STOIC: Serverless Teleoperable Hybrid Cloud for Machine Learning Applications on Edge Device

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

***Abstract*— Serverless computing is a promising new event-driven programming model that was designed by cloud vendors to expedite the development and deployment of scalable web services on cloud computing systems. Using the model, developers write applications that consist of simple, independent, stateless functions that the cloud invokes on-demand (i.e. elastically), in response to system-wide events (data arrival, messages, web requests, etc.).**

In this work, we present STOIC (Serverless TeleOperable HybrId Cloud), an application scheduling and deployment system that extends the serverless model in two ways. First, it uses the model in a distributed setting and schedules application functions across multiple cloud systems. Second, STOIC supports serverless function execution using hardware acceleration (e.g. GPU resources) when available from the underlying cloud system. We overview the design and implementation of STOIC and empirically evaluate it using real-world machine learning applications and multi-tier (e.g. edge-cloud) deployments. We find that STOIC's combined use of edge and cloud resources is able to outperform using either cloud in isolation for the applications and datasets that we consider.

***Keywords*—Serverless computing; Edge computing; Image Processing; Internet of Things**

I. Introduction

With the recent shift of application architectures from monolithic to containers and microservices, serverless computing has risen as a promising cloud service where simple, stateless, event-driven functions comprise applications and services. Serverless platforms relieve developers of the burden of provisioning servers to deploy cloud and web applications. Programmers typically write functions in high-level languages which are triggered by the platform in response to events from external sources or other cloud services.

This function-level abstraction also provides fine-grained computational resource isolation and usage, meaning that each serverless function can autoscale independently based on the rate of incoming events. Providing such elasticity helps avoid a single point failure and performance bottlenecks in data-intensive applications. From this perspective, serverless architecture is an ideal system for machine learning applications, especially for online training [1] and inference, which transfer and manipulate large amounts of data or for which the input sizes vary.

To enable such an event-driven system, one concerning situation is for machine learning applications that receive their data from heterogeneous IoT devices, ranging from temperature sensors to mobile phones to autonomous drones. For such deployments, application execution should be "near" (in terms of network latency) the data sources to achieve fast response times. Such settings motivate us to explore extending the serverless model to the edge for the execution of data analytics applications.

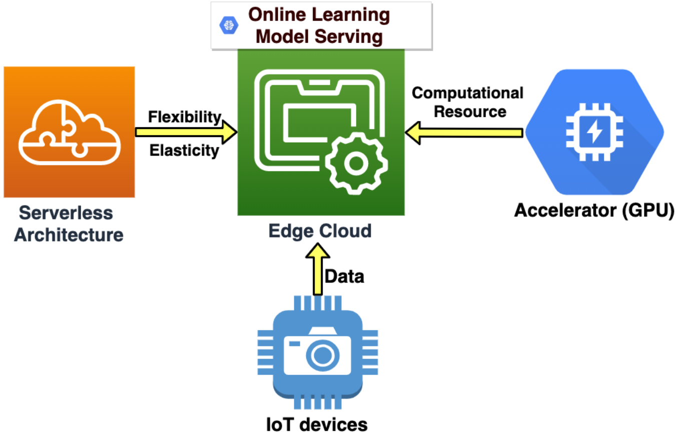

Fig. 1: The abstract design of STOIC – a system for executing distributed machine learning applications in IoT (e.g. edge + cloud) settings.

One challenge with edge computing is the scarcity of computational resources relative to resource-rich public and private clouds. Moreover, public/private clouds may offer specialized hardware (e.g. GPUs) that significantly speed up machine learning applications, which is not commonly available in resource-restricted edge clouds. In our work, we investigate how to extend the serverless computing model to hybrid cloud systems that consist of edge and cloud resources and that integrate GPU acceleration.

