UNIVERSITY OF CALIFORNIA
Santa Barbara

Intelligent Scheduling for IoT Applications at the Network Edge

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Michael Lebo Zhang

Committee in Charge:

Professor Chandra Krintz, Co-Chair

Professor Rich Wolski, Co-Chair

Professor Yufei Ding

September 2021

The Dissertation of
Michael Lebo Zhang is approved:

Professor Yufei Ding

Professor Rich Wolski, Committee Co-Chairperson

Professor Chandra Krintz, Committee Co-Chairperson

August 2021

Intelligent Scheduling for IoT Applications at the Network Edge

To my parents, Chunling and Yuguang, and the memory of my grandmother,

Suyun, who loves me more than anything in this world.

Acknowledgements

Every UCSB student would be asked numerous times throughout their years there: What brings you to UCSB? The answer to me is not only because of moderate weather and gorgeous beach but essentially because of my advisors, Chandra and Rich, who guided me through thick and thin in my academic years. Their infectious passion for research and teaching motivates me when my research project stuck at bottlenecks. Being inspired by weekly meetings and countless causal discussions, I learned from them how to brainstorm research ideas, how to design experiments, how to analyze metrics, how to write an academic paper, and, most importantly, how to be a scientist. The guidance and flexibility they offered lead to our fruitful academic journey and ever-lasting apprenticeship.

Nobody can survive this journey alone without companions. I attribute my success largely to my incredible lab mates, amazing graduate students in the computer science department, and all precious friends around. You are my last resort when my work gets stuck and you never failed to amaze me with how inspiring and encouraging you are. Likewise, I contribute to the community and get others' back when needed. The brotherhood and sisterhood would be remembered the most from all these Ph.D. years.

My whole family has always been supportive of my academic pursuit. They are an emotional panacea for the depression and frustration I experienced through

unsuccessful trials, paper rejections, and failed job interviews. I dedicate my degree to my sweet family.

This work has been supported in part by NSF (CNS-2107101, 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).

And, thank you, Santa Barbara!

Curriculum Vitæ

Michael Lebo Zhang

Education

2021	Doctor of Philosophy in Computer Science, University of California, Santa Barbara
2015	Master of Science in Energy System, Northeastern University, Boston
2008	Bachelor of Science in Computer Science, Kunming University of Science and Technology, China

Selected Professional Experience

2017 – 2021	Research Assistant, Univ. of California, Santa Barbara, CA
2020	Software Engineer Intern, Facebook, Seattle, WA
2019	Research Intern, Microsoft, Redmond, WA
2018	Software Engineering Intern, Appfolio, Santa Barbara, CA
2017	Summer Research Intern, Apple, Cupertino, CA

Publications

- Zhang, Michael; Krintz, Chandra; Wolski, Rich; Sparta: A Heat-Budget-based Scheduling Framework on IoT Edge Systems, International Conference on Edge Computing 2021. Zhang et al. (2021*b*)
- Zhang, M, Krintz, C, Wolski, R. Edge-adaptable serverless acceleration for machine learning Internet of Things applications. Softw Pract Exper. 2020; 1– 16. https://doi.org/10.1002/spe.2944 Zhang et al. (2021*a*)
- M. Zhang, C. Krintz and R. Wolski, "STOIC: Serverless Teleoperable Hybrid Cloud for Machine Learning Applications on Edge Device," 2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops), 2020, pp. 1-6, doi: 10.1109/PerComWorkshops48775.2020.9156239. Zhang et al. (2020)
- M. Zhang, C. Krintz, M. Mock and R. Wolski, "Seneca: Fast and Low Cost Hyperparameter Search for Machine Learning Models," 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), 2019, pp. 404-408, doi: 10.1109/CLOUD.2019.00071. Zhang et al. (2019)
- W. Lin et al., "Tracking Causal Order in AWS Lambda Applications," 2018 IEEE International Conference on Cloud Engineering (IC2E), 2018, pp. 50-60, doi: 10.1109/IC2E.2018.00027. Lin et al. (2018)

Please Note: Text and figures from these papers are used and appear in this dissertation.

Abstract

Intelligent Scheduling for IoT Applications at the Network Edge

by

Michael Lebo Zhang

The convergence of real-time embedded systems, wireless sensor networks, and machine learning, has fueled the rapid development of the Internet of Things (IoT), engendering new computational workloads and generating unprecedented amounts of streaming data. As a result, the computational infrastructure for IoT is facing challenges imposed by scalability, varying availability and performance, and heterogeneity. Highly concurrent event-driven architectures (EDAs) are one potential technological approach to building large-scale IoT systems since they naturally comprise concurrency, scheduling, and service decoupling and isolation. Moreover, serverless computing is an example of an EDA that has emerged as the next-generation event-driven system to address many of these challenges.

In addition, new tiered cloud architectures consisting of low-capability IoT devices, computing and storage resources sited "at the network edge," and public cloud resources provide the opportunity to optimize placement of computation and storage tasks to achieve the performance and reliability requirements of IoT applications. The "edge cloud" is a service-hosting technology (like a public cloud) but

located at the network edge to enable IoT deployments to take advantage of the spatial locality to optimize resource utilization, reduce required wide-area bandwidth, reduce response latency, improve application fault resilience, and improve security. To maximize its benefits, an efficient scheduling system that intelligently places workloads across IoT, edge cloud, and private/public cloud resources is indispensable. In this thesis, we report our research on building a scalable, event-driven, geo-distributed intelligent scheduling system for heterogeneous IoT devices and applications at the network edge.

To achieve the goal, we investigate the efficacy of using a serverless computing platform for tuning machine learning applications in parallel. We also research the usage of serverless computing across the edge and private/public cloud deployments for intelligent scheduling. A third investigation focuses on controlling the system temperature of edge cloud resources by dynamic voltage and frequency scaling (DVFS) to prevent overheating in environments hostile to computational infrastructure. Our work contributes to the corpus of computation offloading research for cyber-physical systems (CPS) that pairs workloads and resources among an available pool of heterogeneous IoT devices, edge cloud resources, and private/public cloud resources.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

I want to stand as close to the edge as I can without going over. Out on the edge, you see all the kinds of things you can't see from the center.

—Kurt Vonnegut

Thanks to the availability of inexpensive, low-power, and relatively high-performance processors and ubiquitous wireless networks, the Internet of Things (IoT) is revolutionizing the generation, streaming, processing, storage, and analytics of large-scale and real-time data. The Internet of Things is a rapidly emerging set of technologies that enables ordinary objects to collect, analyze, and share data across networks with digital intelligence. Connecting the physical and digital worlds, IoT has the potential to improve environmental perception and proactive decision-making without human intervention, to diagnose and remediate health care, to assist with vertical and precision agriculture, to facilitate virtual learning and online education, and to optimize operational procedures throughout the

economy. To realize this impact, IoT devices are being integrated into all aspects of our daily lives, from offices and residences to automobiles and wearables, and globally deployed across urban and rural areas, waters, and open spaces, where they form a vast, geo-distributed, and heterogeneous IoT systems.

Further, as IoT devices increasingly leverage recent technological advances in real-time data streaming, analytics, and machine learning (ML) applications, their development and deployment are faced with unprecedented challenges. These challenges result from

1. the large scale of concurrent requests and network I/O placing enormous demands on underlying resources,

2. immense variations and seasonality in workload that must be processed and analyzed in real time,

3. high latency and frequent unavailability caused by heterogeneous wide-area networks and failures in globally distributed resource collections, and

4. the large heterogeneity of devices leading to programming, debugging, and maintenance complexities across systems.

One promising approach to addressing these challenges focuses on highly concurrent event-driven architectures (EDAs) which combine aspects of asynchronous,

multi-threading, and event-based programming models to manage underlying resources through concurrency and careful scheduling. By decoupling interoperable services (i.e. emitters, routers, and consumers), EDAs effectively manage the underlying resources based on incoming event sourcing; they also enable the independent scaling and partition tolerance that are necessary to make distributed systems reliable; they address the heterogeneity of devices on the system level by the isolation of services, with messaging queues and well-understood REST APIs.

Serverless computing (a cloud service and execution model) has emerged as the next generation of the event-driven system that readily addresses challenges IoT systems are facing. Serverless computing, also known as Functions-as-a-Service (FaaS) is a popular cloud service for hosting and autoscaling applications in a simple and cost-efficient fashion. Originally designed to unload capacity planning, configuration, and maintenance burden from developers, serverless computing provides an event-driven programming model and cloud platform, in which developers write simple functions that are triggered by predefined incoming events, including database transactions, detected anomalies, web requests, state changes, etc. The advantages of serverless computing are reflected in automatically provisioning isolated environments via containers and dynamically adjusting capacity during execution (i.e. autoscaling), which greatly reduces the idle resources across the system. Based on these features, serverless computing's billing model, pay-as-you-

go, only charges the application owner for the time and memory consumed during execution, so that users never pay for over-provisioning and unused resources. In light of its utility as a highly efficient execution model, public cloud providers and open source communities have offered multiple FaaS platforms built upon this paradigm shift (Apa (2021), Ope (2021), Fis (2021), Fn (2021)).

Moreover, serverless computing has been extended to work at the "edge" of networks (Mic (2021), AWS (2021*a*))

1. to reduce latency between devices and the off-board processing and storage resources that they require,

2. to optimize computing resources utilization, and

3. to reduce storage and bandwidth used by data-driven IoT applications.

During an execution cycle, the system rapidly scales up parallel containers, in response to concurrent requests from on-site streaming sensors, at the "crests" of the offered workload and scales down at the troughs. Edge-based serverless execution also saves data transfer time between IoT and data centers that dramatically reduces the end-to-end latency, particularly in real-time data streaming settings. All serverless functions communicate, persist, and access data only through their inputs or via shared storage services (e.g. an external object store, database, etc.)

As a result, serverless computing is inherently elastic and can implement highly concurrent and parallel tasks for IoT applications.

Linking IoT devices and data centers is an intermediate level of resources at the network "edge" (which we term an "edge cloud,") takes advantage of the spatial locality to boost performance, availability, robustness, and security of cloud computing systems. Typically, the computing and storage resources are scarce on terminal IoT devices, relative to resource-rich public and private clouds. This disadvantage becomes aggravated as public/private clouds may offer specialized hardware (e.g. GPUs, FPGAs) that can significantly speed up machine learning applications, which makes the in-situ execution on the resource-restricted IoT devices comparatively inefficient. On the other end of the spectrum, however, the large volume of data transferred by the long-haul, intermittently available networks that connect the IoT device and public cloud causes high response latency and non-trivial operational costs. Such a trade-off describes a need for an edge cloud near IoT devices that offers powerful and cheap computing and storage resources at a smaller scale than that supported by public clouds.

To maximize these advantages, an efficient scheduling system that intelligently places and deploys serverless functions across devices, edge cloud, and private/public cloud resources, aiming to reduce idle resources and total execution time latency, is indispensable. It should span heterogeneous IoT devices,

edge, and public cloud systems, serving IoT requests and leveraging specialized hardware acceleration; it needs to underpin extensible and scalable systems that meet Service-Level Agreement (SLA) and Quality of Service (QoS); it should be used to manage real-time data and be driven by event sourcing; and, it has to support geo-distributed deployment without sacrificing performance. To achieve these goals, to utilize the processing power and spatial locality in end-to-end IoT systems with data-driven, large-scale applications, we propose and explore the following thesis question:

Can we build a scalable, event-driven, geo-distributed intelligent scheduling system for heterogeneous IoT devices and applications at the network edge?

To answer this question, we:

- Investigate the efficacy of using a serverless computing platform for tuning machine learning applications in parallel. To date, identifying the "best" configuration for advanced machine learning models is challenging given the large number of configuration options (represented as "hyperparameters") that are typical for state-of-the-art learning models. This tuning activity is embarrassingly parallel, and, as a result, we believe that it is a good fit for the serverless computing model. We design and develop a new system and toolset called *Seneca*. We investigate its efficiency and its ability to simplify

the use of serverless computing for the training, testing, and evaluation of machine learning models.

- Research the usage of serverless computing across the edge and private/public cloud deployments (e.g. hybrid cloud settings). We develop a scheduling system, called the *Serverless TeleOperable Hybrid Cloud* (STOIC), which automatically selects and places workloads across these systems in hope of lowering the system-wide execution latency versus using either system in isolation. We specifically target image-based, object recognition using Tensorflow (Abadi et al. (2016)) for training and inference in this work. We also enable STOIC to leverage specialized hardware (i.e. GPUs) to accelerate the execution and gauge its overhead based on historical data.

- Study and characterize (using dynamic voltage and frequency scaling – DVFS – to control system temperature) when the ambient temperature might cause edge computing resources to exceed the acceptable operational ranges. DVFS is a technique that has been widely studied in the context of "power capping" (Kim et al. (2008), Von Laszewski et al. (2009), Rountree et al. (2012)) - the implementation of a maximum power draw by the system. Out system, called Sparta, automatically exploits the relationship between system power consumption and generated heat. It does so by ad-

justing processor frequency dynamically so that CPU temperatures do not exceed a specified threshold as ambient temperature changes. Subject to the threshold, the system attempts to minimize the application deceleration that frequency adjustments might introduce.

Our research contributes a new computation offloading mechanism for cyber-physical systems (CPS) that pairs workloads and applicable resources using heterogeneous IoT devices, edge cloud resources, and private/public cloud resources. We show that the naive or random allocation of resources leads to poorly-tuned machine learning models, high execution latency, and CPU overheating on edge devices. Moreover, our work introduces a new feedback mechanism that dynamically deploys streaming workloads between edge and public clouds, based on real-time monitoring data and latency prediction. We also develop a novel algorithm that combines Simulated Annealing (SA) and Additive Increase Multiplicative Decrease (AIMD) to accurately control the CPU temperature under the threshold and to accelerate the execution in the meantime. These developments in intelligent scheduling enable the highly scalable execution of machine learning applications on edge cloud and real-time analytics on heterogeneous and geo-distributed IoT systems.

In the chapters that follow, we present and evaluate these contributions. In Chapter 3, we describe Seneca which addresses the scalability of IoT systems us-

ing an event-driven programming model and a serverless computing framework for hyperparameter tuning. In Chapter 4, we present STOIC – our solution for optimizing deployment in a multi-layer geo-distributed cloud that places serverless functions across the edge and private/public cloud using predicted latency to optimize system-wide efficiency. Chapter 5 presents Sparta – a system that addresses robustness and efficiency challenges across heterogeneous IoT and edge devices by implementing a heat-budget-based scheduling framework on edge cloud for machine learning applications. Finally, in Chapter 6, we present our conclusion and plans for future work.

Chapter 2

Background

Don't explain your philosophy. Embody it.

—Epictetus

The Internet of Things (IoT) is the communication infrastructure for interrelated electronic devices, machines, physical objects in the world around us. A thing in an IoT device or system such as a thermostat controlling the temperature of a data center, a drone hovering over farmland to analyze soil quality, a grizzly bear wearing a collar while hibernating in the cave, or a human wearing a glucose monitor. IoT systems enable objects to transfer data over networks and extend them with capabilities to collect, mine, store, analyze data in situ.

Because these devices are embedded in the environment, many are small, resource-restricted, and run on battery power. Moreover, the networks that connect them can support little bandwidth or are intermittently connected. As a result, IoT deployments require new system architectures, programming paradigms,

computing and storage support, and robust data distribution. Since the advent of ubiquitous network connectivity and low-cost computing, the research community has developed a wide range of innovative solutions to address such demand, which includes Event-Driven Architectures (EDAs) for scaling and managing disparate services, serverless computing for function-oriented and provision-free programming and deployment, edge-based computing for low latency data management, actuation, and control, and wide-area analytics for scalable and robust, distributed data processing. In this chapter, we overview these research advances and discuss their implications for IoT systems.

2.1 Event-driven Architectures

IoT systems have scale, cost, and performance requirements that are similar to those of web and cloud systems. However, by 2023, the scale of interconnected IoT devices is expected to reach 29.4 billion *Cisco Annual Internet Report (2018–2023) White Paper* (2018), which in combination with web and cloud systems is likely to generate billions of concurrent requests from to services which in turn will consume massive CPU cycles, memory, energy, and storage. Such use translates into high network I/O, long system latency, congestion, and high cost if deployed using the established monolithic paradigm of cloud computing Welsh et al. (2001),

Michelson (2006). Furthermore, the deployment of myriad sensors, detectors, and actuators intensifies the pressure that computer systems need to shoulder. These IoT devices demands require computational and bandwidth resources to monitor physical infrastructures and provide real-time, data-driven actuation and control Filipponi et al. (2010).

By leveraging efficient deployment and event-triggered computation, Event-Driven Architectures ((EDAs) Michelson (2006)) attempt to facilitate such response at scale. Events are data flows that carry changes in state and trigger functions (i.e. event handlers) in response to web requests made via Application Programming Interfaces (APIs). EDAs transform the computer systems into decoupled state machines that use events for application functionality interoperation. An event-driven system typically consists of event emitters (producers), event sinks (consumers), and event routers (channels) Taylor et al. (2009). Emitters detect, produce, and transfer events to downstream consumers without knowing if consumers are active or how they will process events. Sinks process events by filtering, transforming, and applying the state change to corresponding domains (e.g. database, file system, user interface, etc.) and forwarding events to downstream sinks if necessary. Channels route messages from upstream emitters to designated sinks. Multiple implementations of messaging queues or peer-to-peer protocols can serve as channels in event-driven architecture (Magnoni (2015),

Chun et al. (2018), Mossissa & Rajaravivarma (2003)). The loosely coupled design of EDAs enables parallelism and concurrency for scalability and high throughput in distributed setting Zeldovich et al. (2003), Rast et al. (2010), Dekate et al. (2012). Moreover, it makes event-driven applications more resilient to partial failures George-Williams & Patelli (2016), Wang et al. (2013), Sučić et al. (2011).

EDA systems also provide abstractions that hide the complexities of distributed computing that facilitate programmer productivity (Gerstlauer & Gajski (2002), Kansal et al. (2013), Daleiden et al. (2020)). These abstractions allow developers to focus on application development and innovation, instead of writing custom, low-level operations code to poll, filter, and route events. Deriving from these abstractions, microservice architectures (Thönes (2015)) encapsulate event topics, code, and data in applications that provide fine-grained modularity and scalability (Newman (2015), Dragoni et al. (2017)).

In an IoT context, this microservice-based approach requires workload scheduling services for IoT devices at the network edge. In this dissertation, we discuss how such architectures streamline the communications between IoT devices and services, achieve scalability, and provide useful abstractions to programmers.

2.2 Serverless Computing

Taking inspiration from physics and economics, elasticity in cloud computing (Herbst et al. (2013)) features the ability of a cloud system to load and unload resources on demand in the execution process. This is realized by monitoring one or more representative metrics (e.g. CPU/memory usage) and provisioning resources correspondingly as workloads alter them (Galante & de Bona (2012), Al-Dhuraibi et al. (2017), Lehrig et al. (2015)). Elasticity is useful in IoT settings because IoT applications generate workloads that fluctuate substantially across different execution phases.

Cloud-based elasticity has fueled remarkable innovations that facilitate computing resource virtualization, isolation, scaling, and sharing. Cloud users "rent" containerized resources while sharing the underlying physical resources on a pay-per-use basis in exchange for availability guarantees specified via service level agreements (SLAs). Uniquely, cloud systems can be configured to add and remove (i.e. auto-scale) resources and services automatically, based on the dynamic resource requirements and service needs of executing applications (McGrath & Brenner (2017), Anderson et al. (1995), Baldini et al. (2017)). Furthermore, such elasticity can be useful for enabling workload placement and autoscaling across IoT deployments to use their limited resources most efficiently.

It is challenging, however, to configure complex IoT deployments for application use, and to leverage the auto-scaling that clouds offer. This is the reason why currently clouds are used more for enterprise services (object stores, databases, application servers, etc.) than for IoT applications. To make clouds more amenable to IoT use, cloud providers have started to offer programming and execution environments (e.g. AWS (n.d.*a*), Int (n.d.), Azu (n.d.*b*), Bos (n.d.), General Electric (n.d.)) that obviate the need for server configuration and that connect clouds to the network edge, under the *serverless* moniker (Jonas et al. (2017), Hellerstein et al. (2018), Wang et al. (2018)).

