ALGORITHMS AND TECHNIQUES FOR AUTOMATED DEPLOYMENT AND EFFICIENT MANAGEMENT OF LARGE-SCALE DISTRIBUTED DATA ANALYTICS SERVICES

By

Anirban Bhattacharjee

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

February 29, 2020

Nashville, Tennessee

Approved:

Aniruddha S. Gokhale, Ph.D.

Abhishek Dubey, Ph.D.

Douglas C. Schmidt, Ph.D.

Gabor Karsai, Ph.D.

Hongyang Sun, Ph.D.

DEDICATION

To my late Grandmother, Bina Roy Chowdhury, infinitely inspirational

and

To my beloved wife, Malabika, unbelievably encouraging

and

To my parents, Ashok and Nilanjana Bhattacharjee, amazingly supportive

ACKNOWLEDGMENTS

I express my sincere gratitude to those who have contributed to this thesis and supported me during this fantastic journey. I am grateful to all of those with whom I have had the pleasure to work during these years.

First and foremost, I would like to express my sincere thanks to my advisor Dr. Aniruddha S. Gokhale, for providing me the opportunity to work in the Distributed Object Computing (DOC) group at the Vanderbilt Vanderbilt School of Engineering. His thoughtful advice, guidance, mentorship, and unwavering support over the years helped me at various stages of my research. I appreciate his valuable suggestions, comments, and leadership, which encouraged me to learn more every day and to become an independent researcher. His support has been the most worthy experience for me, and I owe him a big thanks once again for being a fantastic advisor and mentor.

I want to thank Dr. Abhishek Dubey, Dr. Douglas C. Schmidt, Dr. Gabor Karsai, and Dr. Hongyang Sun, for serving on my dissertation committee. Each of my dissertation committee members has provided extensive professional guidance and taught me a great deal about scientific research. I want to acknowledge the collaboration from Dr. Hongyang Sun in my multiple research works. He helped me remarkably in formulating the research problems in various areas. Thank you, Dr. Hongyang Sun, for your valuable time, cooperation, and generosity, which set this dissertation work possible.

I am grateful to Dr. Douglas Fisher for allowing me to join the Vanderbilt University and for mentoring me during the early stage of my Ph.D. program. I would especially like to thank Dr. Xenofon Koutsoukos for his mentorship, skeptical feedbacks, and deep insights at the DDDAS project meetings. I would also like to thank Dr. Zhifeng Yun for his mentorship and leadership during my internship days at ARM Ltd.

This work would not have been possible without the financial support of various agencies, and I'm grateful for their generous support. This thesis was supported in part by

NEC Corporation, Kanagawa, Japan, and NSF US Ignite CNS 1531079, AFOSR DDDAS FA9550-18-1-0126, and AFRL/Lockheed Martin's StreamlinedML program. I appreciate the feedback and insights from our sponsors, Mr. Thomas Damiano of Lockheed Martin and Dr. Takayuki Kuroda of NEC Corporation.

My sincere gratitude is reserved for my colleague, Ajay Dev Chhokra, for his collaboration and research contributions on multiple projects. His valuable insights and collaboration made the last chapter of the dissertation possible. His contribution towards the third chapter, Barista, is also distinguishable, where he helped me to devise the problem and algorithm. I would also like to thank the members of the DOC group, Shashank Shekhar, Yogesh Barve, Shweta Khare, Shunxing Bao, Subhav Pradhan, Prithviraj Patil, Zhuangwei Kang, and Robert Canady for their collaboration, feedback, and encouragement. I would especially like to thank Shashank Shekhar, Yogesh Barve, Shweta Khare, and Zhuangwei Kang for collaborating with me on multiple projects. I also thank Shreyas Ramakrishna for his deep insights and thoughtful ideas, which helped define the last chapter of this dissertation.

Nobody has been more valuable to me in the pursuit of this Ph.D. journey than my family members. I'm hugely thankful to my late grandmother, Bina Roy Chowdhury, without whose support the Ph.D. journey would have never begun. I want to acknowledge the patience, support, and encouragement of my parents, Ashok and Nilanjana Bhattacharjee. Most importantly, I would like to express my gratitude to my beloved wife, Malabika, for her continuous support during the ups and downs of my Ph.D. journey.

TABLE OF CONTENTS

Page

DEDICATION ii

ACKNOWLEDGMENTS iii

LIST OF TABLES xiii

LIST OF FIGURES xiv

I Introduction 1

I.1 Emerging Trends 1

I.2 Key Research Challenges and Solution Needs 5

I.2.1 Requirement 1: Automation of the ML Development Pipeline . . . 5

I.2.1.1 Challenge 1: Abstraction of ML Pipeline 6

I.2.1.2 Challenge 2: Code-generation for ML Model Training and Evaluation 6

I.2.1.3 Challenge 3: Support for ML Deployment 7

I.2.2 Requirement 2: Automation of Infrastructure and Application Provisioning 7

I.2.2.1 Challenge 4: Abstraction of Application and Infrastructure details 8

I.2.2.2 Challenge 5: Infrastructure Code-generation from Abstract Model 8

I.2.2.3 Challenge 6: Verification of Abstract Deployment Model . 8

I.2.2.4 Challenge 7: Extensibility and Re-usability 9

I.2.3 Requirement 3: Proactive Resource Management 9

I.2.3.1 Challenge 8: Workload Variation 9

I.2.3.2 Challenge 9: Optimal Resource Selection 10

I.2.3.3 Challenge 10: Proactive Resource Provisioning 10
I.2.4 Requirement 4: Interference-aware Strategy for ML Model Update . 10
I.2.4.1 Challenge 11: Heterogeneity-aware Data Management . . 10
I.2.4.2 Challenge 12: Resource Interference-awareness 11
I.3 Organization of the Dissertation 11
II Erudite: A Lifecycle Management Framework for Machine Learning based Predictive Analytics Applications 13
II.1 Introduction 13
II.1.1 Emerging Trends 13
II.1.2 Challenges and State-of-the-art Solutions 13
II.1.3 Overview of Technical Contributions 15
II.1.4 Organization of the Chapter 17
II.2 Related Work 17
II.3 Problem Formulation 20
II.3.1 Motivating Case Study and Key Challenges 21
II.3.1.1 Deployment Challenges 22
II.3.1.2 Data Movement and Management Challenges 22
II.3.1.3 Model Building and Dissemination Challenges 23
II.3.1.4 Challenges in Determining the Right Hardware Needed . . 24
II.3.1.5 Runtime Resource Monitoring Challenges 24
II.3.2 Solution Requirements 24
II.3.2.1 Requirement 1: Automated Deployment of Application components in Heterogeneous environment 25
II.3.2.2 Requirement 2: Framework for Flexible ML Service Development and Encapsulation 25
II.3.2.3 Requirement 3: Performance Monitoring and Intelligent Resource Allocation 26

II.4 Design and Implementation of Erudite 26
II.4.1 Addressing Requirement 1: CloudCAMP - Automated Deployment of Application Components in Heterogeneous Resources 27
II.4.1.1 Meta-model for Heterogeneous Resources 28
II.4.1.2 Meta-model for Data Ingestion Frameworks 29
II.4.1.3 Meta-model for Data Analytics Applications 30
II.4.1.4 Meta-model for Data Storage Services 30
II.4.2 Addressing Requirement 2: Erudite Development Kit for AI/ML Model Development 30
II.4.2.1 Main Meta-model for Erudite Framework 31
II.4.2.2 Meta-model for Machine Learning Algorithms 32
II.4.2.3 Model Evaluation and Flexible ML Service Encapsulation 33
II.4.3 Addressing Requirement 3: Framework for Performance Monitoring and Intelligent Resource Management 34
II.4.3.1 Performance Monitoring 35
II.4.3.2 Resource Management 35
II.4.4 Support for Collaboration and Versioning 37
II.5 Evaluation 38
II.5.1 Evaluating the Rapid Model Development Framework 38
II.5.2 Evaluation of Rapid Application Prototyping Framework 40
II.5.3 Performance Monitoring on Heterogeneous Hardware 41
II.5.4 Resource Management 43
II.6 Conclusion 45
II.6.1 Summary 45

III CloudCAMP: A Model-Driven Approach to Automate Cloud Services Deployment and Management . . . 46
III.1 Introduction . . . 46
III.1.1 Motivation . . . 46
III.1.2 Requirements and State-of-the-art Solutions . . . 47
III.1.2.1 Requirement 1: Reduction in specification details needed for deployment . . . 47
III.1.2.2 Requirement 2: Auto-completion of Infrastructure Provisioning . . . 48
III.1.2.3 Requirement 3: Support for Continuous Integration, Migration, and Delivery . . . 49
III.1.3 Overview of Technical Contributions . . . 51
III.1.4 Organization of the Chapter . . . 52
III.2 Related Work . . . 52
III.3 Design and Implementation of CloudCAMP . . . 55
III.3.1 System Architecture of CloudCAMP . . . 55
III.3.2 System Implementation of CloudCAMP . . . 57
III.3.3 CloudCAMP Domain-specific Modeling Language (DSML) . . . 58
III.3.3.1 Design Rationale for CloudCAMP Meta-models . . . 58
III.3.3.2 Meta-model for the Cloud Platforms . . . 59
III.3.3.3 Meta-model for Application Components . . . 61
III.3.3.4 Defining the Relationship among Components . . . 61
III.3.3.5 Extensibility of the Meta-model . . . 62
III.3.4 Design of CloudCAMP Knowledge Base . . . 63
III.3.4.1 Design of Knowledge Base Database . . . 63
III.3.4.2 Design of Knowledge Base Template . . . 64
III.3.4.3 Extensibility of the Knowledge Base . . . 64

III.3.5 Generative Capabilities of CloudCAMP DSML 65
III.3.5.1 Knowledge Base for Generation of Infrastructure-as-code Solution for Deployment 66
III.3.5.2 Determining the Order of Deployment and Execution . . . 66
III.3.5.3 Generation of Infrastructure-as-code for Migration 68
III.3.5.4 Support for Continuous Delivery 69
III.3.5.5 Constraints Checking for Correctness Business Models . . 70
III.4 Evaluation 70
III.4.1 Case Study 1: LAMP-based Service Deployment Study 70
III.4.1.1 Measurement of Manual Effort 72
III.4.2 Case Study 2: Application Component Migration for LAMP-based Web Service 74
III.5 Conclusion 74
III.5.1 Summary 74
III.5.2 Discussions 75
IV Barista: Efficient and Scalable Serverless Serving System for Deep Learning Prediction Services 76
IV.1 Introduction 76
IV.1.1 Emerging Trends 76
IV.1.2 Challenges and State-of-the-Art Solutions 77
IV.1.3 Overview of Technical Contributions 78
IV.1.4 Organization of the Chapter 79
IV.2 Background and Related Work 79
IV.2.1 Deep Learning-based Prediction Services 79
IV.2.2 Serverless Computing 80
IV.2.3 Dynamic Infrastructure Elasticity 81
IV.2.4 Workload Forecasting 82

IV.3 System Model and Problem Description 83
IV.3.1 Infrastructure Model and Assumptions 83
IV.3.2 VM Flavor Selection and Initial Deployment 84
IV.3.3 Dynamic Resource Provisioning via Workload Forecasting and Infrastructure Elasticity 85
IV.4 Design and Implementation of Barista 87
IV.4.1 Architecture of Barista 87
IV.4.2 Execution Time Distribution Estimation 90
IV.4.3 Workload Forecasting 91
IV.4.3.1 Forecaster 91
IV.4.3.2 Compensator 92
IV.4.4 Resource Estimation 93
IV.4.5 Resource Provisioner 94
IV.5 Evaluation 98
IV.5.1 Experiment Setup 98
IV.5.2 Predicting Execution Time of Predictive Analytics Services 99
IV.5.3 Workload Forecasting 99
IV.5.4 Resources Selection and Provision 101
IV.5.5 Reactive Vertical Scaling for Model Correction 104
IV.6 Conclusion 105
IV.6.1 Summary 105
IV.6.2 Discussions 105
V Deep-Edge: An Efficient Framework for Deep Learning Model Update on Heterogeneous Edge 107
V.1 Introduction 107
V.1.1 Emerging Trends 107
V.1.2 Challenges and State-of-the-Art Solutions 107

V.1.3 Overview of Technical Contributions . . . 110
V.1.4 Organization of the Chapter . . . 111
V.2 Background and Related Work . . . 111
V.2.1 Deep Learning Model Training . . . 111
V.2.1.1 Distributed Deep Learning - Data Parallelism . . . 112
V.2.1.2 Distributed Deep Learning Task Scheduling . . . 114
V.2.2 Model Update Strategy . . . 115
V.2.3 Resource Interference and Performance Modeling . . . 115
V.3 Motivation . . . 117
V.3.1 Motivation for Model Update . . . 117
V.3.2 Motivation for Distributed Training . . . 118
V.3.3 Impact of Heterogeneity on Model Update Time . . . 119
V.3.4 Impact of Resource Interference on Background Tasks . . . 120
V.4 Problem Formulation . . . 121
V.4.1 Cost Models . . . 121
V.4.1.1 Data Transfer Cost . . . 122
V.4.1.2 Initialization Cost . . . 122
V.4.1.3 Training Cost . . . 122
V.4.1.4 Total Cost . . . 123
V.4.2 Optimization Problem . . . 124
V.4.3 Assumptions . . . 125
V.5 Design and Implementation of Deep-Edge . . . 125
V.5.1 Architecture Model of Deep-Edge . . . 125
V.5.2 Components of Deep-Edge Manager . . . 126
V.5.3 Modes of Operation . . . 128
V.5.3.1 Performance and Interference Modeling . . . 128
V.5.3.2 Resource Scheduling . . . 131

V.5.3.3 Fault Tolerance 132
V.6 Evaluation 135
V.6.1 Experiment Setup 135
V.6.1.1 TestBed 135
V.6.1.2 Workloads 135
V.6.2 Performance Modeling 136
V.6.3 Resource Scheduling 138
V.6.3.1 Effectiveness of Data Sharding Strategy on Epoch Time . 139
V.6.4 Model Convergence 143
V.7 Conclusion 143
V.7.1 Summary 143
V.7.2 Discussions 144
VI Summary of Research Contributions 145
VI.1 Stratum Summary 145
VI.2 CloudCAMP Summary 146
VI.3 Barista Summary 147
VI.4 Deep-Edge Summary 148
VI.5 List of Publications 149
BIBLIOGRAPHY 154

LIST OF TABLES

Table Page

II.1 Comparing Stratum with other state-of-the-art solutions. 18

III.1 Survey Questionnaire: For Q1–Q3, rate on a scale of (1-10) 72

III.2 Median and mean±std.dev for deployment time, lines of code written for deployment, migration time and Lines of code written for migration (for Q5–Q6). 73

V.1 Estimator results 139

LIST OF FIGURES

Figure Page

I.1 Generalized Representation of Applications Architecture 4

I.2 A Taxonomy of requirements and challenges for Data Analytics Service development, deployment, and management across the cloud-fog-edge spectrum 6

II.1 Generalized Architecture of Smart Application Framework 21

II.2 Meta-model for (a) Data Ingestion Frameworks and (b) Data Analytics Applications 27

II.3 Erudite Workflow to Deploy Data Analytics Pipeline 29

II.4 Sample Machine Learning Pipeline 31

II.5 Meta-model for Machine Learning Algorithms 32

II.6 Sample generative capabilities of Erudite 33

II.7 Integrated version control to reproduce all historical states 37

II.8 Usability of the Erudite Framework. Box 1 shows the available selection of metamodel elements available to create an ML pipeline as shown in Box 2. Individual metamodel element's attributes can be set using the attribute selection panel in Box 3. Box 4 shows model evaluation. 39

II.9 Example of Data Analytics Application Deployment Model 40

II.10 ML Model Accuracy and Loss trend graph on CIFAR10 dataset (Test and Train) 42

II.11 GPU Performance Metrics for Sample Deep Learning Training 42

II.12 Performance Monitoring of the prediction services (a)The execution latency of InceptionResnetV2 and Xception model on different ML containers with variable configurations, (b) Host CPU utilization of the ML containers (c) Host Memory utilization of ML containers (in MB) 43

II.13 Varying number of machines (each host a ML container) to guarantee QoS on dynamic workload 44

III.1 Desired Level of Abstraction for a WebApp Business Model 47

III.2 A TOSCA-compliant PHP- and MySQL-based Application Deployment Workflow 48

III.3 Box 1 depicts the responsibilities of service deployment team, which is to define the low-level scripts so that existing automation tools can configure the application components and orchestration tools can provision the infrastructure for application components and execute them on heterogeneous cloud environments. Box 2 depicts the contributions of this chapter which introduces a self-service framework and automates whole infrastructure design solutions for these tools. 50

III.4 The CloudCAMP Workflow 55

III.5 A Partial Meta-Object Facility (MOF) model of CloudCAMP DSML and Platform 57

III.6 Main Meta-Model of CloudCAMP framework. The black lines depict containment, the red lines depict inheritence and blue lines depict connection. 60

III.7 Entity-Relation(ER) Diagram of CloudCAMP knowledge base 63

III.8 Sample portion of KnowledgeBase Database tables 63

III.9 (a)Sample DBapplication type template and (b)Sample portion of the Auto-generated code for Deploying MySQL DB application 64

III.10 Specifications related to WebApplication type 71

III.11 Specifications related to DBApplication type 71
III.12 Comparing difficulty percentages to deploy services in different approaches 73
III.13 Likeliness of using CloudCAMP for future cloud services deployment . . 73

IV.1 Box plots of prediction times for different deep learning pre-trained models on different numbers of CPU cores (2, 4 and 8). 81
IV.2 An abstract state machine showing different states and transitions associated with a life cycle of a VM in cloud infrastructure. Edges are labeled with actions and time duration to complete the state transition. 86
IV.3 The setup times (in seconds) for different deep-learning prediction models as per our experiment. The blue bar shows VM deployment time (t_{vm}), orange bars show the specific pre-trained model container download time (t_{cd}), and grey bar show prediction model loading time (t_{ml}). 87
IV.4 Architecture of Barista serving system. 89
IV.5 Data flow model of Barista platform manager. 90
IV.6 Top ranked distribution that describes the variation in the sample data. The distribution (blue) is plotted on top of the histograms (orange) of observations. 98
IV.7 Performance comparison of Barista (blue) and Prophet(green) along with ground truth (First Dataset) (red) 101
IV.8 Performance comparison of Barista (blue) and Prophet(green) along with ground truth (Second Dataset) (red) 101
IV.9 Cumulative Absolute Percentage Error Distribution of First Dataset 102
IV.10 Cumulative Absolute Percentage Error Distribution of Second Dataset . . 102
IV.11 Cost comparison between multiple VM configurations (Cost infinity means the VM is infeasible option, it cannot serve the request within the SLO bound) 103

IV.12 The upper image shows how we guaranteed 2 seconds SLO for Resnet Prediction service and the experienced latency by selecting Barista selected VM configuration on toll dataset, Lower image shows the actual request rate, predicted request rate, and number of allocated VMs (t3.small (2cores)). 104

IV.13 The upper image shows how we guaranteed 1.5 seconds SLO for Wavenet Prediction service and the experienced latency by selecting Barista selected VM configuration on taxi dataset, Lower image shows the actual request rate, predicted request rate, and number of allocated VMs (t3.small (2cores)). 104

IV.14 The upper image shows how we guaranteed 2 seconds SLO for Xception Prediction service and the experienced latency by selecting Barista selected VM configuration on toll dataset, Lower image shows the actual request rate, predicted request rate, and number of allocated VMs(t3.xlarge (4cores)). 104

IV.15 Barista Performance Results on selected VM configuration as backend . . 104

IV.16 Vertical Scaling to allocate the number of CPU cores(red line) while maintaining the SLO bound of 5 seconds. The blue dotted line shows the workload pattern, and solid navy blue line shows the latency of the prediction services if run on maximum allocated cores on a VM of 8 cores. The green line shows the latency if we dynamically (de)-allocate the cores. . . 104

V.1 Life Cycle of Machine Learning Task 108

V.2 Parameter Server architecture for distributed DL (data parallel) training . . 113

V.3 Absolute error between predicted steering value and the actual steering value (with and without model update) 118

V.4 DL model training time (per epoch) for different resource configurations and different batch sizes. 119

V.5 DL model training time (per epoch) on standAlone TX2 and on heterogeneous cluster with different batch sizes. 119
V.6 Variation of step time w.r.t device type 120
V.7 Increase in step time due to resource contention 120
V.8 Initial and Final resource Usage along multiple resource dimensions. . . . 121
V.9 Deep-Edge architecture 126
V.10 Modes of operation 128
V.11 Event Sequence Diagram of Data Sharding and Resource Scheduling . . . 130
V.12 Event Sequence Diagram of Profiling the background tasks along with DL model update task 130
V.13 Event Sequence Diagram of Failure handling 134
V.14 Stressing GPU increases compute time. 136
V.15 Stressing CPU increases compute time. 136
V.16 Increasing Batch size decreases compute time. 137
V.17 Increasing Batch size decreases update time. 137
V.18 Increasing the batch size increases the memory footprint. 137
V.19 Increasing workers nodes increase update time. 137
V.20 Stressing CPU of parameter server increases update time. 138
V.21 Epoch time distribution 140
V.22 Speed Up distribution 141
V.23 Epoch time with different cluster setup. 142
V.24 EpochTime when GPU is occupied by co-located application 142
V.25 Model convergence with batch combination of 64,16,16,16 on cluster of 1tx2 and 3 nanos. 143
V.26 Model convergence with batch combination of 8,8,8,8 on cluster of 1tx2 and 3 nanos. 143

CHAPTER I

INTRODUCTION

I.1 Emerging Trends

The adoption of Cloud computing across business organizations continues to grow because of its attractive offerings such as on-demand self-service, broad network access, rapid elasticity, scalability, resiliency, measured service, and many more. Infrastructure-as-a-service (IaaS), powerful virtual Platform-as-a-Service(PaaS), and advanced cloud Software-as-a-Service(SaaS) enable cloud computing to permeate every business community. To top it off, the Internet of Things (IoT) comprising smart edge computing devices along with Big Data Analytics has pushed the limits of what can be achieved with the cloud computing paradigm.

IoT devices (e.g., surveillance cameras, wearable technologies, industrial sensors, smart building appliances, health monitoring systems, vehicles) have enabled a variety of smart distributed applications where data is collected and transformed in high volumes and velocity. Smart applications process continuous streams of information and provide real-time and robust *predictive analytics* based on the identified data patterns [1]. For finding the patterns of the data over time, the data needs to be stored and then analyzed using batch analytics, e.g., as in the case of finding consumer shopping behaviors in a store over a month or to find a traffic pattern of an area during rush hours.

The backbone of the predictive analytics tasks comprises one or more underlying machine learning (ML) models for detecting the patterns in the dataset. In developing these ML models, developers must be cognizant of the range of feasible ML models (e.g., linear models, decision trees or deep neural networks), and be able to select from among the plethora of ML libraries and frameworks that are available. Moreover, the prediction quality of the developed ML models depends on features and hyperparameters tuning, which

are time-consuming and requires significant expertise. Therefore, developers can benefit from using an intuitive and rapid ML model development framework that enables a faster and robust model building process using automation and seamless deployment on the state-of-the-art computing infrastructure to accomplish training and evaluation tasks.

The predictive data analytics business model is deployed to obtain insights from the new stream of information (live analytics) or to perform batch prediction by querying a set of new observations. Typically, the data analytics pipeline comprises ingestion, storing, pre-processing, transformation, and cleaning, and visualization of the data.

The ML-based data analytics application deployment and management are often hampered by the complex system configuration management, diversity of ML libraries and frameworks, and the range of hardware and cloud platform choices available. The concept of *data analytics service* advocates the view that data can be analyzed efficiently and seamlessly without the need for complex deployment, infrastructure provisioning and configuration, capacity planning, and management procedures during the application lifecycle management phase [2]. With the emergence of microservice-based architecture and infrastructure automation and orchestration tools, there is a trend these days to have these data analytics services be built around serverless computing concepts. All the application deployment and management details of infrastructure are abstracted from the developer or operator.

In the serverless computing realm, infrastructure (re-)configuration, deployment, migration of application components, and performance monitoring of resources are essential to scale the application components iteratively to guarantee that the Service Level Objectives (SLOs) are met under workload variation. Moreover, resource optimization for these services while maintaining the SLOs is of utmost importance to minimize the operational budgets. This is a hard problem due to the diverse set of available resource configurations, each with its deployment and management costs.

Modern smart data analytics applications such as image recognition or voice assistants

often provide live predictive analytics by analyzing the new stream of information in near real-time. Due to low-latency requirements, traditional cloud computing alone may not provide the desired quality of service (QoS) properties of these IoT analytics applications due to the high cost of moving data to distant clouds, network congestion, and the unacceptable round trip delays in obtaining critical insights about the domain problems [3, 4].

Instead, these applications must be able to utilize the edge computing resources [5] to execute the live analytics model closer to the source of the data – often referred to as edge analytics. However, stringent constraints on CPU and memory of edge/fog resources require an intelligent distribution and management of these analytics application components across cloud-edge resources. The generalized architecture of a data analytics application across the cloud-edge spectrum is shown in Figure I.1.

