% of the time. That is, STOIC correctly identifies the fastest option with a success rate of 92%.

Further, we define MIN-LAT (minimum latency scheduler), which is an oracle scheduler that is 100% correct on selections of runtime. Such scheduler would have resulted in an aggregate total latency of 10022 seconds, whereas the worst case, in which the scheduler selects the highest-latency runtime for every run, has an aggregate latency of

35940 seconds, compared to a STOIC aggregate latency of 10770 seconds. Thus STOIC achieves an aggregate latency that is 7.4% slower than MIN-LAT, but 70% (3.33x) faster than the worst case.

We further analyze the data points where STOIC made erroneous selections and found two sources of error. First, the most error occurs around two batch sizes where the total response times of runtime have approximately the same latency. To be specific, the edge and GPU runtimes cross over at 35 image batch size and 90 image batch size for the GPU1 and GPU2 runtimes. At these cross-points, the close predictions of latency lead to incorrect selection. [13] Second, the deployment times for GPU runtime are volatile and error-prone to prediction. As a representative instance, Figure 6 demonstrates the distribution of processing time (T_p), transfer time (T_t), and deployment time (T_d) of GPU1 runtime. We observe geometric distribution from the histogram of processing time and transfer time, whereas deployment time varies irregularly with many outliers. These two phenomenons lead to mistaken selections in the experiment.

4.3 Duplicator Evaluation

Note that the edge cloud node is not a shared resource – it is dedicated to the application. It is implemented using inexpensive hardware that is connected to standard 120 VAC power (in a closet in a management building located at the refuge). As a result, it is possible to use the edge cloud for *every* batch even when it is not the fastest.

Put another way, there is no cost to running the edge cloud speculatively while data is transferring to Nautilus and the application waits for Nautilus to deploy pods for the CPU and GPU runtimes. If STOIC (using Selector) predicts that Nautilus will be faster, and STOIC is correct, the work on the edge cloud is "duplicate work" which is unnecessary. However, because of the deployment variability, it may be that the edge cloud speculative execution finishes ahead of that runtime scheduled to Nautilus.

However, unlike the edge cloud node, Nautilus is a shared resource. Thus we do not wish to "waste" execution time on Nautilus unnecessarily. Thus, in this setting, the cost of duplicate work on the edge is minimal compared to the cost of potentially duplicate work on Nautilus. If this were not true, we would simply launch the job both at the edge and on Nautilus and use whichever finished first.

Thus we explore a second scheduling strategy that attempts to minimize total response time in light of the following assumptions:

- Duplicating unneeded work on the edge carries no penalty.
- Duplicating unneeded work in Nautilus is expensive.
- The STOIC predictions (initial and after transfer and deployment) will be used to choose the resource that yields the fastest response time while using the Nautilus resources parsimoniously.

We call the STOIC scheduler that attempts to minimize response times under these assumptions – the Duplicator.

Further, we noticed that the Nautilus CPU is seldom a good choice in practice. The application must "pay" for the transfer and incur the deployment time variability to acquire a CPU that is almost equivalent to the edge node CPU. Thus, in the "real world" version of the STOIC scheduler for the application, we use the Duplicator with Nautilus GPUs only.

The scheduling algorithm starts the task on the edge cloud node and also begins the transfer to Nautilus. It then waits for the Nautilus deployment time and, when the pod is fully deployed, it predicts whether to use the freshly acquired GPU or GPUs (i.e. to "switch" to the GPU(s)) or to abandon the request and to complete the job on the edge. To do so, STOIC must predict the *remaining* edge time at the moment the GPU pod is deployed, and compare this remaining time to the predicted GPU processing time.

The Duplicator prediction is *conditional* upon the amount of time that has elapsed during transfer and deployment to Nautilus. If STOIC predicts that the GPU pod will start and complete their processing before the edge completes what remains of the job, it allows the Nautilus and edge cloud executions to execute concurrently. If the Nautilus job completes first, the edge cloud execution is terminated. Otherwise, if the edge cloud execution finishes first (i.e. the prediction was incorrect) then the Nautilus job is terminated (and the time between the start of the Nautilus job and the end of the cloud job is "wasted" Nautilus time).