Serverless platforms attempt to automate and simplify event-driven computing (AWS (n.d.*b*), Azu (n.d.*a*), Goo (n.d.), Apa (2021), Ope (2021), Fis (2021), Fn (2021)). Serverless (also referred to as Functions-as-a-Service (FaaS)) was initially developed by Amazon Web Services (AWS) (via a platform called Lambda AWS (2021*b*)), to automatically configure, manage, and scale web and cloud applications to significantly simplify cloud use. Using the serverless model, application developers upload arbitrary computations in high-level languages as stateless functions to cloud-hosted, serverless platforms, where functions are triggered automatically by the cloud in response to updates from other cloud services (e.g. storage, queues, notification services, and API gateways, among others). Serverless functions typically execute under a time-bound (e.g. 15 minutes) and allocated

memory size (e.g. 3 GB) or else the platform will terminate the function. In public clouds, users pay a small fee for the resources their functions use during execution, resulting in very low-cost cloud use. Although now available from all public cloud providers and as open-source for private cloud systems (Mic (n.d.), GCP (2021), Apa (2021), Ope (2021), Fis (2021), Fn (2021)).

Leveraging event-driven architectures, serverless systems provide abstractions for functions that are triggered by events from external network requests or the use of internal cloud services. This programming model significantly improves agility and productivity allowing the programmer to focus on application instead of provisioning and operations. In our work, we leverage serverless computing for IoT deployments and as an infrastructure to enable the scheduling and placement of IoT applications across edge and cloud systems.

2.3 Edge Computing and Edge Clouds

The trend of pervasive computing led by wide IoT adoption has extended to diverse areas, such as manufacturing, agriculture, automotive, retail, etc., that make cloud-native applications more latency-sensitive depending on computing workloads. The former requires powerful computing and memory resources, which are normally not available in situ ("in the wild") across deployments of resource-

restricted IoT devices. Meanwhile, the latter requires high bandwidth and fast networking that are often unavailable in IoT wireless settings. Both factors affect IoT applications often leading to either skyrocketing operational cost for end-to-end systems or the violation of service-level agreement (SLA) from users (Singh & Viniotis (2016), Li et al. (2012)).

Since the centralized cloud computing architecture is unable to meet the latency and connectivity requirements of many IoT applications, recent distributed systems advances place computing and storage "near" IoT devices, at the "edge" of the network. Edge systems or edge clouds embed increasingly capable (cloud-like) services and application functionality within networks of resource-constrained devices and sensors (Shi et al. (2016), Shi & Dustdar (2016), Satyanarayanan (2017), Vaquero & Rodero-Merino (2014), Yi, Hao, Qin & Li (2015), Yi, Li & Li (2015). Such co-location of processing infrastructure and IoT devices significantly reduces the latency between data acquisition and device actuation, enables the extension of device capability via local offloading, and alleviates the cost, power consumption, and congestion of network use versus the centralized, cloud-directed model (Hassan et al. (2018), Bonomi et al. (2012), Simanta et al. (2012), Verbelen et al. (2012)).

One key challenge with building edge clouds is the efficient management of dynamically changing resource demand by applications. Intelligent scheduling

mechanisms are needed to construct and adapt deployments that can respond to dynamically changing resource performance and application demand. Scheduler monitor, store, and interpret key performance indicators (KPIs) from real-time execution, including CPU usage, memory footprint, network I/O or server temperature, etc. (Cao et al. (2019), Kumar et al. (2013)). Using this data, schedulers adjust and adapt resource use and application placement across the heterogeneous devices and tiers (sensors-edge-cloud) of IoT deployments, to reduce latency, energy use, and end-to-end application time. Examples of schedulers for multi-tier deployments that operate in this way include Yang et al. (2019), Tuli et al. (2020), Kaur et al. (2019).

Edge clouds also deliver numerous services and functionality to IoT applications, including those for data analysis and machine learning. Edge-based data analytics and machine learning are prevalent in IoT systems, particularly in real-time streaming environments, featuring facial recognition in surveillance (Muslim & Islam (2017*a*)) and quality control in manufacturing (Feng et al. (2020)). These computation-intensive data analytics applications are offloaded from IoT devices to the edge cloud based on the workloads, network traffic, and server saturation. The offloading brings more computing power from the edge cloud to expedite data analytics and machine learning applications, which is critical to provide feedback in real-time settings. Other uses include applications on mobile devices, VR/AR

glasses, autonomous vehicles, and smart cities (Zhang et al. (2017), Liu et al. (2019), Filipponi et al. (2010)). In these scenarios, the edge cloud reduces the volumes of transfer data and, subsequently, the latency and transmission cost. For complex applications that require massive computing power (e.g. Convolutional Neural Network (CNN) training), the public cloud offers better computing resources that offset the latency caused by transfer time, so that an optimal scheduling scheme depending on workload is the key to efficient edge cloud systems.

2.4 Wide Area Analytics

As the convergence of sensing and automation with physical objects becomes ubiquitous, data is increasingly collected from and distributed across the globe and requires efficient data composition and wide-area analytics to perform analysis, actuation, and control at the edge of the network. In addition, the large-scale datasets are replicated across different data centers to improve scalability, robustness, and fault-tolerance. As a result, by spanning multiple tiers (sensors, edge, cloud) and the wide-area, IoT deployments have become vastly heterogeneous and challenging to manage.

Generally, two approaches are used to simplify wide-area data analytics: batch processing (e.g. MapReduce (Dean & Ghemawat (2008))) and streaming (e.g. Apache Kafka (Kaf (2021))). Batch processing periodically groups dataset into sizable batches and transfer them from storage on-site to data centers for analytics, in which the grouping of data reduces the total round-trip time compared to streaming processing (Shahrivari (2014)). So that, it keeps IoT devices from executing heavy-duty tasks, whereas the remote data centers execute the most computing tasks in exchange for timeliness. Stream processing generates, analyzes, and stores real-time sensing data on the IoT devices and edge cloud for timely delivery. One of the drawbacks of this scenario is that it requires abundant computing and storage resources on embedded devices. A combination of these two processing scenarios, taking both timeliness and resource efficiency into account, presents a promising solution to this trade-off (Pfandzelter & Bermbach (2021)).

The research on wide-area analytics in a geo-distributed network merges edge computing, computational offloading, and application-aware network that balance between responsiveness and accuracy in the system. Many algorithms and frameworks are proposed for more efficient and energy-constrained computational offloading mechanism (Fahim et al. (2013), Flores et al. (2018), Shiraz et al. (2015)). A variety of real-world applications, including distributed video analytics, Aug-

mented Reality / Virtual Reality, smart grid, and appliances, can be optimized by the advance in this field to improve availability and efficiency (Wang & Li (2017), Zhang et al. (2018), Hung et al. (2018)).

This research lays the groundwork for building scalable, resilient, and geo-distributed IoT systems with intelligent scheduling capability: Event-driven architectures enable the system to elegantly scale and decouples different components of complex IoT applications; serverless computing provides a modular and provisioning-free programming model for programmers; edge clouds provide in situ computing support for low latency data processing and analysis. In this dissertation, we investigate how to combine these advances in novel ways to provide intelligent scheduling for IoT applications at the network edge.

Chapter 3

Fast, Low-Cost Hyperparameter Search for Machine Learning Models

We are more often frightened than hurt, and we suffer more in imagination than in reality.

—Seneca

In this chapter, we investigate tuning machine learning models efficiently using the large-scale event-driven system. Since the training and inference of machine learning models are typically computation-intensive, event-driven systems have not been widely applied in this field with a concern that they come with high cost and overhead. Meanwhile, hyperparameter tuning is central to the machine learning process, where hyperparameters affect the speed of training and the quality of the model. Some examples of governing hyperparameters are learning rate,

Parts of this section are adapted, with permission, from IEEE CLOUD 2019 Zhang et al. (2019)

number of hidden layers, activation functions, etc. They are different depending on the model structure and training datasets, and the space of applicable hyperparameter combinations is typically very large. Given the parameter sweeps are embarrassingly parallel, we believe the event-driven serverless computing is suitable for hyperparameter tuning ahead of model training. To realize this potential and evaluate the overhead, we design and develop a hyperparameter tuning system, called Seneca, based on serverless computing.

Seneca implements, packages, and deploys machine learning applications as stateless functions to AWS Lambda. It then orchestrates exhaustive evaluation of specified hyperparameter settings to identify the best performing model for a given dataset by comparing error and accuracy across models. We consider prediction accuracy (as opposed to explanatory power) as the scoring metric (mean squared error for regression and accuracy percentage for classification), to avoid overfitting. Users present Seneca with their application, a range of values for each hyperparameter (or the default can be used), and a representative dataset. Seneca produces, tests, and evaluates models for all combinations of hyperparameters and returns to the user the set of parameters or the model itself that produces the best cross-validation score. Users can employ this model for other datasets (with Seneca if desired) without retraining the model to amortize the cost of Seneca further.

We deploy Seneca on AWS Lambda and evaluate its tuning performance, cost, and memory use for five machine learning applications and datasets. We find that Seneca is fast, inexpensive, and effective for model construction and comparison. Seneca is also able to identify automatically the best memory configuration for each application, further lowering its cost by 10-35%. Relative to execution in AWS EC2, Seneca enables average speedups of 294x for each additional dollar spent for the applications and datasets we consider. We intend to make Seneca, its applications, and its datasets publicly available in foreseeable future. In the following sections, we next overview our design and implementation of Seneca and then present our empirical methodology and results.

3.1 Seneca

The Seneca pipeline consists of packaging, deployment, function optimization, and hyperparameter tuning. Figure 3.1 shows the architecture of Seneca. In the upper-right-front, we show the three inputs that Seneca expects from its users: (A) a hyperparameter configuration file, (B) a dataset URL, and (C) the lambda function of the machine learning application. The configuration file specifies a set of values for each hyperparameter that the application expects. Seneca creates the Cartesian product of all options in this configuration as the search space. The

dataset URL refers to a valid dataset stored in the AWS Simple Storage Service (S3).

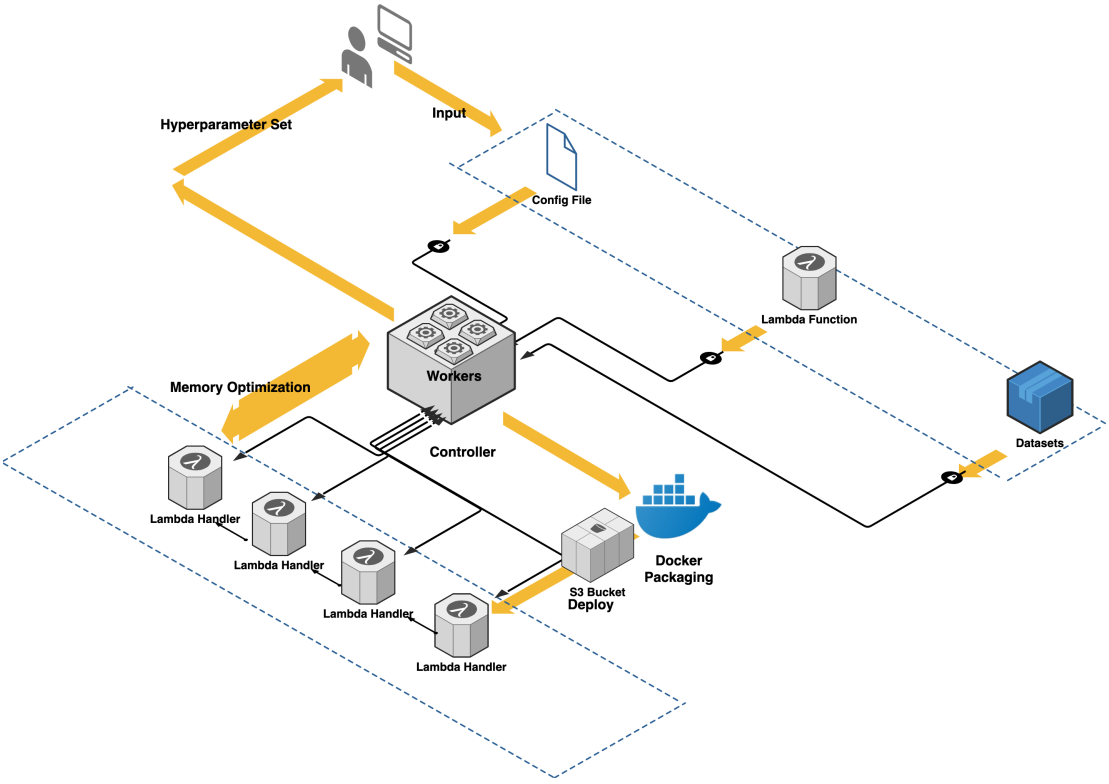

Figure 3.1: The Seneca Architecture

Based on the specified machine learning application, Seneca automatically builds and deploys an AWS Lambda application by launching a Docker container that mirrors the AWS Lambda execution environment, checks and installs the machine learning application and any libraries it requires compresses the application and uploads it to S3 (a work-around for the 10MB AWS Lambda function size restriction). Seneca constructs an AWS Lambda function from a template

that, when executed, will download the dataset and split it into a training and testing set, and construct, test, and evaluate a model using the application and set of hyperparameter values passed in by Seneca as arguments. Users can specify the train/test split ratio that should be used by Seneca; the default is 80%/20% for classification tasks. The function returns a testing score. Upon completion of this process, the container deploys the function to AWS Lambda using the AWS Command Line Interface and the developers' credentials.

3.1.1 Optimizing Memory Use

The cost of using AWS Lambda (i.e. `compute charge`) is the billed duration (execution time rounded up to the nearest 100ms) multiplied by the allocated memory of each invoked function. One goal of our work is to optimize the memory use of these applications to reduce cost and to investigate the trade-offs of doing so.

Currently, allocated memory for a Lambda function can be set from 128MB to 3008MB in increments of 64MB. AWS documentation states that Lambda allocates CPU to functions corresponding to allocated memory size, as is done for general purpose AWS EC2 instance types.

To evaluate the relationship between memory, CPU, and cost, we analyze a 3-D matrix multiplication serverless benchmark using AWS Lambda. We config-

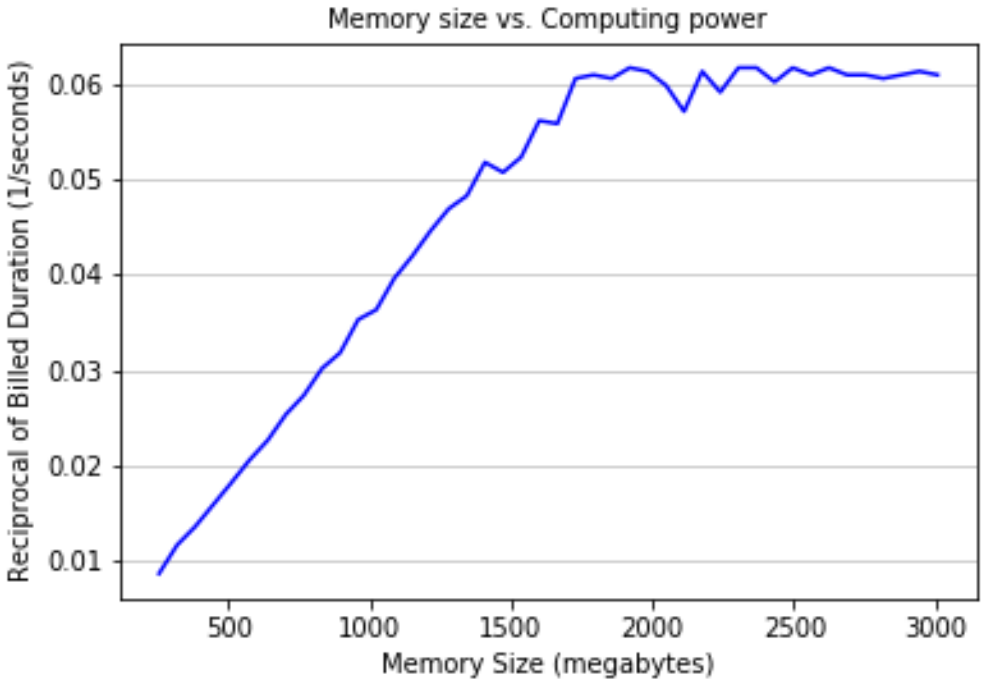

Figure 3.2: The relationship between allocated memory and reciprocal of billed duration, which represents compute power for a compute-bound Lambda function.

ure different functions to use each of the 46 possible allocated memory options. Figure 3.2 shows the relationship between allocated memory (x-axis) and reciprocal of billed duration (y-axis). Figure 3.3 shows the relationship between memory size (x-axis) and compute charge (y-axis). We observe that for this benchmark, billed duration plateaus after 1600 MB, at which point compute charge increases. That is, we achieve no further execution time benefit (only cost increase) after this point.

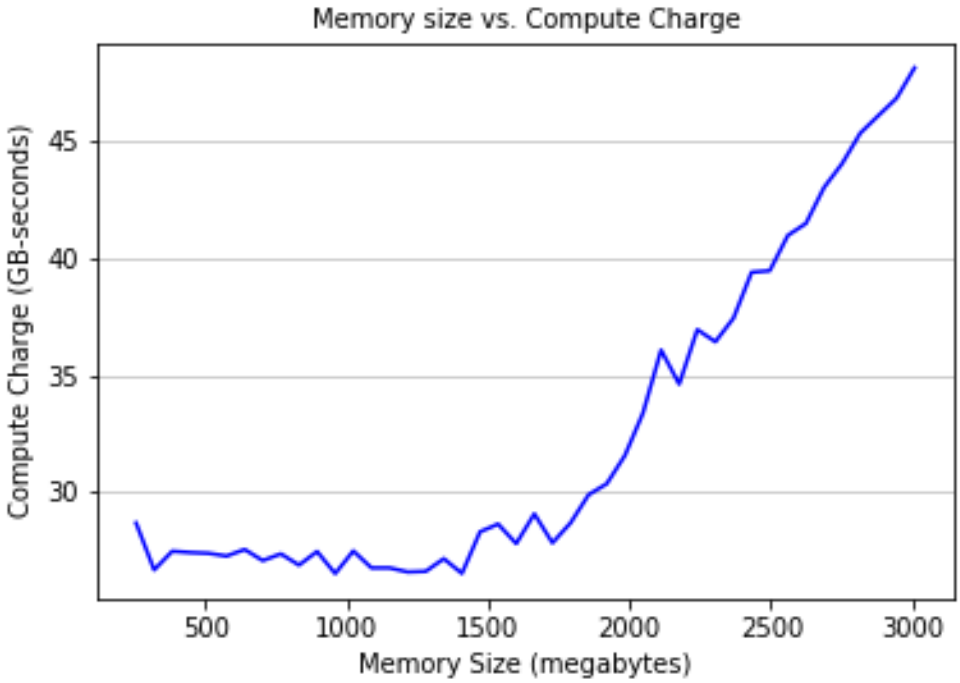

Figure 3.3: The relationship between allocated memory and compute charge for a compute-bound Lambda function.

We use this relationship within Seneca to optimize its cost (`compute charge`) via an extension that enables it to automatically identify the appropriate setting for allocated memory for each application. However, instead of exhaustively testing all 46 possible memory configurations as we did for the matrix benchmark, which may be costly, Seneca employs the heuristic outlined in Algorithm 1.

The Seneca optimizer first configures and invokes the function using a user-defined payload. From this run, Seneca obtains the `maximum memory used` by the function as reported by AWS Cloudwatch, and uses it as the starting point

in its search. Seneca then defines two double-ended queues (*deque*) of length N, to store `allocated memory` and `compute charge` data of different invocations. While the currently allocated memory is less than or equal to 3008 MB, the optimizer reconfigures and invokes the function using the next increment for memory allocation. It calculates the compute charge for each invocation using currently allocated memory and billed duration.

We employ two exit conditions. The first is when the compute charge monotonically increases across *deque*. The second is when the increase in slope is greater than a threshold. When the optimizer finds that both conditions hold, it pops the left-most value from *deque* and configures the function to use that value for allocated memory for all future invocations. If these two conditions can not be satisfied during the search, the allocated memory will be configured as the memory size that results in minimal compute charge within the *deque*. After extensive experimentation, we find that $N = 5$ and a slope threshold of 1 works best, but these values are configurable. In addition, this optimization can be turned on or off via a command-line argument to Seneca.