To develop, deploy, and manage all the predictive data analytics service components, we have developed a framework called *Stratum*. Figure I.1 depicts the general architecture of how an analytics application can be deployed using Stratum using model-driven engineering (MDE) [6]. The Stratum deployment engine can deploy data ingestion tools, stream processing tools, batch analytics tools, machine learning platforms, and frameworks on the target machine (bare metal and virtualized). Given an abstract business model and user-defined attributes, the deployment engine in Stratum called *CloudCAMP* (see Chapter III) is capable of generating Infrastructure-as-Code (IAC) solution, and then deploy it on the respective target machines [7, 8].

At the heart of Stratum is a domain-specific modeling language (DSML)-based rapid ML model development framework called *Erudite* (see Chaper II). Erudite provides ML developers and deployers a user-interface with higher-level abstractions. Using the code-generation capabilities of the DSML, the ML developer can create and evaluate their model using existing ML libraries and frameworks. Once the ML model is built and evaluated, the Stratum framework can save and profile it [9, 10]. Then, based on the provisioning strategy, the user's ML model will be integrated with the data analytics pipeline on the appro-

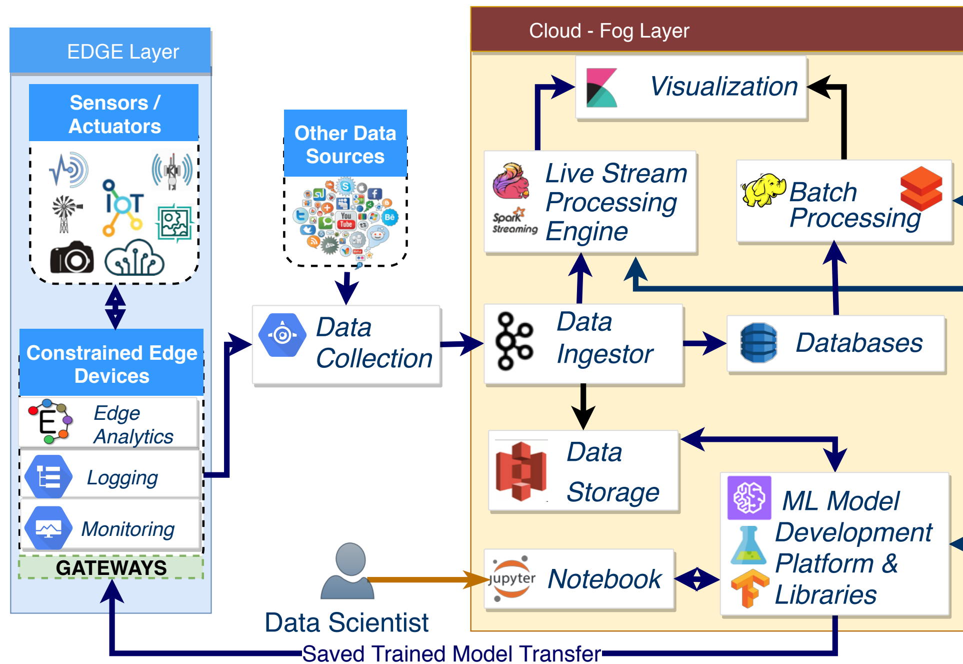


Figure I.1: Generalized Representation of Applications Architecture

priate machine across cloud-fog-edge, and the right number of resources is allocated to the user by Stratum's serverless model serving platform called *Barista* (see Chapter IV) [11]. Moreover, Stratum can also decide to place the ML model on the edge devices based on analyzing the computation and communication overhead. The model evaluation, training(update), and deployment can be done iteratively in an event-driven manner based on business requirements.

Despite advances in ML technology, especially deep learning (DL), predictive analytics-based applications hosted in production environments often experience the arrival of new data, or the existing data patterns change rapidly over time. This leads to *Concept Drift* [12], where the accuracy of the prediction model degrades over time [13], making the deployed DL model a poor fit for changing situations, and thereby requiring updates to the model with the new data. The cloud meets the high computational needs of the model update process though continuously sending the new raw data to the cloud incurs significant communication overhead and can also incur privacy issues. With the advent of GPU-enabled

edge devices, the DL model update can be performed at the edge in a distributed manner using multiple connected edge devices. However, efficiently using the edge resources for deep learning model updates is a hard problem due to the heterogeneity in the edge devices and the resource interference caused by background workloads, such as predictive analytics tasks, already running on the edge devices. *Deep-Edge* (see Chapter V) proposes an efficient framework for ML, especially Deep Learning(DL) model update, to incorporate recent data on heterogeneous edge clusters while minimizing the job completion time for distributed data-parallel deep learning re-training jobs.

I.2 Key Research Challenges and Solution Needs

In the context of deployment and management of data analytics services along with the development of the predictive analytics model, we have identified a set of fundamental underlying requirements and challenges. We have organized these challenges along four dimensions as described in Figure I.2, which form the four focus areas of this research:

① Automation of Infrastructure and Application provisioning, ② Automation of Machine Learning (ML) pipeline development, ③ Proactive Resource Management to handle the dynamic workload using serverless computing paradigm, and ④ Interference-aware strategy for continual ML model update on the heterogeneous edge cluster.

I.2.1 Requirement 1: Automation of the ML Development Pipeline

The goal of the ML model development framework is to enable fast and flexible development of the ML model by raising the level of abstraction and deployment of state-of-the-art ML capabilities. It should aid application developers in prototyping the ML model, and the ML model should be deployed automatically with the desired ML library on the target hardware. Moreover, such an approach should be both scalable and efficient [14]. We will now discuss the challenges to abstract ML framework and library-specific code generation capabilities.

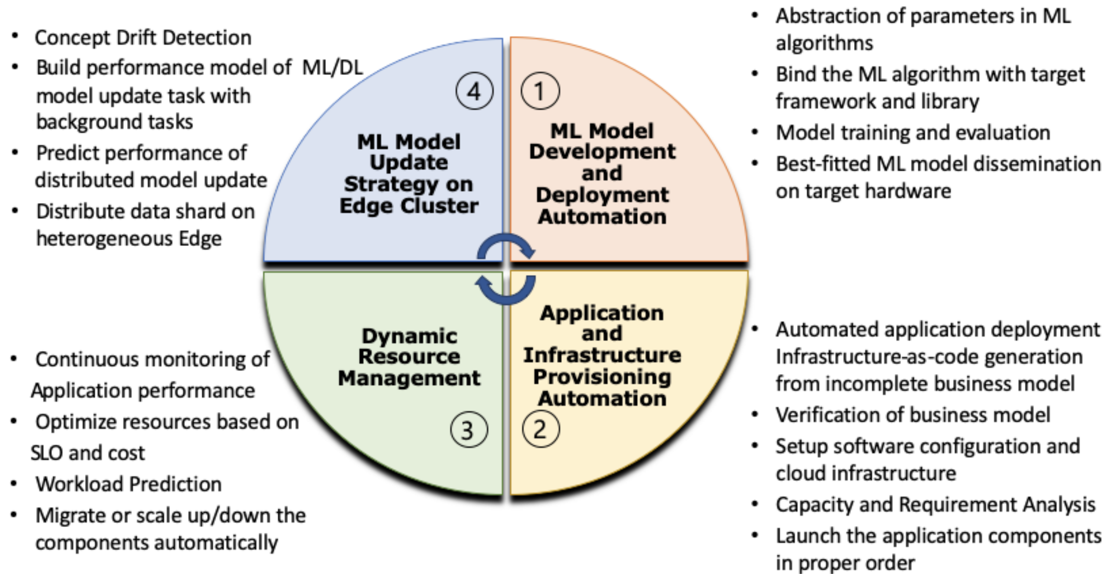


Figure I.2: A Taxonomy of requirements and challenges for Data Analytics Service development, deployment, and management across the cloud-fog-edge spectrum

I.2.1.1 Challenge 1: Abstraction of ML Pipeline

The parameters of feasible ML models (e.g., linear models, decision trees, or deep neural networks) need to be captured and abstracted away from novice users. The developers should be able to select the required ML algorithms from among the plethora of ML libraries and frameworks that are available along with data-preprocessing methodologies. To use the Model-Driven Engineering(MDE) [15] paradigm, we have to capture all the ML algorithms, their specific attributes, and the data pre-processing methods in a metamodel by virtue of reverse engineering. The evaluation methodologies of ML algorithms should be incorporated in a metamodel for model-driven code generation.

I.2.1.2 Challenge 2: Code-generation for ML Model Training and Evaluation

To aid the developer in building their model without writing code, we have to develop an abstraction, which can generate code with the target libraries such as scikit-learn, spark MLlib, based on high-level user-defined choices. Moreover, to ensure the high prediction quality of the developed ML models, the hyperparameter tuning needs to be automated.

The framework should distribute ML model training (if feasible) on different machines (if available) to speed up the process. Based on the evaluation strategy, the framework should give the best-fitted model on the training data.

I.2.1.3 Challenge 3: Support for ML Deployment

The microservice architecture supports the integration and migration of application components with ease. The microservice architecture holds promise for ML model incorporation in the data analytics service pipeline [16]. For instance, microservices expose methods and REST APIs that can be used to control the ML pipeline deployed in the container and can be placed in the cloud, fog, and edge nodes. Moreover, the ML module can be placed on the edge devices for edge analytics, or it can be placed on cloud or fog layer for live or batch analysis of data. These decisions are made based on user requirements and the requirement-capability analysis. However, for automating the encapsulation process with all required software packages, one needs to handle the aforementioned challenges. Moreover, building Linux containers is also dependent on the target hardware, which requires domain expertise.

I.2.2 Requirement 2: Automation of Infrastructure and Application Provisioning

Consider, as an example, a distributed, data-intensive application deployed in the cloud. It requires significant efforts for the designers to:

① configure and deploy the application to use the available big data frameworks [17],

② set up the virtual machines to host the frameworks, and finally

③ establish the connection properly among the application components.

Moreover, adding an application component which requires a different framework exacerbates these problems. We need to abstract away the architecting phase by pre-defining

the configuration of applications and infrastructure as intuitive, high-level modeling artifacts, which will speed up the deployment and migration process. To propose an automated infrastructure and application provisioning framework by addressing these problems, we outline the following key technical challenges:

I.2.2.1 Challenge 4: Abstraction of Application and Infrastructure details

The intriguing infrastructure complexities for the deployment of the application have to be identified and abstracted away from users. The required software packages for the application components depend on the operating systems (OS), the version of OS, and the package manager of OS – which are the commonality points for the application details. The variability points, such as selection of cloud providers, hardware selection, OS selection can be defined by the user. Moreover, all related cloud and application-specific configuration need to be abstracted away from the user. The goal is to reduce the number of variability points for rapid deployment using the framework. Designing the framework by capturing all the required specifications is challenging.

I.2.2.2 Challenge 5: Infrastructure Code-generation from Abstract Model

For the template-based transformation from the abstract business model to an Infrastructure-as-code (IAC) solution (e.g., in Ansible), a language has to be developed. Developing a DSML to realize the abstract model and to parse the user-defined variability points is key to abstracting away the infrastructure specific details.

I.2.2.3 Challenge 6: Verification of Abstract Deployment Model

The framework needs to provide out-of-the-box "correct-by-construction" IAC solution using constraint checkers. We need to validate the abstract business model by conducting requirement-capability analysis on the target hardware or cloud provider offerings before proceeding to actual deployment.

I.2.2.4 Challenge 7: Extensibility and Re-usability

The framework should be implemented in a modularized way, and each module must be easy to reuse via a seamless plug and play architecture. Incorporating these agile methodologies during the design phase needs to be addressed. If a new hardware architecture is introduced or a new service is introduced, we need to add support in a standardized manner. The framework should provide RESTful APIs to communicate among various services including other platform service APIs/RESTful services to enable the deployment and runtime integration of application components or services. Moreover, the framework should support collaboration and version control.

I.2.3 Requirement 3: Proactive Resource Management

Continuously monitoring the performance of the infrastructure and application components is required to derive the best hardware configuration under dynamic workloads. Making effective resource management decisions for these services is a hard problem due to the dynamic workloads and diverse set of available resource configurations that have their deployment and management costs. The goal of proactive resource management is to upgrade the current infrastructure state ahead of time to minimize the SLO violation in a dynamic environment while minimizing the operating budget of the data-analytics service providers.

I.2.3.1 Challenge 8: Workload Variation

The workload generated for predictive analytics applications may be dynamic but may follow a diurnal pattern over a long period. These patterns need to be learned based on historical data and subsequently handled [18]. Identifying the seasonality and trend based on the historical time series pattern of the workload is required to forecast the number of requests.

I.2.3.2 Challenge 9: Optimal Resource Selection

The state-space to search for the optimal cloud configuration is vast, and it is not feasible to try all the options. The total cost needs to be minimized while the SLOs of all the requests can be satisfied, however, as this is a known NP-hard hard, an efficient heuristic needs to be developed based on the application performance model.

I.2.3.3 Challenge 10: Proactive Resource Provisioning

As the resource provisioning time is much higher than the prediction SLO, the resources need to be provisioned proactively ahead of time to meet the stringent SLO requirements. Based on the forecasted workload, current infrastructure state, provisioning time, and profiled execution time of the application on the hardware, the resource manager has to be designed to make horizontal and vertical scaling decisions dynamically.

I.2.4 Requirement 4: Interference-aware Strategy for ML Model Update

As the quality of prediction analytics task degrades over time with the arrival of new data, machine learned models, especially Deep Learning (DL) models, need to be updated with the new data points. Sending all the data to the cloud imposes significant communication overhead [3, 19, 10] and may raise privacy issues. DL models can be updated on the GPU-enable edge cluster; however, there are multiple challenges that need to be handled to minimize the model update time.

I.2.4.1 Challenge 11: Heterogeneity-aware Data Management

The change in the number of data points changes the training time. In the edge cluster, the nodes are heterogeneous, and because of the background load, each node has a different per-sample processing time, and it may vary if we change the number of workers in the cluster. The goal of our solution called Deep-Edge is to minimize the overall cost of

distributed training on a set of heterogeneous edge nodes by choosing the proper data distribution for each edge device while subject to some system and performance constraints.

I.2.4.2 Challenge 12: Resource Interference-awareness

The edge devices are typically assigned to perform some latency-critical jobs. Hence, the DL model update on these devices should not hamper the existing latency-critical jobs. The performance suffers due to interference from the co-located workload and the sharing of non-partitionable resources [20, 21]. This issue has to be handled proactively before deploying the model task. Moreover, due to background latency-critical jobs, the re-training time for the DL model can be affected. Therefore, devising an interference-aware strategy to minimize the job completion time of the distributed DL model update while guaranteeing the SLO of the latency-critical jobs is challenging.

I.3 Organization of the Dissertation

To resolve the range of challenges described in Section I.2, this doctoral research studies the challenges and requirements in detail[1]. For all the requirement and challenges, this doctoral research formulates the problem, design the proposed solution, and validate novel ideas by conducting empirical studies as required in the field of systems research. The contributions of this research include the following tasks:

① ***TASK 1: Automation of ML model development and deployment***

A holistic and intelligent ML model lifecycle management framework, ERUDITE, for development and deployment of ML-based predictive analytics applications is introduced in Chapter II.

② ***TASK 2: Automation of infrastructure and application provisioning***

A model-driven approach to automate cloud services deployment and management is

[1] All the literature surveys specific to the requirements are described in the respective chapters.

discussed in Chapter III. The CloudCAMP framework is introduced as the basis for the deployment time and runtime composition and orchestration through generative programming.

③ ***TASK 3: Proactive resource management to handle the dynamic workload***

Chapter IV proposes an efficient and scalable serverless framework for deployment and management prediction services, especially deep-learning prediction services. Based on workload forecasting and application profiling, how to select the optimal resource configuration, and how to manage the resources, proactively are the main thrusts of this chapter.

④ ***TASK 4: Interference-aware strategy for continual ML model update on heterogeneous edge***

Chapter V proposes a framework to facilitate ML, especially deep learning (DL) model update on the heterogeneous edge cluster. Based on the application performance model with various background stress, Deep-Edge presents an efficient interference-aware strategy to update the ML/DL model to minimize job completion time while guaranteeing the SLO of latency-critical jobs.

Finally, Chapter VI summarizes the dissertation research by alluding to future directions.

CHAPTER II

ERUDITE: A LIFECYCLE MANAGEMENT FRAMEWORK FOR MACHINE LEARNING BASED PREDICTIVE ANALYTICS APPLICATIONS

II.1 Introduction

II.1.1 Emerging Trends

Internet of Things (IoT) provides an ecosystem of interconnected devices (e.g., surveillance cameras, wearable technologies, industrial sensors, smart building appliances, health monitoring systems, vehicles, etc.), which generate and transform high volumes of data at high velocity that is then analyzed to derive valuable insights and make informed decisions for a variety of smart application domains. For instance, smart data analytics applications such as product management, process optimization, video analytics, predictive analytics, and recommendation systems rely on the live and in-depth analysis of the incoming data streams as well as the historical data [22].

II.1.2 Challenges and State-of-the-art Solutions

It is in this context that traditional cloud computing alone may not provide the desired quality of service (QoS) properties of these IoT analytics applications due to the high cost of moving large amounts of data to distant clouds and the unacceptable round trip delays in obtaining critical insights about the domain problems. In particular, applications such as emergency response, health monitoring, intelligent assistants, among others, require real-time, low latency analytics capabilities. Instead, these applications must be able to exploit the edge [5] and fog computing resources opportunistically. Edge/fog computing enables applications to be executed closer to the source of the data, thereby eliminating many of the problems that stem from having to use the distant cloud. At the same time,

however, stringent constraints on CPU and memory of edge/fog resources may require an intelligent distribution and management of these analytics applications across such edge/-fog resources.

Unfortunately, IoT analytics application developers often do not possess the expertise to enforce useful application (re)deployment and dynamic resource management decisions. Thus, there is a compelling need for an approach that relieves the IoT analytics application developer from having to determine the placement of analytics application components, monitoring their resource usage, and controlling different data processing tasks across the cloud-fog-edge devices to enable optimal data control across the edge-to-cloud resource spectrum [23, 20].

Serverless computing shows promise in addressing these issues because it allows application developers to develop the application components without concerns for the intriguing details of the infrastructure. Specifically, *Functions as a Service (FaaS)* is a concept to achieve serverless computing by allowing developers to execute code in an event-driven manner without building or maintaining a complex infrastructure. Scrutiny of an IoT data analytics application reveals an application structure that is made up of a collection of loosely coupled services, such as data ingestion, stream and batch processing, *Machine Learning (ML)-as-a-service*, visualization, and storage [24], where individual components are interconnected by Restful APIs. The loosely coupled nature and event-driven nature of these applications make them highly suitable to be hosted using a serverless paradigm.

An additional challenge faced by IoT analytics application developers pertains to developing artificial intelligence (AI)/machine learning (ML) model using large training data sets. This requires the developers to be cognizant of the range of feasible ML models (e.g., linear models, decision trees or deep neural networks), and be able to select from among the plethora of ML libraries and frameworks that are available. Moreover, they are also responsible for ensuring high prediction quality of the developed ML models, which depends heavily on the choice of features and hyperparameters, which themselves need to be

tuned in an offline evaluation process. IoT analytics application developers are unlikely to be experts in all of these steps, including using the trained models at runtime.

II.1.3 Overview of Technical Contributions

To address the range of the challenges mentioned above that an IoT analytics application developer is likely to face, in this chapter, we present *Erudite,* which exploits the benefits of the serverless computing paradigm thereby relieving the developer of all the deployment and resource management issues while ensuring that applications are delivered their required QoS properties. To make it easier and intuitive for the developer to develop ML models for their analytics application workflow, and to overcome the challenges stemming from having to deal with the variability in ML libraries and hyperparameter optimizations, Erudite defines a domain-specific modeling language that enables declarative specification of application needs. In turn, generative programming mechanisms within Erudite helps to synthesize the metadata needed to provision an ML training and optimization workflow automatically. Subsequently, Erudite automates the deployment and execution of the trained models across the distributed edge-cloud resources using the serverless approach wherein it enforces appropriate and effective auto-scaling mechanisms for individual services and also provides strategies to migrate trained ML model between different nodes as required to maintain the QoS while optimizing the cost of model execution.

The novelty of the Erudite approach lies in how it holistically addresses a range of challenges that IoT analytics application developers face and systematically blends multiple different approaches to realize a complete solution. Specifically, Erudite makes the following contributions:

① Erudite integrated with CloudCAMP (see Chapter III) provides a Domain-Specific Modeling Language (DSML) to hide the lower-level details of infrastructure deployment and provides an easy to use web-interface for the end-users to utilize the platform. Erudite support built-in ‘correct-by-construction’ using constraint checker.

Erudite can validate the application deployment architecture by conducting a requirement-capability analysis on the target hardware before actual deployment.

② We present a rapid AI/ML model development framework, where the developers can connect to a wide range of data sources and build their own AI/ML applications. Specifically, Erudite provides an AI/ML Service Encapsulation approach by leveraging CPU and GPU-enabled containerization architectures and API abstraction for standard ML libraries and frameworks. It provides an easy-to-use framework on which developers can quickly build, train, and evaluate the ML model on historical data requiring little to no-code development.

③ Once the AI/ML model is ready and finalized, the application components can be exposed as RESTful APIs, and seamlessly integrated into the business application workflow using Erudite (as shown in Figure II.1). Erudite manages the lifecycle of models efficiently and triggers retraining of the models also if the model becomes stale over time.

④ Erudite provides an intelligent way to transfer the trained model to the target machines (across the cloud-fog-edge spectrum) as an ML module for inference and analysis of incoming data sources. ML module(s) can be placed on the edge devices for edge analytics, or at Cloud or Fog layer for live or in-depth analysis of data depending on user requirement and requirement-capability analysis. All of these responsibilities are handled by the serverless platform using declarative specifications.

⑤ Erudite takes care of dynamic resource management by providing a strategic design wherein other resource management algorithms can be plugged in. It supports monitoring of data management and models in production by instrumenting finer-grained performance metrics collection on the target hardware. Based on the collected performance metrics such as CPU, memory, IO utilization, the resource management algorithms can enforce horizontal and vertical elasticity of resources to maintain the

QoS requirements. We validated these claims using our default resource management algorithms are enforced.

II.1.4 Organization of the Chapter

The rest of the chapter is organized as follows: Section II.2 presents a survey of existing solutions in the literature and compares them Erudite; Section II.3 presents the background and problem formulation; Section II.4 presents the design of Erudite; Section II.5 evaluates the Erudite resource allocator for a prototypical case study; and finally, Section II.6 presents concluding remarks alluding to future directions.

II.2 Related Work

In this work, we compare and contrast Erudite with the existing state of the art solutions for end-to-end software engineering lifecycle management of machine learning models from the design phase to deployment phase across cloud and edge computing environments. Ease.ml [25] is a training platform providing automatic model selection using a declarative programming approach. It relieves users from determining which models to select for a specific task at hand. Ease.ml introduces a resource scheduler to manage the deployment of the training job in a shared cluster environment used by multiple users at once.t currently supports language constructs for declaring 1) size of the input and output dataset and 2) dataset itself.

Google Vizier [26] is ML platform which supports hyper-parameter tuning service. Users within Google can specify the parameter configuration space and the optimization goals. The Vizier service then proceeds with running experimentation trials until it has met the user-specified goals. Similarly, TFX [27] is another machine learning platform at Google, which provides tools for data preparation and model inference serving. The system is built around the Tensorflow ML framework. Michelangelo [28] is an ML-as-a-service framework deployed at Uber that facilitates, building, and deploying ML models

in the cluster environment. Michelangelo DSL provides users to define the ML tasks to be used for both training and inference ML jobs. However, the DSL restricts the users from choosing only the algorithms that are supported by the platform, thus limiting users from experimenting using different ML algorithms. Alchemist [29] is another internal project at Apple that addresses the training of machine learning models in a cluster environment. It leverages the Kubernetes container orchestration platform for running the training jobs of the ML models.

To the best of our knowledge, Google Vizier, Michelangelo, and Alchemist are internal tools available for use by the users from the respective organizations and not available in the open-source environment. Concerning commercial offerings, services such as Amazon SageMaker, Microsoft Azure service, and IBM's Watson Studio is available to the paid customers. These end-to-end service running the machine learning training and deployment pipelines are restricted to their proprietary runtime infrastructures, which can potentially result in vendor lock-in issues for the end-users.