Toward this end, we present STOIC – a Serverless Tele-Operable Hybrid Cloud. As depicted in Figure 1, STOIC is a framework to which IoT devices stream data in batches for training and inference by machine learning applications. The framework implements serverless computing and deployment for the applications. Unique to STOIC, however, is its scheduling system which intelligently places the application workload on edge and cloud systems that it predicts will result in the fastest time to completion. Moreover, STOIC takes advantage of GPU acceleration when available from the underlying cloud resource. In this paper, we discuss the

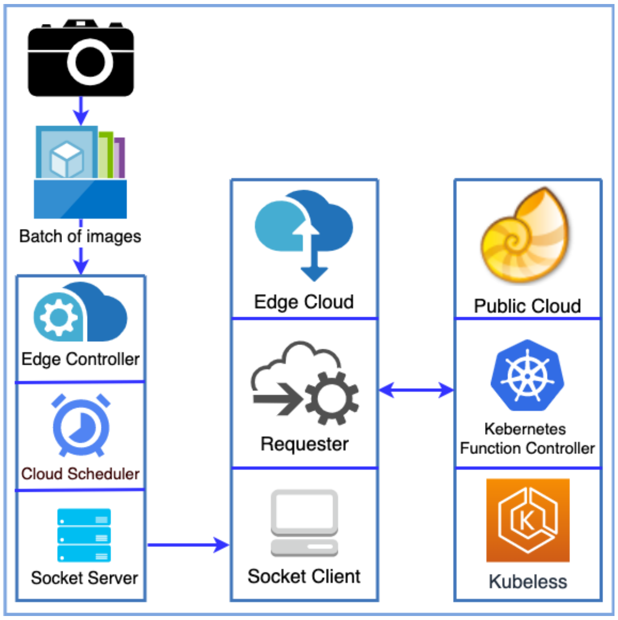

Fig. 2: The STOIC Architecture

design and implementation of this architecture, investigate the efficacy of using this extended serverless model for machine learning applications that span edge-cloud systems, and empirically evaluate the performance of doing so. Using real workloads and deployments, we find that STOIC reduces the total response time of the applications we study from 6.48% to 32.05%, compared with four different runtimes, each running in isolation. Finally, we discuss related and future work and conclude.

II. STOIC

To leverage hardware acceleration and distributed scheduling within a serverless architecture, we have developed STOIC, a framework for executing analytics applications in multi-tier IoT (sensing-edge-cloud) settings. STOIC streamlines the end-to-end process of packaging, transferring, scheduling, executing, and result retrieval for machine learning applications. Figure 2 shows three principal pillars of STOIC's architecture: Edge Controller, Edge Cloud, and Public Cloud.

A. Edge Controller

We deploy a collection of edge devices, including multiple motion-detecting camera traps in open field and a local server as edge controller in the research facility at Sedgwick Natural Reserve [2]. In this deployment, we install motion-triggered cameras (i.e. camera traps) at watering holes to capture images of wildlife in their natural habitat (as part of conservation science studies). The cameras are connected via a wireless radio to a computer located in an outbuilding at the reserve. We refer to this computer as the edge controller. It is connected to a private campus cloud via a microwave link. When a camera trap detects motion, it takes photos and persists the images in flash storage. Periodically, the camera traps transfer saved photos to the edge controller. STOIC runs on the edge controller and its execution is triggered by the arrival of batches of images from camera traps located across the reserve. When a batch arrives, STOIC partitions the image processing application (for object/animal classification) into tasks that it assigns to 1+ cloud components.

B. Edge Cloud

As an intermediate computational tier between the sensors and the public cloud, the edge cloud can be placed anywhere, preferably near the edge devices, to lower the response latency of analytics applications processing the images. Our edge cloud is currently deployed in our lab on campus which is connected to the edge controller via a fast network link.

Our edge cloud consists of a cluster of nine Intel NUCs [3]. Eucalyptus cloud system [4] manages the edge cloud and supports Linux virtual machine instances. Running on an instance, the STOIC socket client listens for the request from edge controller and then either executes the job locally on edge cloud or has STOIC requester interact with public cloud to complete the designated task.

C. Remote Public/Private Cloud

To investigate the use of the serverless architecture with hardware accelerators, we employ a shared, multi-university, GPU cloud Nautilus [5] as our remote cloud system. Nautilus is an Internet-connected, HyperCluster research platform led by researchers at UC San Diego, National Science Foundation, the Department of Energy, and various participating universities globally. Being designed for running data and computationally intensive applications, Nautilus uses Kubernetes [6] as an interface to manage and scale containerized applications. It uses Rook [7] to integrate Ceph [8] data services. As of Nov. 2019, Nautilus consists of 141 computing nodes across the US and 422 GPUs are available in the cluster. All of these nodes are connected via a multi-campus network. In this study, we consider Nautilus as a public cloud that enables us to leverage hardware acceleration (GPUs) in the serverless architecture to serve edge devices.