3.1.2 Tuning Process

To facilitate parallel function invocation, Seneca integrates Celery. Celery is an asynchronous task queue that uses distributed message passing. Celery workers

Algorithm 1: Seneca Optimizer Heuristic

```
   Data: Typical payload
   Result: Optimal allocated memory
 1 Find memory used by payload as starting point;
 2 Define deque for allocated memory & compute charge;
 3 while allocated memory ≤ 3008 MB do
 4 |   if compute charge monotonically increases in deque & slope ≥
   |     threshold then
 5 |   |   popleft from deque;
 6 |   |   configure allocated memory as optimum;
 7 |   |   exit();
 8 |   else
 9 |   |   increase allocated memory by 64 MB;
10 |   |   probe lambda function;
11 |   |   append memory and compute charge to deque;
12 |   end
13 end
```

are processes that take tasks from the queue, execute the tasks with the arguments specified, and store the result that is returned in a database (we use Redis in our prototype).

Based on the configuration file, Seneca creates and enqueues a list of payloads (function arguments) for each combination of hyperparameter values. The Seneca celery workers invoke the application's Lambda function by each payload for model construction. Upon function termination, the worker records a score for the hyperparameter configuration in the database. When the queue is drained and all workers have completed, Seneca extracts and reports the best score, configu-

Application	**Description**
Prophet	Time series decomposition and prediction
Multi-Regression	Multiple linear regression/prediction of time series
XGBoost	Regression and classification by gradient boosting
SVC	Classification based on support vector machine
Neural-Net	Classification by layered artificial neural network

Table 3.1: Machine learning applications used to evaluate Seneca.

ration, and model from the database. Users can then use the model for inference given other datasets without retraining to amortize the time/cost of Seneca.

We assume that the dataset supplied to Seneca by the user is representative of datasets on which the resulting model will be used. In addition, we use prediction error as the score (i.e., mean squared error for regression and accuracy percentage for classification) instead of R^2, which describes explanatory power, to avoid overfitting. As part of future work, we are considering using multiple datasets and ranges of hyperparameter values to preclude the need for users to specify them and to consider a wider range of values.

3.2 Evaluation

In this section, we empirically evaluate Seneca in terms of machine learning (ML) model output quality, performance, and cost. We first overview the ML applications that we consider and our experimental methodology. We then present our results.

Hyperparameter	Default	Tuning options
growth	linear	[linear, logistic]
changepoint prior scale	0.05	[0.05, 0.5]
holidays prior scale	10	[1, 5, 10]
seasonality prior scale	0.5	[0.1, 0.5]
fourier order	10	[5, 10, 15, 20]
seasonality mode	additive	[additive, multiplicative]
interval width	0.8	[0.5, 0.8]

Table 3.2: Hyperparameters Seneca considers for **Prophet**.

Hyperparameter	Default	Tuning options
max depth	3	[3, 4]
learning rate	0.1	[0.1, 0.01]
N estimators	100	[100, 400]
objective	reg:linear	[reg:linear, rank:pairwise]
booster	gbtree	[gbtree, gblinear, dart]
min child weight	1	[0.1, 1]
scale positive weight	1	[1, 2]
base score	0.5	[0.5, 10]

Table 3.3: Hyperparameters Seneca considers for **XGBoost**.

3.2.1 Benchmarks and Training/Testing Datasets

The ML applications that we use to evaluate Seneca are described in Table 3.1. Prophet, Multi-Regression, and XGBoost are regression applications; XGBoost, SVC, and NN are classification applications (XGBoost implements both regression and classification tasks). The regression applications compute the mean square error (MSE) as $\frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y_i})$, where $\hat{Y_i}$ is the ground truth, Y_i is model prediction and n is the number of data points. The applications return the average MSE across cross validations. The classification applications compute and return a

Hyperparameter	**Default**	**Tuning options**
C	1.0	[0.5, 1.0]
kernel	rbf	[rbf, linear, poly, sigmoid]
degree	3	[3, 4]
gamma	auto	[auto, scale]
coef0 init	0.0	[0.0, 1.0]
probability	False	[False, True]
tol	1e-3	[1e-3, 1e-4]
decision function shape	ovr	[ovo, ovr]

Table 3.4: Hyperparameters Seneca considers for **SVC**.

Hyperparameter	**Default**	**Tuning options**
activation	relu	[identity, tanh, relu]
solver	adam	[lbfgs, sgd, adam]
learning rate	constant	[constant, invscaling, adaptive]
learning rate init	0.001	[0.001, 0.0001]
power T	0.5	[0.1, 0.5]
tol	1-e4	[1e-4, 1-e5]
n iter no change	10	[10, 20]

Table 3.5: Hyperparameters Seneca considers for **NN**.

classification accuracy percentage, which is calculated as $\frac{1}{n}\sum_{i=1}^{n} 1(Y_i = \hat{Y}_i)$, where Y_i is the prediction class, $\hat{Y}_i$ is the true class, n is the number of samples, and 1(x) is the indicator function.

Prophet (Taylor & Letham (2017)) is an open-source time series analysis library developed by Facebook. The input dataset we consider is a time series of view counts of Peyton Manning's Wikipedia page (Dec. 2007–Jan. 2016). The dataset exhibits both seasonality and a holiday effect (e.g. around super bowl games). We use the first 6 years as the training set and the last 2 years as the

testing set. We use a cross-validation horizon (sliding window) of 1-year and a period (sliding pace) of 180 days. As such, Seneca performs three cross-validations for a 2-year test range.

Prophet expects multiple hyperparameters: *growth* specifies linear or logistic trend model growth and *prior scale* indicates the strength of the sparse prior probability. There are three prior scale hyperparameters for change point, holidays, and seasonality. Since Prophet uses a Fourier sum to estimate seasonality, the *fourier order* is the number of terms in the partial Fourier sum. *Seasonality mode* indicates that the effect of seasonality is either multiplicative or additive. Finally, the width of uncertainty intervals is set using *interval width*.

Each application has default hyperparameter settings (i.e. default values or those recommended by the application maintainer). The hyperparameters, their default, and optional values that we consider for Prophet are listed in Table 3.2.

Multi-Regression is a regression application developed by others as part of an Internet-of-Things (IoT) project (Krintz et al. (2018)) (which has been extended from linear regression described in the citation to multiple linear regression by the authors of this prior work). The application uses multiple linear regression models to predict outdoor temperature from the processor temperature of single board computers (SBCs). The training dataset consists of eight input time series

(one per SBC, each containing 5-minute measurements) from Apr. 5th to Dec. 10th, 2018.

Hyperparameter configuration for Multi-Regression is a subset of input SBC time series. Seneca considers all $2^N - 1$ non-empty potential subsets (for N input time series). For this application, the default parameterization is the full set of input time series (8 in this case). The test dataset is a time series of the outdoor temperature (ground truth) over the same period. The application makes predictions for each of these outdoor temperatures using the regression coefficients constructed from the training set for each new value in the test set.

XGBoost (sen (2021)), SVC (Hsu et al. (2003)), and NN (Haykin (1994)) are the classification applications that we consider. XGBoost (sen (2021)) is an open source framework for gradient boosting, which performs both regression and classification. The hyperparameters and their default values are listed in Table 3.3, their definitions can be found in (xgb (2021)). SVC uses support vector machines to implement classification as part of the libsvm (Chang & Lin (2011)) library. The hyperparameters and their defaults that Seneca uses for SVC in this study are listed in Table 3.4 with definitions in (svc (2021)). NN is a machine learning application leveraging neural network to identify patterns from an input dataset. Here we implement a feed-forward multi-layer perceptron model (Glorot & Bengio

(2010)) for classification. The hyperparameters and their defaults for NN are listed in Table 3.5 with definitions in (nnp (2021)).

For these classification applications, we use a labeled dataset for training, testing, and evaluation from another IoT project collaborating with Lindcove Research and Extension Center (LRE (2021)). The dataset contains measurements of individual citrus fruit (e.g. oranges, mandarins, lemons, etc.) taken by a fruit sorting and grading device using a large number of sensors. The measurements (i.e. features) include size, shape, weight, color, diameter, flatness, among other characteristics, for each fruit. The dataset has been filtered to remove correlated features (those with an absolute value of the Pearson correlation coefficient greater than 0.8). The dataset has been balanced by down-sampling and the resulting dataset contains 33926 rows (individual fruit) distributed evenly across 5 targets. Each row has 18 features. The label identifies the field from which the individual fruit was harvested.

The applications train a model on a random subset (80%) of the data. Each then uses this model to predict the field from which each fruit originates for the remaining 20%. To study the impact of random data split, we consider multiple 80%/20% splits in our evaluation.

3.2.2 Empirical Methodology

To evaluate Seneca, we measure model output quality, execution time, memory use, and monetary cost. For output quality (prediction accuracy) our metrics are mean squared error (MSE) for regression and percentage accuracy for classification as described above. We score model prediction accuracy and no explanatory power (R^2) to avoid overfitting. Seneca can compare models for each type of application using these scores because it uses the same total number of hyperparameters for each model. Thus, the penalty function in terms of Bayesian Information Criteria is the same. comments: BIC is based on parameters, not on hyperparameters.

We compare results for the default, best (Seneca's recommendation), and worst-performing hyperparameter configurations for each application type. Seneca computes all possible combinations of the hyperparameter settings specified in the configuration to extract each of these results. `default` represents results that a novice or first-time user might experience when using these applications as a "black box." The `worst` shows how bad the results can be when parameters are poorly tuned. Finally, the `best` is the upper bound on what is possible from tuning the hyperparameters for the values and datasets specified (e.g. using expert knowledge or Seneca).

Seneca deploys the applications automatically over AWS Lambda and extracts execution time and memory use from AWS CloudWatch logs. We compute mon-

etary cost using the AWS Lambda pricing model. Each function downloads the training/testing dataset of the application from AWS S3 upon function invocation. We do not consider the cost of dataset storage in our cost computations, because it is very small. For XGBoost that makes most S3 requests among others, the cost is 2.5 cents for storage and request combined in a month.

We also evaluate Seneca's automatic memory optimization capabilities. To do so we compare the execution performance and cost of the applications using the maximum allocatable memory size to the performance and cost when run with Seneca's automatically determined memory size. Even though `maximum memory used` reported by AWS CloudWatch can fluctuate, we have verified that the optimized allocated memory is sufficient for all hyperparameter configurations to complete successfully. We have also verified that the memory requirements across hyperparameter settings do not vary significantly. We plan to consider applications for which hyperparameter settings require different maximum memory sizes as part of future work.

3.2.3 Application Efficacy

We first evaluate the quality of the output generated by each ML application when Seneca determines the hyperparameter settings. We first show the results for the regression applications in Table 3.6. The first row of data is the number

	Prophet	**Multi-Regression**	**XGBoost**
# of Combinations	384	255	768
Default MSE	0.284	11.446	0.118
Worst MSE	1.266	43.752	8.981
Best MSE (Seneca)	0.220	9.621	0.065

Table 3.6: Hyperparameter configuration count and MSE for the default, best (Seneca's recommendation), and worst configurations for the three regression applications. For the MSE values (rows 3-5), lower is better.

of hyperparameter configurations that Seneca considers for each. The last three rows show the MSE for the default, worst, and best performing (Seneca's recommendation) hyperparameter configuration (lower is better). Seneca reduces MSE by 22.56%, 15.94%, and 44.88%, for Prophet, Multi-Regression, and XGBoost, respectively, for the datasets and training methodologies that we consider. Compared to the worst case, Seneca reduces MSE by 82.62%, 78.01%, and 99.28%, respectively.

Figure 3.4 shows the MSE box plot for the hyperparameter search space for these applications (lower is better). The central rectangle covers the interquartile range (IQR), which is defined as the range of data points from the first quartile to the third quartile ($Q3 - Q1$). The upper whisker extends to the last datum less than ($Q3 + 2 * IQR$) and the lower whisker extends to the first datum greater than ($Q1 - 2 * IQR$). The data points beyond the whiskers are considered outliers and are plotted as colored diamonds. The red notch identifies the MSE that results from training the model using the default settings of the hyperparameter. The

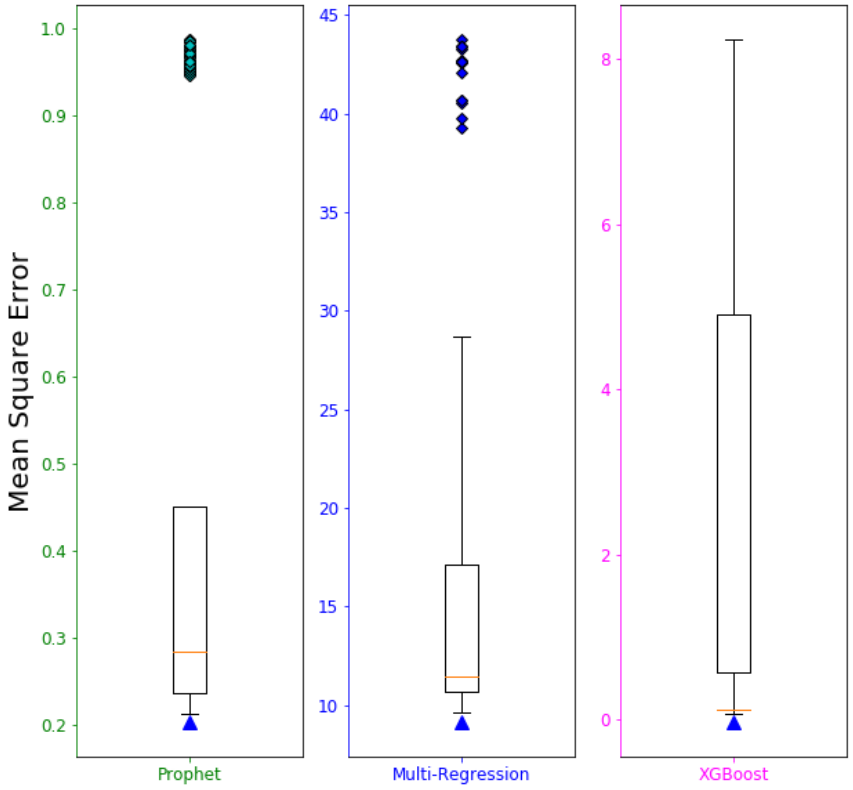

Figure 3.4: Box plot of MSE from the three regression applications across the hyperparameter tuning search space. The red notch shows the MSE from the default settings. The colored diamonds are outliers beyond two interquartile ranges. Seneca selects the points indicated by blue triangle. Lower MSE values are better.

blue triangle identifies the MSE of Seneca. The difference between the red notch and blue triangle is the improvement brought about by the use of Seneca, over using the default parameter setting. The plot also shows that Prophet and Multi-

80%-20%	**XGBoost**	**SVC**	**NN**
# of Combinations	768	512	432
Default Accuracy	95.65%	21.77%	79.53%
Worst Accuracy	0.00%	14.08%	19.15%
Best Accuracy (Seneca)	98.11%	40.81%	83.32%

Table 3.7: Accuracy for the default, best (Seneca's recommendation), and worst hyperparameter configurations for the three classification applications using 80% of the data to train and 20% of the data as a test set. Higher accuracy is better.

	Exec Time (Secs)	**Memory Use (MB)**	**Best Accuracy**
XGBoost_1	1244.42 (32.58)	228.74 (15.92)	98.11%
XGBoost_2	1280.56 (38.47)	225.10 (19.67)	97.70%
SVC_1	116.73 (1.11)	224.44 (19.64)	40.81%
SVC_2	115.33 (3.96)	228.55 (16.20)	44.12%
NN_1	116.10 (6.05)	328.84 (16.40)	83.32%
NN_2	121.29 (2.18)	327.57 (16.44)	83.92%

Table 3.8: The mean and standard deviation (in parentheses) for execution time and memory use (across 30 runs), and best accuracy score for the classification applications using two different random splits.

Regression have a significant number of outliers, indicating that a comprehensive search is critical to finding the best configurations.

We next empirically evaluate Seneca's model output quality for the three classification applications: XGBoost (classification), SVC, and NN. Table 3.7 presents the accuracy percentage (higher is better) for each application (3 right-most columns) for the default, worst, and best (Seneca's recommendation) hyperparameter tuning configurations (data rows 2-4). The first row reports the number of configurations that Seneca considers in its search space. Using a random 80/20 (train/test) percent split, Seneca increases accuracy by 2.46%, 19.04%, and 3.79%, for XGBoost (classification), SVC, and NN applications, respectively. Be-

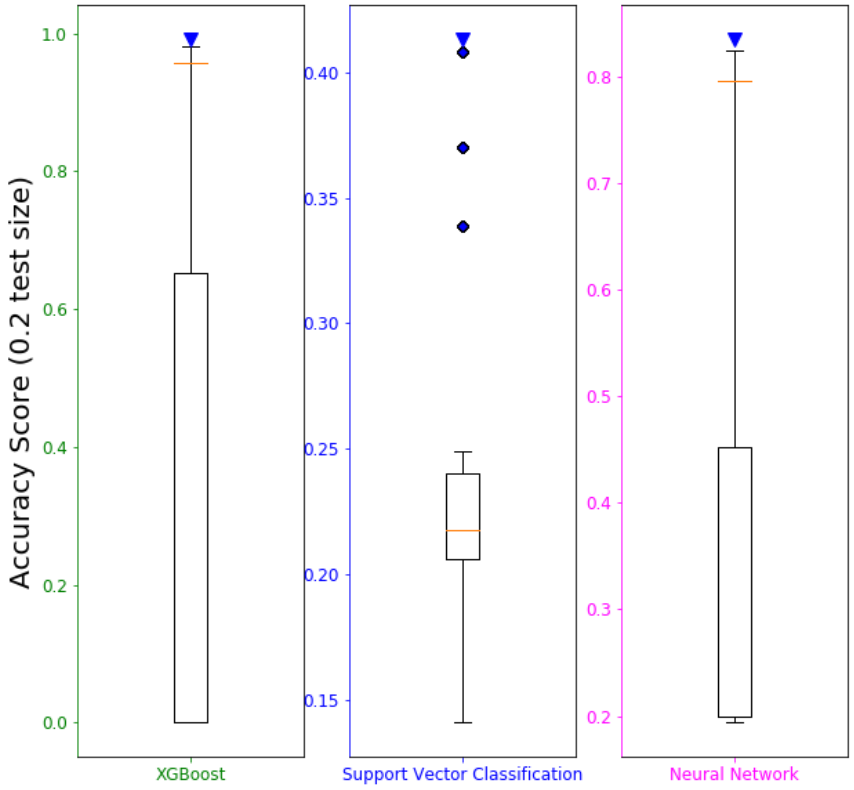

Figure 3.5: Box plot of accuracy reported for three classification applications across the hyperparameter search space. The red notch indicates the accuracy that results from default hyperparameter values. The diamonds are outliers beyond two interquartile ranges. Seneca selects the points indicated by blue triangle. Higher accuracy is better.

cause XGBoost and NN use a well-tuned default parameter set that works well for most datasets, Seneca provides only modest improvements. Compared to the

worst case, however, Seneca improves accuracy by 98.11%, 26.74%, and 64.17%, respectively.

Figure 3.5 presents the accuracy box plot across the hyperparameter search space for these applications (higher is better). The central rectangle covers the first-third quartile ($Q3 - Q1$) and the whiskers span from ($Q3 + 2 * IQR$) to ($Q1 - 2 * IQR$). The red notch indicates the accuracy metric from the model trained using the default settings and colored diamonds show outliers beyond the whiskers. The blue triangle at the top identifies the accuracy percentage reported by Seneca.

The model output quality results across applications show that prediction accuracy (for a given dataset) is dramatically affected by hyperparameter settings. Predictably, the default settings are near the "good" end of the spectrum, however, Seneca can find the parameterization that improves output quality over the default settings in each case.