	FEATURES										
PROJECT	**ML Model Development**	**Model Training**	**Inference Serving**	**Resource Monitoring**	**Code Generation**	**Resource Management**	**Collaborative Environment**	**Model Versioning**	**DSML**	**Cloud(C)/Edge(E) Computing**	**Open-Source**
Ease.ML	✗	✓	✗	✗	✓	✓	✗	✗	✓	C	✗
Vizier	✗	✓	✗	✗	✗	✗	✗	✗	✓	C	✗
Clipper	✗	✗	✓	✗	✗	✗	✗	✗	✗	C	✓
Michelangelo	✓	✓	✓	✗	✓	✗	✗	✗	✓	C	✗
Alchemist	✓	✓	✗	✓	✗	✓	✗	✗	✗	C	✗
Amazon ML Microsoft ML IBM Watson	✓	✓	✓	✗	✓	✗	✓	✓	✗	C/E	✗
MLFlow	✓	✓	✓	✗	✗	✗	✗	✗	✗	C	✓
InferLine	✗	✗	✓	✗	✗	✓	✗	✗	✗	C	✗
DLHub	✗	✗	✓	✗	✗	✗	✗	✓	✗	C	✗
ML.Net	✓	✗	✗	✗	✗	✗	✗	✗	✗	C	✓
Acumos	✓	✓	✓	✗	✗	✗	✗	✓	✓	C	✓
ModelDB	✗	✓	✗	✗	✗	✗	✗	✓	✗	C	✓
Weka Mahout Scikit-Learn	✓	✓	✗	✗	✗	✗	✗	✗	✗	C	✓
Rafiki	✗	✓	✓	✗	✗	✓	✗	✗	✓	C	✗
STRATUM	✓	✓	✓	✓	✓	✓	✓	✓	✓	C/E	✓

Table II.1: Comparing Stratum with other state-of-the-art solutions.

Clipper [14] focuses on low-latency prediction serving or inference serving aspect of the ML stage. Clipper uses an ensemble of prediction models to select and combine the best models that have higher accuracy from different ML frameworks. Unlike Clipper,

Rafiki [30] addresses training and inference service deployment in the cloud environment. Rafiki also provides a hyperparameter tuning service for parameter exploration. For inference serving, it provides an ensemble approach to allow for better accuracy of prediction results. MLFlow [31] is an open-source ML platform project that covers end to end lifecycle phases of ML development and deployment. It has Python-based generic APIs that allow binding to different ML libraries. It also has support for data preparation, training, and deployment of the ML models across heterogeneous service providers such as Microsoft Azure, Amazon Sagemaker, Apache Spark. InferLine [32] presents a prediction pipeline provisioning in a cloud environment. It provides inference serving across a DAG of ML models subject to latency constraints. It provides a hybrid approach for maintaining end-to-end latency constraints by changing model configurations.

ML.NET [33] is an open-source ML pipeline framework by Microsoft. ML models can be integrated directly into application codebase natively. Also, predictions can be served by the OS-agnostic platform supported by the .NET Core framework. This feature is useful for prediction serving at the edge computing environments as the application need not communicate with external service to get prediction results, as the model itself is packaged within the application. DLHub [34] is a platform that supports publishing trained models to model repositories and provides model serving capabilities for ML. DLHub implements an executor model for deploying inference tasks to the serving infrastructures. The currently supported infrastructures in DLHub comprise of TensorFlow Serving, SageMaker, and Parsl-based execution platforms.

Acumos [35] is another open-source effort towards easing packaging, cataloging, and sharing activities of ML models. Acumos provides a marketplace where developers can search and find pre-trained models for downloads. It also allows users to publish their custom models to the marketplace for easy sharing. Similarly, ModelDB also provides support for training and managing ML models. It provides a web-based GUI that allows for easy visualization of the ML pipelines. It also supports model versioning, visual exploration of

models, and has support for collaborations. Other efforts such as Weka, Apache Mahout, Scikit-Learn provides declarative programming means to design and create ML pipelines and models. However, these platforms do not provide means for model versioning and model deployment.

Compared to these existing work, Erudite provides a unified framework that supports design-time tools and deployment tools for model construction and deployment. Also, Erudite handles deployment across a heterogeneous set of platforms spanning from cloud to edge computing platforms using CloudCAMP [7]. Erudite also provides version support while creating designing models using a visual drag and drop GUI interface. Erudite leverages model-driven engineering technologies that facilitate creating custom domain-specific modeling language, automated code generation facility, and orchestration of models to be deployed on the target platforms. We believe these end-to-end capabilities are lacking in the current state of the art integrated tool suites which Erudite addresses. Table II.1 gives an overview of different feature supports provided by the existing works.

In the literature, there exist several model-based approaches to managing end-to-end application performance and resource management autonomically based on the domain-specific modeling language (DSML) [36, 37], however they do not consider about distributing the model across the cloud-edge spectrum like Erudite.

Similar to Erudite integrated with Barista [11], several platforms leverage virtual machine and container technology to deploy and scale the application components [38, 39]; however, they do not provide end-to-end data analytics deployment and management platform.

II.3 Problem Formulation

In this section, we use a motivating case study to derive the key solution requirements for the Erudite framework that we present in this chapter.

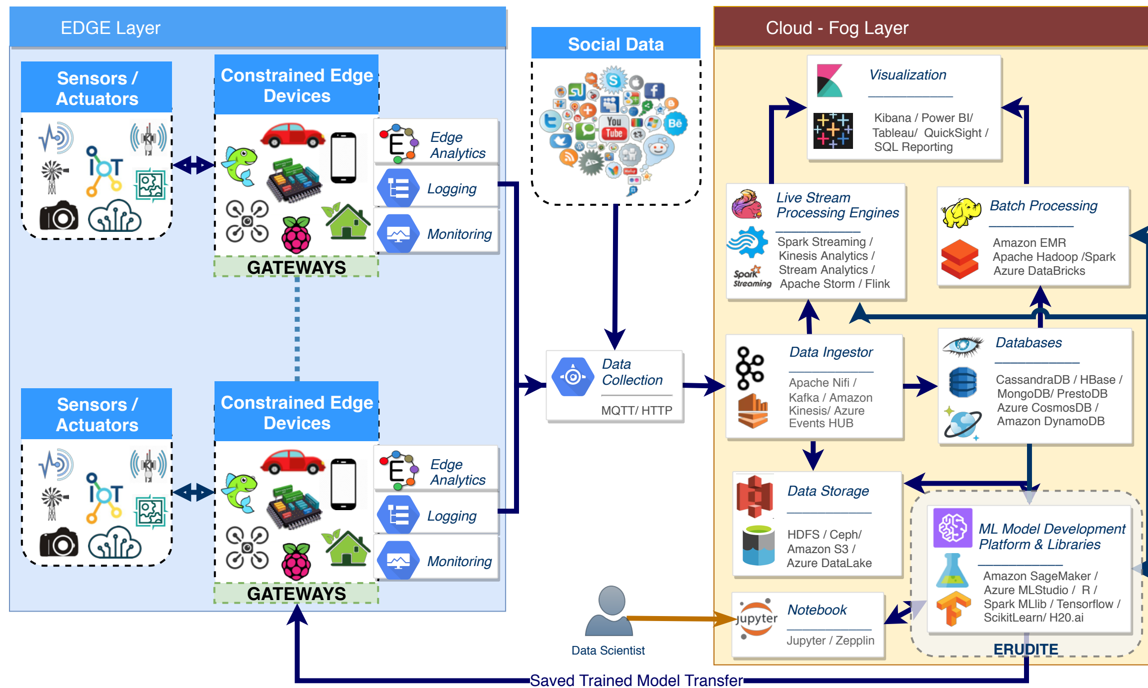


Figure II.1: Generalized Architecture of Smart Application Framework

II.3.1 Motivating Case Study and Key Challenges

Consider an IoT use case of an automated toll booth that takes images of a vehicle's license plate in order to charge the toll [40]. This application will involve a real-world workload of an image recognition service that needs predictive analytics so that it can automatically detect the license plate of the entering or leaving car from a toll plaza. The overall system will thus involve a camera that takes a picture of the license plate and analyzes the image to identify the license plate and accordingly charge the associated account with the appropriate toll. Our use case belongs to a category of applications for which many design considerations must be taken into account when developing and deploying such applications. These design considerations and the range of challenges faced by a developer are depicted in Figure II.1 and described below.

II.3.1.1 Deployment Challenges

For applications involving computer vision (e.g., the toll booth use case) or for automation assistants like Google Home using the natural language inference model, the device-to-cloud data round trip latency is considerable. Thus, processing at the edge is attractive as it reduces latency and makes connected applications more responsive. Therefore, the ability to analyze, filter, or aggregate the data before sending it to the cloud data center can lead to significant savings in network and computing resources. As shown in Figure II.1, however, an IoT analytics application developer will need the expertise to deploy such applications on a plethora of edge device types, such as Raspberry Pi, MinnowBoard, Beagle Bone, and Arduino.

Since not all analytics can be performed at the edge because an edge device may not contain all the data, for that, it may be necessary to aggregate information at fog or cloud data center servers. Examples of applications that need these capabilities include route planning for a vehicle in nearby-locations, which can be done in fog/cloud servers if the localized edge devices send the traffic data, weather data on the fog server to aggregate. Fog servers can be energy-efficient multi-core ARM64 processors and GPGPUs (e.g., NVidia TX1 with a GPU), or it can be a private cloud with limited processing power (e.g., few Intel Xeon Servers). Thus, a developer needs to be cognizant of the capabilities of the fog/cloud resources in order to make the best use of these resources.

II.3.1.2 Data Movement and Management Challenges

IoT applications generate enormous amounts of data and operate on data streams from edge devices to cloud-based resources. As shown in Figure II.1, edge devices must send the filtered data to the fog/cloud servers using one of the various communication protocols, e.g., HTTP, MQTT, CoAP. Moreover, Data Ingestion services such as Apache Kafka [41], Apache Nifi or Amazon Kinesis must be programmed to listen to incoming data streams and be able to retain the data in databases or data lakes [23]. We also require adaptive batch-

ing to acquire data at higher throughput. Once data is ingested in the server, it is possible that different subscribers may be interested in it so they can run separate live analytics on the window of streaming data and visualize the live patterns. Stream processing platforms like Apache Flink and Spark Streaming can aid in building live data analytics models. Such a system, however, must scale to the needs and numbers of the subscribers. All of these become the responsibility of the analytics application developer, who are unlikely to be experts in all of these technologies and protocols.

II.3.1.3 Model Building and Dissemination Challenges

Building predictive analytics requires the development of AI/ML learning models based on historical datasets. For that, data preparation strategies, and different ML algorithms need to be tried using one of many available ML frameworks. Developing highly accurate models requires feature engineering, model selection, hyperparameter tuning, and so on. Moreover, to speed up the training process, it needs to be driven by high-performance computing using GPU and CPU and needs to be distributed if possible. The model development platform needs to be ready rapidly for best model selection by evaluating a large number of models with automatic scoring. The model evaluation phase needs to be also parallelized.

Once the AI/ML model is trained, it may need to be pushed to edge devices for inference. For example, at the toll booth, vehicle license plate detection model can be placed at the roadside unit, which detects the license plate number, and debits a certain amount of money from the account associated with the license plate. The inference algorithm converts the license plate image to a text string, and because the size of an image is significantly less than the text strings, the amount of information flow in the network is much less, and it also reduces the latency.

II.3.1.4 Challenges in Determining the Right Hardware Needed

For more in-depth analytics, it may be necessary to build a deep learning model from the traffic cameras of the city or to analyze the behavioral and driving pattern of a car that requires a detailed analysis of data, all of which need massive computation resources. The storage and computation can be scaled in the cloud seamlessly. Batch processing frameworks such as Apache Hadoop or Spark can run deep analytics by aggregating data from multiple data sources. These frameworks can integrate the trained ML model for their diagnostic or predictive analytics as required. The challenges lie in determining proper hardware from the wide variety of device classes, selecting correct connectivity protocol, choosing deployment options (on-premise or over the cloud), and configuring the deployment platform for prototyping the business application to deliver real value.

II.3.1.5 Runtime Resource Monitoring Challenges

Once the analytics framework is deployed, it needs to be scaled automatically to handle the dynamic workload cost-effectively. The framework also needs to monitor model decay to retrain the ML model and need to push the trained model in business workflow efficiently. Moreover, all the performance metrics of the hardware components need to be monitored for dynamic decision making.

II.3.2 Solution Requirements

Based on the discussion of the wide range of challenges that an analytics application developer is likely to face, we need a solution approach that will maximally decouple the developer from these challenges and automate many of these tasks on their behalf. The solution requirements that Erudite ought to support are described next.

II.3.2.1 Requirement 1: Automated Deployment of Application components in Heterogeneous environment

In the cloud-fog-edge spectrum, significant numbers of application components can use different hardware with different operating systems (OS). The software package manager differs based on OS, and software requirements can differ based on OS versions. Software package requirements are necessary to reverse engineer the template-based creation of the microservice [8, 42]. For example, to install Apache Kafka on multiple Ubuntu machines, we need to gather all the software packages required to install Kafka, and we need to write the infrastructure-as-Code(IAC) script to deploy Kafka on target machines. We have to repeat the whole process if we want to deploy Kafka on a different cluster of Windows machine. To alleviate the labor for the application developer, we aim to build the framework so that the user only requires to fill some user-specific details, and then through one click, the Kafka service will be deployed seamlessly on target hardware. Using the framework, lifecycle services must be easily pluggable, reusable, flexible, and customizable. All the components are exposed by RESTful APIs, and can easily be interconnected. We also aim to provide a version-controlled and collaborative environment to the application developer to deploy their business workflow.

II.3.2.2 Requirement 2: Framework for Flexible ML Service Development and Encapsulation

There is a need for an AI/ML model building framework that can abstract away the ML algorithms from existing ML frameworks. A diverse set of ML capabilities, including classification (e.g., logistic regression, naive Bayes), regression, decision trees, random forests, and gradient-boosted trees, recommendation (ALS), clustering (K-means, GMMs), and many others provided by different libraries and frameworks such as Scikit-learn, Weka, Spark MLlib needs to the captured, encapsulated and abstracted in the framework. Such a development kit for the ML models also needs to provide Repository APIs, Commu-

nication Services APIs, Orchestration APIs, and other platform services APIs/RESTful services to enable the integration of various software components/services as required for model development. The model developer selects the required building blocks and tunes the hyperparameters as needed to develop and fine-tune the model. The framework also aims to integrate the autoML model to select the best model or the ensemble of models based on the evaluations.

II.3.2.3 Requirement 3: Performance Monitoring and Intelligent Resource Allocation

Erudite aims to integrate data and storage services, schedule and workload management, and container deployment and orchestrated by an execution engine with continuous monitoring of system metrics. The framework should integrate a performance data collection strategy, and then based on the metrics, the decision engine needs to allocate the resources for the specific tasks. The duration of computation can range from milliseconds (e.g., taking action based on the inference model) to hours (e.g., training a complex policy). Additionally, training often requires various hardware (e.g., CPUs, GPUs, or TPUs), whereas inference can be made in low-powerful edge devices. Therefore, application component profiling is needed before actual deployment. In the management phase, based on profiled data, and current performance data, the framework will be able to provide scalability across different dimensions, including parallel pipeline executions, event, and stream processing.

II.4 Design and Implementation of Erudite

Based on identified requirements specified in Section II.3, we have designed Erudite, which is a framework to deploy and manage data analytics pipelines. The design and implementation of the Erudite modeling environment are developed around the core concepts of Model-Driven Engineering (MDE). As part of the serverless offering, Erudite provides the user with intuitive abstractions to specify their needs. These specifications are then used

to automate the entire workflow for the user maximally. The rest of this section delves into the details of its design, outlining how it addresses the solution requirements.

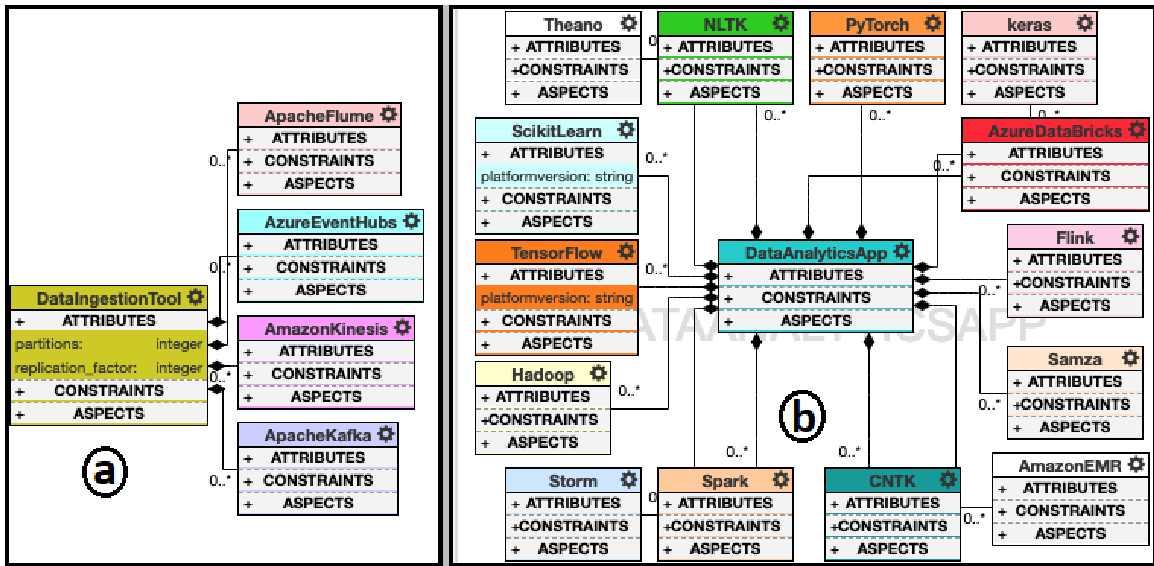


Figure II.2: Meta-model for (a) Data Ingestion Frameworks and (b) Data Analytics Applications

II.4.1 Addressing Requirement 1: CloudCAMP - Automated Deployment of Application Components in Heterogeneous Resources

We address requirement 1 by providing the user with a higher level of abstraction to work with and automate most of the tasks. Figure II.2 depicts the meta-model that captures all the concepts necessary to specify the commonalities and variabilities of the data analytics pipeline based on our Domain-Specific Modeling Language (DSML). We build our DSML on the WebGME modeling environment [43]. Using the WebGME framework, we have defined our meta-models for the DSML, created the semantics for model interpreters, and encoded generative logic for synthesizing artifacts. We capture various facets of the application and target machine hardware specifications in the meta-model.

The Erudite meta-models were developed through a combination of (1) reverse engineering, (2) dependency mapping across heterogeneous hardware, (3) dependency mapping across different operating systems and their versions. We abstract the specification

details from the end-users by identifying the commonalities of the provisioning stacks, which become the high-level reusable building blocks of the deployment and management pipeline that are captured as domain-specific artifacts. The high-level meta-model for the data-analytics and data ingestion framework are shown in the figure. The user-defined variability points (e.g., Number of Machines, pre-defined business policy) are needed to be specified by the user.

The visual GUI environment provides component blocks, which can be dragged and dropped easily to construct a prototype deployment. This reduces the barrier to entry for developers who can rapidly and concisely describe the abstract deployment model. Then, the Erudite *model interpreter* verifies the correctness of the abstract model. A NodeJS-based *code Generator* of Erudite then realizes the user-defined model through one or more interrelated meta-models that capture the syntax and semantics, and generate the Infrastructure-as-Code (IAC) solution by parsing the user-defined model and deploy it on the target machine using a template-based transformation and knowledge base. Finally, the IAC solution is executed by the *code executor* to deploy the desired data analytics architecture on the target machines across the cloud-fog-edge spectrum, as shown in Figure II.3. The deployment of the tools can be done using the Erudite model-driven approach without writing a single line of code. More details about the cloudCAMP platform are described in Chapter III. The details of the transformation approach can be found in our previous works [7, 8], and are out of scope for this chapter.

II.4.1.1 Meta-model for Heterogeneous Resources

As noted before, the resources used can be target hardware such as Raspberry Pis, NVidia Jetson TX1 GPU, any of bare metal CPU machines such as Intel Xeon, or GPU machines such as NVidia Tesla or it could be any cloud platform such as Amazon AWS or Microsoft Azure. The target OS residing on it can be Linux, Windows, Raspbian, etc., and their versions can be different. In the meta-model, we capture hardware details, operating

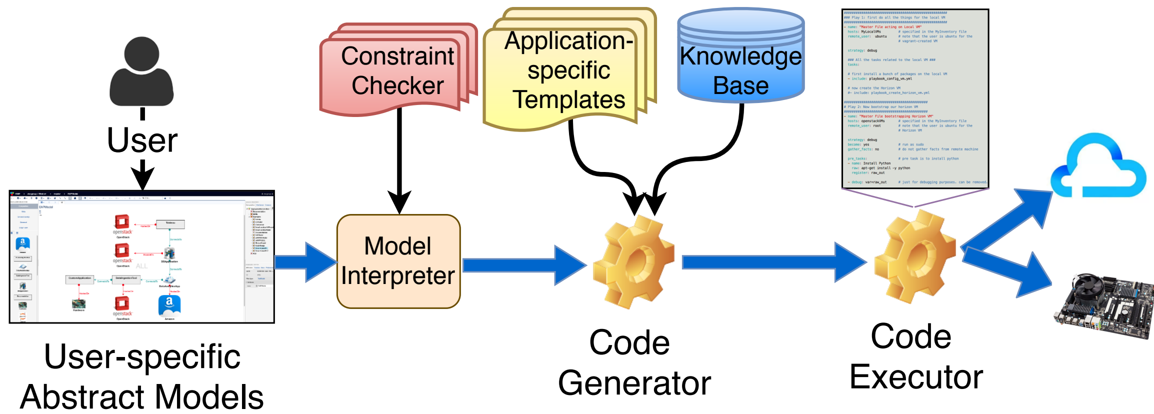


Figure II.3: Erudite Workflow to Deploy Data Analytics Pipeline

system type, and their version. The package manager of the software packages such as apt, yum, depends on the underlying hardware, whereas the software installment process depends on OS type and versions.

II.4.1.2 Meta-model for Data Ingestion Frameworks

Figure II.2(a) illustrates the meta-model for data ingestion tools (e.g., RabbitMQ, Kafka), which receive the data from the edge devices, who serve as the publisher. The data is forwarded to the subscribers and can be stored in Data Repositories. Here we capture the specifications for the Data Ingestion tools, and a specific Data Ingestion tool such as Kafka is contained within the parent dataIngestionTool class. The modeling environment includes services for interacting with the Data Repositories and other microservices using RESTful APIs, as shown in Figure II.1. The user must select the specific data ingestion tool to deploy it on the target platform. We also verify the correctness of the user model based on the semantics defined in the meta-model. For example, if the user selects the AzureEventHubs as there desired Data Ingestion Tool, then Amazon AWS or OpenStack or bare metal cannot be its' target deployment platform.

II.4.1.3 Meta-model for Data Analytics Applications

The live or in-depth analytics frameworks such as Hadoop, Spark, Flink, Samza, and more need to be deployed on the target distributed systems. Henceforth, those frameworks' specifications need to be contained in the Data Analytics parent class. Moreover, as shown in Figure II.1, the trained ML model needs to be integrated with the stream and batch analytics frameworks. So to deploy the production-ready machine-learning pipeline we need to capture the specifications for the ML libraries and frameworks such as Tensorflow, sci-kit learn, PyTorch, CNTK and more as shown in Figure II.2(b). We also capture the specifications to start the AmazonEMR service on the cluster of Amazon EC2 machines or AzureDataBricks service on the cluster of Azure Virtual Machines.

II.4.1.4 Meta-model for Data Storage Services

We have captured the storage service specifications in the meta-model also. The subscribers from the data ingestion tools can consume the data in an event-driven way and store in the storage service. Our storage class contains AmazonS3, HDFS, AzureBlobStorage, Ceph based distributed storage services. The storage service can easily be deployed with user specifications such as folder name, bucket name. The data can also be stored in relational databases such as MySQL or NoSQL databases such as MongoDB, Cassandra. The user has to select the database of their choice, and the database attributes need to be configured by the user by providing database names, user id, password, replication factor, etc.

II.4.2 Addressing Requirement 2: Erudite Development Kit for AI/ML Model Development

The Erudite Development Kit is part of the Erudite framework and is built with a focus on ML service development and encapsulation. It is an integrated model environment

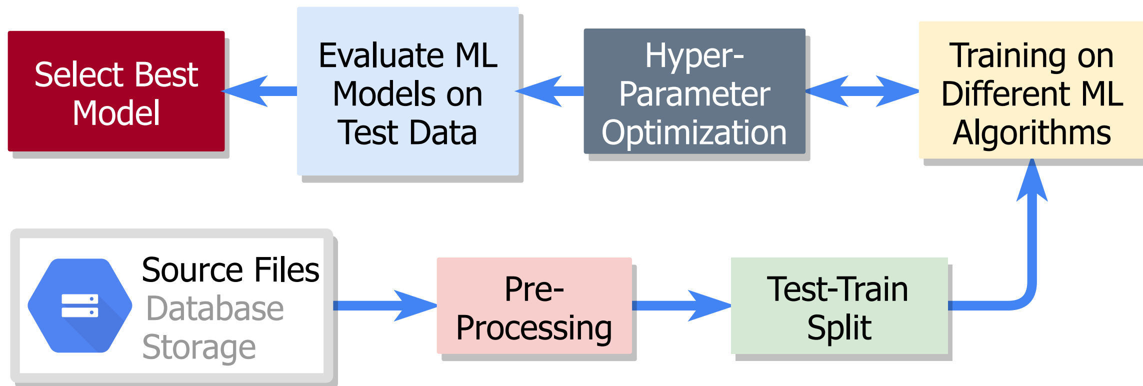


Figure II.4: Sample Machine Learning Pipeline

aid to build machine learning (ML) pipeline by abstracting data pre-processing strategies, ML algorithms on an existing ML framework, and evaluation strategies. The model-driven ML development framework will benefit the novice and expert data scientists to rapidly prototype their model using the generative capabilities provided by the framework. It provides a diverse set of encapsulated ML capabilities, which can be easily bound to the ML pipeline as required using the user interface, as shown in Figure II.4. The data scientist needs to specify the data storage type and location, their pre-processing strategies, selected ML techniques, hyper-parameter tuning strategies, evaluation techniques to create the ML pipeline.