Alternatively, when STOIC predicts that the edge cloud will finish first, it returns the GPU resources to Nautilus and run only the edge cloud job. If the Nautilus job would have completed first (i.e. the conditional prediction in favor of the edge is incorrect) then the time between when the Nautilus job would have finished and the time that the edge cloud job completes is an additional delay (compared to having made a correct prediction).

Thus, choosing incorrectly (i.e. a failure) occurs when the actual completion time exceeds the time of the runtime corresponding to the minimum prediction (in either edge or GPU case) made by STOIC. That is, a "failure" for the Duplicator occurs when STOIC makes a conditional choice (i.e. continue on edge or to include Nautilus) and the choice results in a longer *actual* response time than the one not chosen. Table 4 shows the performance of the Duplicator using the edge and one GPU and, separately, the edge and two GPUs from Nautilus.

These results are both expected and surprising. As expected, restricting the choice to the edge and a single Nautilus request and using a conditional prediction at deployment time (as opposed to a ranking at the beginning) as a success criterion improves the success rate dramatically. We do not claim that Duplicator is better than Selector in terms of success rate. Instead, Duplicator enables a more dependable scheduling strategy for the classification application based on conditional predictions rather than resource ranking. Surprisingly, however, requesting 2 GPUs improves both success rate and aggregate

	Success Rate	versus MIN-LAT	versus Worst Case
Selector	92%	105%	30%
Duplicator Edge vs GPU1	97%	102%	30%
Duplicator Edge vs GPU2	95%	101%	30%

Table 4: The comparison of Selector and Duplicators. The table demonstrates that the duplicator(GPU1) achieves the highest success rate in predicting optimal runtime, whereas duplicator(GPU2) obtains the lowest total latency.

STOIC Choice	Nautilus Savings (+) or Loss (-)
Edge	$+1393$s
GPU1	-440s
GPU2	-257s

Table 5: Nautilus savings (positive values) and loss (negative values) for STOIC Duplicator. Savings are the time returned to Nautilus due to edge execution. Loss is the "wasted" time on Nautilus when the GPU runtimes are terminated because of faster edge execution. All units are in seconds. In the GPU2 case, the time is for both GPUs.

response time relative to choosing one.

This result surprised us for two reasons. First, because there was greater deployment variance and a larger mean deployment time for two GPUs, we expect that the edge (which is more predictable) would generate a greater success rate, but a larger aggregate response time. Put another way, we expected that STOIC would make safer predictions favoring the edge in the GPU2 case, but the cost of this safety would be greater aggregate response time. Empirically, however, we observe that STOIC "risks" predicting the GPU2 deployment more frequently, but that it amortizes this risk effectively because the two GPU execution is faster.

Note that the cost is not large. In practice, the application will use the one GPU case to get a better success rate at the cost of 2% in aggregate response time. However, it is interesting that STOIC is able to make this risk-reward trade-off explicit. Note also that the worst case is unchanged. This result indicates that there are unusually bad response time, but that *all* STOIC scheduling methods can mitigate them to approximately the same degree.

We conclude our analysis with quantification of the savings and unnecessary loss of Nautilus time that STOIC Duplicator is able to achieve. Table 5 shows the savings and loss of Nautilus time that are realized by the Duplicator heuristic.

Recall that the total MIN-LAT time (the time associated with the minimum execu-

tion of each batch) is 10022 seconds. The positive values in the table indicate the total time returned to Nautilus (that would have otherwise been used) by selecting the edge for execution. Note that these savings correspond to the results shown in Table 4 for the Duplicator. That is, they are the savings that STOIC was able to achieve while implementing a schedule within either 1% or 2% of MIN-LAT. The loss (negative values) shows the amount of Nautilus time that was used unnecessarily. That is, when STOIC Duplicator chose conditionally to use the GPU or GPUs and the edge finishes first, the elapsed time on Nautilus is unnecessarily "lost." Clearly from the table, Duplicator saves more Nautilus time than it loses. Thus, we infer that STOIC in duplicator mode optimizes the time to solution (Table 4) while utilizing the expensive Nautilus resource efficiently (Table 5) by using the edge cloud node speculatively.