D. Implementation

Considering performance and interface, we implement STOIC using Golang [9]. Golang provides high performance (vs scripting languages)and a user-friendly interface [10] to Kubernetes. STOIC currently supports machine learning applications developed using the TensorFlow framework [11].

1) Serverless framework: For our serverless architecture, STOIC employs kubeless [12] and Docker [13] in the Nautilus Cloud. As a Kubernetes-native serverless framework, kubeless uses the Custom Resource Definition (CRD) [14] to dynamically create functions as Kubernetes custom resources and launches runtimes on-demand. For specific machine learning tasks that STOIC executes, we use Docker to build customized runtime images that we upload to Docker Hub [15] in advance. When the function controller at Nautilus Cloud receives a task request, it pulls the latest image from Docker Hub before

TABLE I:
PERFORMANCE COMPARISON ON THREE BENCHMARKS BETWEEN STANDARD AND CUSTOM TENSORFLOW LIBRARY COMPILED WITH AVX2 / SSE / FMA CPU INSTRUCTION SET SUPPORT

	Mean Std. (sec)	Mean Custom. (sec)	Speed-up %
Iris	53.17	41.86	21.3
MNIST	268.81	189.80	29.4
InceptionV3	958.47	791.28	17.4

launching the function. This deployment pipeline makes the runtime flexible and extensible for evolving applications.

For the edge cloud, we execute tasks by directly invoking the application function binaries. We make this design decision to simplify STOIC's control plane in our prototype but we are investigating the use of a consistent serverless architecture across edge and public/private cloud as part of future work.

2) STOIC Library Support: To leverage the computational power of the CPU systems available in the Edge and Public Cloud, we compile Tensorflow from source with AVX2, SSE4.2 [16] and FMA [17] instruction set support. We then test the performance of customized Tensorflow package on three common machine learning training tasks: **(A)** *Iris* [18] with 10-fold cross-validation; **(B)** *MNIST* [19] on 20 Epochs; **(C)** *InceptionV3* [20] on 10 epochs with 1,000 images.

We execute these applications 10 times on standard and customized Tensorflow packages. Table I shows the mean execution time of three benchmarks for each package. We calculate speed-up as $(T_s - T_c)/T_s$, where T_s and T_c represent the execution time by the standard and customized Tensorflow library respectively. We observe that, in all three benchmarks, the customized library achieves a speed-up that ranges from 17.4% to 29.5%. STOIC uses the customized TensorFlow packages for cloud systems that implement these instruction sets. In our prototype, both the edge and public cloud have these extensions.

3) GPU Accessibility: To enable GPU access by serverless functions, we build a container with NVIDIA Container Toolkit [21] support. Such support includes the NVIDIA runtime library and utilities which link serverless functions to NVIDIA GPUs. We also install CUDA 10.0 and cuDNN 7.0 in the image.

4) STOIC Runtime: To schedule the machine learning tasks across hybrid cloud deployments, we define four runtime scenarios: **(A)** *edge* - A VM instance on the edge cloud with AVX2 support; **(B)** *cpu* - A Kubernetes pod on Nautilus containing a single CPU with AVX2 support; **(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.

E. Execution Time Estimation

As depicted in Figure 2, the STOIC socket server executes in the edge cloud and listens for requests from the edge controller (machine learning job requests). After a preset period (parameterizable but currently set to 1 hour), STOIC estimates total response time (T_r) of a requested batch, based on 4 different runtime scenarios. The total response time includes data transfer time (T_t), runtime deployment time (T_d) and corresponding processing time (T_p):

1) Transfer time (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.

2) Runtime deployment time (T_d): It measures the time Nautilus uses to deploy requested kubeless function. Since the scarcity of computation, it is common that *gpu2* runtime takes longer to deploy than *gpu1* and *cpu* runtimes. We analyze the deployment log and calculate the average deployment time for each Nautilus runtime. In future work, we plan to develop a feedback control loop to dynamically update deployment time for each runtime. Note that, for *edge* runtime, the transfer and runtime deployment time zero out since STOIC executes the task locally in the edge cloud.