II.4.2.1 Main Meta-model for Erudite Framework

Erudite modeling environment consists of a standard data exchange format that provides read and write capabilities in various data formats from various data sources. It also provides unified data access across different machine learning frameworks such as scikit-learn, SparkMLlib, and tensorflow. The integrated ML pipeline allows retrieval (input block), and manipulation of data stored in cloud storage such as Amazon S3, or Azure DataLake Store, or locally or in a scalable (HDFS-based) datastore via RESTful and other microservice interfaces (DataPreprocessing block).

Machine Learning frameworks also provide ML pipeline construction, model evalua-

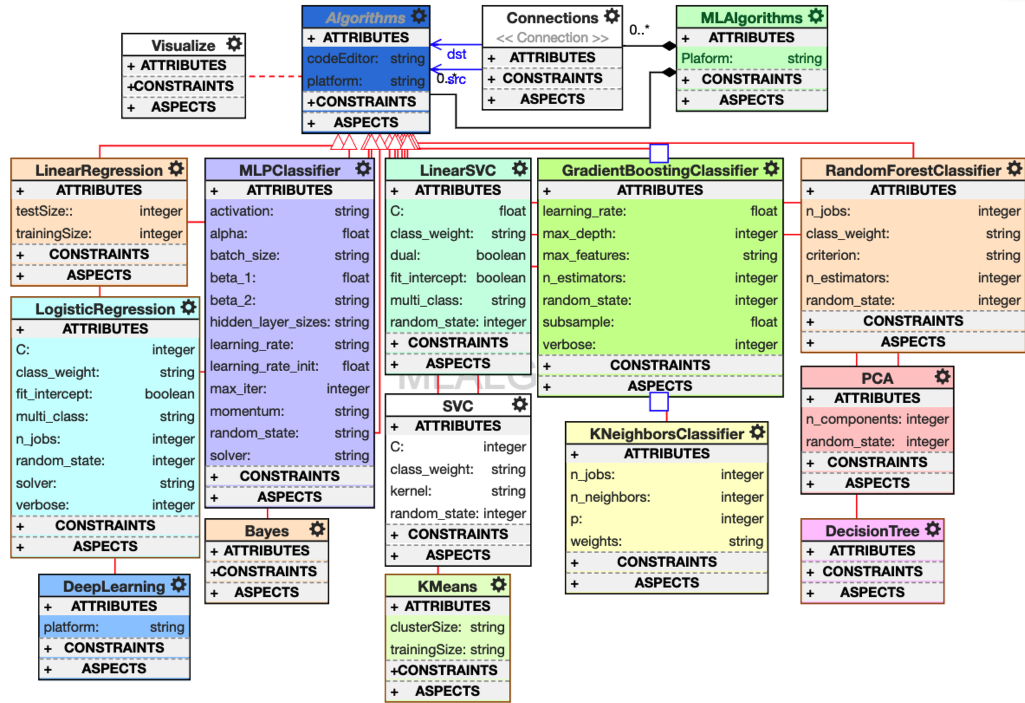


Figure II.5: Meta-model for Machine Learning Algorithms

tion, and hyper-parameter tuning capabilities, which forms the basis for continuous evaluation. We also integrated Jupyter Notebook and Apache Zeppelin (notebook-based environment) to provide data-scientists the ability to train their models interactively. We fill up the iPython notebook with basic code blocks as per the user-defined model, and if the expert data-scientists want to play more with the ML model in the notebook, they can easily do it as shown in Figure II.6. The data-scientist can also directly generate python code if needed.

The Deployment Platform can be distributed systems or standalone hardware, and deployment of the machine learning pipeline on the target machine for training is handled by CloudCAMP as described in Section II.4.1.

II.4.2.2 Meta-model for Machine Learning Algorithms

The meta-model for the supported ML algorithms is shown in Figure II.5. We captured the specifications for a diverse set of ML algorithms including classification (e.g., logistic regression, naive Bayes), regression, decision trees, random forests, and gradient-boosted

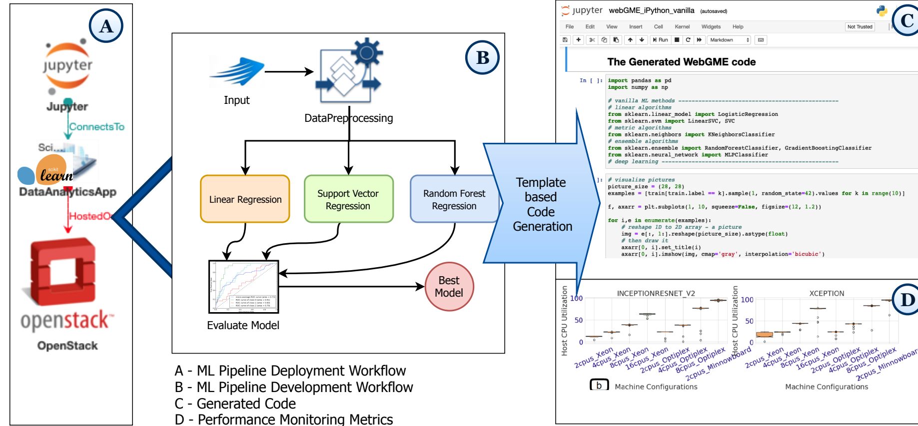


Figure II.6: Sample generative capabilities of Erudite

trees, recommendation (ALS), clustering (K-means, GMMs), and many others.

Using this meta-model, the data-scientist needs to drag relevant machine learning blocks and needs to define all the parameters such as fit_intercept, normalize, n_jobs for scikit-learn linear regression block or have to specify the type of layers such as dense, CNN, RNN for deep learning.

II.4.2.3 Model Evaluation and Flexible ML Service Encapsulation

Users express pipelines as a directed acyclic graph (DAG) where each node represents a task such as data pre-processing, training based on different ML techniques, and deployment to evaluate the model. To allow for cyclic hyperparameter tuning based on customizable optimization method, the optimization methods include Grid Search, Bayesian optimization, Gradient-based Optimization, Reinforcement Learning based optimization, etc. These hyperparameter optimization methods [44] are an advanced mechanism to evaluate the model. Hence we have a placeholder where the data-scientist needs to write their logic for their purpose.

Figure II.6 shows a sample pipeline prototypes for building various regression analysis model for a specific use case using the Erudite meta-model and DSML. The DAG structure of creating an ML pipeline is flexible, and other ML techniques for the specific dataset can easily be plugged with the existing pipeline. We built the WebGME based GUI for the data scientist where they can rapidly prototype their model, and specify the ML model related attributes. Then our DSML parses the user-specific model and generates the code based on the choice of user ML libraries and framework using template-based transformation. All the ML algorithms are containerized, and they can process the data and produce the desired output, such as prediction accuracy, error values. The ML algorithms are encapsulated in Linux containers and exposed using endpoints. The DSML encapsulates the ML algorithms in Linux containers to support the parallel execution of the pipeline based on the availability of the resources. After training the model, we evaluate the model based on different scoring methods such as accuracy, f1 score, precision, r2 score, mean square error. The ML model developer needs to specify the evaluation method of their choice. Based on the evaluation score, we find the model and save it for prediction jobs. The Erudite framework pushes the saved model into the business application workflow.

Discussions. Our Erudite model, is capable of generating only Python-based code as of now, and only scikit-learn and tensorflow are integrated with it for our demonstration. However, other languages such as Java, C++ can easily be plugged into Erudite and other cloud libraries such as Amazon SageMaker, AzureML, can be integrated with Erudite very easily. The design of Erudite uses agile methodologies so that it can be extended with ease.

II.4.3 Addressing Requirement 3: Framework for Performance Monitoring and Intelligent Resource Management

After configuring the runtime framework using the domain-specific modeling language, the resource management logic such as scaling, load-balancing can be integrated with the Erudite framework with ease. We show how Erudite provides these capabilities.

II.4.3.1 Performance Monitoring

To understand the runtime performance of the infrastructure and the application, it is critical to monitor the status of these systems. Considering the distributed nature of the systems spanning from cloud to edge computing environments, monitoring of such systems can become challenging. Also, given the heterogeneity in the computing platforms, there is a need for a monitoring tool that can integrate different architecture-specific monitoring tools in a unified fashion. To address these challenges, we built our monitoring infrastructure leveraging *CollectD* [45] monitoring daemon and different metric specific tools such as *Linux Perf*, *nvidia-smi* [46] to monitor various system metrics. We leveraged *rabbitmq* [47] based publish/subscribe system to provide data dissemination in the distributed infrastructure. The collected data is stored in InfluxDB [48] which is a time-series database utilized for analysis and triggering resource resource management decisions. The system metric data, which is collected includes GPU specific metrics such as power consumption, GPU utilization, temperature and host-specific metrics such as CPU, disk, network, low level cache utilization, memory bandwidth utilization.

II.4.3.2 Resource Management

Erudite contains a *Resource Manager* to maintain the QoS of the application components by scaling and migrating the application components. The ML-based data analytics applications' total latency comprises the round trip latency(l_{rt}) of data and the ML model execution time. We profile the ML model execution time on various data and target hardware before actually deploying the model, and consider it's 95th percentile execution latency as estimated execution time $(exec_{hw_id,mlmodel})$. In the cloud, fog, and edge analytics spectrum, the ML prediction model can be deployed for various purposes such as image recognition, speech recognition.

ML Prediction Task Migration. Before considering edge devices as a potential node for executing predictive analytics, we check if it has sufficient memory to keep the model in

memory. If the edge node can host the ML model, we profile the ML model on the edge devices. We also profile 95th percentile network latency(l_{rt}) between edge and cloud node. We consider the migration of the ML model in the edge when the below condition is true.

$$exec_{edge,mlmodel} < exec_{cloud,mlmodel} + l_{rt} \tag{II.1}$$

Discussion: The transfer of the ML model happens asynchronously, so the transfer cost is amortized. We also consider that the rate of incoming data is less than the execution time in the edge device so that we do not have to build a queue for data at the edge. If the incoming data rate is high, we have to consider a scenario for the cluster of edge devices and have to build some queuing mechanism.

Auto-scaling of Application Components. Let χ denote the constraint specified by the Service Level Objective (SLO) of the ML model execution latency, and let $exec_{hw_id,mlmodel,p}$ denote 95th percentile execution latency on p CPU cores. For each server configuration, we can compute the number of requests n_req it can serve for a prediction service while meeting the SLOs using p CPU cores.

$$n_{req} = \chi / exec_{hw_id,mlmodel,p} \tag{II.2}$$

By monitoring the total number of incoming requests, we can easily calculate the total number of machines ($total_incomingrequests/n_{req}$) require to handle the workload. Based on the difference between the current state and desired state, we can calculate how many more machines to start. Erudite *Resource Manager* can deploy the ML model on the required machines and start the prediction service automatically to handle dynamic workload reactively. During the scaling down phase, we monitor the number of requests for a certain period (e.g., 30 minutes), and if we have more machines than required based on the calculation, we decide to scale down by turning off the machines one by one. Moreover, a batching mechanism is considered to send bulk data at an interval or based on the num-

ber of messages, using data ingestion tools like Apache Kafka. Similarly, to handle the incoming message rate, we can autoscale the data ingestion tools also.

Discussions: We consider that the resources are homogeneous in the cloud, and during scaling, the configuration of the resources does not change. The configuration of the resource is fixed during deployment time based on the user's choice and requirement-capacity analysis. Moreover, the management of resources can be done proactively also; however, that problem is out of the scope of this chapter.

II.4.4 Support for Collaboration and Versioning

Erudite core concepts are developed using WebGME, which supports collaboration in a version-controlled environment. We also incorporated the automatic version control in the Erudite framework so that we can recall a specific version of the framework later if required. We save data, code, and attributes of the modeling environment to guarantee that every state is reproducible. Because of collaborative editing support, the developers can easily tag branches when they are developing their part of the model, and the branches can be merged easily to integrate the whole model. Figure II.7 shows an integrated version control for all developers to make any historical state reproducible in the collaborative environment.

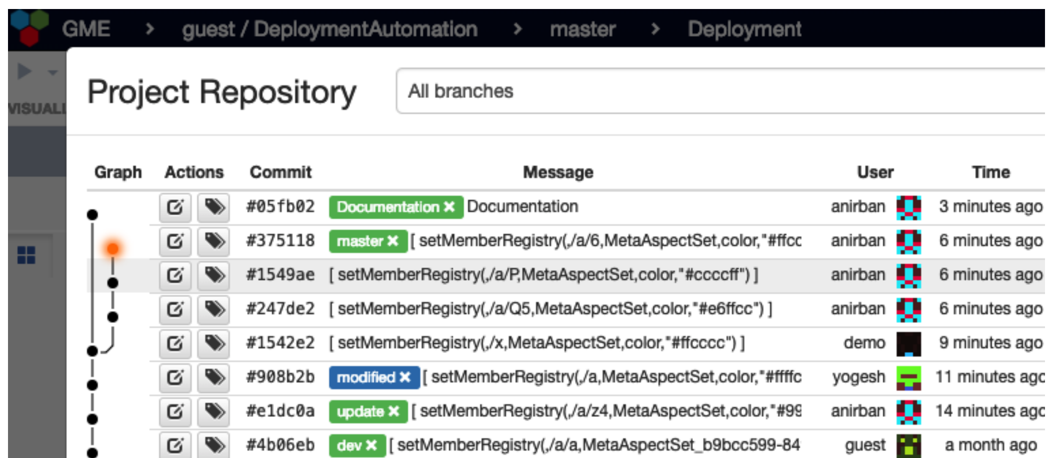


Figure II.7: Integrated version control to reproduce all historical states

II.5 Evaluation

In this section, we evaluate the simplicity, rapid deployment, and resource management capabilities of Erudite, along with the accessibility, scalability, and efficiency of the Erudite ML model development framework of Erudite.

II.5.1 Evaluating the Rapid Model Development Framework

The Erudite Rapid Model Development framework leverages the strength of Model-Driven Engineering(MDE) to provide a visual development environment for machine learning pipelines. It provides a hybrid visual-textual interface for various phases of ML model development in a version-controlled, collaborative visual environment. The Erudite model transformer can distribute different jobs with different ML techniques over a cluster of connected machines and aids the developer to select the best model or ensemble of models based on the user's choice of evaluation methods.

As shown in Figure II.8, the ML developer can build their machine learning pipeline using the visual interface of Erudite. In the left-hand pane (box1), all the building blocks are defined using the metamodel. The ML model developer has to drag and drop the required blocks in the design pane (box 2) and must connect the blocks to define the ML pipeline, as shown in Figure II.4. All the attributes of the selected ML algorithms, such as max_depth, criteria need to be specified by the user (or can take default values) from the right pane (box 3). The name of the attributes are dependent on ML algorithms, and this aspect is captured by reverse engineering. The ML execution framework needs to be mentioned to bind the workflow with a specific library or framework such as Scikit-learn or Tensorflow.

All the ML algorithms are encapsulated in Docker containers, and different algorithms can be executed in parallel to speed up the training and tuning phases. Similarly, in the Input building block, the source data type, and path, e.g., database, HDFS needs to be mentioned, and also data source type, e.g., csv, Avro, text is required. In the data preprocessing block,

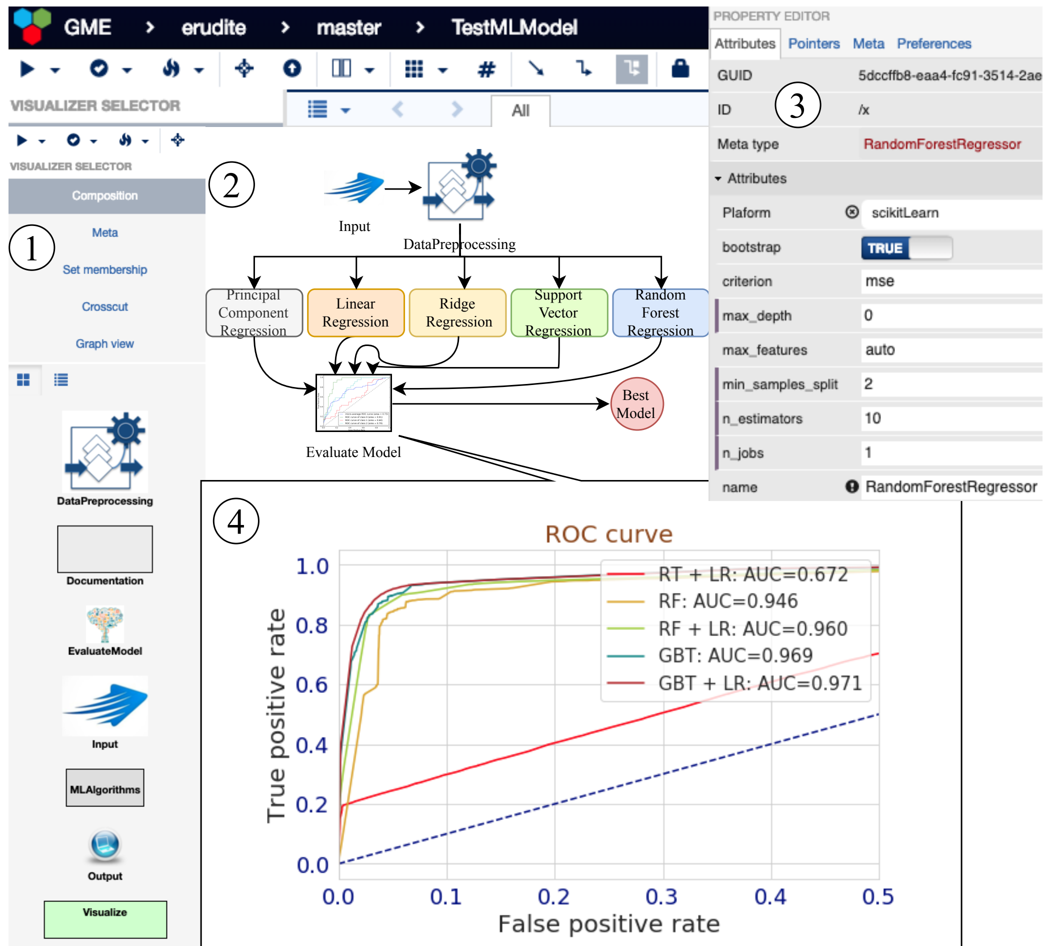


Figure II.8: Usability of the Erudite Framework. Box 1 shows the available selection of metamodel elements available to create an ML pipeline as shown in Box 2. Individual metamodel element's attributes can be set using the attribute selection panel in Box 3. Box 4 shows model evaluation.

we only support simple data cleaning methods, such as filtering and normalization. In the evaluation building block, the method of evaluation needs to be specified, and based on that, Erudite selects the best model. A sample ROC curve is shown in box 4, which is showing that the ensemble of two ML methods has the highest accuracy in the training phase on a sample dataset, and it should be selected as the best model. Thus, Erudite helps to build the ML model using MDE techniques, and the ML developer does not need to write any code. The framework can also generate the code in the notebook environment as depicted in Figure II.6 for the expert user, where they can tune the ML model as required. We save the model in the ML framework-specific format.

II.5.2 Evaluation of Rapid Application Prototyping Framework

As depicted in Figure II.9, using the visual interface of Erudite, the application developer can develop the data analytics application. As described in our previous work [8, 7], the building blocks for application components and infrastructure components need to be dragged from the left panel in the design space. Then all the building blocks need to be connected to build the business application workflow. The 'hostedOn' connection illustrates on which target machine the application components are deployed, and 'connectsTo' connection represents that the source component needs to be started before destination components because the destination component is dependent on the source component. As shown in Figure II.3, by parsing the user-defined abstract model tree, the Erudite DSML creates the deployable model (Ansible-specific in our case) and using NodeJS based plugin it executes the deployable model and creates the infrastructure of the application as described in Section. II.4.1.

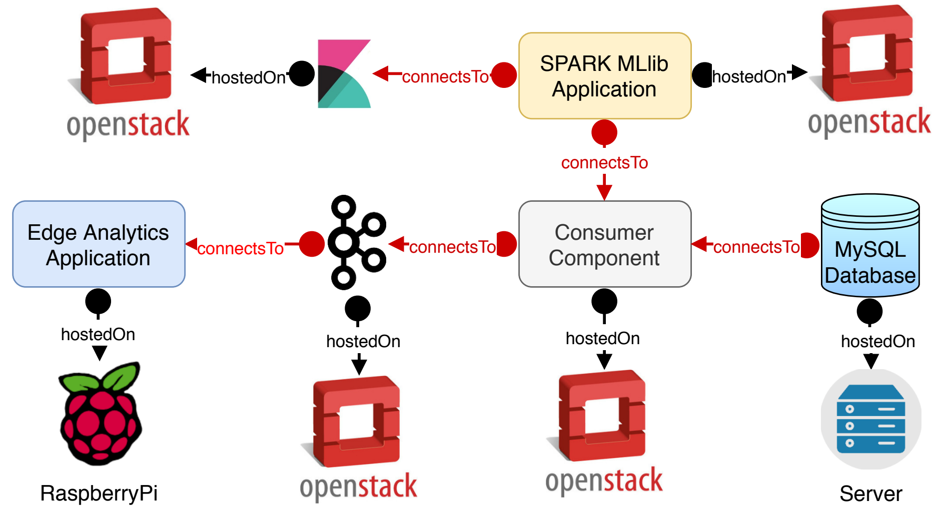


Figure II.9: Example of Data Analytics Application Deployment Model

Figure II.9 illustrates that using the Erudite modeling environment, edge analytics application component can be deployed on Raspberry Pi machine, and data ingestion tool, e.g., Kafka can be deployed on cloud VMs, which is maintained by OpenStack. The data consumer application component can be similarly deployed on OpenStack VM, which will

consume the data from Kafka in a batch or stream and store it in MySQL database, which is deployed on top of the bare-metal server. Then, Spark with the MLlib library can be deployed and configured on OpenStack VMs, and a visualization engine like Kibana can be integrated with workflow. RESTful APIs connect all the application components, so the ML model can easily be pushed into a predictive analytics application during the management phase.

To deploy the complex workflow such as this, Erudite provides easy to use rapid deployment environment and using the in-built DSML of CloudCAMP (see Chapter III)the entire workflow can be deployed without writing a single line of code.

II.5.3 Performance Monitoring on Heterogeneous Hardware

As described in Section II.4.3.1, we need to monitor the performance of the infrastructure as well as the application components to take dynamic resource management decisions. To describe the monitoring capabilities of Erudite, we set up the training experiments on NVIDIA GeForce Titan X Pascal GPU machine integrated with Intel(R) Xeon(R) CPU E5-2620 v4. For prediction experiments, we set up a cluster of Intel(R) Xeon(R) CPU E5-2620 v4 machines in the private cloud, Dell OptiPlex 3020 machines in the fog nodes, and MinnowBoard with 64-bit Intel Atom devices as edge devices.

We developed a deep learning model for image classification using CIFAR10 datasetWe monitor the accuracy and loss of the custom-developed ML model, as shown in Figure II.10. Figure II.11 show the GPU performance metric, such as GPU utilization, GPU memory utilization per core, the power drawn by GPU cores in watt, and the temperature of the GPU machine in Celsius, during the training phase of the ML model.