5 Conclusion

In this paper, we propose a framework, called STOIC, for executing machine learning applications in IoT-cloud settings using the serverless architecture. STOIC integrates an edge controller and a public cloud with GPU acceleration. When the scheduler at the edge controller receives a batch of images from open field camera traps, it predicts the total response time for processing the batch based on batch size and historical log data. In the selector mode, STOIC schedules the task to the runtime with the least predicted latency. In the duplicator mode, STOIC co-schedules the task on the edge cloud and GPU runtime in the public cloud. If the latter is deployed and predicted to be faster, the edge cloud job is terminated. Otherwise, STOIC terminates the public cloud job and completes the task on the edge cloud. This mode further optimizes the selection process by avoiding volatile deployment times.

We present the design principles, implementation details, the feedback control mechanism, and different modeling methodologies to address the variability in the edge and public cloud deployments. Our empirical evaluation demonstrates STOIC can schedule tasks on local and remote deployments to achieve a speedup of 3.3x versus our baseline scenario. STOIC's success rate for prediction placement ranges from 92% to 97% for the application and datasets that we study.

As part of future work, we plan to investigate substituting RANSAC with Gradient Boosting Regression Trees (GBRT) to capture the non-linearity in the processing time due to heterogeneous hardware across deployment options (runtimes). We also plan to investigate model check-pointing in duplicator mode to better utilize computational resource on edge cloud and to improve the overall performance of the STOIC system.

Acknowledgments

This work has been supported in part by NSF (CNS-1703560, CCF-1539586, ACI-1541215), ONR NEEC (N00174-16-C-0020), and the AWS Cloud Credits for Research program. This work was performed in part at the University of California Natural Reserve System Sedgwick Reserve DOI: 10.21973/N3C08R.

References

[1] [AWS Lambda] https://aws.amazon.com/lambda/. [Online; accessed 12-July-2020].

[2] [Serverless] https://www.serverless.com/ [Online; accessed 12-July-2020].

[3] [Azure Functions] https://azure.microsoft.com/en-us/services/functions/. [Online; accessed 12-July-2020].

[4] [AWS Lambda for Web Services] 2019. ”https://aws.amazon.com/getting-started/projects/build-serverless-web-app-lambda-apigateway-s3-dynamodb-cognito/” [Online; accessed on 12-Aug-2019].

[5] [AWS Lambda for Microservices] 2019. ”https://aws.amazon.com/microservices/” [Online; accessed on 12-Aug-2019].

[6] [Google Cloud Functions] https://cloud.google.com/functions/docs/. [Online; accessed 12-July-2020].

[7] Wolski R., Krintz C., Bakir F., George G., Lin W-T.. CSPOT: Portable, Multi-scale Functions-as-a-Service for IoT in *ACM Symposium on Edge Computing* 2019.

[8] [IBM OpenWhisk] https://developer.ibm.com/openwhisk/. [Online; accessed 12-July-2020].

[9] [Iron.io] https://www.iron.io. [Online; accessed 12-Sep-2017].

[10] [GreenGrass and IoT Core] https://aws.amazon.com/iot-core,greengrass/. [Online; accessed 2-Mar-2019].

[11] [IoT Hub: Microsoft Azure] https://azure.microsoft.com/en-us/services/iot-hub/.

[12] [IoT Edge: Microsoft Azure] https://azure.microsoft.com/en-us/services/iot-edge/.

[13] Zhang M, Krintz C, Wolski R.. STOIC: Serverless TeleOperable Hybrid Cloud for Machine Learning Applications on Edge Device *International Workshop on Smart Edge Computing and Networking.* 2020.

[14] Pu Qifan, Ananthanarayanan Ganesh, Bodik Peter, et al. Low Latency Geo-Distributed Data Analytics in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*SIGCOMM '15(New York, NY, USA):421–434Association for Computing Machinery 2015.

[15] Vulimiri Ashish, Curino Carlo, Godfrey P. Brighten, Jungblut Thomas, Padhye Jitu, Varghese George. Global Analytics in the Face of Bandwidth and Regulatory Constraints in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*(Oakland, CA):323–336USENIX Association 2015.