3) Processing time (T_p): is the execution time of a specific machine learning task. As a primary component for scheduling tasks across the hybrid cloud, we regress processing time based on prior experiment data by Bayesian Ridge Regression [22] due to its robustness to ill-posed problems compared to Ordinary Least Squares regression [23]. Thus, STOIC formulates the regression and uses it to predict the processing time based on the file size of the current batch.

F. Workflow

The STOIC workflow is as follows: based on the three time components, STOIC predicts the total response times (T_r) of the four deployment options. The scheduler selects the runtime with the shortest estimated response time. Then the edge controller sends a request, including the payload of compressed image batch and runtime information, to edge cloud. Upon acceptance, the edge cloud executes the task locally if the choice is the *edge* runtime. Such deployment is common when a batch of images is small.

For large batch sizes, STOIC typically schedules one of the three public runtime options. For these three scenarios, the edge cloud first requests the deployment on the public cloud. It then sends the payload to public cloud storage. The public cloud then deploys and executes the kubeless function. As a design decision, instead of running a requester pod in the public cloud, we run an instance on the edge cloud. We do so because the edge cloud is more stable and fault resilient than Nautilus which experiences intermittent downtime.

Once Nautilus successfully deploys the serverless function, it informs the edge cloud's requester to trigger the function via an HTTP request. When the task completes, the requester retrieves the results and runtime metrics, and transmits them back to the edge controller. Finally, the controller saves the results and metrics to persistent storage.

G. Intelligent Probing

With a series of experiments, we have found that the processing times from the same image batch and kubeless function can vary significantly between the first run and successive runs. This is due to the differences between cold and warm starts [24]. A cold start is when a machine learning task requires retrieval of stored model and dataset from cloud storage, which takes time. Once the function retrieves and caches this information, successive invocations of the function (using the same container) avoid this cost (i.e. experience a warm start).

STOIC accounts for cold and warm starts in its scheduling estimate using intelligent probes. When STOIC schedules an incoming task in a different runtime than the previous one, it triggers the function with the least amount of input data to ensure the function caches the model and dataset in memory. Following this, STOIC triggers the actual tasks. To avoid redundant probing, STOIC starts the task directly when the designated runtime is the same as the previous batch.

III. Evaluation

In this section, we empirically evaluate STOIC's performance when executing machine learning applications. We compare its use of multiple runtimes versus solely using a single runtime for all batches. In the sections that follow, we first describe the machine learning application that we consider. We then present our experimental setup and results.

A. Benchmark Application and Dataset

We evaluate STOIC using an image processing application that classifies animal images from a wildlife monitoring system called "Where's The Bear" (WTB) [25]. "Where's The Bear" is an end-to-end distributed data acquisition and analytics system that implements an IoT architecture and edge cloud. Our application makes inferences for each photo taken by deployed camera traps in Sedgwick Natural Reserve using a convolutional neural network (CNN) [26]. We train the model using labeled images from the WTB dataset. Technically, the application employs Tensorflow and Scikit-learn [27] to implement image classification.

In total, there are five classes that we consider in the CNN model training: Bird, Fox, Rodent, Human and Empty. Since class size is unbalanced due to frequencies of animal occurrences, we up-sample minority classes (e.g. fox) using the Keras ImageDataGenerator [28]. Doing so ensures that the classification model is not biased. We resize every image in the WTB 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 cloud storage at both edge cloud (disk storage) and Nautilus (a shared volume in a Ceph file system).

B. Performance Evaluation

We first test the efficacy of STOIC by processing an image batch of fixed size at four runtimes individually and then compare them with STOIC. To make the result reliable, we again experiment 10 times and list the mean and standard deviation of total response time (T_r) and processing time (T_p) in Table II. We can observe from Table II that STOIC schedules 40-image batch to *gpu1* runtime, based on its prediction that *gpu1* would have the least total response time (T_r). One important observation is that *gpu2* runtime has even lower processing time (T_p) than *gpu1*, but STOIC disregard *gpu2* in this scenario, because its gain in processing time (T_p) does not compensate for its lengthy deployment time (T_d) on the Nautilus cloud.