We encapsulate the trained models in Docker containers and build the container for the target hardware. We monitor the performance of the Docker containers along with host machines on which the ML models reside. We collect various metrics of the container from the host, and it includes execution time, CPU, memory, network, disk utilization along with

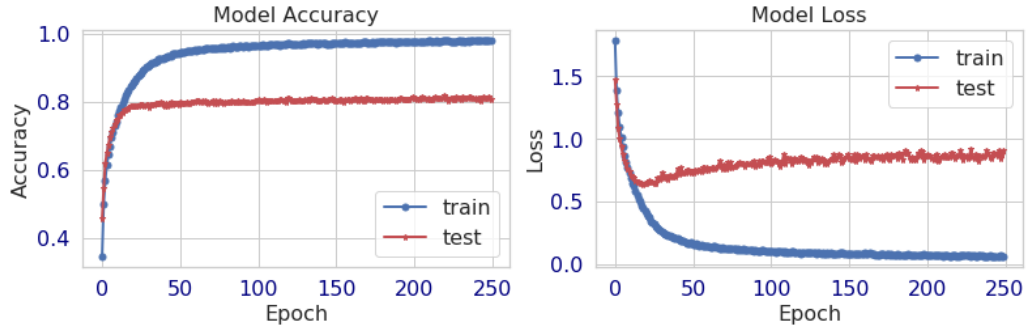


Figure II.10: ML Model Accuracy and Loss trend graph on CIFAR10 dataset (Test and Train)

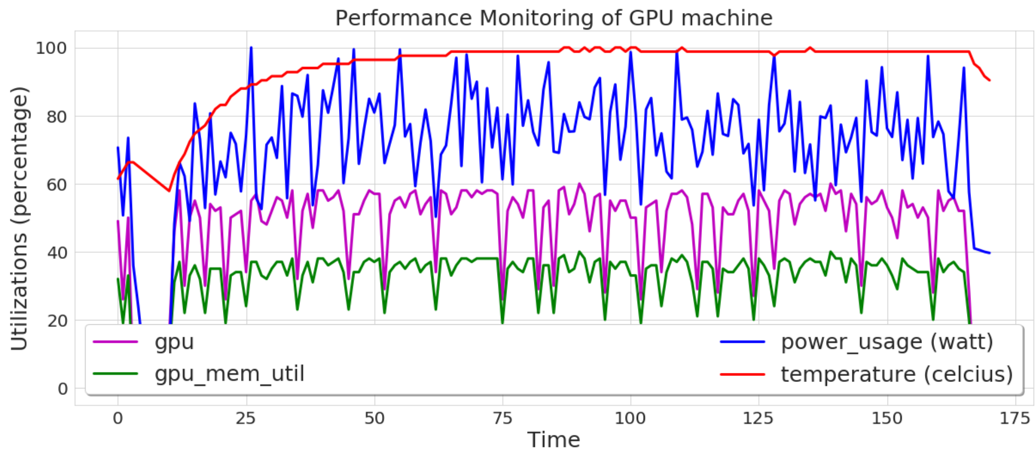


Figure II.11: GPU Performance Metrics for Sample Deep Learning Training

L2, L3 cache bandwidth, cache miss ratio, and many more.

Figure II.12 illustrates a glimpse of collected performance metrics during the prediction serving phase. Figure II.12(a) shows the execution latency of InceptionResnetV2 and Xception model on different ML containers with variable configurations, which is hosted on different bare-metal machines. Figure II.12(b) shows CPU utilization of the ML containers from the host machines(such as two cores container only use around 12% of CPU resources of a sixteen cores machine, while two cores container use around 100% of CPU resources of a two cores machine), and Figure II.12(c) shows memory utilization of ML containers (in MB) from the host machines.

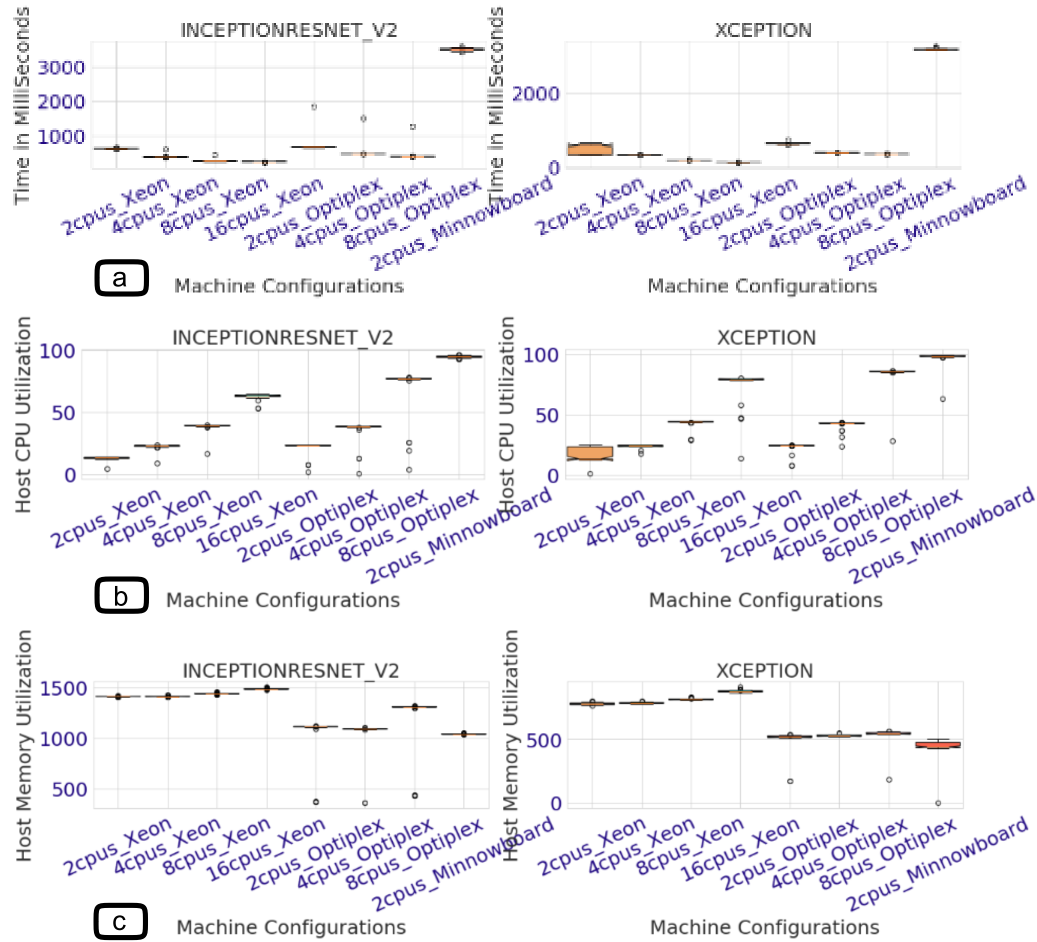
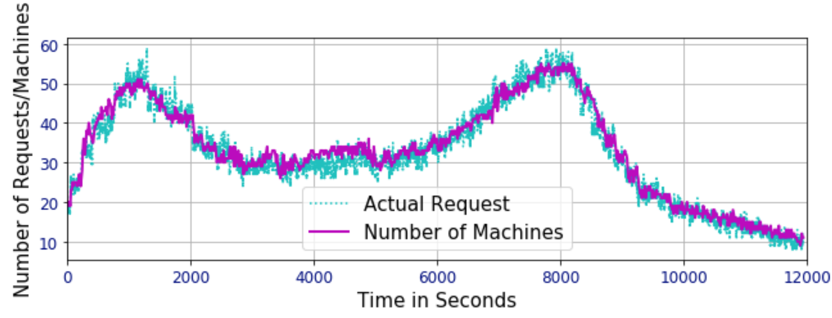


Figure II.12: Performance Monitoring of the prediction services (a)The execution latency of InceptionResnetV2 and Xception model on different ML containers with variable configurations, (b) Host CPU utilization of the ML containers (c) Host Memory utilization of ML containers (in MB)

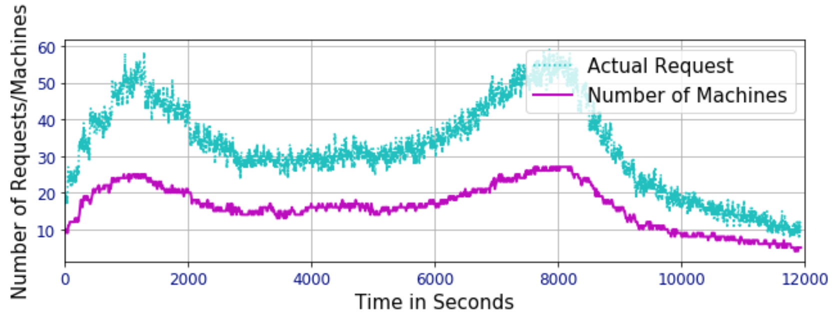
II.5.4 Resource Management

As mentioned Section. II.4.3.2, we profile the prediction service on the specific hardware before deploying it on the cluster of machines. The prediction service is encapsulated in a docker container, and based on the number of incoming requests (dynamic workload), we scale our system in an event-driven manner. Using the Docker swarm cluster management tool, we can easily scale up and down the number of ML model containers to guarantee the pre-defined QoS.

As shown in Figure II.13, we profiled InceptionResnetV2 image classification algorithm on four cores and eight cores virtual machines(VM), which is running on a cluster



(a) Number of 8 Cores machines are required to maintain 800 ms execution latency of inceptionresnetv2 model on dynamic workload



(b) Number of 4 Cores machines are required to maintain 2 seconds execution latency of inceptionresnetv2 model on dynamic workload

Figure II.13: Varying number of machines (each host a ML container) to guarantee QoS on dynamic workload

of Intel(R) Xeon(R) CPU E5-2640 v4 machines. The classification algorithms are docker container, which is pinned on all available cores of the VM. Figure II.13(a) depicts that to guarantee 800 ms execution latency on eight cores machines, how we change the number of machines to handle the dynamic workload. Similarly, Figure II.13(b) shows that to guarantee 2 seconds execution latency on four core machines, how we change the number of machines. This approach remains the same for any parallelizable prediction services, as shown in our recent work [11]. For the non-parallelizable ML algorithms, we currently run each container on a single core and spawn the required containers every time. In the future, we will look into adaptive batching strategies to handle the problem.

Moreover, to migrate the machine learning application component as described in Section. II.4.3.2, we monitor the round trip latency to send the data from the edge to cloud, and also the execution latency of the ML model on cloud and edge hardware as shown in

II.12(a). If the total execution time (round trip latency + execution latency) is more in the cloud than the edge, we migrate the saved ML model to edge. We do not provide container migration from cloud to edge, because the same Docker container cannot run on edge because of hardware mismatch. We do the requirement-capacity analysis on edge, and the edge device is feasible to deploy the ML model by installing all the dependent software packages on edge to serve the ML model.

II.6 Conclusion

II.6.1 Summary

As the realm of IoT-based analytics becomes increasingly sophisticated, developers are finding themselves lacking expertise in a wide range of skills while at the same time are overwhelmed by the plethora of frameworks, libraries, protocols, programming languages, and hardware available to design and deploy these analytics applications. To address these highly practical challenges, this chapter presents a novel holistic framework that systematically integrates a number of underlying platforms and technologies, hiding most of the details of these underlying artifacts while providing the user with higher-level, intuitive abstractions based on model-driven engineering (MDE) and domain-specific modeling. The MDE capabilities are offered in our Erudite framework as part of a serverless offering so that the user-supplied specifications are used to automate the application lifecycle management. Erudite focuses on three key dimensions: automating the deployment of the application, simplifying the machine learning model building process, and ensuring that the models get their desired quality of service properties when the models are used in the inference phase. The Erudite framework capabilities are available for download from https://github.com/doc-vu/EruditeMLplatform.git.

CHAPTER III

CLOUDCAMP: A MODEL-DRIVEN APPROACH TO AUTOMATE CLOUD SERVICES DEPLOYMENT AND MANAGEMENT

III.1 Introduction

Self-service application deployment and management in a fault-free manner is desired for enterprises to speed up time-to-market for their cloud services. Enterprises often suffer from service outages and delays that stem predominantly from the use of tedious and error-prone manual efforts that are expended in the service configuration, integration, and infrastructure provisioning across the heterogeneous platforms. Modern cloud services are architected as microservices, and each of the components must be configured and deployed on cloud platforms – sometimes federated – in a specific order. The capabilities of the entire service are realized through a collection of distributed, loosely coupled service components [49, 6]. Script-centric efforts to deploy and manage these complex scenarios, degrade productivity, and adversely impact the product time-to-market.

III.1.1 Motivation

Consider the case of a LAMP [Linux, Apache, MySQL, and PHP] -based service deployment on a cloud platform. Figure III.1 shows the desired cloud application topology consisting of two connected software stacks, i.e., a PHP-based web front-end and a MySQL database backend. The front-end WebApplication stack holds the business logic, and it will be deployed on Ubuntu 16.04 server virtual machine (VM), which is managed using the OpenStack cloud platform. The backend DBApplication stack holds the relational database, which is used to store and query the product data. The backend database is a MySQL DBMS, which will be deployed on the Amazon Elastic Compute Cloud (EC2) VM instance with an Ubuntu 14.04 server.

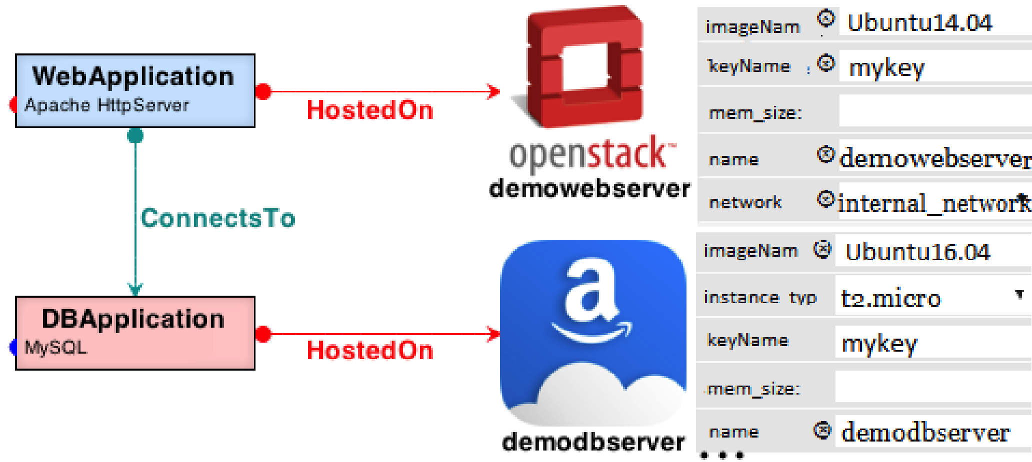


Figure III.1: Desired Level of Abstraction for a WebApp Business Model

III.1.2 Requirements and State-of-the-art Solutions

Based on this use case, we elicit the following requirements that drive the solution presented in this chapter.

III.1.2.1 Requirement 1: Reduction in specification details needed for deployment

As depicted in Figure III.2, the deployer needs to provision the PHP and MySQL-based e-commerce application stack from two aspects. In the cloud infrastructure provisioning aspects, the application topology needs to be woven into the execution environment, which can be virtual machines (VM), containers, or third party services. To provision the cloud infrastructure, the deployer needs to select a proper image for their VM, along with the security group, roles, network, number of instances, the storage unit in the target cloud providers' platform. In the service provisioning aspects, all the dependent software needs to be installed, and all the constraints need to be configured. For example, for the front-end of our motivating example, Apache Httpd needs to be installed and configured along with PHP and Java. Similarly, in the backend, MySQL needs to be installed and configured. Moreover, the database service should start before the PHP application service so as to run the WebApp properly. The IAC solution for the application provisioning requires all the

installation and configuration details to execute the deployment plan.

This scenario shows that a user must possess extensive domain knowledge to provision even a simple web application correctly, and we aim to abstract these detailed specifications from the users.

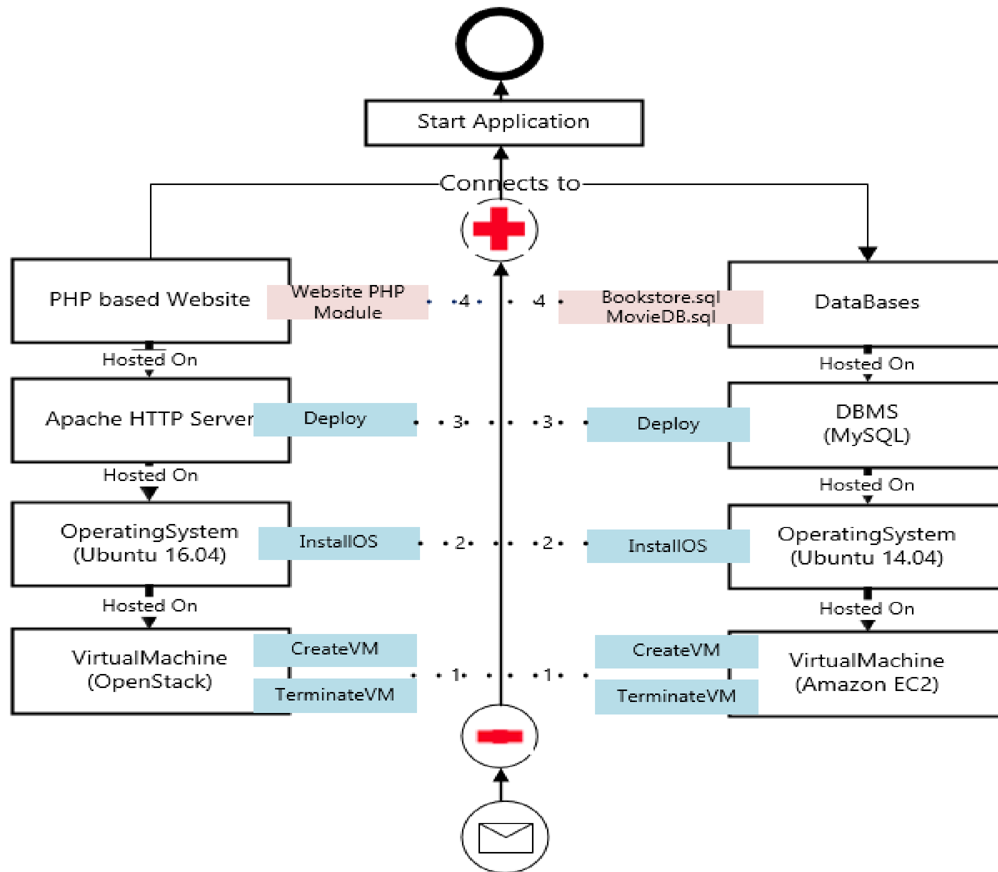


Figure III.2: A TOSCA-compliant PHP- and MySQL-based Application Deployment Workflow

III.1.2.2 Requirement 2: Auto-completion of Infrastructure Provisioning

Writing the complex low-level scripts to provision the infrastructure for the motivating scenario is time-consuming. To improve productivity by significantly alleviating such efforts requires a self-service framework, which should be capable of transforming the abstracted business models to complete, deployable TOSCA-compliant[1] Infrastructure-as-

[1]TOSCA [50] is an OASIS standard for vendor-neutral topology and orchestration specification for cloud-based applications.

Code (IAC) solution [51].

III.1.2.3 Requirement 3: Support for Continuous Integration, Migration, and Delivery

Suppose that for the use case of Figure III.2, the enterprise wants to execute a management task to migrate the web front-end to Amazon's EC2 with the purpose of reducing the number of cloud providers used by their services. To migrate the front-end, the user must perform the following steps: (1) shut down the old virtual machine on OpenStack, (2) create a new virtual machine on Amazon EC2, (3) install the Apache HTTP server and the other dependencies, (4) deploy the PHP-based front-end, and so on.

This migration activity gives rise to several issues, such as having to deal with missing database drivers and missing configurations of the target database service. Manually performing the migration tasks often require sheer technical expertise about the different cloud APIs and its underlying technologies. Application extensibility (such as adding one database server node or data analysis toolkits with the existing application) will also incur additional challenges.

All of these challenges motivate the need for a fully automated platform that can generate robust deployment plans. Nevertheless, the challenge here also lies in capturing the application and cloud specifications in the metamodel and the DSML. However, apart from that, there are a few more problems, which are listed below:

III.1.2.3.1 Extensibility and Reusability of the Application Components

New application components need to be added at runtime to the existing application by leveraging the platform. The specifications captured in the metamodel should be modularized and loosely coupled with a particular application. DSMLs should do all the binding after querying for the specification for a particular application type in the knowledge base, and then the DSML will generate a concrete cloud-specific, operating-system specific infrastructure-as-a-code solution. The IAC is idempotent, so it will not change the existing

deployment if configured correctly. The correctness of the added application components can be validated using a constraint checker at the model level.

III.1.2.3.2 Extensibility of the Platform

The platform can transform the business-relevant model to actionable infrastructure-as-a-code, which produces application deployments in the cloud. However, the challenge is to make the platform loosely coupled with any DevOps or orchestrating tool, so that later different tools can be added if required. Moreover, adding new application requires reverse-engineering the application components, and capturing the application specifications in the metamodel of our platform, and adding new cloud providers also requires a similar approach. Defining commonality and variability points is critical to building a modularized platform so that the extensibility of the platform will be relatively easy.

In our proof-of-concept solution, we only generate the Ansible specific code from the business model, and our WebGME metamodel handles the TOSCA specification.

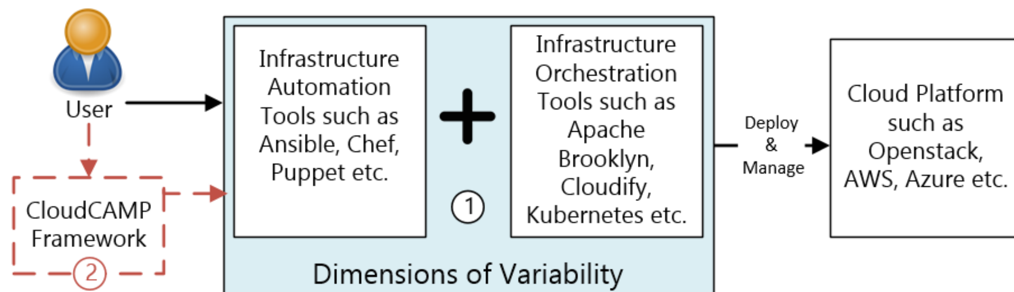


Figure III.3: Box 1 depicts the responsibilities of service deployment team, which is to define the low-level scripts so that existing automation tools can configure the application components and orchestration tools can provision the infrastructure for application components and execute them on heterogeneous cloud environments. Box 2 depicts the contributions of this chapter which introduces a self-service framework and automates whole infrastructure design solutions for these tools.

Self-service application provisioning requires extensive planning for their smooth operations. In the context of cloud-based service hosting, service provisioning includes two key steps: (a) orchestration, where the deployment and ordering of individual components of the service must be managed across distributed resources of a cloud platform or federated cloud platforms, and (b) service automation, where defining and executing individual resource-specific configurations, such as a virtual machine or container configurations, and

deployment of service components on these resources are automated. *Infrastructure as Code (IAC)* is a term used by the DevOps community in which the cloud infrastructure is viewed as software for which code is developed to automate the entire cloud-based service provisioning. To that end, the DevOps community today leverages orchestration solutions such as Cloudify, Apache Brooklyn, and Kubernetes, among others, in conjunction with automation tools such as Ansible, Puppet, and Chef, among others. This state of the art (i.e., the extensive choices available to the developer) is reflected in Figure III.3.

The choice of services provided by different cloud providers needs to be selected and configured by the deployer. It requires elaborate specifications of service topologies comprising requirements, functionalities, dependencies , and relationships of the components. For instance, depending on the technology used, e.g., MySQL versus PostgreSQL or PHP versus Node.js, the script must include the appropriate drivers. Also, architecting the solution for different cloud providers is different. For instance, creating data flow architecture using AWS Kinesis and DynamoDB is much different from creating the same architecture using Azure Event Hubs and CosmosDB. Additional dimensions of variability (i.e., addressing application's compatibility and cloud providers' incompatible APIs) as depicted in Box 1 of Figure III.3 complicates the manual effort, which is already daunting, tedious, and error-prone. Finally, existing approaches do not account for pre-deployment validation to check if the end-user requirements and software dependencies are met.

III.1.3 Overview of Technical Contributions

Our motivating example shows that a user must possess extensive domain knowledge to provision even a simple web application stack correctly. Users need to write Infrastructure-as-Code (IAC) solutions via low-level scripting. Instead, the desired capability would require a self-service platform, in which (1) a deployer specify only the application components, such as a Web App, and (2) the framework automatically transforms the business model into deployable artifacts. To achieve the goal, we propose a model-driven and scal-

able, rapid provisioning framework called CloudCAMP. It complies with TOSCA (Topology and Orchestration Specification for Cloud Applications) specification, which enables the creation of portable and interoperable *plans-as-a-service* template for cloud services. TOSCA provides the standardization for decoupling software applications and its dependencies from the cloud platform specifications.

The key contributions in this chapter include:

① We present key elements of CloudCAMP's domain-specific modeling language (DSML) that masks low-level details of the application component specifications and cloud provider specifications and instead offers intuitive high-level representations;

② We present the use of an extensible knowledge base and algorithms to perform Model-to-Infrastructure-as-code (IAC) transformations automatically; and

③ We present a concrete realization of CloudCAMP and validation in the context of real-world use cases.

III.1.4 Organization of the Chapter

The rest of the chapter is organized as follows: Section III.2 presents a brief survey of existing literature and compares to our solution; Section III.3 presents the design of CloudCAMP; Section III.4 evaluates our metamodel for a prototypical case study and presents a user survey; and finally, Section III.5 concludes the chapter alluding to future directions.

III.2 Related Work

The problem of deployment and management abstraction has been explored in the area of cloud automation and orchestration. In this section, we compare existing efforts in the literature with our work. The use of these toolchains adds the burden of configuring the application components and integrating pre-deployment verification on application developers. The script-centric DevOps community provides toolchains for elimi-

nating the disconnect between developers and operations providers [52], and these tools incur limitations in providing a self-service provisioning platform. Cloud orchestration tools like Apache Scalr (https://scalr-wiki.atlassian.net/wiki/display/docs/Apache), Cloud-Foundry (https://www.cloudfoundry.org/), Cloudify (http://getcloudify.org/) etc. are excellent toolchains to deploy and manage applications on any cloud providers. They provide techniques to monitor the health of the application and to migrate between the cloud providers using standardized approaches. However, they all suffer from the limitations of requiring the users to define the complete and correct deployable model with all the functionalities and features. In this context, Alien4Cloud [53] proposes a visual way to generate TOSCA topology model, which can be orchestrated by Apache Brooklyn. However, building the proper topology, even using an MDE approach combined with the TOSCA specification needs domain expertise. Unlike these approaches, CloudCAMP abstracts all the application and cloud-specific details in the metamodel of its DSML and transforms the business model to TOSCA-compliant IAC.