[16] Cuervo Eduardo, Balasubramanian Aruna, Cho Dae-ki, et al. MAUI: Making Smartphones Last Longer with Code Offload in *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*MobiSys '10(New York, NY, USA):49–62Association for Computing Machinery 2010.

[17] McMahan Brendan, Ramage Daniel. Federated Learning: Collaborative Machine Learning without Centralized Training Data 2017.

[18] Hellerstein Joseph M., Faleiro Jose, Gonzalez Joseph E., et al. Serverless Computing: One Step Forward, Two Steps Back 2018.

[19] Jonas Eric, Schleier-Smith Johann, Sreekanti Vikram, et al. Cloud Programming Simplified: A Berkeley View on Serverless Computing 2019.

[20] Ishakian Vatche, Muthusamy Vinod, Slominski Aleksander. Serving deep learning models in a serverless platform in *2018 IEEE International Conference on Cloud Engineering (IC2E)*:257–262IEEE 2018.

[21] Naranjo Diana M, Risco Sebastián, Alfonso Carlos, Pérez Alfonso, Blanquer Ignacio, Moltó Germán. Accelerated serverless computing based on GPU virtualization *Journal of Parallel and Distributed Computing.* 2020;139:32–42.

[22] Mohanty Sunil, others . Evaluation of Serverless Computing Frameworks Based on Kubernetes *master thesis.* 2018.

[23] Muslim Nasif, Islam Salekul. Face recognition in the Edge Cloud in *Proceedings of the International Conference on Imaging, Signal Processing and Communication*:5–9 2017.

[24] Teerapittayanon Surat, McDanel Bradley, Kung Hsiang-Tsung. Distributed deep neural networks over the cloud, the edge and end devices in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*:328–339IEEE 2017.

[25] Lv Bojie, Hong Yuncong, Tan Haisheng, Han Zhenhua, Wang Rui. Cooperative Job Dispatching in Edge Computing Network with Unpredictable Uploading Delay *arXiv preprint arXiv:1912.10732.* 2019.

[26] Paščinski Uroš, Trnkoczy Jernej, Stankovski Vlado, Cigale Matej, Gec Sandi. QoS-Aware Orchestration of Network Intensive Software Utilities within Software Defined Data Centres: An Architecture and Implementation of a Global Cluster Manager *Journal of Grid Computing.* 2017;16.

[27] Kochovski Petar, Drobintsev P., Stankovski Vlado. Formal Quality of Service assurances, ranking and verification of cloud deployment options with a probabilistic model checking method *Information and Software Technology.* 2019.

[28] Kochovski Petar, Gec Sandi, Stankovski Vlado, Bajec Marko, Drobintsev P.. Trust management in a blockchain based fog computing platform with trustless smart oracles *Future Generation Computer Systems.* 2019;101.

[29] Cech H. L., Großmann M., Krieger U. R.. A Fog Computing Architecture to Share Sensor Data by Means of Blockchain Functionality in *2019 IEEE International Conference on Fog Computing (ICFC)*:31-40 2019.

[30] Lin Wei-Tsung, Bakir Fatih, Krintz Chandra, Wolski Rich, Mock Markus. Data repair for Distributed, Event-based IoT Applications in *Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems*:139–150 2019.

[31] Lin W-T., Krintz C., Wolski R.. Tracing Function Dependencies Across Clouds in *IEEE IC2E* 2019.

[32] Lin W., Krintz C., Wolski R., et al. Tracking Causal Order in AWS Lambda Applications in *IEEE International Conference on Cloud Engineering* 2017.

[33] [AWS X-ray] "https://aws.amazon.com/xray/" [Online; accessed 12-July-2020].

[34] Shahrad Mohammad, Balkind Jonathan, Wentzlaff David. Architectural implications of function-as-a-service computing in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*:1063–1075 2019.

[35] Xiao Liang, Wan Xiaoyue, Dai Canhuang, Du Xiaojiang, Chen Xiang, Guizani Mohsen. Security in mobile edge caching with reinforcement learning *IEEE Wireless Communications.* 2018;25:116–122.