TABLE II:
Mean and Stdev. of Total Response Time (T_r) and Processing Time (T_p) of 40-Image Batch: STOIC Schedules Tasks onto the Runtime (*gpu1*) that Has the Least Total Response Time (T_r).

	Mean T_r (sec)	**Stdev. T_r (sec)**	**Mean T_p (sec)**	**Stdev. T_p (sec)**
edge	108.88	1.65	108.88	1.65
cpu	100.0	4.93	86.99	4.92
gpu1	98.90	4.03	50.65	4.05
gpu2	106.29	5.53	39.21	5.55
STOIC	**97.73**	**3.13**	**50.49**	**3.11**

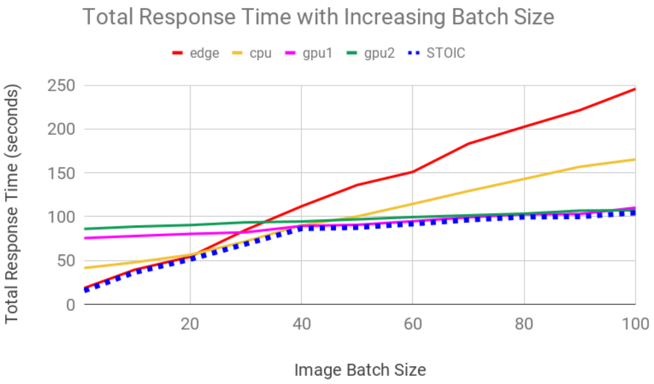

Fig. 3: Total Response Time (T_r) of image batches of growing sizes: The x-axis represents the batch size, while the y-axis is the total response time (T_r). STOIC, which is depicted in the blue dashed line, schedules the task on the runtime with the least total response time.

We next test STOIC by processing a series of image batches of growing sizes on the four runtimes and STOIC. Figure 3 shows their total response times. The x-axis is the size of the image batch and the y-axis is the total response time (T_r) in seconds. The red curve shows that latency increases linearly over the *edge* runtime. The yellow curve shows the performance of the *cpu* runtime in the Nautilus cloud. We observe that its slope is more moderate than *edge* runtime since CPUs in nodes of Nautilus cloud are usually more powerful than those in the edge cloud. The pink and green curves represent the *gpu1* and *gpu2* runtimes, respectively, and they intersect at a batch size of 95, at which STOIC would switch the deployment of tasks from *gpu1* to *gpu2*. The blue dashed line depicts the total response time (T_r) of STOIC, which is able to schedule a series of tasks to the runtime with the least latency. According to such result, STOIC improves system performance by determining the best runtime for the given task dynamically.

We next perform an empirical evaluation of STOIC by

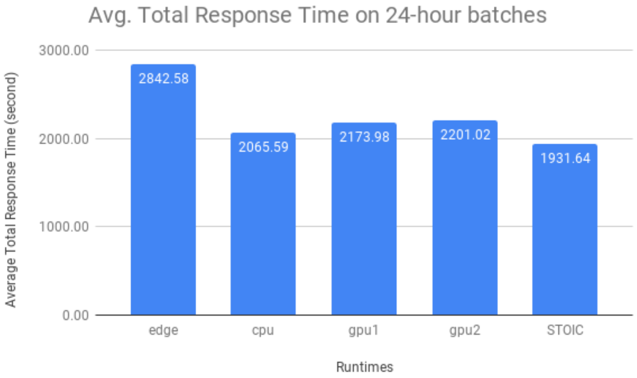

Fig. 4: Average total response time (T_r) on the 24-hour dataset: The x-axis represents runtimes, while the y-axis represents the average total response time (T_r) by STOIC and four other runtimes on the 24-hour dataset. The data labels on columns are specific numbers of T_r. The seeded simulator generates the 24-hour batch sizes from the distribution of historical data.

comparing the total response time (T_r) of multiple image batches of different sizes across the four single runtimes and STOIC. To accelerate the repetitive experiment, we developed a simulator to generate image batches based on the frequency distribution of the WTB dataset.