To investigate the potential impact of Seneca's 80/20 percent data split for the classification applications, we next evaluate the quality of the output generated from each when we consider different 80/20 random splits. For that purpose, we run Seneca 30 times to obtain execution time, memory use, and best accuracy score. We report the mean and standard deviation (in parentheses) for execution time and memory use across runs, and the best accuracy score in Table 3.8.

Each pair of rows shows the results for two different random splits. Our earlier results use input 1; this table adds results for a second, 80/20 random split of the input (we also considered other random splits, which we omit for brevity, and the results are similar). The performance and Seneca score are similar across splits. This result indicates that for these applications, users can repeatedly employ the recommended models for inference on other datasets or splits, to amortize the cost of using Seneca.

3.2.4 Cost Analysis

We next analyze the monetary cost incurred by Seneca with and without Seneca's memory optimization. We consider the use of the maximum allocatable memory (3GB) and Seneca's automatic detection and configuration of allocated memory. This optimization requires that Seneca intelligently probe to determine the best memory size to use. We report the cost of these probes as `Optimizer Cost`.

Table 3.9 shows the results with and without the Seneca memory optimization for each of the five applications. The first two rows show the results when we use the maximum allocated memory for the Lambda functions. We present execution time in minutes (row 1) and monetary cost in cents (row 2). Rows 3–6 show the performance and cost when using Seneca's memory optimization. `Exec time`

	Prophet	**MR**	**XGBoost**	**SVC**	**NN**
Exec time max (mins)	7.78	2.71	20.85	0.87	2.06
Cost max (cents)	22.16	7.58	59.92	2.21	5.92
Exec time opt (mins)	12.60	4.09	29.07	2.08	3.06
Optimizer Cost (cents)	2.02	1.27	0.05	0.04	0.04
Cost opt (cents)	17.81	4.47	39.76	1.38	4.39
Total Cost (cents)	19.83	5.74	39.81	1.43	4.44
Savings (cents)	2.33	1.84	20.11	0.78	1.48
Savings (%)	10.49%	24.23%	33.57%	35.42%	24.98%

Table 3.9: Seneca Memory Optimization: Rows 1–2 show the execution time and monetary cost of using Seneca without its memory optimization (allocated memory = 3G). Rows 3-6 is the execution time and cost, respectively, when using the Seneca memory optimizer. Rows 7-8 show the savings in cents and percentage, respectively.

`opt` is the execution time in minutes. `Optimizer Cost` is monetary cost in cents of Seneca's memory size detector. `Cost opt` is the monetary cost in cents of using Seneca's memory optimizer. `Total cost` is the overall cost of using Seneca to perform hyperparameter tuning for these applications and datasets (sum of Optimizer Cost and Cost opt). The last two rows show the monetary savings in cents (row 7) and percent savings (row 8) of using Seneca's memory optimization. Seneca's memory optimization reduces the monetary cost of its use from 10–35% (25% on average).

Table 3.9 illustrates two important points. First, using AWS Lambda, full hyperparameter space exploration is inexpensive in *absolute* dollar-cost terms, and Seneca's automatic memory size optimization decreases this cost further. Second, memory optimization reduces cost but can increase the total execution time

	Prophet	**Multi_Reg**	**XGBoost**	**SVC**	**NN**
EC2 exec time (mins)	73.79	21.99	359.87	7.74	15.92
EC2 total cost (cents)	8.30	4.20	25.00	4.20	4.20
yield	50.86	340.23	83.36	0.00	1875.56
ideal yield	25.43	170.12	41.68	0.00	937.78

Table 3.10: Seneca VS EC2 cost analysis. Execution time (mins) and cost (cents) for executing the applications serially in EC2 (t2.medium). Rows 3–4 show yield – the additional speedup that Seneca can achieve for each additional dollar spent for these applications. The yield for SVC is 0 (infinite) because Seneca costs less than EC2 in this case. Ideal yield shows the yield when we execute the applications in parallel (assuming 2x perfect parallelism).

for parameter search since the functions must operate under additional memory constraints (versus using the maximum allocated memory). In addition, this cost fluctuates depending on the quality of the Lambda execution environment (number of CPUs, Linux container overhead, multitenancy, etc.). We omit this data due to space constraints but analyze it here. The average absolute difference in cost across the five applications (30 runs) is $0.05. Moreover, we have verified that the highest cost of execution under optimized memory is still cheaper than the lowest cost of execution under maximum memory for all five applications.

Finally, we compare the cost of Seneca to AWS Elastic Compute Cloud (EC2) use. We measure the execution time of Seneca using the least expensive EC2 instance type in which the applications will run (t2.medium, which has 2 multi-tenant cores and 4GB of memory). Note that EC2 instances are charged for by the hour; Lambda charges are only imposed when functions execute. We execute the applications serially using the instance. For this setting, Seneca enables a speedup

over EC2 of 3.72x – 12.38x (6.51x on average). Doing so, however, imposes an additional cost of \$0.01–\$0.15 over EC2 for all but SVC. Seneca costs \$0.03 less than EC2 for SVC (because SVC runs in significantly less than the next hour boundary).

To further understand the relationship between Seneca speedup and cost when Seneca is more expensive (but faster) than using EC2, we define `yield` as $Y = \frac{T_{ec}}{T_{sc}}/(C_{sc} - C_{ec}) \mid if\ C_{sc} > C_{ec}$ where T_{ec} and T_{sc} are the execution time, C_{ec} and C_{sc} are the total cost of EC2 instance and Seneca, respectively. For applications for which Seneca is cheaper (e.g. SVC), we report `yield` as 0.00. This metric reveals the amount of speed up that Senence can achieve for each additional dollar spent. To understand the yield if we were to parallelize the EC2 deployment (perhaps a more "fair" comparison), we also estimate yield for perfect parallelism (2x in our case for the t2.medium).

We present results for this yield metric in Table 3.10 for each of the applications. Rows 1 and 2 show the average execution time (in minutes) and cost (in cents) from using EC2 for each. Rows 3 and 4 show the Seneca yield (speedup/\$). Row 3 shows yield for serialized execution in EC2 (t2.medium instance) and row 4 shows estimated yield if we were to achieve perfect parallelism(i.e. 2x) using the EC2 instance. On average across the four applications for which EC2 is cheaper, Seneca achieves a yield of 294 (assuming perfect parallelism in EC2). That is,

Seneca can provide a speedup of 294x on average, for each additional dollar spent for these applications. We plan to compare Seneca's cost and performance to other EC2 instances and the AWS Elastic Container Service as part of future work.

Overall, given the AWS Lambda pricing model and its Lambda performance variability, Seneca is still able to find the sweet spot between cost and execution time. Thus Seneca can be used to trade-off time-to-solution for cost as desired by users, to automatically evaluate the impact of hyperparameter settings for ML models.

3.3 Related Work

As related work, we consider recent advances in evaluating serverless computing for different application domains, automatic deployment for serverless, and machine learning (ML) model optimization. For the former, much work has investigated the efficacy and overhead of the serverless programming model and implementations (Jonas et al. (2017), Hellerstein et al. (2018), Baldini et al. (2017), Lin et al. (2018)). The authors identify challenges with using AWS Lambda to train machine learning (ML) models. Our work, however, shows that it is possible to leverage the concurrency and parallelism in AWS Lambda to perform a fast grid search for the subset of ML applications that we consider.

PyWren (Jonas et al. (2017)) uses serverless for different distributed computing models. The technique abstracts away cluster management overhead and is ideal for embarrassingly parallel jobs. ExCamera (Fouladi et al. (2017)) presents a framework for running general-purpose parallel tasks (encoding 4K video) on a commercial serverless platform using multithreading. Cirrus (Jonas et al. (2019)) attempts to train ML models using a parameter server and serverless functions.

The serverless framework (Ser (2021)) provides automated packaging and deployment for serverless functions across clouds. GammaRay (Lin et al. (2018)) does so for AWS Lambda to insert profiling instrumentation. The serverless framework uses CloudFormation (Clo (2021)) for deployment in AWS Lambda, which introduces additional cost. The cloud infrastructure provisioning framework Terraform (Ter (2021)) also provides automated deployment of functions to serverless platforms. Seneca uses a local Docker container to avoid cost and overhead (vs these related works), which guarantees execution compatibility for AWS Lambda.

Automated hyperparameter tuning is the focus of many projects. Google Vizier (Golovin et al. (2017)) provides a service for black-box optimization. Optunity (Claesen et al. (2014)) and Hyperopt (Hyp (2021*b*)) provide a Python library for hyperparameter tuning. Hyperas (Hyp (2021*a*)) adds another abstraction layer to hyperopt to facilitate hyperparameter tuning for Keras. However,

we are not aware of any work that leverages serverless to perform hyperparameter tuning and memory optimization in parallel for ML applications.

3.4 Conclusion

We present a new framework, called Seneca, for simplifying and expediting the training and testing of machine learning models in AWS Lambda using low-cost cloud services. Specifically, Seneca leverages the AWS Lambda for autoscaled (elastic) and parallel execution of hyperparameter search and selection for machine learning models. Users provide Seneca with the application code and libraries, 1+ datasets, and a list of possible hyperparameter settings. Seneca uses this information to automatically configure and deploy these functions concurrently for all possible combinations of hyperparameter values specified. Seneca returns the best scoring model and configuration to the user for future use on other datasets.

We present the design, implementation, and cost optimization for Seneca. The optimizer automatically optimizes function memory use to reduce the cost of AWS Lambda use. Our empirical evaluation using multiple applications for regression and classification shows that Seneca can quickly identify the best performing hyperparameter setting for the applications and datasets that we consider. We also find that Seneca enables average speedups of 294x for each additional dollar spent

and that its memory optimization reduces the cost of using Seneca by 10-35% for the applications studied.

Chapter 4

Edge-Adaptable Serverless Acceleration for IoT Applications

It never ceases to amaze me: we all love ourselves more than other people, but care more about their opinion than our own.

—Marcus Aurelius

In this chapter, we investigate the use of serverless computing across the edge and public cloud deployments (i.e. in 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.

Parts of this section are adapted, with permission, from IEEE PerCom 2020 Zhang et al. (2020) and Software: Practice and Experience 2021 Zhang et al. (2021*a*)

STOIC automatically places serverless functions at the edge (without GPUs) or in public cloud instances (equipped with 1+ GPUs) using predicted solution 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 deployment over 24 hours.

In summary, with this research, 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 mechanisms 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. We then present the design and implementation of STOIC, following by our experimental methodology and empirical evaluation of the system and application workloads, using a distributed serverless deployment. Finally, we present our conclusions and future work plans.

4.1 Related Work

A significant body of work (Pu et al. (2015), Vulimiri et al. (2015), Cuervo et al. (2010)) 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 (McMahan & Ramage (2017)), 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. Hellerstein et al. (2018) and Jonas et al. (2019) 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. Ishakian et al. (2018) provides a prototype for a deep learning model serving in a serverless platform. Naranjo et al. (2021) 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 acceleration and distributed function placement across the edge and public clouds. Mohanty et al. (2018) evaluates several serverless frameworks that use Kubernetes to manage and orchestrate the 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. Muslim & Islam (2017*b*) 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. Teerapittayanon et al. (2017) proposes a distributed deep neural network that allows fast and localized inference at the edge device using truncated layers of a neural network. Lv et al. (2019) 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, Paščinski et al. (2017) proposes a Global Cluster Manager for orchestrating network-intensive programs within Software-Defined Data Centers (SDDCs) targeting high Quality of Service (QoS) and, further, Kochovski, Drobintsev & Stankovski (2019) classifies available cloud deployment options by a stochastic Markov model, namely Formal QoS Assurances Method (FoQoSAM), to optimize the automated offloading process. Due

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

to its practical utility, such a method can guarantee that QoS requirements are satisfied. Kochovski, Gec, Stankovski, Bajec & Drobintsev (2019) proposes a fog computing platform (DECENTER) and a trust management architecture based on Smart Contracts. Related to this work, Cech et al. (2019) develops an architecture (HCL-BaFog) by the blockchain functionality to share sensor data. Table 4.1 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 (Lin, Bakir, Krintz, Wolski & Mock (2019), Lin, Krintz & Wolski (2019), Lin et al. (2018), AWS (n.d.*c*)). FaaSProfiler (Shahrad et al. (2019)) integrates testing and profiling within a FaaS platform. Xiao et al. (2018) proposes a security solution that applies reinforcement learning (RL) to provide secure offloading to the edge nodes to prevent jamming attacks. These related

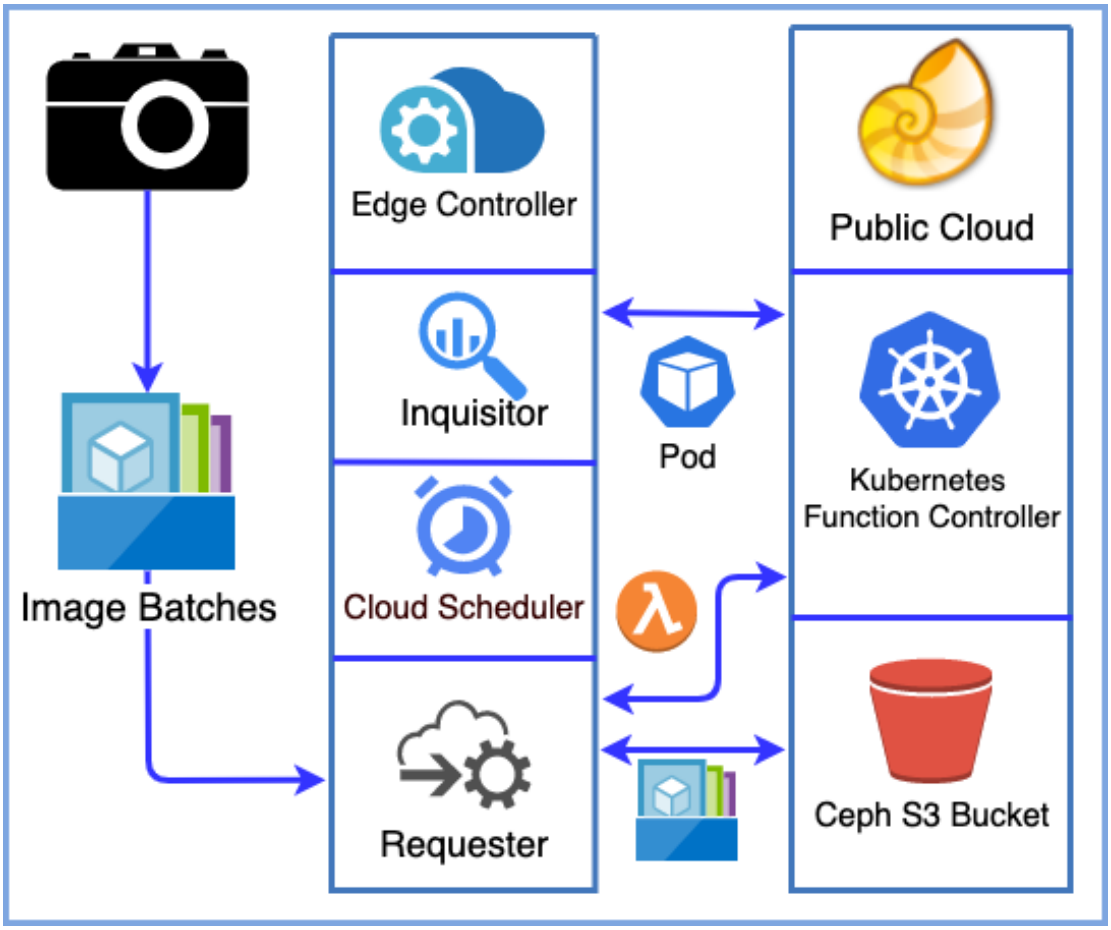

Figure 4.1: The STOIC Architecture

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

4.2 STOIC

To leverage hardware acceleration and distributed (multi-cloud) scheduling within a serverless architecture, we have developed STOIC, a framework for dis-

tributing 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 the result retrieval for machine learning applications in these settings.

Figure 4.1 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* (2021), Burns et al. (2021)) overlaid with kubeless (kub (2021)), 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.

4.2.1 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 the UCSB campus (which has full Internet connectivity) via a microwave link. When a camera trap detects motion, it takes photos and persists the images in a 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* (2021)) 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 (nuc (2021)) (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 (euc (2021)), 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.

4.2.2 Public/Private Cloud

To investigate the use of the serverless architecture with hardware acceleration, we employ a shared, multi-university, GPU cloud, called Nautilus (nau (2021)), 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 (Burns et al. (2021)) to manage and scale containerized applications. It also uses Rook (roo (2021)) to integrate Ceph (Weil et al. (2006)) 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 holds 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.

4.2.3 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* (2021) support; **(B)** *CPU* - A Kubernetes pod on Nautilus containing a

single CPU with AVX2 support *AVX2* (2021); **(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.

4.2.4 Execution Time Estimation

As depicted in Figure 4.1, 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 $T_s = T_t + T_d + T_p$.

Transfer time (T_t)

T_t measures the time spent in transmitting a compressed batch of images from the edge controller to the 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

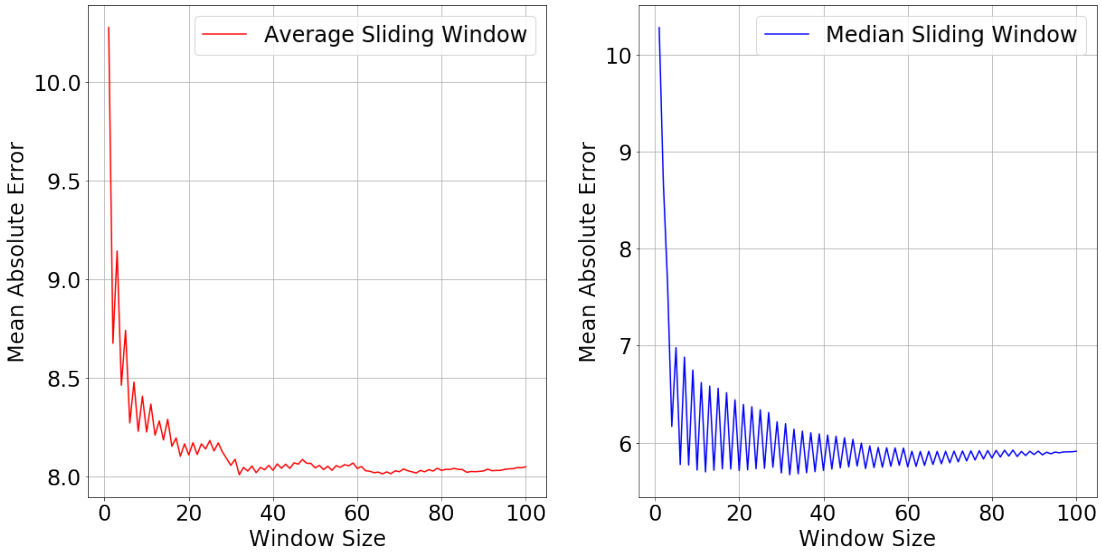

Figure 4.2: The Mean Absolute Error (MAE) of deployment time for the GPU1 runtime. The x-axis is the window (history) size. The left subplot is MAE when STOIC uses the average sliding window, the right subplot is MAE when STOIC uses the median sliding window.

the bandwidth at the moment provided by a bandwidth monitor at the edge controller.

Runtime deployment time (T_d)

T_d measures the time Nautilus uses to deploy the 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.

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 4.2: 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.

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) auto-regression modeling, (2) average sliding window, and (3) median sliding window. Auto-regression (*Auto-Regression* (2021)) 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 (*Moving Average* (2021)) scanning through the time series by a fixed-length window. Similarly, the median sliding window captures the moving median across the time series. All window 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 (4.2) 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 (4.2), 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 deployment time when jobs arrive. The inquisitor period, calibration period, and maximum window size are all modifiable.

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 (brr (2021)) due to its robustness to ill-posed problems (relative to ordinary least squares regression (*Ordinary Least Squares* (2021))). 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 (ran (2021)), which iteratively removes outliers from the regression. The algorithm 2 illustrates our RANSAC approach in STOIC.