Several patterns-based approaches are proposed to reduce the complexity of service deployment [54, 55]. They differentiate between business logic and the deployment of a service-oriented architecture platform. Each pattern offers a set of capabilities and characteristics. Likewise, model-based patterns of proven solutions are used for service deployment in cloud infrastructures [56, 57]. For instance, MODAClouds [58] allows users to design, develop, and re-design application components to operate and manage in multi-cloud environments using a Decision Support System. In Computation Independent Model, the design artifacts are semi-automatically translated to the Cloud-Provider Independent Model level, where an entirely deployable abstract cloud model is generated by matching the application patterns. The abstract deployment model is concretized to Cloud-Provider Specific Model (CPSM) by a domain-specific language. Similar to CloudCAMP, they also support reuse and role-based iterative refinement in a component-based approach. However, their deployment plan generation lacks verification and extensibility. They also did

not consider distributing application components in a heterogeneous cloud environment.

Several efforts come close to the CloudCAMP idea. For instance, ConfigAssure [59] is a requirement solver to synthesize infrastructure configuration in a declarative fashion. All the requirements are expressed as constraints by the developer, and the provider predefines a configuration database containing variables as a deployment model. Kodkod [60] is a relational model finder that takes these arguments as a first-order logic constraint in the finite domain. Engage [61] deploys and manages the application from a partial specification using a constraint-based algorithm. Aeolus Blender [36] comprises the configuration optimizer Zephyrus [62], the ad-hoc planner Metis [63], and deployment engine Arnomic. Zephyrus automatically generates an abstract configuration of the desired system based on a partial description. They guarantee meeting all the end-user requirements for software dependencies and provide an optimal solution for a given number of active virtual machines. In contrast to the use of the knowledge base in CloudCAMP, these efforts use a CSP solver to transform the business model. CSP solvers, however, can take significant time to execute. Moreover, defining constraints on the configurations requires domain expertise, which is not needed in CloudCAMP.

Similar to CloudCAMP, Hirmer et al. [64] focus on producing complete TOSCA-compliant topology from users' partial business-relevant topology. Users have to specify the requirements directly using definitions of the corresponding node types or are added manually for refinement. Their completion engine compares user specifications with target models and combines the missing components to make it a fully deployable model, and then the service components can be executed in the right order using an OpenTOSCA toolchain [65]. CELAR [66] combines MDE and TOSCA specification to automate deployment cloud applications, where topology completion is fulfilled by requirement and capability analysis on node template. Unlike these efforts, the model transformation in CloudCAMP is based on querying the knowledge base and idempotent infrastructure code generation.

III.3 Design and Implementation of CloudCAMP

This section delves into the design details of CloudCAMP (Figure III.4) and shows how it meets the requirements discussed in Section III.1.2.

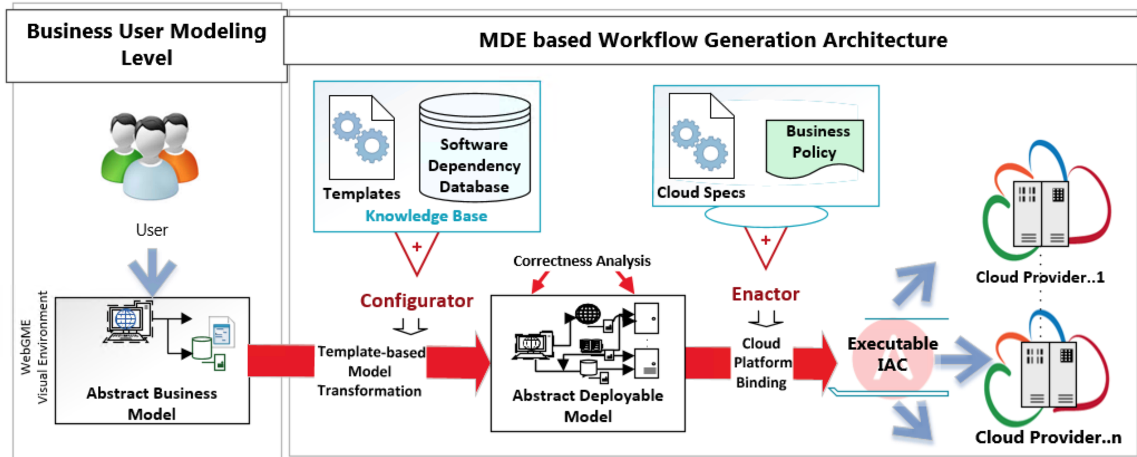


Figure III.4: The CloudCAMP Workflow

III.3.1 System Architecture of CloudCAMP

To better appreciate the CloudCAMP solution presented below, consider the fundamental requirements outlined earlier and depicted in Figure III.4. As per our framework design, 1) A deployer needs to specify only the application components, such as a Web App, using intuitive notations provided by the framework, 2) The DSML transforms the business model into deployable artifacts. Thus, the first step is for the user to utilize an intuitive, higher-level modeling framework that simplifies the modeling of business logic and automatically takes care of non-business centric deployment and management artifacts.

To that end, we have architected CloudCAMP's cloud-based service provisioning workflow, as depicted in Figure III.4. Below we explain the roles of the different actors involved [8]:

① **Business User Modeling**: A business application is modeled as a compendium of different application components. The user has to select appropriate application com-

ponent types from the CloudCAMP application pane to deploy the associated application code. The user needs to specify the variability points for application components' deployment, as depicted in Figure III.1. The design details of CloudCAMP DSML is described in Section III.3.3.

② **Configurator**: This actor is responsible for transforming each abstract description of an application component to a deployable cloud automation task (e.g., Ansible-specific) for each application component. Configurator realizes a user-defined abstract description of a cloud application model, and then maps the application components with the operating system, and query the knowledge base to find the software dependency tree; it generates full 'correct-by-construction' Ansible (https://www.ansible.com/) specific code from the application type template. The details of template-based transformation and code generation are described in Section III.3.5.

③ **Enactor**: It generates the infrastructure design workflow of IAC solutions by integrating the generated automation code with the business rules and cloud infrastructure specifications. The users define the connection types between the application components. There are four types of connections: 'hostedOn', 'connectsTo', 'deleteFrom', 'migrateTo'. The details of the connection types and their role is described in Section III.3.3.4. The orchestration tool executes all the automation tasks based on the connection types to deploy and run the business application components in proper order.

④ **Knowledge Base:** A knowledge base is needed for auto-completing the partially specified deployment models. We pre-define the software dependencies for application type in a relational table with a key-value pair. All the software packages needed for a particular application component are defined in the tables, along with their dependency on the operating system and its version. The application developer needs to populate the tables with all software dependencies for including the new application

component type in the CloudCAMP. The design details of the knowledge base are described in Section III.3.4.

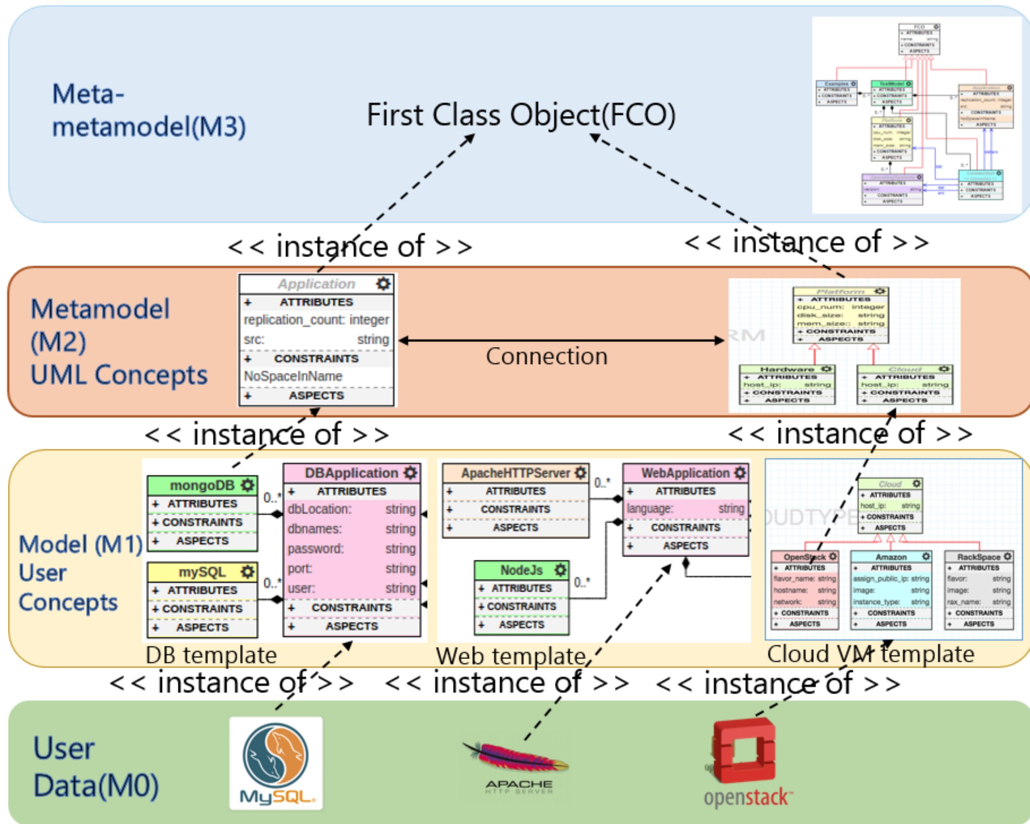


Figure III.5: A Partial Meta-Object Facility (MOF) model of CloudCAMP DSML and Platform

III.3.2 System Implementation of CloudCAMP

The CloudCAMP DSML shown in Figure III.5 is developed using the WebGME MDE framework (www.webgme.org). WebGME is a cloud-based framework that offers an environment for DSML developers to define their language and create model parsers that can serve as generators of code artifacts. The CloudCAMP runtime platform uses a microservices architecture comprising three services: (a) the modeling infrastructure, i.e., the WebGME UI, and orchestration and automation frameworks forming one service, (b) the WebGME modeling details are stored in a MongoDB NoSQL database, and (c) the knowledge base is hosted as a MySQL database service. The microservices are connected

through the API endpoints and placed behind an HAproxy (http://www.haproxy.org/) load balancer. Thus, all the services can independently scale to support parallel spawning and configuration of multiple VMs or containers in the cloud platform.

III.3.3 CloudCAMP Domain-specific Modeling Language (DSML)

The CloudCAMP DSML abstracts the design complexities by separating the application from deployment and infrastructure technologies according to TOSCA specification as described in Requirement III.1.2.1.

III.3.3.1 Design Rationale for CloudCAMP Meta-models

DSMLs are realized through one or more interrelated meta-models that capture the DSML's syntax and semantics. In our case, to transform the business model to a full-blown deployment model, we needed to capture various facets of the application and cloud specifications in our meta-model. CloudCAMP's deployment modeling automation meta-model was developed by harnessing a combination of (1) reverse engineering, (2) dependency mapping across heterogeneous clouds, (3) dependency mapping across different operating systems and their versions, (4) semantic mapping, (5) business policy, and (6) prototyping. Capturing this variability helps to enrich the expressive power, multi-cloud tool support and interoperability of the platform. Prototyping and reverse engineering helped to identify the different application components, cloud and operating system specific endpoints. The dependent software packages, their relationship mapping, and configuration templates were realized in the meta-model by querying the knowledge base. The set of available building blocks, requirements, policies, and other information concerning the implementation of the services and all other known constraints are pre-defined in the high-level application meta-model.

To that end, CloudCAMP provides different node types, which are the application components such as Web Application, Database Application, DataAnalytics Application, and

various cloud providers such as OpenStack, Amazon AWS, Microsoft Azure. The goal is to concretize the abstract application node type by matching the application deployers' desired specification with the pre-defined functionalities captured in the CloudCAMP meta-model and knowledge base. The concrete node templates are then woven to specific cloud provider types and their VMs to create a dependency graph that has to be executed to deploy the application on the desired target machine, as shown in Figure III.4. Using our DSML, the deployer can configure the node in a defined cloud platform or particular target system with ease.

Snippets of the meta-models for CloudCAMP are shown in the M1 and M2 level of Figure III.5, which are based on the Meta-Object Facility (MOF) standard provided by Object Management Group (OMG [http://www.omg.org/]). Using our DSML, the application deployer can configure the node in a defined cloud platform or particular target system without providing any deployment or implementation artifacts that contain code or logic.

The high-level meta-model is depicted in the Figure III.6. It shows the M2 and M3 level of the MOF standard. The First Class Object(FCO) is the root node, and under it, the meta-model for the cloud platform, operating systems, containers, and application components (M2 level) are defined. The connection type is also defined at the M2 level. We will now dive down into the meta-model for Cloud Platforms and Application components.

The CloudCAMP meta-models are extensible and reusable, so new component types and platforms can be added as required in the CloudCAMP meta-model.

III.3.3.2 Meta-model for the Cloud Platforms

In designing the meta-model for cloud platforms, we observed (i.e., reverse engineered) the process of hosting applications across different cloud environments, and captured all the commonalities and variabilities. The specifications for different cloud platforms such as OpenStack, Amazon AWS, Microsoft Azure, etc. for provisioning virtual machines (VMs) with different operating systems (OS) are captured. The deployers can choose their desired

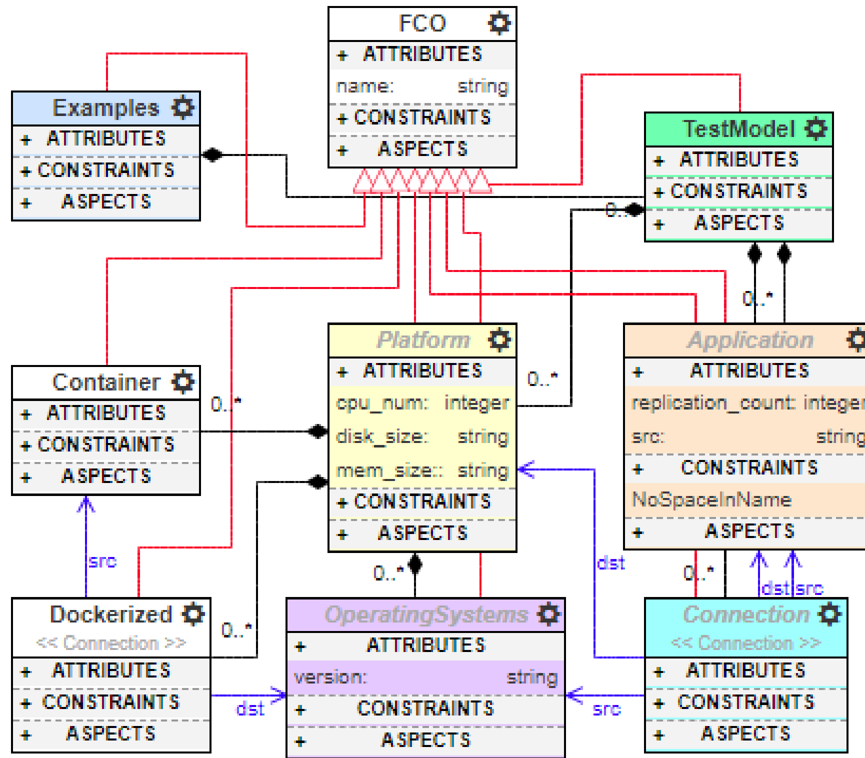


Figure III.6: Main Meta-Model of CloudCAMP framework. The black lines depict containment, the red lines depict inheritence and blue lines depict connection.

OS images to spawn the VMs/containers.

```
_cloudSpec_(type, vmtype, services, ostype)
type ::= Openstack | Amazon | Azure | Hardware
services ::= Function
vmtype ::= Function
os_type ::= ubuntu | redhat | windows
```

The deployer can select a pre-defined VM flavor, available networks, security groups, roles, and the available images, all of which are defined as variabilities in our meta-model. They also must specify their environment file, the secret key for the selected cloud host types, which are the endpoints to bind to a particular cloud provider as shown in Figure III.1. Optionally, a pre-deployed machine can be specified by providing the IP address

and OS. Available services and VM types for cloud platforms are pre-defined in the meta-model.

III.3.3.3 Meta-model for Application Components

For cloud-hosted services, CloudCAMP provides different node types for application components such as Web Application, Database Application, DataAnalytics Application, etc. For instance, the meta-model enables a deployer to choose the web server attribute, language for the code, the database server attribute, or the NoSQL database attributes from the provided list. The deployer has to specify the variable attributes to deploy the desired application component type.

```
_appSpec_(type, os_type, swdependency, attributes)
type ::= Web | Database | DataAnalytics | ...
os_type ::= ubuntu | redhat | windows
swdependency ::= Query to knowledgeBase
attributes ::= Function
```

III.3.3.4 Defining the Relationship among Components

Four relationship types bind the node types in the meta-model as follows:

① *'hostedOn'* relationship type implies the source node type is required to be deployed on the destination node type, e.g., Webserver is hosted on Ubuntu 16.04 in OpenStack.

② *'connectsTo'* relationship type is used for deployment order to relate the source node type's endpoint to the required target node type endpoint if they are dependent on each other. The node types linked by 'connectsTo' can be configured in parallel, but the service at the source node needs to deploy only after starting the target node.

③ *'deleteFrom'* connection type defines the source node type is required to be removed from the end node type.

④ *'migrateTo'* connection type defines the source node type that is to be migrated to the end node type. The 'migrateTo' relation type cannot be defined without a 'deleteFrom' connection type to assure the correctness of the business model.

III.3.3.5 Extensibility of the Meta-model

CloudCAMP is an opinionated framework, however, with lots of freedom. The meta-model has been designed for extensibility so that in future we can add more application node types.

① If the application type is defined, e.g., SQLite needs to be added, it will go under the DBApplication branch, and all the specific attributes will be automatically inherited from the parent node e.g., DBApplication. If more attributes need to defined, the framework designer can add it under the SQLite component type.

② If the application type is not defined, such as if the framework designer wants to add a stream processing engine, then the StreamProcessingEngine component should be added under the parent Application node and should capture the commonalities as attributes. Then as a child of StreamProcessingEngine node type (at M1 level) specific engine, such as Apache Kafka, Storm needs to be added. The variability points specific to be the engines needs to be added as attributes. Reverse engineering can obtain variable attributes.

Adding a new application component is time-consuming; however, it is a one-time effort, and it is reusable.

III.3.4 Design of CloudCAMP Knowledge Base

The Knowledge Base of CloudCAMP comprises a database and the application type templates.

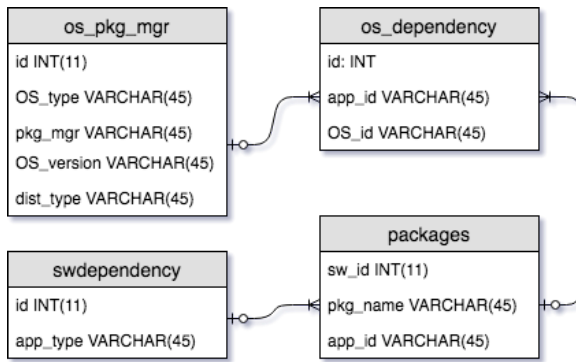


Figure III.7: Entity-Relation(ER) Diagram of CloudCAMP knowledge base

sw_id	pkg_name	app_id
1	mysql-server	2
2	mysql-client	2
3	python-mysqldb	2
4	libmysqlclient-dev	2

packages

id	AppType
1	php
2	mysql

swdependency

id	OS_type	pkg_mgr	OS_version	distribution
1	ubuntu	apt	14.04	linux
2	centos	yum	7	linux
3	rhel	yum	7.3	linux

os_pkg_mgr

Figure III.8: Sample portion of KnowledgeBase Database tables

III.3.4.1 Design of Knowledge Base Database

The ER diagram of the knowledge base database is depicted in Figure III.7, and it reflects the artifact sets stored in the knowledge base. We have structured it as four tables: `os_pkg_mgr`, `os-dependency`, `packages` and `swdependency` to build the knowledge database. We store 1) all the operating systems, their distributions, and versions in the `os_pkg_manager` table, 2) all available application component types, e.g., PHP based web application, MySQL based DB applications, etc. are stored in `swdependency` table, and 3) all the software packages needed for a particular application type is found using reverse engineering and stored in the packages table. For example, to install the scikit-learn package (http://scikit-learn.org), one needs to install python, python-dev, python-pip, python-numpy, etc. using the apt-manager package, and then the scikit-learn package can be installed from the pip package manager. In a relational table called `os_dependency`, we map the software packages and their versions with operating systems and their versions and store it as a key-value pair. For instance, to install java8 on Ubuntu 16.04, we need

different packages than to install java8 on windows10. We build the lookup table manually to handle these variability points. For new application component types, the application developer needs to populate the tables with all software dependencies. The sample section of the database table structure is shown in Figure III.8.

III.3.4.2 Design of Knowledge Base Template

The knowledge base templates are designed by capturing the commonality point of the application components, and it leaves the placeholders which need to be filled up by the CloudCAMP DSML by querying the knowledge base database. One sample ansible-specific template is shown in Figure. III.9(a). The algorithms to fill the template from the user-defined specifications are described in Algorithm 1 and 2. The generated filled template is shown in Figure. III.9(b). Different templates are designed to serve specific application components with different configurations.

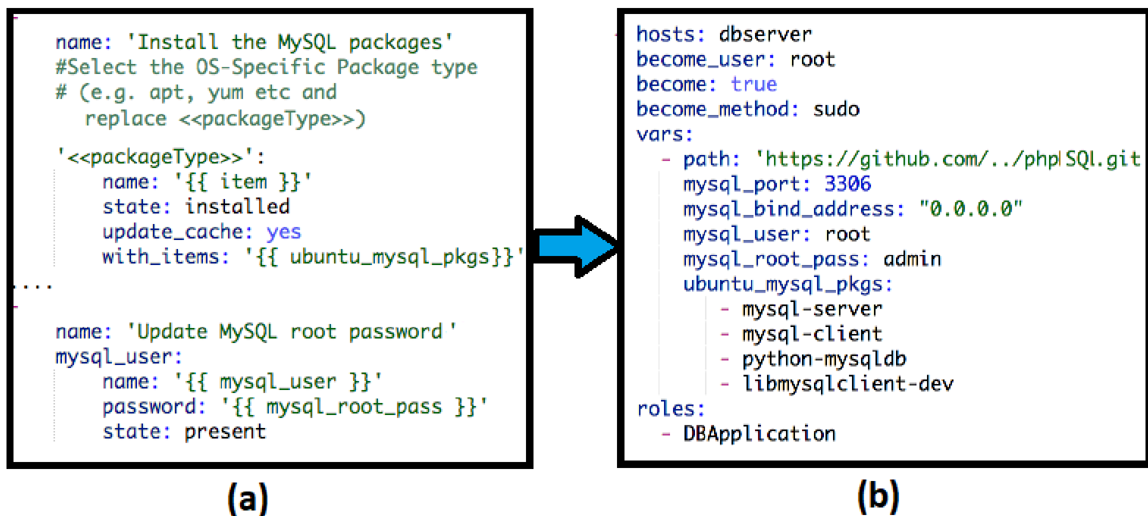


Figure III.9: (a)Sample DBapplication type template and (b)Sample portion of the Auto-generated code for Deploying MySQL DB application

III.3.4.3 Extensibility of the Knowledge Base

The knowledge base is extensible by design. Addition of new application components require the design of new templates (at least by part) by reverse-engineering the software

stack. The commonalities and variabilities need to be identified, and according to that, the template needs to be designed. The software dependencies for the application components need to be identified, and this information should be inserted in the knowledge base database tables: `os_pkg_mgr`, `os-dependency`, `packages` and `swdependency`. Similarly, for the new application components, the framework designer should insert and manipulate the records in these tables correctly.

III.3.5 Generative Capabilities of CloudCAMP DSML

CloudCAMP DSML provides generative capabilities for an IAC solution by interpreting the instances of models for which it incorporates a built-in knowledge base. The CloudCAMP DSML in WebGME is built using JavaScript, NodeJS, and a MySQL database.

As an example, we will walk through the specifications needed to be captured for the WebApplication and DBApplication component types. As shown in the M0 level of Figure III.5, the HTTP servers (e.g., Apache web server) for the webEngines are captured in WebApplication component type, and that is related to the node template for a WebApplication. The development languages and frameworks (Node.js, PHP, Django, etc.) of the webApplication is taken as attributes in the software property as depicted in the M1 level of Figure III.5, which is derived from Application type of M2 level, and our modeling tools meta-model is shown in M3 level. Similarly, as shown in the M0 level of Figure III.5, the software for the database types (e.g., Relational Databases such as MySQL, PostgreSQL, or NoSQL databases such as Cassandra, MongoDB) are captured in DBApplication component type, and that is related to the node template for the Database Application. Related features, such as the user id, password, specific binding port number of the Database application, etc. are stated as attributes, which is captured in the M1 level of the MOF.