According to 2016 WTB dataset, the size of image batch fits to normal distribution $\mathbf{N}(\mu = 42.75, \sigma^2 = 39.5)$. Thus, the simulator generates 24 image batches in the edge controller to emulate streaming data in one day from open field camera traps. To conduct an unbiased evaluation, we seed the simulator to make these 24 image batches consistent across all runtimes and STOIC.

To ensure the validity of the outcome, we run each experiment 10 times for each runtime scenario and report the average value. Figure 4 shows the average total response time (T_r) for STOIC and the four individual runtimes. STOIC achieves the lowest average latency versus the four other single runtimes. STOIC reduces total response time (T_r) by 32.05% (versus *edge*), 6.48% (versus *cpu*), 11.15% (versus *gpu1*) and 12.24% (versus *gpu2*) respectively. According to such a result, we conclude that STOIC outperforms any single-runtime scheduling mechanism on the empirical datasets and real-world machine learning applications.

IV. Related Work

As related work, we consider recent advances in both machine learning infrastructure and serverless computing domains. In the former area, much research has extended efforts into designing efficient systems for inference and deployment of machine learning models. As a complement to the Tensorflow framework, Tensorflow-serving [29] integrates new models and updates versions from training to serving. Though it makes seminal exploration on the multi-tenant model hosting service, Tensorflow-serving does not realize authentic high-performing parallelism to handle concurrent heavy query loads.

Clipper [30] constructs a general-purpose low-latency prediction serving system, which attempts to solve the problem of demanding real-time prediction at the client-side and handling heavy query load at the server-side. It also enables the model composition and online learning to improve accuracy and render more reliable predictions. To explore the multi-pipeline techniques, PRETZEL [31] casts model-serving as a database problem and applies multi-query optimizations to maximize performance. However, both Clipper and PRETZEL require considerable compute resources in caching, batching, adaptive model selection and off-line training to maximize throughput. Therefore, they are not optimized for resource-constrained and heterogeneous, multi-tier IoT systems. To the best of our knowledge, STOIC is the first work to addresses this problem by integrating machine learning applications into a serverless architecture that leverages GPU as additional computational resources for IoT devices. We consider it as a promising and extensible solution for high-throughput and low-latency system for online training and machine learning applications in general.

To build an end-to-end system for practical machine learning applications, we require several other components. Seneca [32] fine-tunes hyper-parameters of machine learning models on a general-purpose serverless architecture (AWS Lambda [33]). It provides a fast and low-cost method to grid search for the best-performing hyper-parameter set, which is essential in the deployment pipeline of machine learning applications. Velox [34] offers a low-latency and scalable solution for complex analytical model-serving, in which it completes a missing piece of personalized prediction serving using Apache Spark [35]. For calibrating performance, McGrath et al. [36] propose an empirical methodology to measure the design and performance of serverless platforms, including latency and auto-scaling capability. These related systems are complementary to STOIC and can be combined to provide a robust serverless ecosystem for machine learning applications.

V. Conclusion

In this paper, we propose a framework, called STOIC, for executing machine learning applications in hybrid cloud settings based on serverless architecture. STOIC integrates three components: Edge Controller, Edge Cloud, and Public Cloud. 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. It then schedules the task to the runtime that it predicts to have the least total response time. Our STOIC prototype considers four different runtime scenarios. When STOIC schedules the task to the public cloud, the edge cloud deploys a serverless function and then relays the request and payload to the public cloud. STOIC returns the result and metrics to the edge controller when the task completes.

We present the design principles, implementation details, workflow and empirical evaluation on real-world machine learning application for STOIC. Our evaluation demonstrates

STOIC is able to intelligently schedule machine learning tasks across hybrid cloud deployments and obtain better performance that using any single deployment option in isolation. Our speed-up percentages range from 6.48% to 32.05% for the application and datasets that we study.

As part of future work, we are developing a feedback control loop to dynamically update the deployment and processing time of STOIC tasks. We plan to also investigate the feasibility of executing model-training tasks using STOIC. Finally, we plan to investigate ways of not having sufficient labeling for image classification tasks and to unify the serverless architecture across all edge and cloud systems within the STOIC system and for the applications that execute using it.

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.

References

[1] S. Shalev-Shwartz *et al.*, "Online learning and online convex optimization," *Foundations and Trends® in Machine Learning*, vol. 4, no. 2, pp. 107–194, 2012.