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 a 150-image batch using the GPU1 runtime. Depicted in Figure 4.3, we observe that actual

Algorithm 2: 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

```
while iterations ≤ k do
    curr_sample := n data points from observation;
    curr_model := M regressed on curr_sample;
    fit_data := empty set;
    for every data point p in curr_sample do
        if error of p ≤ t then
            p → fit_data;
    end
    if fit_data size ≥ d then
        curr_error := average error in fit_data;
        if curr_error < e then
            Update M and e
    else
        Increment iteration
    end
end
return M
```

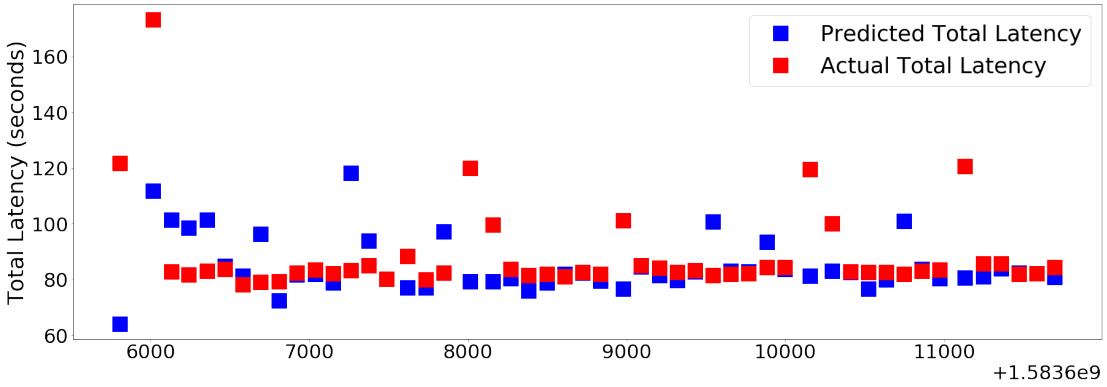

Figure 4.3: The comparison of predicted and actual total latency on 50 GPU1 benchmark executions with 150-image batch size. The x-axis is the epoch time and the y-axis is the total latency.

	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 4.3: 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.

total latency varies significantly, and predicted total latency has a non-negligible difference from the actual total latency at the beginning of the experiment. However, over time, as STOIC learns the various latencies of the system, the difference is significantly reduced. In Table 4.3, 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 shows STOIC's adaptability.

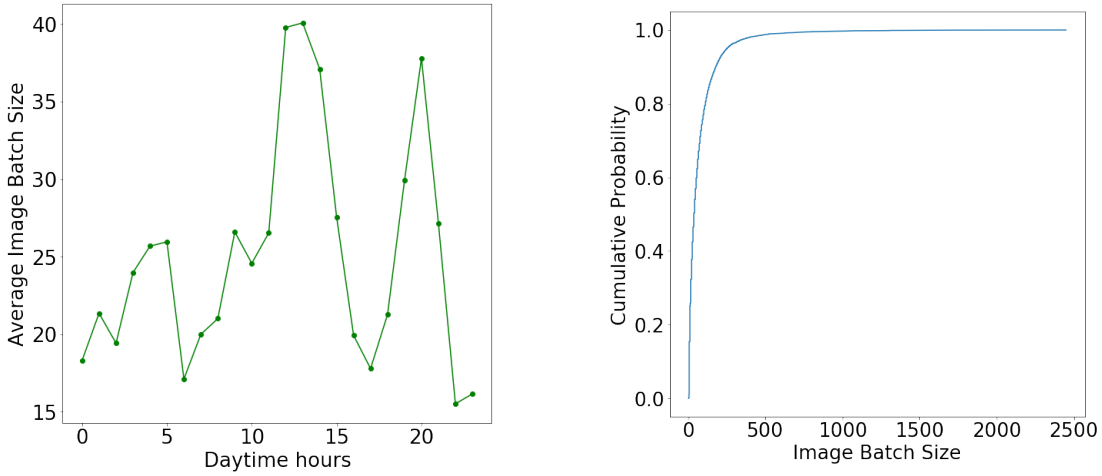

Figure 4.4: Wildlife Hourly Activity Level (left graph) and its Conditional Empirical Cumulative Distribution Function (right graph). The left graph demonstrates the mean activity level of wildlife throughout the daytime. Based on the curve, 1 PM and 8 PM are two peak hours of animal activities. The 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.

4.2.5 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 on 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 bears, deer, coyotes, puma, and birds as well as wind-triggered empty images and other animals.

After excluding camera maintenance periods (gaps), we extract 1136 effective days (27264 hours) of data. The maximum size of the 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 4.4 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 4.4 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 can evaluate and conclude by replaying the image stream from the camera traps in faster-than-real-time for comparative evaluation.

4.2.6 Implementation

We implement STOIC using Golang (gol (2021)). Golang provides high-performance execution (vs scripting languages) and a user-friendly interface (cli (2021)) to Kubernetes and database technologies. STOIC currently supports machine learning applications developed using the TensorFlow framework (Abadi et al. (2016)) and can be easily extended to permit other machine learning libraries.

As mentioned previously, the STOIC serverless architecture leverages kubeless (kub (2021)). As a Kubernetes-native serverless framework, kubeless uses the Custom Resource Definition (CRD) (crd (2021)) 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 (doc (2021)) 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 (*AVX2* (2021)), and FMA (fma (2021)) instruction set support. We use this optimized version of Tensorflow on both the edge and public clouds.

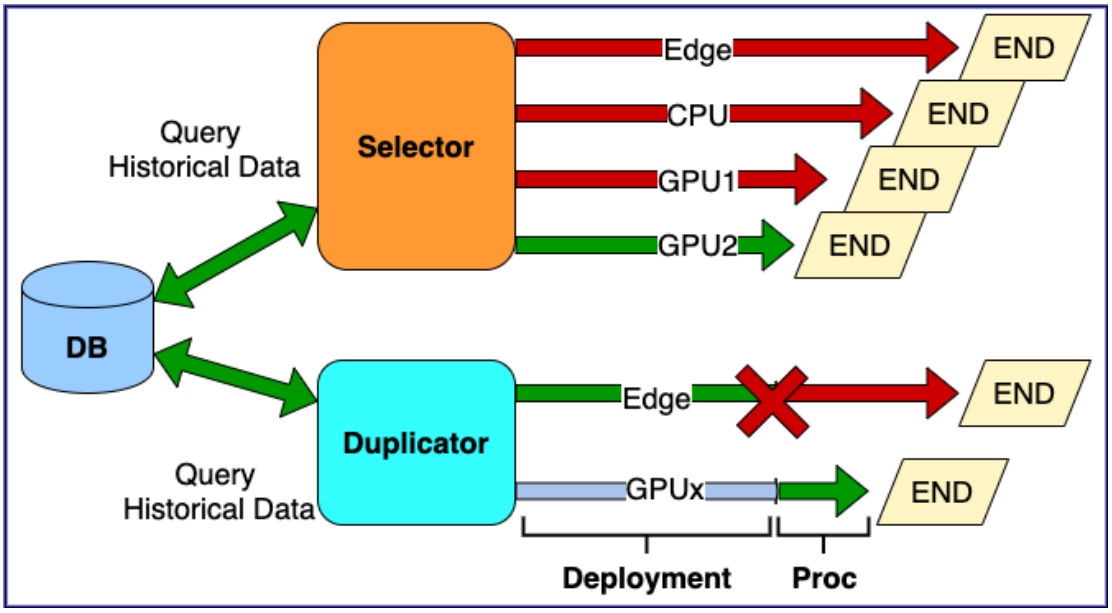

Figure 4.5: The selector and duplicator modes of STOIC.

To enable GPU access by serverless functions (available in the public cloud), we equip our Docker container with NVIDIA Container Toolkit (nvi (2021)). 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.

4.2.7 Workflow

STOIC considers two workflows upon receiving an image batch: selector mode and duplicator mode. Both are depicted in Figure 4.5. 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 the 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 a single image to retrieve and cache the base model in memory at each pod.

We observe from Table 4.3 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 termi-

nates 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.

4.3 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.

4.3.1 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 (Elias et al. (2017))). "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 (sed (2021)) 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 (nau (2021)) as the public cloud, in conjunction with the edge cloud to optimize image classification on a convolutional neural network (CNN) (LeCun et al. (1995)) implemented by Tensorflow and Scikit-learn (Pedregosa et al. (2011)).

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 the 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 (Ker (2021*a*)). 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

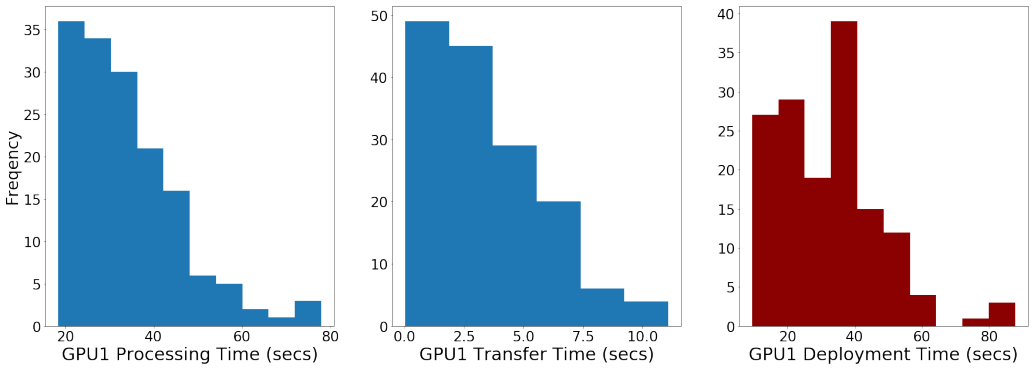

Figure 4.6: The distribution of three components in total response time (T_s) of 150 executions on GPU1 runtime: Processing time (T_p), Deployment time (T_d), and Transfer time (T_t). The x-axis represents the time range, while the y-axis is the frequency of executions. The deployment time, which is depicted in the red histogram, is volatile and error-prone to prediction.

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 loading and memory transfer overhead (Campos et al. (2017)).

To drive this experiment, we use the workload generator described in Section 4.2.5 to facilitate a 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 4.6 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 respec-

tively. Note that processing time and transfer time are relatively stable compared to deployment time.

4.3.2 Selector Evaluation

We first evaluate STOIC selector mode for 24 hours 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 on 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 (Zhang et al. (2020)). Second, the deployment times for GPU runtime are volatile and error-prone to prediction. As a representative instance, Figure 4.6 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.

4.3.3 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 its processing before the edge completes what remains of the job, it allows the Nautilus and the 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).

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

Alternatively, when STOIC predicts that the edge cloud will finish first, it returns the GPU resources to Nautilus and runs only the edge cloud job. If the Nautilus job would have been 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 4.4 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 predic-

tion 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 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 can make this risk-reward trade-off explicit. Note also that the worst case is unchanged. This result indicates that

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

Table 4.5: 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.

there is 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 can achieve. Table 4.5 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 execution 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 4.4 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 4.4) while utilizing the expensive Nautilus resource efficiently (Table 4.5) by using the edge cloud node speculatively.

4.4 Conclusion

In this work, 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 (run-times). We also plan to investigate model check-pointing in duplicator mode to better utilize computational resources on edge cloud and to improve the overall performance of the STOIC system.

Chapter 5

Heat-Budget-based Scheduling on IoT Edge Systems

> *There is no use for bravery unless justice is present, and no need for bravery if all men are just.*
>
> —Agesilaus

Edge processing introduces new challenges for IoT deployments. Unlike the devices themselves, edge computing elements are often designed for office or home use – environments in which the ambient environmental conditions are controlled and kept within a narrow operational range. However for many IoT applications, the operational settings in which these edge systems (in our work we deploy miniaturized "edge clouds" using clusters of commodity small-board computers to support IoT analytics) must function can be harsh, hard, or costly to access, and exposed to harmful environmental elements (heat, moisture, dust, animals, other objects, humans, weather, etc.). For example, we currently support an IoT de-

Parts of this section are adapted, with permission, from EDGE 2021 Zhang et al. (2021*b*)

ployment for image processing and deep learning for the automatic, real-time identification of animals using camera traps deployed across UCSB Sedgwick Reserve, an ecology and wildlife education and research reserve in California sed (2021). The reserve is 6,000 acres that comprise critical wildlife habitats, two watersheds at the foot of Figueroa Mountain in Santa Ynez, California, and a 300-acre farm easement. Our edge clouds fuse and analyze images from within out-buildings on the property. Sedgwick yearly outdoor temperatures range between 30° and 116° Fahrenheit; within enclosures (e.g. shelters for electrical pumping equipment where grid electricity is available) our cloud systems are subjected to much higher ambient operating temperatures.

Excessive heat can degrade the performance and reliability of devices and negatively impact their longevity (requiring more human intervention and frequent replacement). Commodity computers are particularly sensitive to high temperatures and extended exposure can cause these machines to break down, degrade in functionality, and fail prematurely – even they are protected using operational safeguards such as throttling and automatic shutdown int (2021). For this reason, most manufacturers include an onboard thermal CPU temperature sensor and the ability to set a "shut-down" temperature if the CPU exceeds the manufacturer's maximum supported temperature. Figure5.1 shows a time series of CPU temperature in degrees Fahrenheit gathered from one of our edge clouds deployed at

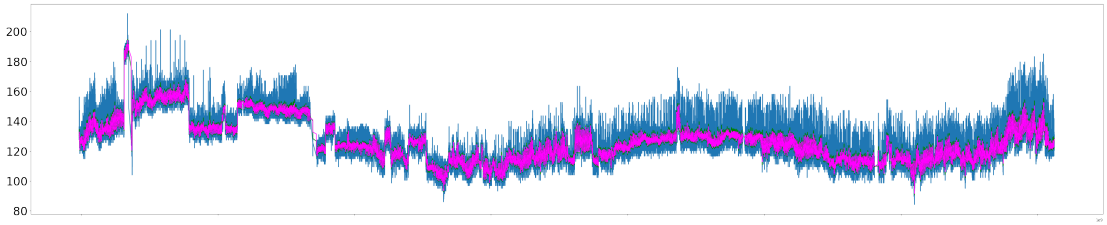

Figure 5.1: The time series of CPU temperature in the edge cloud deployed at Sedgwick Natural Reserve from Feb. 28th, 2018 to Jun. 3rd, 2020. The x-axis is the epoch time and the y-axis is the CPU temperature in Fahrenheit.

Sedgwick between February 2018 and June 2020. The cut-off temperature was set to 200° F and the temperature drop early in the trace records the system's automatic shutdown.

In this work, we investigate the use of dynamic voltage and frequency scaling (DVFS) Liu et al. (2007), Wang et al. (2010), Wu et al. (2014) to control system temperature when the ambient temperature might cause it to exceed the acceptable operational range. DVFS is a technique that has been widely studied in the context of "power capping" – the implementation of a maximum power draw by the system. Our system – called *Sparta* – automatically exploits the relationship between system power consumption and generated heat. It does so by adjusting processor frequency dynamically so that CPU temperatures do not exceed a specified threshold as ambient temperature changes. Subject to the threshold, the system attempts to minimize the application "slow down" (relative to maximum CPU frequency) that frequency adjustments might introduce. We use Sparta to

study the relationships between CPU frequency, temperature, power dissipation, and execution behavior. Moreover, we consider IoT workloads that employ a wide range of machine learning algorithms, including image recognition, natural language processing, decision forest, and time series prediction.

We consider three modes for the Sparta frequency scheduler: **Annealing**, **AIMD**, and **Hybrid**. Annealing employs an epsilon-greedy strategy to extrapolate an appropriate CPU frequency in real time. AIMD uses the linear growth of CPU frequency when the temperature is under the threshold and exponential reduction when it detects temperature anomalies to determine its CPU frequency. With Hybrid, we combine the best features of the two modes to overcome their drawbacks. Our results show that Sparta in Hybrid mode speeds up the execution of our applications by **1.16x** and **1.14x** on average in three thermal environments compared to Annealing and AIMD. Moreover, Sparta in Hybrid mode maintains CPU temperature below threshold **94.4%** of the time (as measured via temperature sampling), on average across all benchmarks.

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

- We investigate the relationship between CPU frequency and sampling temperature to precisely model and manage processor power dissipation during execution;

- We design and implement a heat-budget-based scheduling framework that protects edge systems from overheating and potential damage;

- We empirically evaluate the efficacy of using Sparta to control CPU temperature and accelerate machine learning applications on six real-world benchmarks in three thermal deployment environments.

In the following sections, we first present the design and implementation of Sparta. We then describe our experimental methodology and empirical evaluation of the system using multiple machine learning applications in different thermal environments. In the next section, we discuss related work. Finally, we present our conclusions and future work plans.

5.1 Sparta

5.1.1 Architecture

To address the processor overheating challenge and accelerate the execution of applications under a CPU temperature threshold, we develop Sparta, a heat-budget-based scheduling framework for edge devices and machine learning applications. The architecture of Sparta is shown in Figure 5.2. The scheduler consists of three components: a control plane, a data plane, and a decision plane. Sparta

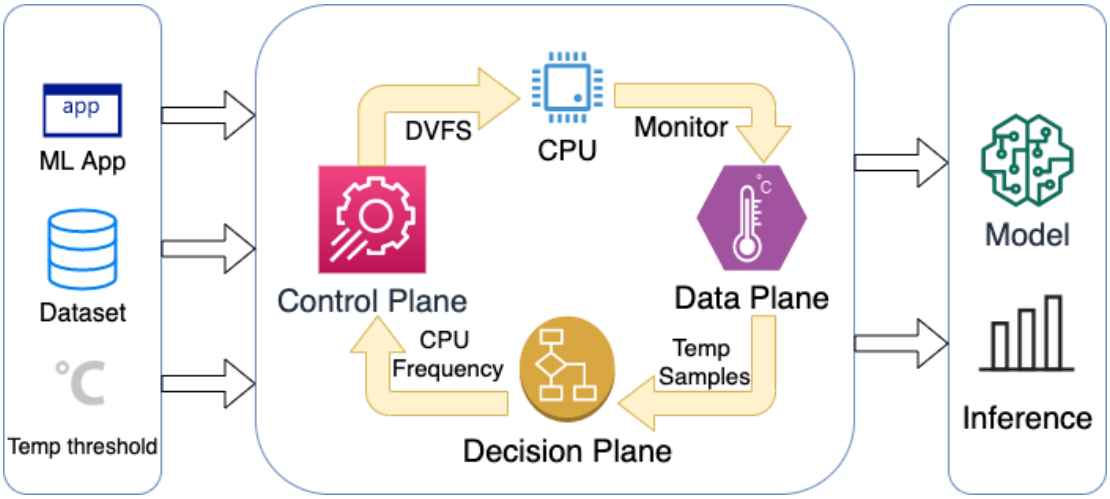

Figure 5.2: The Architecture of Sparta

takes a machine learning application, datasets, and a CPU temperature threshold as input. During the execution, the scheduler utilizes a feedback control mechanism that controls the CPU temperature by dynamically adjusting CPU frequency via system-level dynamic voltage and frequency scaling (DVFS). Sparta returns the trained model and inference results at the end of the execution.

The data plane monitors, samples, and records the CPU real-time temperature via the lm-sensors interface lm- (2021) and selects the maximum temperature within a sliding time window. Both the sampling rate and window size are configurable. (1/second and 5 seconds by default) To signify the authentic temperature of multi-core processors, the data plane records the temperature samples of the entire CPU package instead of any specific ones. Being accessible by the decision

plane, all structured temperature data helps determine the proper CPU frequency in real time to keep the CPU temperature under a threshold.

The control plane manages the CPU power and temperature. In the design phase, we consider two methods: Sleep injection and DVFS. The first method injects sleep time in the iteration loop that lowers the CPU usage, whereas the second method adjusts the CPU frequency by tuning the CPU voltage. We experiment with these two methods on a multi-threaded matrix multiplication benchmark and monitor the CPU temperature. Figure 5.3 shows the CPU temperature time series using these two methods. We observe the latter method generates a controllable and stable temperature curve, and thus choose DVFS as the control plane interface. Upon the execution of the scheduler, the control plane receives the determined CPU frequency and sets the max clock speed of all cores in the CPU package on the fly. This way the control plane effectively manages the power consumption and heat generation of the processor.