III.3.5.1 Knowledge Base for Generation of Infrastructure-as-code Solution for Deployment

CloudCAMP's generative capabilities (Requirement III.1.2.2) are enabled via a WebGME plugin, which is invoked by a user after the modeling process. It generates and executes IAC as described in Algorithm 1. The VMs are spawned in the specified cloud platform based on the destination of 'HostedOn' connection [Lines 8-14]. Wherever possible, CloudCAMP will ensure that scripts specific to provisioning run in parallel to provide faster deployment. Once the VMs are spawned, *GenerateConfig()* queries the knowledge base [line24-34] to populate the appModel [line17] based on the user's specifications. Then, the query result fills application-specific pre-defined configuration templates and generates IAC, e.g., Ansible, for specific application components [line 29-34] using template-based transformation. A similar approach is taken to configure the service-specific containers or to start the cloud-specific services.

A sample of the automated SQL script used to query the knowledge base for deployment script generation is shown below:

```sql
SELECT  pkg.pkg_name
FROM    packages pkg,
        swdependency dep
WHERE   pkg.app_id = dep.id
    AND   pkg.apptype = <language>
    AND   pkg.sw_id IN
    (     SELECT app_sw_id
          FROM   os_dependency
          WHERE  os_id IN
              (     SELECT id
                    FROM   os_pkg_mgr
                    WHERE  concat(os_type, os_version)=<os>, <version>))
```

III.3.5.2 Determining the Order of Deployment and Execution

The Enactor component, which is a NodeJS script, builds the dependency tree for the application types defined in the meta-model and feeds it to the orchestration workflow

Algorithm 1: Deployment Script Generation

```
1  cloudModel ← Objects to store cloud specs
2  appModel ← Objects to store app specs
3
4  Procedure GenerateIAC()
5  if ConectionType == 'HostedOn' then
6  |   cloudType ← the destination node of connection
7  |   appType ← the source node of connection
8  |   if cloudType == 'Desired Cloud Platform' then
9  |   |   while !cloudModel.empty() do
10 |   |   |   Traverse the cloudModel
11 |   |   |   Fill 'cloudType' specific API Template
12 |   |   |   Generate 'cloudType' specific script
13 |   |   |   Execute script to spwan VMs
14 |   |   end
15 |   end
16 |   IPAddress(es) ← IP Address of target machine
17 |   GenerateConfig(IPAddress(es),appType)
18 |   if ConectionType == 'connectsTo' then
19 |   |   Find the source and destination application type
20 |   |   Prepare workflow to execute destination script(s)
21 |   |   Prepare workflow to execute source script(s)
22 |   end
23 end
24
25 Procedure GenerateConfig()
   Input: IPAddress(es) of Application Component Type
26 Create empty Tree Structure
27 Fill 'hosts' with IPAddress(es) of App Component Location
28 if appComponent == 'Desired Application Type' then
29 |   while !appModel.empty() do
30 |   |   Traverse the appModel
31 |   |   Query dataBase for appType = 'appComponent'
32 |   |   Fill 'appType' specific API Templates
33 |   |   Create complete Tree Structure
34 |   end
35 end
36 Wait for SSH in target machine(s)
37 Run workflow to execute tasks in parallel
```

engine. We generate scripts for automation tools (e.g., Ansible playbooks) for different component types, and these tools can, in turn, dispatch tasks to multiple hosts in parallel. If there is a 'connectsTo' relationship in the model, we let the dependent script complete first by defining the dependency chain [Line 18-21]. All the 'HostedOn' dependent building blocks run in a linear fashion. Thus, the Enactor remotely connects to the deployment hosts and deploys the application in proper order.

III.3.5.3 Generation of Infrastructure-as-code for Migration

The algorithm for generating a migration workflow (Requirement III.1.2.3) is portrayed in Algorithm 2. The 'deleteFrom' connection type specifies from where the user wants to move the application components and attaches a 'migrateTo' connection type to indicate the destination. The migrationType (stateless or stateful) must be selected, and depending on that, CloudCAMP decides to checkpoint the application state or not before terminating the old VMs/containers [Line 17-23]. The 'migrateTo' relation type cannot be defined without 'deleteFrom' connection type to ensure the correctness of the model.

Although actions are taken for live migration, an application component from one VM to another depends on the application component type, which is a hard problem. For example, live migration of DBApplication needs a two-phase commit protocol, and a consensus algorithm to make it reliable. For the sake of simplicity, in the Algorithm 2, we generalize our approach. Our future work will consider more complicated scenarios of live migration and application consistency and availability issues.

According to Algorithm 2, it will spawn a new VM with the new operating system for the 'migrateTo' destination node. For Stateful migration[line 20-23], our platform creates a manager node with a load balancer, and deploy the application on the current node. From that point of time, the load balancer redirects all the new requests to the current node, and it checkpoints the current state of the old node and restores it in the current node. Finally, it detaches the load balancer node. Thus, it produces the full infrastructure-as-code solution

```
Algorithm 2: Migration Script Generation
1  cloudModel ← Objects to store cloud specs
2  appModel ← Objects to store app specs
3
4  Procedure MigrationIAC()
5  if ConectionType == ' deleteFrom' then
6  |   cloudType ← the destination node of connection
7  |   appType ← the source node of connection
8  |   IPAddress(es) ← IP Address of target machine
9  |   if cloudType == 'Desired Cloud Platform' then
10 |   |   Generate 'cloudType' specific workflow script
11 |   |   Execute script to terminate VMs
12 |   end
13 end
14 if ConectionType == 'migrateTo' then
15 |   GenerateIAC()
16 end
17 if migrationType == 'stateless' then
18 |   Execute deletion and migration scripts in parallel
19 end
20 else if migrationType == 'stateful' then
21 |   Checkpoint current application state on old machine
22 |   Restore checkpoint on the current machine
23 |   Execute deletion and migration scripts in parallel
24 end
```

along with the related configuration files. All of these complete the Ansible layout tree structure helps to migrate application components from one node to another node.

III.3.5.4 Support for Continuous Delivery

CloudCAMP can also handle continuous delivery and component addition/deletion, which is just a matter of updating the model with addition or removal of a component. For instance, to add a new database server, a user extends the model with a DBApplication node type and 'connectsTo' relationships from the webserver to the database server. CloudCAMP will generate IAC for the newly added component and executes it to deploy added component without hampering the availability of the existing application. Since Ansible is idempotent, it always sets the same configuration in the target environment regardless of their current state.

III.3.5.5 Constraints Checking for Correctness Business Models

We also validate the business model by checking for constraint violations, thereby ensuring that the models are "correct-by-construction." We verify the correctness of the endpoint configurations for application component types, the relationship types, cloud-specific types, etc., and the business model as a whole before generating any infrastructure code. Examples of some of the constraints are shown below:

- $\forall$ Applications $\in$ WebApplication $\exists!$ WebEngine
- $\forall$ Applications $\in$ DBApplication $\exists!$ DBEngine
- $\forall$ Platform $\in$ Openstack $\exists!$ imageName
- $\forall$ Applications $\in$ DataAnalyticsApp $\exists$ processEngine etc.

We also verify other rule-based constraints to verify the compatibility of the component. For example, Amazon Kinesis delivery stream destination has to be Amazon Services (e.g., Redshift, S3), it cannot be Azure or OpenStack Services. We gather this information using reverse engineering. Thus, we validate the business model by satisfying the constraints and notify the user if there are any discrepancies in the business model.

III.4 Evaluation

This section describes results comparing the time and effort incurred in deploying application use cases using (a) manual efforts, where the deployer must log into each machine and type the commands to install packages and deploy the applications, (b) manually writing scripts to deploy these applications, and (c) using the CloudCAMP framework.

III.4.1 Case Study 1: LAMP-based Service Deployment Study

This is a prototypical three-tier Linux, Apache, MySQL, and PHP (LAMP)-based microservice architecture deployment similar to the motivating example described in Sec-

tion III.1.1. Figure III.1 shows the application topology illustrating the modeling effort in CloudCAMP.

Here, we describe the details of template-based transformation that happens behind the scenes within CloudCAMP DSML. As stated in Algorithm 1, the DSML traverses the business logic tree of Figure III.1, which is defined by the deployer, and collects all the user-defined attributes as shown in Figure III.10 and Figure III.11. It populates the pre-defined template for the specific application type with the user-defined attributes. The 'mysql_user' and 'mysql_root_pass' will be filled from specifications related to DBApplication type (Figure III.11).

Meta type	WebApplication
Attributes	
language	PHP
name	WebApplication
replication_count	1
src	https://github.com/Anirban2

Figure III.10: Specifications related to WebApplication type

Meta type	DBApplication
Attributes	
dbLocation	mySqlDB/movieDB.sql,myS
dbnames	moviedb,bookstore
name	DBApplication
password	admin
port	3306
replication_count	1
src	https://github.com/Anirban2
user	root

Figure III.11: Specifications related to DBApplication type

The application components' software dependencies are gathered by querying the knowledge base database. For example, to install MySQL on a Ubuntu16.04 machine, the mysql-server and mysql-client software packages are needed. So, CloudCAMP DSML will query the knowledge base database and runs the template-based transformation to concretize the pre-defined partial template. The DSML copies the related configuration files in specific folders to configure MySQL correctly. Thus, the DSML will populate the pre-defined template file with all the details, and generate deployable Ansible-specific deployable IAC. After generating all the Ansible-specific files, the CloudCAMP executes these files in proper order to deploy the application by provisioning the cloud infrastructure as described in Algorithm 1.

III.4.1.1 Measurement of Manual Effort

We conducted a small user study in a Cloud Computing course for case study 1 involving sixteen teams of three students each. We requested users to manually configure the files, create the handlers to specify the deployment order in the desired host, log into each host where the application components are deployed and manually install the packages, configure the software packages and finally start the different components in the correct order. We have also requested them to write the ansible script to provision the same application stack and infrastructure. We measured the time taken, and efforts for (a) a fully manual effort, (b) for writing scripts in Ansible and executing these manually, and (c) using the CloudCAMP framework to deploy the scenario.

▷ *Quantitative Evaluation based on a User Study:* The questionnaire as shown in Table III.1 was created to conduct the study. For each question, the evaluation scale was 1–10 where one is the easiest and ten is the hardest.

Num	***Question***
Q1	How easy is it to deploy PHP MySQL application manually?
Q2	How easy is it to deploy PHPMySQL using DevOps tool like Ansible?
Q3	How easy is it to deploy PHPMySQL using CloudCAMP?
Q4	How much time and effort did you require to deploy the application manually (in minutes)?
Q5	How much time and effort are required in deploying the application using DevOps tool like Ansible (in minutes)?
Q6	How much time and effort are required deploying the application using CloudCAMP (in minutes)?
Q7	How likely are you to use the CloudCAMP platform to deploy applications in future?

Table III.1: Survey Questionnaire: For Q1–Q3, rate on a scale of (1-10)

▷ *Responses to Q1, Q2, and Q3 - Ease of use:* As seen from Figure III.12, the “ease of use” rating for the CloudCAMP platform is much higher compared to manual and scripting efforts. The median difficulty in the manual effort is rated as 72.2%, and median difficulty in scripting effort is rated as 71.6%, while the median difficulty rating for CloudCAMP use is 30.9%.

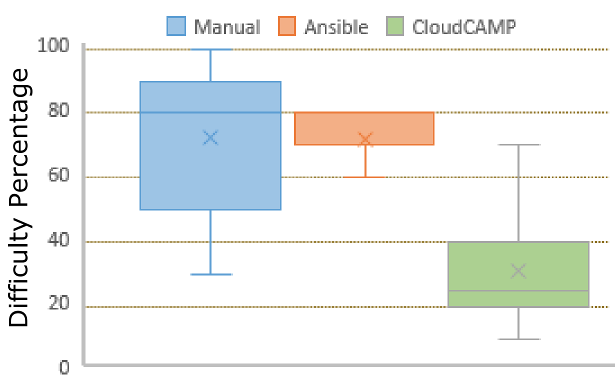


Figure III.12: Comparing difficulty percentages to deploy services in different approaches

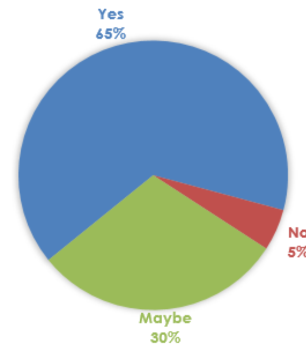


Figure III.13: Likeliness of using CloudCAMP for future cloud services deployment

	Deployment Time(mins)	***Lines to Deploy***	***Migration Time(mins)***	***Lines to Migrate***
median	510	300	720	550
mean $\pm$ std.dev.	516$\pm$244	315$\pm$47	653$\pm$231	553$\pm$142

Table III.2: Median and mean±std.dev for deployment time, lines of code written for deployment, migration time and Lines of code written for migration (for Q5–Q6).

▷ *Responses to Q4, Q5, and Q6 - Time to complete the entire deployment:*

The effort incurred by the user to deploy the LAMP model in the Cloud is shown in Table III.2, whereas using the CloudCAMP the same topology deployment time is approximately 15-20 minutes for the first time users.

▷ *Response to Q7:*As shown in Figure III.13, 65% of the respondents agreed to use CloudCAMP tool to deploy cloud applications in the future, whereas 30% are still unsure.

Discussions. Results from our user study strengthen our belief that the CloudCAMP platform will be a very resourceful and productive tool for business application deployers. We have also conducted a user study specifying to create Docker Containers (https://www.docker.com/) and deploy the LAMP architecture inside it using scripting tools and found very similar results. The visual drag and drop environment helps users to quickly deploy various scenarios of business application topology in distributed systems. Therefore, the benefits of automated provisioning accrued using CloudCAMP can easily be understood.

III.4.2 Case Study 2: Application Component Migration for LAMP-based Web Service

CloudCAMP platform also supports application component migration with ease for which we have two connection types 'deleteFrom' and 'migrateTo'. As described in Scenario III.1.2.2, suppose the user wants to migrate the database application component from one machine to another machine, which resides on a different OpenStack cloud platform. This assignment was to migrate the 'stateful' MySQL database service from one node to another node, and the students are asked to add load balancer node to make the service available all the time. CloudCAMP generates a new workflow structure based on the changed user specifications as described in section III.3.5.3.

▷ *Responses to Q4, Q5, and Q6: Time to complete the whole migration:* The average time the students took to write the scripts to complete the entire migration process is 653 minutes, with a median of 720 minutes as shown in Table III.2. Whereas our rough estimates for students using the CloudCAMP-based topology migration will be only 10-15 minutes for the first time users. The average lines of code written using manual effort for the migration process are 553 lines as per the survey is shown in Table III.2.

III.5 Conclusion

III.5.1 Summary

This chapter presented a model-driven engineering and generative programming approach for an automated deployment and management platform for cloud applications. It aids the application deployer in modeling service provisioning at a higher level of abstraction, and deploy its code without requiring significant domain expertise while requiring only minimal modeling effort and no low-level scripting. All the application components are the building blocks in our modeling environments and can be connected using exposed endpoints as a pipeline. The DSML will generate a "correct-by-construction" IAC solu-

tion from the pipeline and execute the IAC to provision the application stack on the target cloud environment. Using WebGME to define the CloudCAMP framework enables us to decouple its metamodel(s) and knowledge base from the generative aspects while permitting extensibility. CloudCAMP significantly increases the productivity and efficiency of the application deployment and management team. CloudCAMP is available in open source from https://doc-vu.github.io/DeploymentAutomation.

III.5.2 Discussions

CloudCAMP provides the flexibility to modify the application components as needed and facilitates selecting the building blocks for business requirements. We leave the choice of application design to the application deployer. For example, one can select MongoDB and Cassandra for their backend in the development phase, deploy and do a performance test without much hassle. As the framework matures, we can support more application components.

CHAPTER IV

BARISTA: EFFICIENT AND SCALABLE SERVERLESS SERVING SYSTEM FOR DEEP LEARNING PREDICTION SERVICES

IV.1 Introduction

IV.1.1 Emerging Trends

Cloud-hosted, predictive analytics services based on *pre-trained deep learning models* [67] have given rise to a diverse set of applications, such as speech recognition, natural language processing, fitness tracking, online recommendation systems, fraudulent behavior detection, genomics, and computer vision. End-users of these services query these pre-trained models using an interactive web or mobile interface through RESTful APIs. Based on the supplied input, these pre-trained models infer the target values and return the prediction results to the end-users. As an example, a speech recognition system transcribes spoken words into text [68].

These prediction services are usually containarized [69] and encapsulated with all the required software packages. Thus, deployer of these services can preferably use the *function-as-a-service* (FaaS) approach to hosting these services in an event-driven manner, wherein the functions are executed on the occurrence of some trigger or event (e.g., incoming request). This overall approach can be handled using *serverless computing* [70] since the service creator needs only to provide the function logic, the trigger conditions, and the *service level objectives*(SLOs), such as latency bounds, which are on order of few seconds. It is then the responsibility of the serverless platform provider to provide the hosting environment for these services and ensure that the SLOs are met.

IV.1.2 Challenges and State-of-the-Art Solutions

The execution environments for deep learning-based prediction services typically comprise containers running on a cluster of virtual machines (VMs). These prediction services are usually stateless, parallelizable(multi-threaded), and compute-intensive. The model sizes of these prediction services are large (hundreds of megabytes to gigabytes), which takes a significant time to load them into the containers and provision the infrastructure. Moreover, due to the parallelizable nature of these models, their running times can be substantially reduced by assigning more CPU cores (see Figure IV.1). However, allocating more memory only marginally improves the running times. Thus, a naïve approach to assuring the SLOs is to over-provision the service infrastructure; however, doing so imposes undue costs on the serverless provider. The efficient management of computing resources dynamically is required to minimize the cost of hosting these services in the cloud environment [71, 70, 14].

Although a substantial amount of literature exists on finding the sweet spot between resource overprovisioning (which wastes resources and increases the cost) and underprovisioning (which can violate the SLOs) [72, 73, 74], these works focus primarily on long-running analytics jobs for which the goal is to find optimal configurations to meet the SLOs and scale the resources dynamically to handle their variable workloads.

In contrast, the prediction services have short running times. Moreover, the incoming request (workloads) patterns can fluctuate significantly and follow a diurnal model, which requires rapid management of resources to meet the variable workload demands. A reactive approach is not suitable as the prediction latency may increase significantly due to infrastructure provisioning time (e.g., an order of minutes due to VM creation and model loading times). Hence, the desired solution is one that can forecast the workload patterns and can estimate the required resources for the application to maintain the SLOs under the forecasted workload. Determining the right cluster configurations and allocate the resources dynamically is hard due to the variability of cloud configurations (VM instance types), the

number of VMs, and their deployment and management costs [72, 75, 9].

Only recently have some solutions started to emerge to address these concerns [71, 76]. Nevertheless, there remain many unresolved problems. First, current horizontal elasticity solutions for prediction services often do not account for the container-based service lifecycle states (e.g., whether the VM is already up or not, or whether the container is running, and if so, whether the model is loaded or not). Each such state incurs a hosting cost and impacts the running time. Second, vertical elasticity solutions for containers do not yet fully exploit the parallelizable aspects of the pre-trained models. Third, proactive resource scaling decisions require effective workload forecasting and must be able to monitor the service lifecycle states, both of which are missing in prior efforts. Finally, existing strategies for container allocation tend to overlook performance interference issues from other co-located containerized services, which may cause unpredictable performance degradation [20].

IV.1.3 Overview of Technical Contributions

To address these unresolved problems, we present a serverless framework called Barista which hosts containerized, pre-trained deep learning models for predictive analytics in the cloud environment, and meets the SLOs of these services while minimizing hosting costs. Barista comprises an efficient, data-driven, and scalable resource allocator, which estimates the resource requirements ahead of time by exploiting the variable patterns of the incoming requests and the forecast-aware scheduling mechanism improves resource utilization while preventing physical resource exhaustion. We show how serverless computing concepts for dynamically scaling the resources vertically and horizontally can be utilized under different scenarios. Barista efficiently and cost-effectively provisions (scale-up and scale-down) resources for the prediction analytics services to meet its prediction latency bound. Specifically, we make the following contributions:

① ***Workload Forecast:*** We propose a hierarchical methodology to forecast the workload based on historical data.

② ***Resource Estimation:*** Barista allows service providers to communicate the performance constraints of their service models regarding their SLOs. An analytical model is provided to predict resource estimation based on latency bound, workload forecasting, and the profiled execution time model on different cloud configurations.

③ ***Serverless Resource Allocation:*** Barista provides a novel mechanism using the serverless paradigm to allocate resources proactively based on the difference between resource requirement estimation and current infrastructure state in an event-driven fashion.

IV.1.4 Organization of the Chapter

The rest of the chapter is organized as follows: Section IV.2 presents a survey of existing solutions in the literature and compares them Barista; Section IV.3 presents the problem formulation; Section IV.4 presents the design of Barista; Section IV.5 evaluates the Barista resouce allocator for a prototypical case study; and finally, Section IV.6 presents concluding remarks alluding to future directions.

IV.2 Background and Related Work

This section provides an overview of and literature survey along the dimensions of Deep Learning-based Prediction Services, Infrastructure Elasticity, Serverless Computing, and Workload forecasting, all of which are critical for the success of the presented work on Barista.

IV.2.1 Deep Learning-based Prediction Services

Pre-trained models based on Deep Learning techniques are increasingly being used in prediction analytics services. In this approach, all the learned internal parameters are stored in the form of a vector of scores for each category along with their weights in a pre-trained

deep learning model of desired accuracy [67]. Once the models are trained, these prediction models are seamlessly integrated into applications to predict outcomes based on new input data.

IV.2.2 Serverless Computing

Serverless Computing focuses on providing zero administration by automating deployment and management tasks. In this paradigm, the responsibility of deployment and management is delegated to another entity, which could be the cloud infrastructure provider itself or a mediating entity. The execution platform leverages container technology to deploy and scale the prediction service components, which helps to minimize idle resource capacity [38]. These features are beneficial to the design and deployment of parallelizable (multi-threaded) deep learning prediction services.

These features are beneficial to the design and deployment of deep learning prediction services. Moreover, isolation and decentralization of the pre-trained models due to containers help to isolate both scaling on a per-model execution level and failures. However, due to variation in workloads, the providers of prediction services are required to modify their resource requirements by monitoring the resources continuously [77], and that reactive approach can often violate the SLOs. Barista intelligently and efficiently manages the containerized allocation based on resource estimation by workload forecasting and profiling the execution time of prediction services.

The MxNet deep learning framework [78] shows the feasibility of using serverless computing AWS Lambda framework [79]. There are several efforts [39, 7, 80, 36, 8, 36] to deploy and orchestrate VMs or containers dynamically with all software dependencies. Barista's focus is orthogonal to these efforts; instead, it is to trigger the deployment process ahead of time so that the system can handle the workload surges by utilizing the aforementioned efforts.

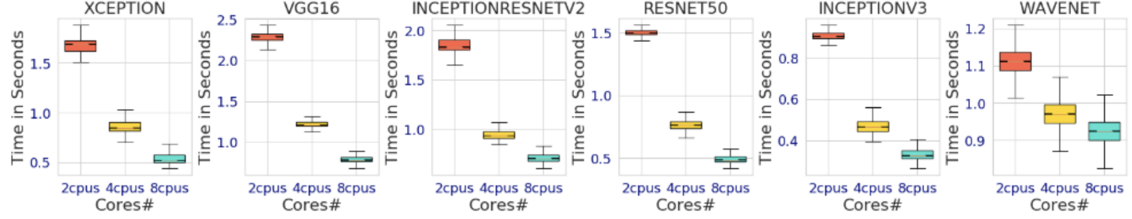


Figure IV.1: Box plots of prediction times for different deep learning pre-trained models on different numbers of CPU cores (2, 4 and 8).

IV.2.3 Dynamic Infrastructure Elasticity

Most state-of-the-art technology and research strategies to *horizontally* or *vertically* scale the resources are heuristics-driven or rule-based and have custom triggers. Selecting optimal cloud configurations is an NP-hard problem, and various models are presented based on heuristics [81, 20, 82, 37, 83, 73]. In Barista, we consider leasing VMs from the cloud provider to meet the latency bounds by relying on time series forecasting of the incoming workloads. We propose an efficient heuristic to select configuration types to guarantee bounded prediction latency while minimizing the cost.

Swayam [71] presented a short-term predictive provisioning model to guarantee SLO while minimizing resource waste. However, they only consider horizontal scaling by allocating more backend containers from the resource pool. Model loading time for deep learning models is significant, especially when the container is in cold state [78]. In contrast, Barista proactively considers infrastructure provisioning time to scale the system and also allows vertical resource scaling.