[36] [Kubernetes] 2020. "https://kubernetes.io/docs/reference/" [Accessed 20-May-2020].

[37] Burns Brendan, Grant Brian, Oppenheimer David, Brewer Eric, Wilkes John. Borg, omega, and kubernetes *System Evolution.* 2016.

[38] [kubeless] https://github.com/kubeless/kubeless [Accessed 20-July-2020].

[39] [Kubernetes Pods] https://kubernetes.io/docs/concepts/workloads/pods/pod/ [Accessed 20-July-2020].

[40] [Intel NUCs] https://www.intel.com/content/www/us/en/products/boards-kits/nuc.html [Accessed 20-July-2020].

[41] [Eucalyptus] https://www.eucalyptus.cloud/ [Accessed 20-July-2020].

[42] [Nautilus] http://ucsd-prp.gitlab.io/nautilus/ [Accessed 20-July-2020].

[43] [Rook Ceph Block] https://rook.io/docs/rook/v1.0/ceph-block.html [Accessed 20-July-2020].

[44] Weil Sage A, Brandt Scott A, Miller Ethan L, Long Darrell DE, Maltzahn Carlos. Ceph: A scalable, high-performance distributed file system in *Proceedings of the 7th symposium on Operating systems design and implementation*:307–320USENIX Association 2006.

[45] [AVX2] https://docs.oracle.com/cd/E36784_01/html/E36859/gntae.html [Accessed 20-July-2020].

[46] [Auto-Regression] ”https://en.wikipedia.org/wiki/Autoregressive_model” [Accessed 20-July-2020].

[47] [Moving Average] https://en.wikipedia.org/wiki/Moving_average.

[48] [Bayesian Ridge Regression] https://scikit-learn.org/stable/auto_examples/linear_model/plot_bayesian_ridge.html [Accessed 20-July-2020].

[49] [Ordinary Least Squares] https://scikit-learn.org/stable/modules/linear_model.html#ordinary-least-squares [Accessed 20-July-2020].

[50] [RANSAC] https://en.wikipedia.org/wiki/Random_sample_consensus [Accessed 20-July-2020].

[51] [Golang] https://golang.org/ [Accessed 20-July-2020].

[52] [client-go] https://github.com/kubernetes/client-go [Accessed 20-July-2020].

[53] Abadi Martín, Barham Paul, Chen Jianmin, et al. Tensorflow: A system for large-scale machine learning in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*:265–283 2016.

[54] [CRD] https://kubernetes.io/docs/tasks/access-kubernetes-api/custom-resources/custom-resource-definitions/ [Accessed 20-July-2020].

[55] [Docker Hub] https://hub.docker.com/ [Accessed 20-July-2020].

[56] [Fma] https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-fma-qfma [Accessed 20-July-2020].

[57] [Nvidia Container Toolkit] https://github.com/NVIDIA/nvidia-docker [Accessed 20-July-2020].

[58] Elias Andy Rosales, Golubovic Nevena, Krintz Chandra, Wolski Rich. Where's the bear?-automating wildlife image processing using iot and edge cloud systems in *2017 IEEE/ACM Second International Conference on Internet-of-Things Design and Implementation (IoTDI)*:247–258IEEE 2017.

[59] [Sedgwick Natural Reserve] https://sedgwick.nrs.ucsb.edu/ [Accessed 20-July-2020].

[60] LeCun Yann, Bengio Yoshua, others . Convolutional networks for images, speech, and time series *The handbook of brain theory and neural networks.* 1995;3361:1995.

[61] Pedregosa Fabian, Varoquaux Gaël, Gramfort Alexandre, et al. Scikit-learn: Machine learning in Python *Journal of machine learning research.* 2011;12:2825–2830.

[62] [Keras image data generator] https://keras.io/preprocessing/image/#imagedatagenerator-class [Accessed 20-July-2020].

[63] Campos Víctor, Sastre Francesc, Yagües Maurici, Torres Jordi, Nieto Xavier. Scaling a convolutional neural network for classification of adjective noun pairs with tensorflow on gpu clusters in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*:677–682IEEE 2017.