The decision plane determines the CPU frequency based on historical and real-time temperature data throughout the execution. To provide the historical dataset, on which the decision plane decides the initial CPU frequency, we collect CPU temperature and frequency data from a multi-threaded matrix multiplication (MATMUL) benchmark that simulates the underlying operations in machine learning applications.

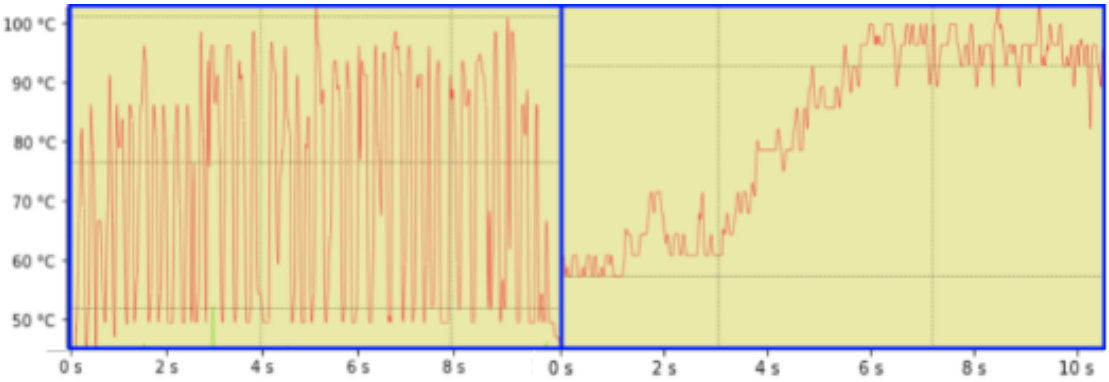

Figure 5.3: The CPU temperature time series by sleep injection (left) vs DVFS (right). The x-axis is the time frame and the y-axis is the CPU temperature ranging from 48° C to 100° C.

We gather the data in the ambient temperature ranging from 2.6° C to 43.8° C to cover different thermal environments. In the experiment, we found the sequence of CPU frequency and maximum temperature in a time window demonstrate a better linear relationship than the sequence of all temperatures, because of its inherent oscillating feature. To verify the correlation between MATMUL and machine learning applications, we collect the same data from an image recognition application written in Tensorflow Abadi et al. (2016). As depicted in Figure 5.4, we found the correlated linear relationship between the CPU frequency and logarithmic delta temperature defined as $log(T_{max} - T_i)$, where T_{max} is the maximum temperature sample in the time window and T_i is the starting CPU temperature in idle state.

Depending on this correlation, the decision plane extrapolates the appropriate CPU frequency by linear regression from the MATMUL dataset and assigns the initial CPU frequency before the execution starts. During the process, the decision plane starts to extrapolate CPU frequency from real-time data that accurately reflects the ambient temperature and the execution pattern of ML applications. The extrapolation frequency is 12/minute by default and configurable by users.

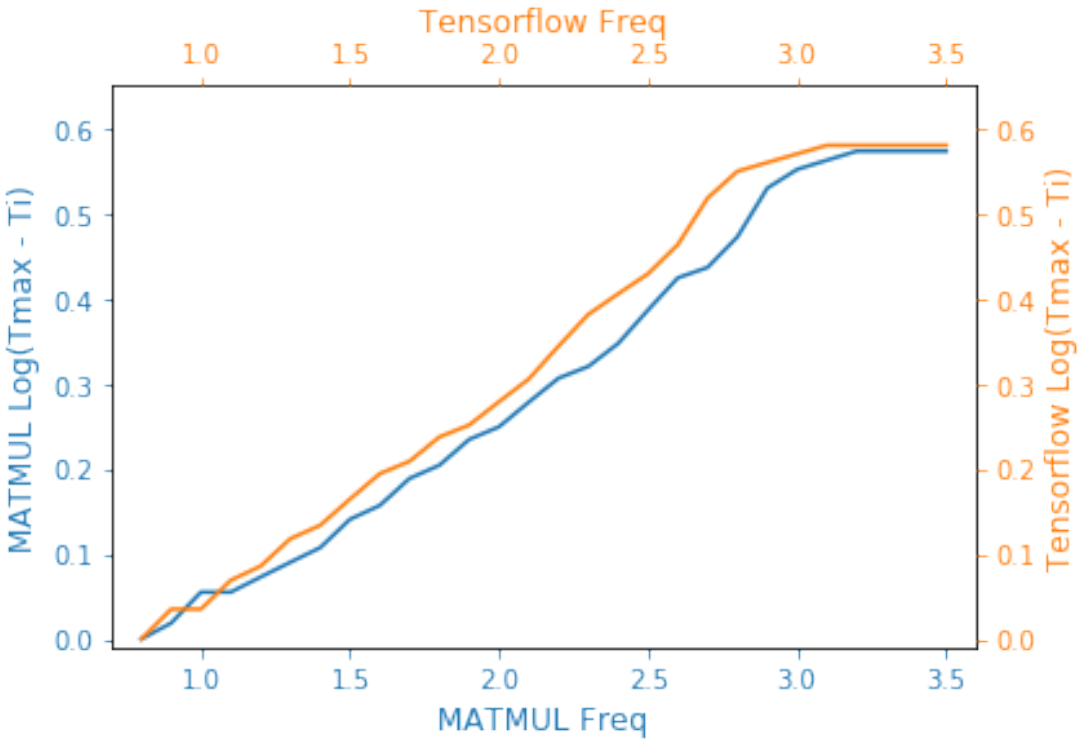

Figure 5.4: The linear relationship between CPU frequency and logarithmic delta temperature of two benchmarks. The blue curve represents MATMUL and the orange curve represents the image recognition application. The plateaus at the right side of curves are caused by CPU hardware temperature throttling.

5.1.2 Operating Modes

In the testing phase of Sparta, we identified two major problems in the decision plane. First, the extrapolation from linear regression oftentimes gets stuck at a local minimum. Thus, the determined CPU frequency is frequently lower than the ideal one, which leaves computational resources idle during execution. Second, the response time to correct the CPU from overheating is longer than expected when CPU temperature surpasses the threshold. To solve these two problems, we construct three operating modes for Sparta: Annealing, AIMD, and Hybrid.

Annealing is a probabilistic algorithm that leverages the epsilon-greedy strategy that balances exploration and exploitation by choosing randomly. In this mode, Sparta scheduler picks a value(P) in the range [0, 1] uniformly at random and compares it with ϵ/K, where ϵ is a probability of taking random actions (0.5 by default) and K is the number of extrapolation decision plane has made. The scheduler assigns a random CPU frequency when P is greater, whereas it keeps the extrapolated frequency when P is less than ϵ/K. With a decreasing probability of ϵ/K as the application proceeds, the scheduler stabilizes and chooses to exploit what it has learned so far. When the ambient temperature or the execution pattern shifts dramatically, the scheduler resets the ϵ/K that allows more random exploration. This mode effectively addresses the problem of CPU frequency stuck

at a local minimum and expedites the execution of machine learning applications under the temperature threshold.

AIMD is a feedback control mechanism that responds to CPU temperature anomaly faster. The scheduler configures the CPU frequency according to the historical data extrapolation at the start of execution. During the execution, it decreases the CPU frequency by a multiplicative factor (0.5 by default) when CPU temperature surpasses the threshold. Subsequently, it increases the frequency by a fixed amount (0.07 GHz by default) every iteration until the CPU temperature stabilizes right below the threshold. The decision plane turns into hibernation at this point to prevent redundant tuning on CPU frequency that leads to inefficient execution. Meanwhile, the data plane keeps monitoring the CPU temperature and wakes up the decision plane if any anomalies caused by ambient temperature or execution patterns are detected. AIMD reduces the response time to temperature deviation and keeps most samples under the threshold.

Hybrid combines Annealing and AIMD modes to address each other's disadvantage: if the probabilistic actions in Annealing drive CPU temperature above threshold, AIMD brings the anomaly back to normal fast; when AIMD settles at a local minimum of CPU frequency and leaves resources idle, Annealing boosts the execution by assigning a random CPU frequency. This way, Hybrid mode pro-

vides a complement to accelerate the machine learning execution while keeping the CPU temperature under the threshold.

5.2 Evaluation

In this section, we empirically evaluate Sparta's performance in a series of experiments on six benchmarks, ranging from image recognition, natural language processing to random forest and time series prediction, which are implemented based on Tensorflow and executed through Sparta's actuator interface. We first overview the machine learning benchmarks that we consider and our experimental methodology. We then present our results.

5.2.1 Machine Learning Benchmarks

To comprehensively evaluate the efficacy and efficiency of Sparta, we implemented 6 machine learning benchmarks, which consist of four categories: image recognition, natural language processing, ensemble learning, and time series analysis. We aim to test Sparta on a variety of machine learning applications that represent different execution patterns.

WTB_Train is an image recognition application that we use as a benchmark to train a convolutional neural network (CNN) LeCun et al. (1995) based on

ResNet50 He et al. (2016). The training dataset contains animal images from a wildlife monitoring system called "Where's The Bear" (WTB) Elias et al. (2017). "Where's The Bear" is an end-to-end distributed data acquisition and analytical system that automatically analyzes camera trap images collected by cameras sited at the Sedgwick Natural Reserve sed (2021) in Santa Barbara County, California. In total, there are five classes that we consider: Bear, Coyote, Deer, Bird, and Empty, by which we label images for training tasks. We also up-sampled minority classes using the Keras Image Data Generator ker (2021*b*) since the class size is unbalanced due to the frequency of animal occurrences. Doing so ensures that the classification model is not biased. We resized every image in the dataset to 1920×1080, and for each class, the dataset contains 60 images used to train the CNN model. Once the training is complete, the application stores this model in hdf5 format in object storage.

The WTB_Train application has a cold start at the beginning of the execution since it loads a pre-trained neural network model and a large dataset. Once it completes loading, the entire training process has relatively consistent CPU usage and temperature.

WTB_Inf inferences the type of wildlife in camera trap pictures based on the model trained by WTB_Train. It loads the trained hdf5 model at the beginning and, for each picture, it assigns probabilities to five classes we consider

in the training dataset by Softmax function. In each experiment, we assign 20 pictures for WTB_Inf to inference. In terms of the execution pattern, WTB_Inf runs in short bursts as opposed to WTB_Train. Therefore, the CPU usage and temperature fluctuate dramatically throughout the execution of this benchmark.

MNIST is a dataset containing grayscale pictures of handwritten digits, in which it has 60,000 examples as the training set and 10,000 examples as the testing set. Based on the dataset, we train a 2-layer convolutional neural network mni (2021) and test its accuracy in the third application. In contrast to WTB benchmarks, the size of pictures is smaller (28×28) and the model is simplified in MNIST.

BiLSTM is a sentiment analysis application based on a dataset of the Internet Movie Database (IMDB) movie reviews. It consists of 25,000 sequences each for training and testing. The model is constructed as a bidirectional LSTM with a classification layer using the sigmoid activation function. We train the model by the training dataset and validate its performance in classifying sentiment by the testing dataset. Since it has a large dataset and a complex model, the execution pattern is long-running and consistent in CPU usage and temperature.

Decision_Forest is an implementation of deep neural decision forests Kontschieder et al. (2015) that classifies high-earning individuals from the pool. The benchmark leverages the United States Census Income Dataset cen (2021) that has

48,843 instances with 14 features, including age, education, occupation, etc. The dataset is split up that the training part has 32,561 instances and the testing part has 16,282 instances. The application has three phases: it firstly processes the dataset by encoding input features. Then, it trains a deep neural decision tree model. Based on that, the application trains a neural decision forest model consists of a set of neural decision trees. Therefore, the usage and temperature of the CPU increasingly grow throughout the process.

Time_Series is a time series prediction application built on the climate data recorded by the Max Planck Institute for Biogeochemistry jen (2021). The dataset has 14 features such as temperature, pressure, humidity, etc. and the sampling frequency is 10 minutes. The time frame of the dataset ranges from Jan. 10th, 2009 to Dec. 31st, 2016. The application uses 300,693 rows to train a single-layer LSTM model, by which we can predict the outdoor temperature in the next 72 timestamps (12 hours) given the samples in the past 720 timestamps (120 hours). By this benchmark, we intend to evaluate Sparta on an application with a lightweight model and a large dataset.

5.2.2 Experimental Setup

Each edge cloud node used in the experiments is an Intel NUC nuc (2021) (6i7KYK) with two Intel Core i7-6770HQ 4-core processors (6M Cache, 2.60 GHz)

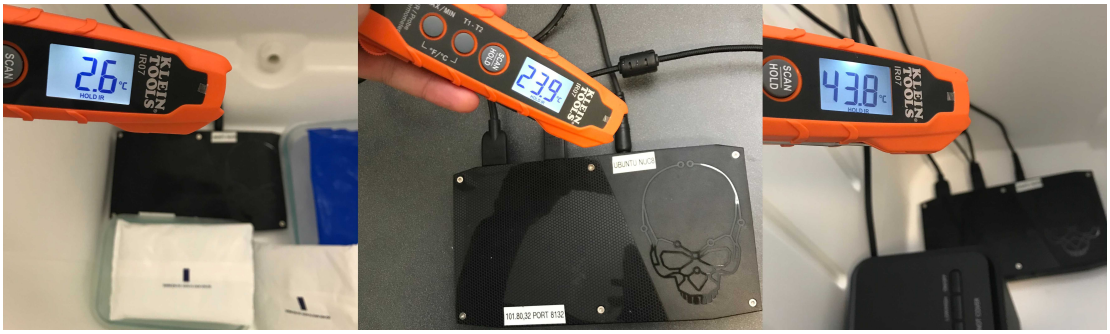

Figure 5.5: Three thermal environments in the experiment

and 32GB of DDR4-2133+ RAM connected via two channels. We use dynamic voltage and frequency scaling (DVFS) to control the frequency of the CPU from 0.8GHz to 3.5GHz.

To simulate the natural temperature in Sedgwick natural reserve, we create three thermal environments in an isolated cooler that represent cold, neutral, and hot ambient temperature. In the cold scenario, the ambient temperature is 2.6° C and the CPU of NUC runs under 40° C in idle status. In the neutral scenario, the CPU of NUC starts at 51° C under the ambient temperature of 23.9° C. The hot scenario increases the ambient and CPU temperature to 43.8° C and 68° C respectively.

There are two goals of the Sparta scheduler: the first is to limit the CPU temperature under the threshold; the second is to accelerate tasks without overheating the edge cloud. To evaluate these two objectives, we execute 6 machine learning benchmarks under 3 modes of Sparta scheduler. In each experiment, Sparta takes

inputs of the task program, corresponding workload dataset, and a threshold temperature. To keep the comparison consistent across 3 thermal environments, we use 75° C as the threshold temperature for all experiments.

In 3 modes of Sparta, we execute each machine learning benchmark repeatedly 100 times under 3 thermal environments (totally $3 \times 3 \times 6 \times 100 = 5400$ executions) and report relevant metrics, both mean and standard deviation, to compare the efficacy and efficiency among Annealing, AIMD, and Hybrid modes.

5.2.3 Application Efficacy

We first measure the stabilization time for six benchmarks. We define stabilization time as the elapsed time from the start to the point all CPU temperatures in the sampling time window are within $[T_s - T_d, T_s]$, where T_s is the threshold and T_d is a slack variable (3 ° C by default). During each of the 10 consecutive executions (1 epoch) of benchmarks, we record the duration when the Sparta scheduler stabilizes the CPU temperature according to the threshold. As shown in the first part of Table 5.1, we report the mean and stdev of stabilization time for each benchmark in 3 modes. Hybrid mode uses less time to stabilize CPU temperature than Annealing and AIMD in all six benchmarks. It performs even better in the WTB_Inf benchmark that has a short burst execution pattern and volatile CPU temperature.

	WTB Train	WTB Inf	MNIST	BiLSTM	Decision Forest	Time Series
Annealing	53.79 (30.1)	50.7 (21.1)	62.02 (32.2)	69.02 (33.8)	59.13 (31.9)	72.31 (28.8)
AIMD	61.73 (24.9)	63.26 (25.0)	58.91 (14.0)	59.9 (11.2)	78.17 (30.7)	73.41 (28.9)
Hybrid	38.59 (26.11)	23.24 (17.33)	38.66 (22.17)	52.83 (20.6)	55.46 (25.5)	52.91 (29.8)

	Neutral	Cold	Hot	Average
Annealing	68.15	67.27	48.07	61.16
AIMD	63.10	77.40	57.20	65.90
Hybrid	43.03	52.43	35.39	43.61

Table 5.1: The mean and stdev of **stabilization time** in seconds for 6 machine learning benchmarks in 3 Sparta modes. Compared to Annealing and AIMD, Hybrid mode uses less time to stabilize CPU temperature across all benchmarks and all thermal scenarios.

As the second part of Table 5.1 presents the result in the thermal dimension, the Hybrid mode also uses less time to stabilize CPU temperature across all 3 thermal environments, comparing to Annealing and AIMD. Averagely, Hybrid mode uses 43.61 seconds in the stabilization phase, in contrast to 61.16 seconds in Annealing and 65.9 seconds in AIMD. Hybrid mode's performance is even better in the hot scenario, which is the key use case for edge devices to prevent overheating in Sedgwick natural reserve.

We next empirically evaluate the execution time of six benchmarks by the Sparta scheduler. In the first part of Table 5.2, we report the mean and stdev of execution time for each benchmark under 3 modes. On average, the Hybrid mode

	WTB Train	WTB Inf	MNIST	BiLSTM	Decision Forest	Time Series
Annealing	374.67 (9.8)	60.94 (3.9)	39.85 (2.9)	222.34 (3.5)	48.21 (3.6)	130.65 (7.6)
Speedup	1.17x	1.10x	1.21x	1.04x	1.15x	1.32x
AIMD	393.55 (5.8)	64.32 (3.9)	36.51 (4.3)	234.92 (5.2)	45.38 (2.2)	110.06 (6.2)
Speedup	1.22x	1.16x	1.13x	1.10x	1.09x	1.11x
Hybrid	318.55 (4.2)	55.31 (3.2)	32.53 (2.3)	212.91 (2.6)	41.56 (4.4)	98.78 (7.2)

	Neutral	Cold	Hot	Average
Annealing	145.72	116.14	176.42	146.09
Speedup	1.17x	1.06x	1.26x	1.16x
AIMD	134.67	122.28	185.42	147.46
Speedup	1.07x	1.15x	1.18x	1.14x
Hybrid	124.86	107.68	147.28	126.61

Table 5.2: The mean and stdev of **execution time** in seconds for 6 machine learning benchmarks in 3 modes of Sparta. Compared to Annealing and AIMD, Hybrid mode uses less time to complete tasks across all benchmarks and all thermal scenarios.

completes the task of each benchmark faster than Annealing and AIMD. Given the stdev and degree of freedom, we also run a student t-test among 3 modes for each benchmark and confirm that the execution time by Hybrid is smaller than Annealing and AIMD with a statistical significance level of 5%. Table 5.2 also indicates the speedup of Hybrid over Annealing and AIMD, ranging from 1.04x to 1.32x.

The second part of Table 5.2 demonstrates the average execution time in 3 thermal scenarios. On average, Hybrid mode completes the task in 126.61 seconds,

	WTB Train	WTB Inf	MNIST	BiLSTM	Decision Forest	Time Series
Annealing	5.04 (1.0)	7.88 (1.7)	9.22 (0.8)	5.07 (1.3)	9.91 (1.5)	9.63 (2.4)
AIMD	4.39 (0.6)	6.24 (0.9)	8.35 (1.0)	5.81 (2.1)	9.95 (1.6)	8.67 (3.1)
Hybrid	4.32 (0.6)	5.79 (1.2)	6.11 (1.8)	4.90 (1.2)	9.48 (2.4)	7.25 (3.0)

	Neutral	Cold	Hot	Average
Annealing	7.12	9.59	6.67	7.79
AIMD	5.69	9.39	5.79	6.96
Hybrid	4.92	9.10	4.99	6.34

Table 5.3: The mean and stdev of **RMSE** of all temperature samples for 6 benchmarks in 3 modes of Sparta. Compared to Annealing and AIMD, Hybrid mode has less RSME to threshold temperature across all benchmarks and all thermal scenarios.

in comparison with 146.09 seconds by Annealing and 147.46 seconds by AIMD. Hybrid mode provides 1.16x and 1.14x speedup respectively over Annealing and AIMD. These results show that Sparta in Hybrid mode efficiently executes more workloads than Annealing and AIMD mode under the same temperature threshold.