[2] (2019, Nov.) Sedgwick natural reserve. [Online]. Available: https://sedgwick.nrs.ucsb.edu/

[3] (2019, Nov.) Intel nucs. [Online]. Available: https://www.intel.com/content/www/us/en/products/boards-kits/nuc.html

[4] (2019, Nov.) Eucalyptus. [Online]. Available: https://www.eucalyptus.cloud/

[5] (2019, Nov.) Nautilus. [Online]. Available: http://ucsd-prp.gitlab.io/nautilus/

[6] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *System Evolution*, 2016.

[7] (2019, Dec.) Rook ceph block. [Online]. Available: https://rook.io/docs/rook/v1.0/ceph-block.html

[8] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 307–320.

[9] (2019, Dec.) Golang. [Online]. Available: https://golang.org/

[10] (2019, Nov.) client-go. [Online]. Available: https://github.com/kubernetes/client-go

[11] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.

[12] (2019, Nov.) kubeless. [Online]. Available: https://github.com/kubeless/kubeless

[13] (2019, Nov.) Docker. [Online]. Available: https://www.docker.com/

[14] (2019, Nov.) Crd. [Online]. Available: https://kubernetes.io/docs/tasks/access-kubernetes-api/custom-resources/custom-resource-definitions/

[15] (2019, Nov.) Docker hub. [Online]. Available: https://hub.docker.com/

[16] (2019, Nov.) Avx2. [Online]. Available: https://docs.oracle.com/cd/E36784_01/html/E36859/gntae.html

[17] (2019, Nov.) Fma. [Online]. Available: https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-fma-qfma

[18] (2019, Nov.) iris. [Online]. Available: https://scikit-learn.org/stable/auto_examples/datasets/plot_iris_dataset.html

[19] (2019, Nov.) mnist. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[20] (2019, Nov.) inceptionv3. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/applications/InceptionV3

[21] (2019, Nov.) Nvidia container toolkit. [Online]. Available: https://github.com/NVIDIA/nvidia-docker

[22] (2019, Nov.) Bayesian ridge regression. [Online]. Available: https://scikit-learn.org/stable/auto_examples/linear_model/plot_bayesian_ridge.html

[23] (2019, Nov.) Ordinary least squares. [Online]. Available: https://scikit-learn.org/stable/modules/linear_model.html#ordinary-least-squares

[24] (2019, Nov.) Cold start. [Online]. Available: https://en.wikipedia.org/wiki/Cold_start_(computing)

[25] A. R. Elias, N. Golubovic, C. Krintz, and R. Wolski, "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)*. IEEE, 2017, pp. 247–258.

[26] Y. Kim, "Convolutional neural networks for sentence classification," *arXiv preprint arXiv:1408.5882*, 2014.

[27] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *Journal of machine learning research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[28] (2019, Nov.) Keras image data generator. [Online]. Available: https://keras.io/preprocessing/image/#imagedatagenerator-class

[29] C. Olston, N. Fiedel, K. Gorovoy, J. Harmsen, L. Lao, F. Li, V. Rajashekhar, S. Ramesh, and J. Soyke, "Tensorflow-serving: Flexible, high-performance ML serving," *CoRR*, vol. abs/1712.06139, 2017. [Online]. Available: http://arxiv.org/abs/1712.06139

[30] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system," in *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, 2017, pp. 613–627.

[31] Y. Lee, A. Scolari, B.-G. Chun, M. D. Santambrogio, M. Weimer, and M. Interlandi, "{PRETZEL}: Opening the black box of machine learning prediction serving systems," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 611–626.

[32] M. Zhang, C. Krintz, M. Mock, and R. Wolski, "Seneca: Fast and low cost hyperparameter search for machine learning models," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 404–408.

[33] (2019, Dec.) Aws lambda. [Online]. Available: https://aws.amazon.com/lambda/

[34] D. Crankshaw, P. Bailis, J. E. Gonzalez, H. Li, Z. Zhang, M. J. Franklin, A. Ghodsi, and M. I. Jordan, "The missing piece in complex analytics: Low latency, scalable model management and serving with velox," *arXiv preprint arXiv:1409.3809*, 2014.

[35] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

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