To investigate the error from the sampling temperature and threshold, we next evaluate the Root Mean Square Error (RMSE) of all temperature samples in the executions. We define $RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(T_i - \hat{T})^2}$, where T_i is a sample of CPU temperature, $\hat{T}$ is the temperature threshold and n is the number of temperature samples. In Table 5.3, we display the mean and stdev of RMSE of all CPU temperature samples. The RMSE of the Hybrid mode is the least across all six benchmarks among the other two modes. The Hybrid mode also has the lowest

	WTB Train	WTB Inf	MNIST	BiLSTM	Decision Forest	Time Series
Annealing	71.8% (0.05)	83.0% (0.14)	83.4% (0.09)	72.6% (0.10)	84.3% (0.07)	91.0% (0.13)
AIMD	97.2% (0.07)	99.7% (0.01)	98.0% (0.08)	99.6% (0.09)	98.7% (0.04)	99.5% (0.25)
Hybrid	93.0% (0.11)	95.2% (0.14)	92.7% (0.07)	97.2% (0.23)	91.1% (0.10)	96.9% (0.17)

	Neutral	Cold	Hot	Average
Annealing	88.3%	79.7%	75.1%	81.1%
AIMD	99.7%	98.6%	98.2%	98.8%
Hybrid	98.1%	92.29%	92.7%	94.4%

Table 5.4: The mean and stdev of **PTBT** (Percentage of Temperature Below Threshold) for 6 benchmarks in 3 modes of Sparta. Due to their inherent algorithm, Annealing has the lowest PTBT value and AIMD has the highest, whereas the Hybrid mode has the PTBT value in-between across all benchmarks and all thermal scenarios.

RMSE in all three thermal scenarios. On average, Hybrid has 6.34 as RMSE for all temperature samples from the threshold.

Lastly, we report the percentage of samples below threshold temperature in six benchmarks. The first part of Table 5.4 manifests the mean and stdev of PTBT (Percentage of Temperature Below Threshold) for six benchmarks. Because Annealing mode uses a probabilistic algorithm, it results in the lowest PTBT metric among the three modes. Since AIMD mode multiplicatively decreases the CPU frequency whenever a temperature over the threshold is detected, it has the highest PTBT metrics in all six benchmarks. Combined with Annealing and AIMD, the PTBT of the Hybrid mode is between the other two modes. This

relationship holds for all three thermal scenarios, as depicted in the second part of Table 5.4. Hybrid mode maintains 94.4% of all temperature samples below the threshold. Thus, we consider the above results as strong proof of Sparta's efficacy in preventing overheating of edge devices and executing a variety of tasks as efficiently as possible.

5.3 Related Work

As related work, we consider recent advances in edge cloud's energy consumption and power management. Zahedi et al. (2017) proposes computational sprinting which is a class of mechanisms that supply additional power on processors for a short duration to improve performance. It also introduces phase change materials onto processors to absorb additional heat primarily concerning the performance. ThriftyEdge Chen et al. (2018) presents a resource-efficient edge computing paradigm that consists of an offloading mechanism based on delay-aware task graph partition and a virtual machine selection method. To augment existing resources, Hossain et al. (2021) manifests a dynamic fog computing framework that schedules computing tasks to Citizen Fog (CF) with the highest computational ability. Different from the above systems, Sparta focuses on preventing CPU

overheating caused by ambient temperature and program execution patterns on edge clouds deployed in natural conditions.

By offering distributed, reliable, and low-latency machine learning services, edge-based ML as a fast-growing area has a great appeal both for AI and the system research community. Thus, we also consider the cutting-edge development in machine learning based on the edge cloud. Park et al. (2019) explores the building blocks and principles of wireless intelligence at edge networks concerning latency reduction, reliability guarantees, scalability enhancement, and privacy constraints. Murshed et al. (2019) provides a comprehensive survey of techniques in the scope of machine learning systems at the network edge, including distributed training and inference, real-time video analytics and speech recognition, autonomous vehicles and smart cities, etc. Cruz et al. (2021) presents an approach to estimate the performance of ML application on edge cloud and to load appropriate computing resources for an edge-based application. The above work provides guiding principles and examples for Sparta and serves as one of the key motivations for our work.

5.4 Conclusion

In this work, we propose a heat budget-based scheduling framework, called Sparta, aiming to prevent edge cloud CPU overheating in executing machine learning applications. Sparta's scheduler integrates three components – data plane, decision plane, and control plane: Decision plane extrapolates the initial CPU frequency from historical benchmark data and dynamically adjusts it based on real-time data monitored by the data plane, while the control plane modified the CPU frequency via DVFS throughout the execution. Sparta strives to accelerate the execution of applications without sacrificing CPU overheating protection.

We present the design principles and implementation details of Sparta's components and operating modes that address the drawback we encounter in the testing phase. Our empirical evaluation demonstrates Sparta effectively protects CPU from overheating, putting **94.4%** temperature samples under the threshold in Hybrid mode. In the meantime, it speeds six benchmarks' execution up to **1.04x - 1.32x** in all three thermal environments compared to Annealing and AIMD.

As part of future work, we plan to investigate using non-uniform distributions in generating random values for exploration in Annealing mode that potentially improves the PTBT metrics. We also plan to extend the deployment of Sparta at

edge cloud clusters and investigate its performance in the distributed execution of the training and inference process.

Chapter 6

Conclusions, Impact, and Future Work

The great tragedy of science - the slaying of a beautiful hypothesis by an ugly fact.

—Thomas Huxley

The introduction of edge computing creates new demands on the way computing resources are allocated and managed across computing infrastructure. It accelerates wide-area data analytics in a geographically distributed system compared to a stand-alone private/public cloud. Enabling edge clouds in a heterogeneous IoT system, however, requires multiple technology pillars to underpin scalability, robustness, and energy efficiency.

In this thesis, we present three pillars supporting the implementation and management of intelligent scheduling for IoT applications on the network edge, bridging the gap in the computational offloading research in IoT systems: the

scalability achieved by event-driven programming model, the optimal deployment in multi-layer geo-distributed cloud, and energy efficiency across heterogeneous IoT devices. We are inspired by existing work in event-driven architecture, mobile computational offloading mechanism, and dynamic voltage and frequency scaling, aiming to construct scheduling systems for real-world environments.

Seneca is our effort to simplify and accelerate the hyperparameter tuning in the machine learning training process using a scalable, event-driven architecture, specifically serverless computing systems. The framework can be deployed at the edge, private or public cloud depending on demand from users. Seneca automatically configures possible combinations of hyperparameter settings and swipes concurrently to find the best scoring model for future use. It envisions a promising field for large-scale distributed machine learning.

We demonstrate a new scheduling mechanism in STOIC that integrates a monitoring and extrapolation system. STOIC deploys and executes machine learning applications in hybrid IoT-cloud settings using serverless architecture, providing specialized hardware (e.g. GPUs) accelerations. The central feedback control mechanism makes STOIC highly adaptive to varying workloads and runtime deployment time in the public cloud. Two execution modes, selector and duplicator, further optimize the selection process by suppressing the residuals between volatile deployment time and predictions.

Consisting of a data plane, decision plane, and control plane and leveraging DVFS, Sparta implements a heat-budget-based scheduler aiming to prevent edge cloud CPU overheating in executing machine learning applications. We create the hybrid temperature prediction model, combining simulated annealing and AIMD, which achieves the most acceleration without sacrificing the CPU overheating protection. Effectively addressing the energy efficiency issue, this framework runs smoothly on heterogeneous edge cloud devices in the open field settings. It becomes the third pillar that supports a scalable, event-driven, and heterogeneous edge cloud system executing IoT applications.

In future work, we expect to extend the scalability of edge cloud and IoT systems using master-replica architecture. We also plan to optimize the data structures used to represent events to not only accelerates event sourcing but also to ensure the integrity of the end-to-end system given the malicious requests are presented.

More prediction modeling techniques, including LSTM and neural network, will be investigated to make the feedback loop more resilient and adaptive, as we consider it as the key to a more optimized intelligent scheduling system. The consensus algorithm is another area that we want to explore due to the nature of geo-distribution of edge cloud and IoT systems in wide-area data analytics. Besides PAXOS-based consensus algorithms (Pease et al. (1980), Lamport et al.

(2001), Lamport (2006)), blockchain and smart contracts (Christidis & Devetsikiotis (2016), Zheng et al. (2018), Pan et al. (2018), Wüst & Gervais (2018), Suliman et al. (2019)) have developed various interesting mechanisms for reaching consensus. The positive and negative impacts, in terms of security, latency, and resource consumption, are essential to building an efficient scheduling system on edge cloud and IoT clusters.

To more effectively and accurately manage devices' temperature in the edge cloud, we will investigate other controlling mechanisms for CPU and specialized hardware. We also plan to extend the deployment of intelligent scheduling systems to large-sale cloud clusters and empirically evaluate different modes, prediction models, and execution latency.

Bibliography

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

Al-Dhuraibi, Y., Paraiso, F., Djarallah, N. & Merle, P. (2017), 'Elasticity in cloud computing: state of the art and research challenges', *IEEE Transactions on Services Computing* **11**(2), 430–447.

Anderson, T. E., Dahlin, M. D., Neefe, J. M., Patterson, D. A., Roselli, D. S. & Wang, R. Y. (1995), Serverless network file systems, *in* 'Proceedings of the fifteenth ACM symposium on Operating systems principles', pp. 109–126.

Apa (2021), 'Apache openwhisk'.
URL: *https://openwhisk.apache.org/*

Auto-Regression (2021). "https://en.wikipedia.org/wiki/Autoregressive_model" Accessed 20-July-2021.

AVX2 (2021). https://docs.oracle.com/cd/E36784_01/html/E36859/gntae.html Accessed 20-July-2021.

AWS (2021*a*), 'Aws greengrass'.
URL: *https://aws.amazon.com/greengrass/*

AWS (2021*b*), 'Aws lambda developer guide'.
URL: *https://docs.aws.amazon.com/lambda/latest/dg/lambda-dg.pdf*

AWS (n.d.*a*), 'AWS IoT Core'. "https://aws.amazon.com/iot-core/" [Online; accessed 12-Sep-2017].

AWS (n.d.*b*), 'AWS Lambda', `https://aws.amazon.com/lambda/`. [Online; accessed 12-Sep-2017].

AWS (n.d.*c*), 'AWS X-ray'. "https://aws.amazon.com/xray/" [Online; accessed 12-Sep-2017].

Azu (n.d.*a*), 'Azure Functions', `https://azure.microsoft.com/en-us/services/functions/`. [Online; accessed 12-Sep-2017].

Azu (n.d.*b*), 'Azure Internet of Things', `https://www.microsoft.com/en-us/cloud-platform/internet-of-things-azure-iot-suite`. [Online; accessed 22-Aug-2016].

Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A. et al. (2017), Serverless computing: Current trends and open problems, *in* 'Research Advances in Cloud Computing', Springer, pp. 1–20.

Bonomi, F., Milito, R., Zhu, J. & Addepalli, S. (2012), Fog computing and its role in the internet of things, *in* 'Proceedings of the first edition of the MCC workshop on Mobile cloud computing', pp. 13–16.

Bos (n.d.), 'Bosch Internet of Things Platform', `https://www.bosch-si.com/iot-platform/iot-platform/iot-platform.html`. [Online; accessed 22-Aug-2019].

brr (2021), 'Bayesian ridge regression'. https://scikit-learn.org/stable/auto_examples/linear_model/plot_bayesian_ridge.html Accessed 20-July-2021.

Burns, B., Grant, B., Oppenheimer, D., Brewer, E. & Wilkes, J. (2021), 'Borg, omega, and kubernetes', *System Evolution* .

Campos, V., Sastre, F., Yagües, M., Torres, J. & Giró-i Nieto, X. (2017), 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)', IEEE, pp. 677–682.

Cao, B., Zhang, L., Li, Y., Feng, D. & Cao, W. (2019), 'Intelligent offloading in multi-access edge computing: A state-of-the-art review and framework', *IEEE Communications Magazine* **57**(3), 56–62.

Cech, H. L., Großmann, M. & Krieger, U. R. (2019), A fog computing architecture to share sensor data by means of blockchain functionality, *in* '2019 IEEE International Conference on Fog Computing (ICFC)', pp. 31–40.

cen (2021), 'Us census income dataset'.
URL: *https://archive.ics.uci.edu/ml/datasets/census+income*

Chang, C.-C. & Lin, C.-J. (2011), 'Libsvm: a library for support vector machines', *ACM transactions on intelligent systems and technology (TIST)* **2**(3), 27.

Chen, X., Shi, Q., Yang, L. & Xu, J. (2018), 'Thriftyedge: Resource-efficient edge computing for intelligent iot applications', *IEEE network* **32**(1), 61–65.

Christidis, K. & Devetsikiotis, M. (2016), 'Blockchains and smart contracts for the internet of things', *Ieee Access* **4**, 2292–2303.

Chun, B., Oh, B., Cho, C. & Lee, D. (2018), Design and implementation of lightweight messaging middleware for edge computing, *in* 'Proceedings of the 6th International Conference on Control, Mechatronics and Automation', pp. 170–174.

Cisco Annual Internet Report (2018–2023) White Paper (2018). Online; accessed 12-July-2021.
URL: *https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html.*

Claesen, M., Simm, J., Popovic, D. & Moor, B. (2014), Hyperparameter tuning in python using opportunity, *in* 'Workshop on Technical Computing for Machine Learning and Mathematical Engineering'.

cli (2021), 'client-go'. https://github.com/kubernetes/client-go Accessed 20-July-2021.

Clo (2021), 'Aws cloudformation'.
URL: *https://aws.amazon.com/cloudformation/*

crd (2021), 'Crd'. https://kubernetes.io/docs/tasks/access-kubernetes-api/custom-resources/custom-resource-definitions/ Accessed 20-July-2021.

Cruz, B. D., Paul, A. K., Song, Z. & Tilevich, E. (2021), Stargazer: A deep learning approach for estimating the performance of edge-based clustering applications, *in* '2021 IEEE International Conference on Smart Data Services (SMDS)', IEEE, pp. 9–17.

Cuervo, E., Balasubramanian, A., Cho, D.-k., Wolman, A., Saroiu, S., Chandra, R. & Bahl, P. (2010), Maui: Making smartphones last longer with code offload,

in 'Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services', MobiSys '10, Association for Computing Machinery, New York, NY, USA, p. 49–62.
URL: *https://doi.org/10.1145/1814433.1814441*

Daleiden, P., Stefik, A. & Uesbeck, P. M. (2020), 'Gpu programming productivity in different abstraction paradigms: a randomized controlled trial comparing cuda and thrust', *ACM Transactions on Computing Education (TOCE)* **20**(4), 1–27.

Dean, J. & Ghemawat, S. (2008), 'Mapreduce: Simplified data processing on large clusters', *Commun. ACM* **51**(1), 107–113.
URL: *https://doi.org/10.1145/1327452.1327492*

Dekate, C., Anderson, M., Brodowicz, M., Kaiser, H., Adelstein-Lelbach, B. & Sterling, T. (2012), 'Improving the scalability of parallel n-body applications with an event-driven constraint-based execution model', *The International Journal of High Performance Computing Applications* **26**(3), 319–332.

doc (2021), 'Docker hub'. https://hub.docker.com/ Accessed 20-July-2021.

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R. & Safina, L. (2017), 'Microservices: yesterday, today, and tomorrow', *Present and ulterior software engineering* pp. 195–216.

Elias, A. R., Golubovic, N., Krintz, C. & Wolski, R. (2017), 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, pp. 247–258.

euc (2021), 'Eucalyptus'. https://www.eucalyptus.cloud/ Accessed 20-July-2021.

Fahim, A., Mtibaa, A. & Harras, K. A. (2013), Making the case for computational offloading in mobile device clouds, *in* 'Proceedings of the 19th annual international conference on Mobile computing & networking', pp. 203–205.

Feng, Y., Yang, C., Wang, T., Zheng, H., Gao, Y. & Fan, W. (2020), Quality control system of automobile bearing production based on edge cloud collaboration, *in* '2020 International Conference on Advanced Mechatronic Systems (ICAMechS)', IEEE, pp. 319–322.

Filipponi, L., Vitaletti, A., Landi, G., Memeo, V., Laura, G. & Pucci, P. (2010), Smart city: An event driven architecture for monitoring public spaces with heterogeneous sensors, *in* '2010 Fourth International Conference on Sensor Technologies and Applications', IEEE, pp. 281–286.

Fis (2021), 'Fission'.
URL: *https://fission.io/*

Flores, H., Hui, P., Nurmi, P., Lagerspetz, E., Tarkoma, S., Manner, J., Kostakos, V., Li, Y. & Su, X. (2018), 'Evidence-aware mobile computational offloading', *IEEE Transactions on Mobile Computing* **17**(8), 1834–1850.

fma (2021), 'Fma'. https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-fma-qfma Accessed 20-July-2021.

Fn (2021), 'Fn project'.
URL: *fnproject.io*

Fouladi, S., Wahby, R. S., Shacklett, B., Balasubramaniam, K. V., Zeng, W., Bhalerao, R., Sivaraman, A., Porter, G. & Winstein, K. (2017), Encoding, fast and slow: Low-latency video processing using thousands of tiny threads, *in* '14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)', USENIX Association, Boston, MA, pp. 363–376.
URL: *https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi*

Galante, G. & de Bona, L. C. E. (2012), A survey on cloud computing elasticity, *in* '2012 IEEE Fifth International Conference on Utility and Cloud Computing', IEEE, pp. 263–270.

GCP (2021), 'Google cloud platform'.
URL: *https://www.cloud.google.com/*

General Electric (n.d.), 'GE Predix', `https://www.predix.io/`. [Online; accessed 22-Aug-2019].

George-Williams, H. & Patelli, E. (2016), 'A hybrid load flow and event driven simulation approach to multi-state system reliability evaluation', *Reliability Engineering & System Safety* **152**, 351–367.

Gerstlauer, A. & Gajski, D. D. (2002), System-level abstraction semantics, *in* 'Proceedings of the 15th international symposium on System Synthesis', pp. 231–236.

Glorot, X. & Bengio, Y. (2010), Understanding the difficulty of training deep feedforward neural networks, *in* 'International conference on artificial intelligence and statistics'.

gol (2021), 'Golang'. https://golang.org/ Accessed 20-July-2021.

Golovin, D., Solnik, B., Moitra, S., Kochanski, G., Karro, J. & Sculley, D. (2017), Google vizier: A service for black-box optimization, *in* 'ACM SIGKDD'.

Goo (n.d.), 'Google Cloud Functions', `https://cloud.google.com/functions/docs/`. [Online; accessed 12-Sep-2017].

Hassan, N., Gillani, S., Ahmed, E., Yaqoob, I. & Imran, M. (2018), 'The role of edge computing in internet of things', *IEEE Communications Magazine* **56**(11), 110–115.

Haykin, S. (1994), *Neural networks: a comprehensive foundation*, Prentice Hall PTR.

He, K., Zhang, X., Ren, S. & Sun, J. (2016), Deep residual learning for image recognition, *in* 'Proceedings of the IEEE conference on computer vision and pattern recognition', pp. 770–778.

Hellerstein, J. M., Faleiro, J., Gonzalez, J. E., Schleier-Smith, J., Sreekanti, V., Tumanov, A. & Wu, C. (2018), 'Serverless computing: One step forward, two steps back', *arXiv preprint arXiv:1812.03651* .

Herbst, N. R., Kounev, S. & Reussner, R. (2013), Elasticity in cloud computing: What it is, and what it is not, *in* '10th international conference on autonomic computing ({ICAC} 13)', pp. 23–27.

Hossain, M. R., Whaiduzzaman, M., Barros, A., Tuly, S. R., Mahi, M. J. N., Roy, S., Fidge, C. & Buyya, R. (2021), 'A scheduling-based dynamic fog computing framework for augmenting resource utilization', *Simulation Modelling Practice and Theory* **111**, 102336.

Hsu, C.-W., Chang, C.-C., Lin, C.-J. et al. (2003), 'A practical guide to support vector classification'.

Hung, C.-C., Ananthanarayanan, G., Golubchik, L., Yu, M. & Zhang, M. (2018), Wide-area analytics with multiple resources, *in* 'Proceedings of the Thirteenth EuroSys Conference', pp. 1–16.

Hyp (2021*a*), 'Hyperas'.
URL: *http://maxpumperla.com/hyperas/*

Hyp (2021*b*), 'Hyperopt'.
URL: *http://hyperopt.github.io/hyperopt/*

int (2021), 'Intel processors manual'.
URL: *https://www.intel.com/content/www/us/en/support/articles/000005597/processors.html*

Int (n.d.), 'Internet of Things Solutions - Google Cloud Platform', `https://cloud.google.com/solutions/iot/`. [Online; accessed 12-Sep-2017].

Ishakian, V., Muthusamy, V. & Slominski, A. (2018), Serving deep learning models in a serverless platform, *in* '2018 IEEE International Conference on Cloud Engineering (IC2E)', IEEE, pp. 257–262.

jen (2021), 'Max planck institute for biogeochemistry'.
URL: *https://www.bgc-jena.mpg.de/wetter/*

Jonas, E., Pu, Q., Venkataraman, S., Stoica, I. & Recht, B. (2017), Occupy the cloud: Distributed computing for the 99%, *in* 'Proceedings of the 2017 Symposium on Cloud Computing', ACM, pp. 445–451.

Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.-C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N., Gonzalez, J. E., Popa, R. A., Stoica, I. & Patterson, D. A. (2019), 'Cloud programming simplified: A berkeley view on serverless computing'.

Kaf (2021), 'Apache kafka'.
URL: *https://kafka.apache.org/*

Kansal, A., Saponas, S., Brush, A. B., McKinley, K. S., Mytkowicz, T. & Ziola, R. (2013), 'The latency, accuracy, and battery (lab) abstraction: programmer productivity and energy efficiency for continuous mobile context sensing', *ACM SIGPLAN Notices* **48**(10), 661–676.

Kaur, K., Garg, S., Kaddoum, G., Ahmed, S. H. & Atiquzzaman, M. (2019), 'Keids: Kubernetes-based energy and interference driven scheduler for industrial iot in edge-cloud ecosystem', *IEEE Internet of Things Journal* **7**(5), 4228–4237.

Ker (2021*a*), 'Keras'.
URL: *https://keras.io/*

ker (2021*b*), 'Keras image data generator'.
URL: *https://keras.io/preprocessing/image/#imagedatagenerator-class*

Kim, W., Gupta, M. S., Wei, G.-Y. & Brooks, D. (2008), System level analysis of fast, per-core dvfs using on-chip switching regulators, *in* '2008 IEEE 14th International Symposium on High Performance Computer Architecture', IEEE, pp. 123–134.

Kochovski, P., Drobintsev, P. & Stankovski, V. (2019), 'Formal quality of service assurances, ranking and verification of cloud deployment options with a probabilistic model checking method', *Information and Software Technology* .

Kochovski, P., Gec, S., Stankovski, V., Bajec, M. & Drobintsev, P. (2019), 'Trust management in a blockchain based fog computing platform with trustless smart oracles', *Future Generation Computer Systems* **101**.

Kontschieder, P., Fiterau, M., Criminisi, A. & Bulo, S. R. (2015), Deep neural decision forests, *in* 'Proceedings of the IEEE international conference on computer vision', pp. 1467–1475.

Krintz, C., Wolski, R., , Golubovic, N. & Bakir, F. (2018), Estimating outdoor temperature from cpu temperature for iot applications in agriculture, *in* 'International Conference on the Internet of Things'.

kub (2021), 'kubeless'. https://github.com/kubeless/kubeless Accessed 20-July-2021.

Kubernetes (2021). "https://kubernetes.io/docs/reference/" Accessed 20-May-2021.

Kubernetes Pods (2021). https://kubernetes.io/docs/concepts/workloads/pods/pod/ Accessed 20-July-2021.

Kumar, K., Liu, J., Lu, Y.-H. & Bhargava, B. (2013), 'A survey of computation offloading for mobile systems', *Mobile networks and Applications* **18**(1), 129–140.

Lamport, L. (2006), 'Fast paxos', *Distributed Computing* **19**(2), 79–103.

Lamport, L. et al. (2001), 'Paxos made simple', *ACM Sigact News* **32**(4), 18–25.

LeCun, Y., Bengio, Y. et al. (1995), 'Convolutional networks for images, speech, and time series', *The handbook of brain theory and neural networks* **3361**(10), 1995.

Lehrig, S., Eikerling, H. & Becker, S. (2015), Scalability, elasticity, and efficiency in cloud computing: A systematic literature review of definitions and metrics, *in* 'Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures', pp. 83–92.

Li, L., Jin, Z., Li, G., Zheng, L. & Wei, Q. (2012), Modeling and analyzing the reliability and cost of service composition in the iot: A probabilistic approach, *in* '2012 IEEE 19th International Conference on Web Services', IEEE, pp. 584–591.

Lin, W.-T., Bakir, F., Krintz, C., Wolski, R. & Mock, M. (2019), Data repair for distributed, event-based iot applications, *in* 'Proceedings of the 13th ACM International Conference on Distributed and Event-based Systems', pp. 139–150.

Lin, W.-T., Krintz, C. & Wolski, R. (2019), Tracing Function Dependencies Across Clouds, *in* 'IEEE IC2E'.

Lin, W.-T., Krintz, C., Wolski, R., Zhang, M., Cai, X., Li, T. & Xu, W. (2018), Tracking causal order in aws lambda applications, *in* 'Cloud Engineering (IC2E), 2018 IEEE International Conference on'.

Liu, S., Liu, L., Tang, J., Yu, B., Wang, Y. & Shi, W. (2019), 'Edge computing for autonomous driving: Opportunities and challenges', *Proceedings of the IEEE* **107**(8), 1697–1716.

Liu, Y., Yang, H., Dick, R. P., Wang, H. & Shang, L. (2007), Thermal vs energy optimization for dvfs-enabled processors in embedded systems, *in* '8th International Symposium on Quality Electronic Design (ISQED'07)', pp. 204–209.

lm- (2021), 'lm-sensors'.
URL: *https://github.com/lm-sensors/lm-sensors*

LRE (2021), 'Lindcove research and extension center'.
URL: *http://lrec.ucanr.edu/*

Lv, B., Hong, Y., Tan, H., Han, Z. & Wang, R. (2019), 'Cooperative job dispatching in edge computing network with unpredictable uploading delay', *arXiv preprint arXiv:1912.10732* .

Magnoni, L. (2015), Modern messaging for distributed systems, *in* 'Journal of Physics: Conference Series', IOP Publishing, p. 012038.

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

McMahan, B. & Ramage, D. (2017), 'Federated learning: Collaborative machine learning without centralized training data'.
URL: *https://ai.googleblog.com/2017/04/federated-learning-collaborative.html*

Mic (2021), 'Microsoft iot hub'.
URL: *https://azure.microsoft.com/en-us/services/iot-hub/*

Mic (n.d.), 'Microsoft windows azure'. "http://www.microsoft.com/windowsazure/".

Michelson, B. M. (2006), 'Event-driven architecture overview', *Patricia Seybold Group* **2**(12), 10–1571.

mni (2021), 'mnist'. http://yann.lecun.com/exdb/mnist/ Accessed 20-July-2021.

Mohanty, S. et al. (2018), 'Evaluation of serverless computing frameworks based on kubernetes', *master thesis* .

Mossissa, D. A. & Rajaravivarma, V. (2003), Mechanisms for building intelligent messaging, *in* 'Proceedings of the 35th Southeastern Symposium on System Theory, 2003.', IEEE, pp. 291–295.

Moving Average (2021). https://en.wikipedia.org/wiki/Moving_average.

Murshed, M., Murphy, C., Hou, D., Khan, N., Ananthanarayanan, G. & Hussain, F. (2019), 'Machine learning at the network edge: A survey', *arXiv preprint arXiv:1908.00080* .

Muslim, N. & Islam, S. (2017*a*), Face recognition in the edge cloud, *in* 'Proceedings of the International Conference on Imaging, Signal Processing and Communication', pp. 5–9.

Muslim, N. & Islam, S. (2017*b*), Face recognition in the edge cloud, *in* 'Proceedings of the International Conference on Imaging, Signal Processing and Communication', ICISPC 2017, Association for Computing Machinery, New York, NY, USA, p. 5–9.
URL: *https://doi.org/10.1145/3132300.3132310*

Naranjo, D. M., Risco, S., de Alfonso, C., Pérez, A., Blanquer, I. & Moltó, G. (2021), 'Accelerated serverless computing based on gpu virtualization', *Journal of Parallel and Distributed Computing* **139**, 32–42.

nau (2021), 'Nautilus'. http://ucsd-prp.gitlab.io/nautilus/ Accessed 20-July-2021.

Newman, S. (2015), *Building microservices: designing fine-grained systems*, " O'Reilly Media, Inc.".

nnp (2021), 'Sklearn neural network mlpclassifier'.
URL: *https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html*

nuc (2021), 'Intel nucs'. https://www.intel.com/content/www/us/en/products/boards-kits/nuc.html.

nvi (2021), 'Nvidia container toolkit'. https://github.com/NVIDIA/nvidia-docker Accessed 20-July-2021.

Ope (2021), 'Openfaas'.
URL: *https://www.openfaas.com/*

Ordinary Least Squares (2021). https://scikit-learn.org/stable/modules/linear_model.html#ordinary-least-squares Accessed 20-July-2021.

Pan, J., Wang, J., Hester, A., Alqerm, I., Liu, Y. & Zhao, Y. (2018), 'Edgechain: An edge-iot framework and prototype based on blockchain and smart contracts', *IEEE Internet of Things Journal* **6**(3), 4719–4732.

Park, J., Samarakoon, S., Bennis, M. & Debbah, M. (2019), 'Wireless network intelligence at the edge', *Proceedings of the IEEE* **107**(11), 2204–2239.

Paščinski, U., Trnkoczy, J., Stankovski, V., Cigale, M. & Gec, S. (2017), '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* **16**.

Pease, M., Shostak, R. & Lamport, L. (1980), 'Reaching agreement in the presence of faults', *Journal of the ACM (JACM)* **27**(2), 228–234.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V. et al. (2011), 'Scikit-learn: Machine learning in python', *Journal of machine learning research* **12**(Oct), 2825–2830.

Pfandzelter, T. & Bermbach, D. (2021), Iot data processing in the fog: Functions, streams, or batch processing?, *in* '2021 IEEE International Conference on Fog Computing (ICFC)', IEEE, pp. 201–206.

Pu, Q., Ananthanarayanan, G., Bodik, P., Kandula, S., Akella, A., Bahl, P. & Stoica, I. (2015), Low latency geo-distributed data analytics, *in* 'Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication', SIGCOMM '15, Association for Computing Machinery, New York, NY, USA, p. 421–434.
URL: *https://doi.org/10.1145/2785956.2787505*

ran (2021), 'Ransac'. https://en.wikipedia.org/wiki/Random_sample_consensus Accessed 20-July-2021.

Rast, A. D., Jin, X., Galluppi, F., Plana, L. A., Patterson, C. & Furber, S. (2010), Scalable event-driven native parallel processing: the spinnaker neuromimetic system, *in* 'Proceedings of the 7th ACM international conference on Computing frontiers', pp. 21–30.

roo (2021), 'Rook ceph block'. https://rook.io/docs/rook/v1.0/ceph-block.html Accessed 20-July-2021.

Rountree, B., Ahn, D. H., De Supinski, B. R., Lowenthal, D. K. & Schulz, M. (2012), Beyond dvfs: A first look at performance under a hardware-enforced power bound, *in* '2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum', IEEE, pp. 947–953.

Satyanarayanan, M. (2017), 'The emergence of edge computing', *Computer* **50**(1), 30–39.

sed (2021), 'Sedgwick natural reserve'. https://sedgwick.nrs.ucsb.edu/ Accessed 20-July-2021.

sen (2021), 'Xbgoost'.
URL: *https://xgboost.ai*

Ser (2021), 'Serverless framework'.
URL: *https://serverless.com/*

Shahrad, M., Balkind, J. & Wentzlaff, D. (2019), Architectural implications of function-as-a-service computing, *in* 'Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture', pp. 1063–1075.

Shahrivari, S. (2014), 'Beyond batch processing: towards real-time and streaming big data', *Computers* **3**(4), 117–129.

Shi, W., Cao, J., Zhang, Q., Li, Y. & Xu, L. (2016), 'Edge computing: Vision and challenges', *IEEE internet of things journal* **3**(5), 637–646.

Shi, W. & Dustdar, S. (2016), 'The promise of edge computing', *Computer* **49**(5), 78–81.

Shiraz, M., Gani, A., Shamim, A., Khan, S. & Ahmad, R. W. (2015), 'Energy efficient computational offloading framework for mobile cloud computing', *Journal of Grid Computing* **13**(1), 1–18.

Simanta, S., Lewis, G. A., Morris, E., Ha, K. & Satyanarayanan, M. (2012), A reference architecture for mobile code offload in hostile environments, *in* '2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture', pp. 282–286.

Singh, A. & Viniotis, Y. (2016), An sla-based resource allocation for iot applications in cloud environments, *in* '2016 Cloudification of the Internet of Things (CIoT)', IEEE, pp. 1–6.

Sučić, S., Dragičević, T., Capuder, T. & Delimar, M. (2011), 'Economic dispatch of virtual power plants in an event-driven service-oriented framework using standards-based communications', *Electric Power Systems Research* **81**(12), 2108–2119.

Suliman, A., Husain, Z., Abououf, M., Alblooshi, M. & Salah, K. (2019), 'Monetization of iot data using smart contracts', *IET Networks* **8**(1), 32–37.

svc (2021), 'Sklearn support vector classifier'.
URL: *https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html*

Taylor, H., Yochem, A., Phillips, L. & Martinez, F. (2009), *Event-driven architecture: how SOA enables the real-time enterprise*, Pearson Education.

Taylor, S. & Letham, B. (2017), 'Forecasting at scale', *PeerJ Preprints* **5**.

Teerapittayanon, S., McDanel, B. & Kung, H.-T. (2017), Distributed deep neural networks over the cloud, the edge and end devices, *in* '2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)', IEEE, pp. 328–339.

Ter (2021), 'Terraform by hashicorp'.
URL: *https://www.terraform.io/*

Thönes, J. (2015), 'Microservices', *IEEE software* **32**(1), 116–116.

Tuli, S., Ilager, S., Ramamohanarao, K. & Buyya, R. (2020), 'Dynamic scheduling for stochastic edge-cloud computing environments using a3c learning and residual recurrent neural networks', *IEEE Transactions on Mobile Computing* .

Vaquero, L. M. & Rodero-Merino, L. (2014), 'Finding your way in the fog: Towards a comprehensive definition of fog computing', *ACM SIGCOMM computer communication Review* **44**(5), 27–32.

Verbelen, T., Simoens, P., De Turck, F. & Dhoedt, B. (2012), 'Cloudlets: Bringing the cloud to the mobile user', *MCS'12 - Proceedings of the 3rd ACM Workshop on Mobile Cloud Computing and Services* .

Von Laszewski, G., Wang, L., Younge, A. J. & He, X. (2009), Power-aware scheduling of virtual machines in dvfs-enabled clusters, *in* '2009 IEEE International Conference on Cluster Computing and Workshops', IEEE, pp. 1–10.

Vulimiri, A., Curino, C., Godfrey, P. B., Jungblut, T., Padhye, J. & Varghese, G. (2015), Global analytics in the face of bandwidth and regulatory constraints, *in* '12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)', USENIX Association, Oakland, CA, pp. 323–336.
URL: *https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/vulimiri*

Wang, H. & Li, B. (2017), Lube: Mitigating bottlenecks in wide area data analytics, *in* '9th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 17)'.

Wang, J.-B., Wang, J.-Y., Chen, M., Zhao, X., Si, S.-B., Cui, L., Cao, L.-L. & Xu, R. (2013), 'Reliability analysis for a data flow in event-driven wireless sensor networks using a multiple sending transmission approach', *EURASIP Journal on Wireless Communications and Networking* **2013**(1), 1–11.

Wang, L., Li, M., Zhang, Y., Ristenpart, T. & Swift, M. (2018), Peeking behind the curtains of serverless platforms, *in* 'USENIX ATC'.

Wang, L., Von Laszewski, G., Dayal, J. & Wang, F. (2010), Towards energy aware scheduling for precedence constrained parallel tasks in a cluster with dvfs, *in* '2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing', IEEE, pp. 368–377.

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

Welsh, M., Culler, D. & Brewer, E. (2001), Seda: An architecture for well-conditioned, scalable internet services, *in* 'Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles', SOSP '01, Association for Computing Machinery, New York, NY, USA, p. 230–243.
URL: *https://doi.org/10.1145/502034.502057*

Wu, C.-M., Chang, R.-S. & Chan, H.-Y. (2014), 'A green energy-efficient scheduling algorithm using the dvfs technique for cloud datacenters', *Future Generation Computer Systems* **37**, 141–147.
URL: *https://www.sciencedirect.com/science/article/pii/S0167739X13001234*

Wüst, K. & Gervais, A. (2018), Do you need a blockchain?, *in* '2018 Crypto Valley Conference on Blockchain Technology (CVCBT)', IEEE, pp. 45–54.

xgb (2021), 'Xgboost scikit-learn api'.
URL: *https://xgboost.readthedocs.io/en/latest/python/python_ api.html#module-xgboost.sklearn*

Xiao, L., Wan, X., Dai, C., Du, X., Chen, X. & Guizani, M. (2018), 'Security in mobile edge caching with reinforcement learning', *IEEE Wireless Communications* **25**(3), 116–122.

Yang, L., Liu, B., Cao, J., Sahni, Y. & Wang, Z. (2019), 'Joint computation partitioning and resource allocation for latency sensitive applications in mobile edge clouds', *IEEE Transactions on Services Computing* .

Yi, S., Hao, Z., Qin, Z. & Li, Q. (2015), Fog computing: Platform and applications, *in* '2015 Third IEEE workshop on hot topics in web systems and technologies (HotWeb)', IEEE, pp. 73–78.

Yi, S., Li, C. & Li, Q. (2015), A survey of fog computing: concepts, applications and issues, *in* 'Proceedings of the 2015 workshop on mobile big data', pp. 37–42.

Zahedi, S. M., Fan, S., Faw, M., Cole, E. & Lee, B. C. (2017), 'Computational sprinting: Architecture, dynamics, and strategies', *ACM Transactions on Computer Systems (TOCS)* **34**(4), 1–26.

Zeldovich, N., Yip, A., Dabek, F., Morris, R. T., Mazieres, D. & Kaashoek, M. F. (2003), Multiprocessor support for event-driven programs., *in* 'USENIX Annual Technical Conference, General Track', pp. 239–252.

Zhang, B., Jin, X., Ratnasamy, S., Wawrzynek, J. & Lee, E. A. (2018), Awstream: Adaptive wide-area streaming analytics, *in* 'Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication', pp. 236–252.

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

Zhang, M., Krintz, C. & Wolski, R. (2020), Stoic: Serverless teleoperable hybrid cloud for machine learning applications on edge device, *in* '2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)', pp. 1–6.

Zhang, M., Krintz, C. & Wolski, R. (2021*a*), 'Edge-adaptable serverless acceleration for machine learning internet of things applications', *Software: Practice and Experience* .
URL: *https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2944*

Zhang, M., Krintz, C. & Wolski, R. (2021*b*), Sparta: A heat-budget-based scheduling framework on iot edge systems, *in* 'International Conference on Edge Computing 2021'.

Zhang, W., Chen, J., Zhang, Y. & Raychaudhuri, D. (2017), Towards efficient edge cloud augmentation for virtual reality mmogs, *in* 'Proceedings of the Second ACM/IEEE Symposium on Edge Computing', pp. 1–14.

Zheng, Z., Xie, S., Dai, H.-N., Chen, X. & Wang, H. (2018), 'Blockchain challenges and opportunities: A survey', *International Journal of Web and Grid Services* **14**(4), 352–375.