Vertical elasticity adds flexibility as it eliminates the overhead in starting a new VM and loading the service model. Prior efforts to scale the CPU resources vertically appear in [84, 85] including an approach that uses the discrete-time feedback controller leveraging MAPE-K loop for containerized applications [69]. Barista uses an efficient, proactive method to trigger the scaling of resources horizontally while relying on vertical scaling reactively to allocate and de-allocate CPU cores for model correction when our estimation model cannot predict accurately. Our reactive approach can also handle sudden workload

spikes within a threshold.

IV.2.4 Workload Forecasting

Workload forecasting is indispensable for service providers to anticipate changes in resource demands and make proactive resource allocation decisions. Various forecasting methods based on time series analysis are described in [86]. In AGILE [87], a resource prediction algorithm is proposed to scale up the server pool by renting the VMs from the cloud providers *a priori* to guarantee the SLOs of the services. Similarly, Dejavu [88] and Bubble-Flux [89] proposed self-adaptive resource management algorithms, which leverage workload prediction and application performance characterization to predict resource requirements. These efforts employ a linear model for workload prediction, which often results in low-quality forecasts with high uncertainty.

Several non-linear methodologies based on Support Vector Machine [90], Error Correction Neural Network (ECNN) [91], Gaussian processes [92, 18] are proposed to predict workloads. However, these models fail to capture longer-term trends, which are characteristics of cloud-hosted services [93]. In [93], a hybrid model called *Prophet* is proposed for forecasting workloads by combining linear/logistic trend models with a Fourier series-based seasonality model. According to the authors, Prophet is easier to use than the widely used ARIMA models [69, 94, 95] as it handles missing values automatically. Moreover, the ARIMA models generally struggle to produce good quality forecasts as it lacks seasonality detection.

In general, workload forecasting methods tend to lack feedback to update predictions based on recent performance. Therefore, Barista extends Prophet with a non-linear decision-based model that modifies the forecast according to previous prediction errors. Barista workload forecasting model estimates the resource requirement and proactively scales the infrastructure to guarantee application SLOs.

IV.3 System Model and Problem Description

This section first describes the system model and assumptions and then presents the problem formulation, which has two subproblems. First, given the SLO of the service, the pre-trained model properties, and the costs of VM configurations, we determine the cost-effective VM types by solving an optimization problem to meet their SLOs (Section IV.3.2). We then consider the dynamic management of VM and container resources through workload forecasting and infrastructure elasticity (Section IV.3.3).

IV.3.1 Infrastructure Model and Assumptions

To explore the /emphserverless capabilities, we assume that the Deep Learning pre-trained model for the predictive analytics service is encapsulated inside containers that are executed in a cluster of VMs. All the requests to the service are assumed to be *homogeneous*, i.e., they execute the same prediction model, and that the service is stateless. Since deep learning models are generally compute-intensive, they benefit from executing on multiple cores. We validate this claim in Figure IV.1, which shows the range of prediction time latencies for several pre-trained deep-learning models on a VM hosted on AMD Opteron 2300 (Gen 3 Class Opteron) physical machine with different numbers of assigned CPU cores, demonstrating good speedup behaviors. The results are obtained by running 10,000 trial executions in isolation for each model. Further, let *min_mem* denote the minimum amount of memory required to run a prediction model.

The VM lifecycle in cloud infrastructure for service deployment and management is considered as follows.

① `VM Cold`: VM has not been deployed.

② `VM Warm`: VM is deployed, but the container inside the VM has not been downloaded.

③ `Container Cold`: the container is downloaded, but the pre-trained deep-learning model has not been loaded into the container's memory.

④ `Container Warm`: the deep-learning model is loaded, and the container is ready to serve the prediction requests.

Finally, we assume that once a VM is deployed, it is leased from the cloud provider for a minimum duration of τ_{vm} time. During this time, the deployment cost is paid for even if the VM is not used (because we do not scale down the system immediately). This could happen when the prediction model is unloaded from the container's memory during lightly-loaded periods so that the VM could serve other batch jobs in the background (e.g., deep analytics applications).

IV.3.2 VM Flavor Selection and Initial Deployment

We first consider the static problem of serving a fixed set of requests that execute a deep-learning prediction model by finding the most cost-effective VM flavor type. Let λ denote the constraint specified by the SLO of the model regarding its execution latency[1], and let t_p denote the latency when the model is executed on p CPU cores.

To serve the prediction requests, we need to deploy a set of VMs from the cloud provider that offers a collection of m possible configurations (flavors), denoted as $\{vm_1, vm_2, \ldots, vm_m\}$. We consider three parameters to specify each configuration vm_i, i.e. ① p_i, the number of available cores, ② mem_i, the memory capacity, and ③ $cost_i$, the cost of deployment. In particular, the cost includes both the running cost and the management cost of deploying the VM. For each configuration vm_i, suppose $\alpha_i \in \{0, 1, 2, \ldots,\}$ number of VMs are deployed. Then, the total deployment cost is given by $total_cost = \sum_{i=1}^{m} \alpha_i \cdot cost_i$. While each deployed VM is assumed to serve only one request at a time (because each request is benefited from consuming all the cores as the prediction service is highly-parallelizable)

[1]Depending on the SLO; the execution latency can be flexibly defined, based on, e.g., worst-case latency, x-percentile latency. In this chapter, we consider the 95th percentile latency.

and could serve multiple requests one after another. In this case, the request that is served later in the pipeline needs to wait for the preceding requests to be first completed, which will delay its prediction time.

The optimization problem concerns deploying a set of VMs, i.e., to choose α_i's for all $i = 1,\dots,m$, so that the total cost is minimized while the SLOs of all the requests can be satisfied (i.e., with latency not larger than λ). Section IV.4.4 presents our VM deployment solution.

IV.3.3 Dynamic Resource Provisioning via Workload Forecasting and Infrastructure Elasticity

The resources must be provisioned dynamically to meet the SLOs under the workload variations. Since the reactive approach can be detrimental to response times, we use a proactive approach to handle the variation of the workload by forecasting the future service demands based on historical workload patterns of a deep-learning prediction model.

With the forecasted future workload (i.e., number of service requests), the VM deployment decision (as described in Section IV.3.2) must also be adapted accordingly. Horizontal scaling [96] is a promising approach to provision the resources dynamically. To that end, we exploit the four cloud infrastructure states explained earlier. Figure IV.2 illustrates the actions needed to transition between the states. Each action incurs a state transition time. Specifically, we denote the VM deployment time by t_{vm}, the container service download time as t_{cd}, and the pre-trained model loading time as t_{ml}.[2]

Figure IV.3 shows concrete timings for the different prediction services we tested on our experimental infrastructure. Further, we refer to total time to set up the service as $t_{setup} = t_{vm} + t_{cd} + t_{ml}$. This motivates the need to forecast the future workload $t'_{setup} = t_{setup} + t_{forecast}$ time ahead to guarantee the SLO with certain accuracy, where $t_{forecast}$ is

[2]The time required to unload the model from memory, denoted as t_{mu}, is negligible and thus not considered. The time taken to move from any state to `VM Cold` is denoted as t_{exp}. This duration does not impact the resource manager logic and hence is also ignored.

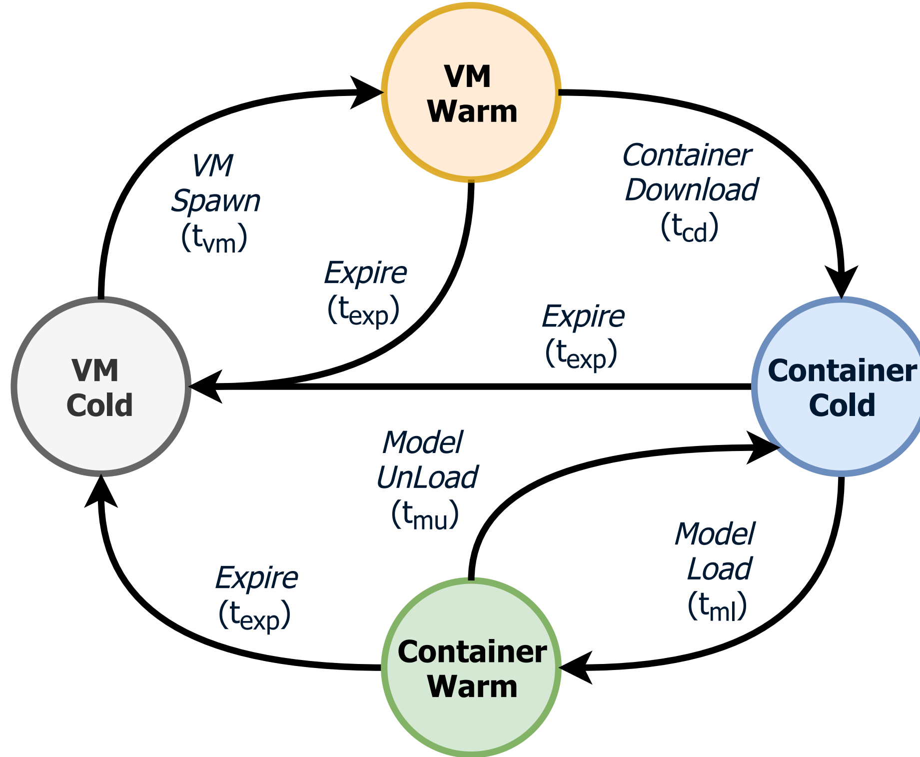


Figure IV.2: An abstract state machine showing different states and transitions associated with a life cycle of a VM in cloud infrastructure. Edges are labeled with actions and time duration to complete the state transition.

the time taken to obtain the forecast. The forecasting needs to be performed for t'_{setup} time into the future to account for the infrastructure setup time.

When the VM is not servicing any requests, the infrastructure state transitions from `Container Warm` to `Container Cold`. Later on, when the load increases and the VM needs to serve requests again, the model will be reloaded. As a result, we also need to check the infrastructure state ahead of time to make decisions for downloading the container and loading the model if it is not already in the `Container Warm` state. We account for all of these times in meeting the SLOs while avoiding excess over-provisioning of the infrastructure resources. Section IV.4.5 presents our combined solution to the dynamic resource management problem that incorporates infrastructure elasticity, workload forecasting, and VM deployment.

Moreover, if the workload forecaster overestimates the workload, we allow the excess resources to be utilized by the low-priority batch jobs by vertical scaling. Co-locating

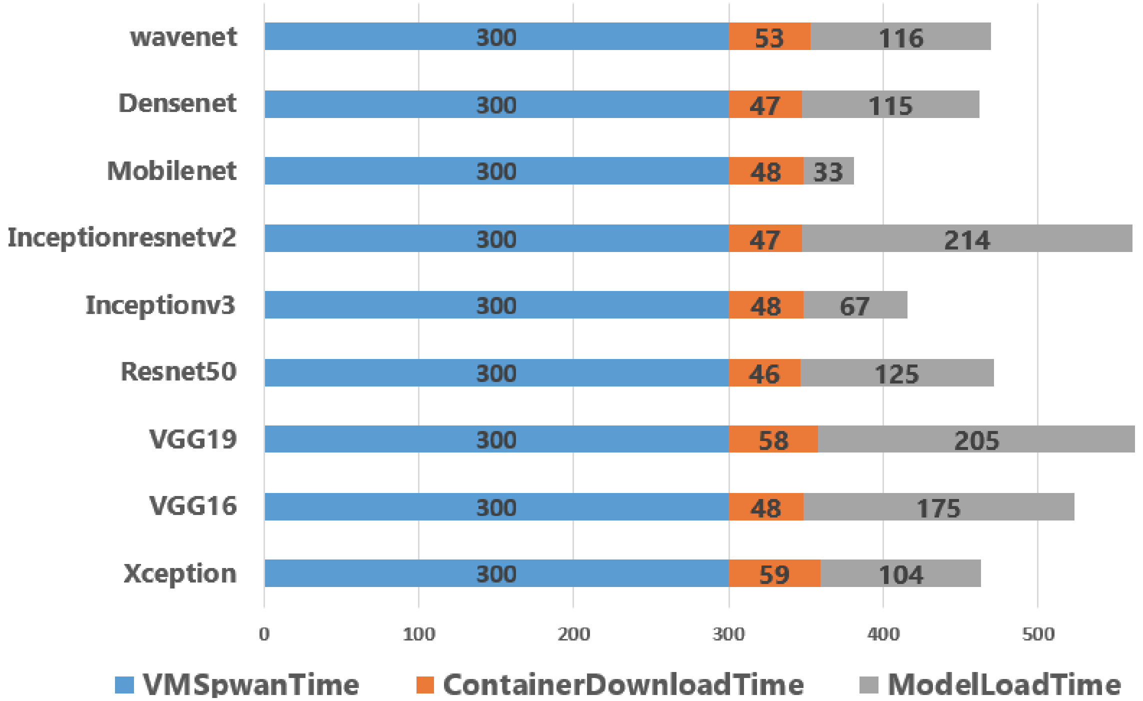


Figure IV.3: The setup times (in seconds) for different deep-learning prediction models as per our experiment. The blue bar shows VM deployment time (t_{vm}), orange bars show the specific pre-trained model container download time (t_{cd}), and grey bar show prediction model loading time (t_{ml}).

various jobs on a server can cause performance interferences. In our approach, we assume 20% performance degradation (for the worst-case scenario based on our experiment) if the latency-sensitive prediction service is co-hosted with batch jobs.

IV.4 Design and Implementation of Barista

In this section, we give the architectural insights of Barista by describing its various components. We also explain our solutions to the problems of static VM deployment and dynamic resource provisioning, as mentioned in Section IV.3.

IV.4.1 Architecture of Barista

Barista architecture consists of a pool of frontend and backend servers, load balancers to distribute the requests, a platform manager to allocate and scale backends for different prediction services, as shown in Figure IV.4. Frontend servers are the virtual machines,

which host the user interface, whereas backend servers host the containerized pre-trained deep learning model. End users send their requests to the frontend load balancer, which redirects the requests to the frontend servers based on the *round robin* policy. Each frontend server forwards the request to the backend load balancer, which then redirects the request to one of the backend servers assigned to serve the prediction query based on the *least loaded connection* policy. Each backend processes a single request at a time and gives the prediction result back. The platform manager is an integral part of Barista, which is responsible for dynamic provisioning of resources in cloud infrastructure by forecasting workload patterns and estimating the execution time of various prediction queries. The platform manager (as zoomed-in Figure IV.4) consists of a prediction service profiler, a request monitor, a request forecaster, a prediction latency monitor, and a resource manager. They are described as follows:

① **Prediction Service Profiler:** It profiles the execution time of a prediction service on different numbers of CPU cores and finds the best distribution (as shown in Figure IV.5). This provides the 95th percentile latency estimate of the execution time for the prediction service based on the assigned number of cores.

② **Request Monitor:** It monitors and logs the number of aggregated incoming requests received every minute by the backend load balancer.

③ **Request Forecaster:** It predicts the number of requests t'_{setup} time steps into the future based on the historical data. The forecaster also updates its model every minute based on the previous prediction errors and the actual data from the request monitor to diminish uncertainty.

④ **Prediction Latency Monitor:** It monitors and logs the SLO violations for the incoming requests every five seconds. SLO is defined over the response time of the backend servers to a prediction query requested by the frontend servers.

⑤ **Resource Manager:** It allocates the required number of virtual machines for the forecasted workloads and performs intelligent scaling based on resource estimation and provisioning strategies, as discussed in Section IV.4.4.

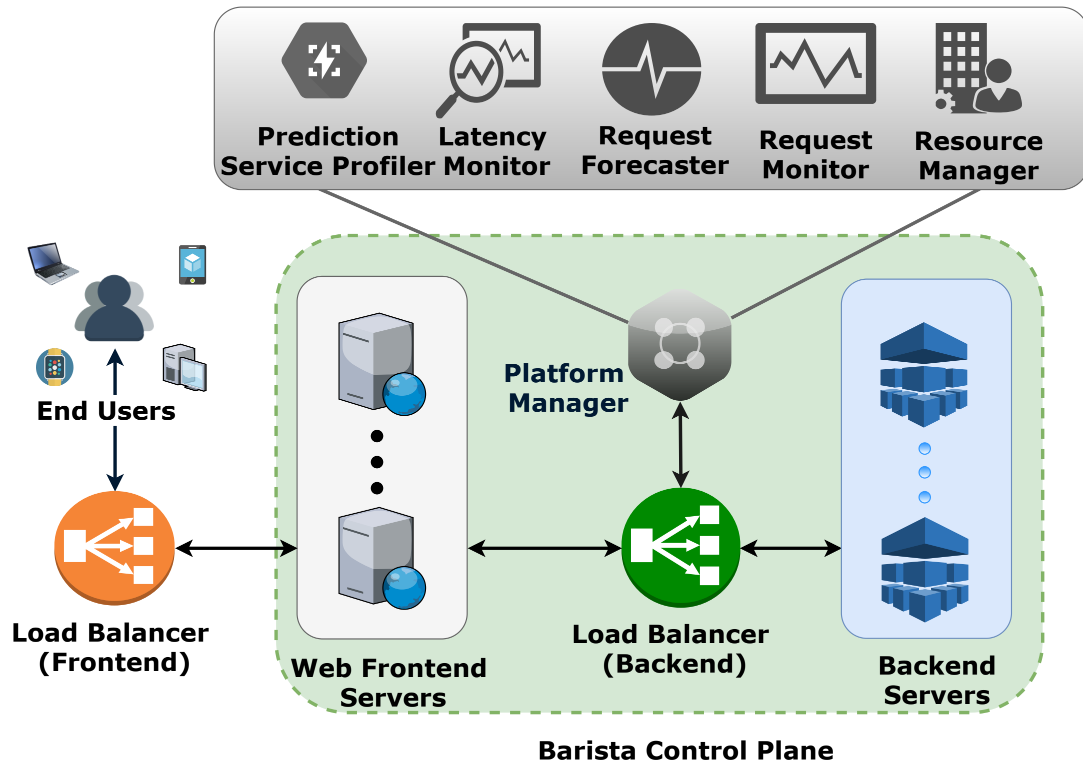


Figure IV.4: Architecture of Barista serving system.

The operation of the platform manager can be categorized into two phases, *online* and *offline*, as shown in Figure IV.5. In the offline or design phase, the execution time of the deep learning model is profiled on different VM configurations followed by distribution estimation. The workload forecasting model is also trained in this phase. In the online or runtime phase, based on the output of workload forecaster and execution time estimator, resources are estimated and provisioned. The actual workload is also being monitored and stored, which is used to update the forecasting model based on the last five error predictions and rolling training window. The following subsections provide more details on these operations.

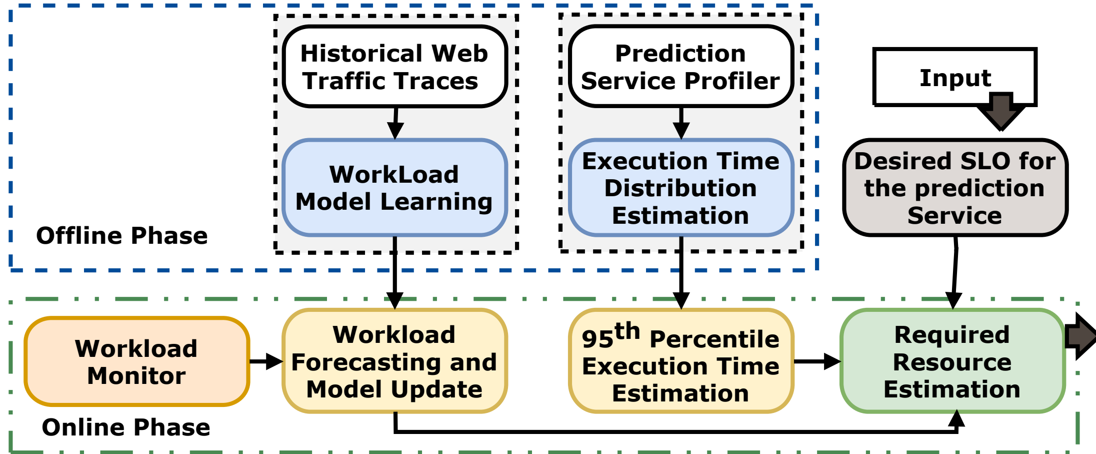


Figure IV.5: Data flow model of Barista platform manager.

IV.4.2 Execution Time Distribution Estimation

An inaccurate estimate of the execution time of a pre-trained deep learning model can result in erroneous output produced by the resource manager, which may lead to over- or under-provisioning of resources. Thus, in Barista, extensive offline profiling of different deep learning models is performed on various VM configurations. Figure IV.1 shows our experiments on configurations involving 2, 4, and 8 CPU cores with required memory size, where each experiment contains 10,000 trails. The execution times are random and follow an unknown distribution. In Barista, the resource manager uses the 95th percentile statistic of the execution time to provision resources. In order to accurately calculate the percentile values, we remove sample bias and estimate the unknown distribution.

Barista prediction service profiler estimates the distribution using parametric methods based on Maximum Likelihood Estimation (MLE) for fitting different distributions and finding their respective unknown parameters. For quantifying the goodness of fit, we use one-sample Kolmogorov-Smirnov (K-S) test [97] to rank different hypothesized distributions. Given the cumulative distribution function F_0 of the hypothesized distribution and the empirical distribution function $F_{data}(x)$ of the observed dataset, the test statistic (D_n) can be calculated by Equation (IV.1), where sup_x is the *supremum* of the set of distances

and n is the size of the data. According to [98], if the sample comes from distribution $F_0(x)$, then D_n converges to 0 almost surely in the limit when n goes to infinity.

$$D_n = sup_x |F_0(x) - F_{data}(x)| \tag{IV.1}$$

IV.4.3 Workload Forecasting

Barista uses an online rolling window based forecasting methodology to predict workload in order to allocate resources proactively. Request forecaster is composed of two main components: ① *Forecaster*, which is responsible for modeling both periodic and non-periodic elements associated with time-varying workloads, and ② *Compensator*, which modifies the forecast produced by the first component according to the last five forecast errors. These two components are briefly described as follows:

IV.4.3.1 Forecaster

A time-varying workload can be composed of three main elements - *trend*, *seasonality* and *holidays* [99, 93], and they are combined as shown in Equation (IV.2) below:

$$y(t) = g(t) + s(t) + h(t) + \varepsilon_t \tag{IV.2}$$

where $g(t)$, $s(t)$, $h(t)$ model non-periodic changes, periodic changes (e.g., daily, weekly and yearly seasonality), and effects of holidays which occur on potentially irregular schedules over one or more days, and ε_t represents noise.

The trend function ($g(t)$) models how the workload has grown in the past and how it is expected to continue growing. Modeling web traffic is often similar to population growth in natural ecosystem, where there is a non-linear growth that saturates at carrying capacity [93]. This kind of growth is typically modeled using the logistic function shown in Equation (IV.3), where C is the carrying capacity, k is the growth rate, and m is an offset

parameter.

$$g(t) = \frac{C}{1+\exp(-k(t-m))} \tag{IV.3}$$

The seasonality function ($s(t)$) models multi-period seasonality that repeats after a certain period. For instance, a five day work week can produce effects on a time series that repeat each week. A standard Fourier series, as shown in Equation (IV.4), is used to provide a flexible model of periodic effects [100], where P is the expected time series period and N is the order.

$$s(t) = \frac{1}{2}a_0 + \sum_{n=1}^{N}\left[a_n \cos(\frac{2\pi nt}{P}) + b_n \sin(\frac{2\pi nt}{P})\right] \tag{IV.4}$$

The holiday function ($h(t)$) models the predictable variations in workload caused due to holidays. However, these variations do not follow a periodic pattern, so their effects are not well modeled by a cycle. The holidays are added in the form of a list and are assumed independent of each other. An indicator function is added for each holiday that shows the effect of a given holiday on the forecast. Barista leverages Prophet [93] for implementing Forecaster.

IV.4.3.2 Compensator

This component adjusts the output of the forecaster based on the past forecast errors. It can be modeled as a transformation function c, which changes the output of the forecaster y based on the errors from the last m forecasts $E = \{e_1, e_2, \ldots, e_m\}$, as shown in Equation (IV.5) below:

$$y' = c(y, y_{upp}, y_{low}, E) \tag{IV.5}$$

where y_{upp} and y_{low} are the upper and lower estimation bounds of y. The transformation can be learned using data-driven methods. In Barista, we use H2O's AutoML framework [101] to find the best hyper-parameter tuned algorithm.

IV.4.4 Resource Estimation

Resource estimation is one of the two main tasks performed by the resource manager. Depending upon the forecasted value, the SLO, and the service type, the resource manager, solves the static VM deployment problem described in Section IV.3.2. Due to the NP-hardness of the problem, this subsection presents a greedy heuristic to perform static VM deployment.

For each VM configuration vm_i, we can compute the number of requests n_req_i it is able to serve for a deep learning prediction service while meeting the SLOs:

$$n_req_i = \begin{cases} \lfloor \frac{\lambda}{t_{p_i}} \rfloor, & \text{if } mem_i \geq min_mem \\ 0, & \text{otherwise} \end{cases}$$

Recall that λ is the model's SLO timing constraint, min_mem is the model's minimum memory requirement, t_{p_i} is the latency to serve each request of the model using configuration vm_i with p_i CPU cores, and mem_i is amount of memory available in vm_i. We can then define the *cost per request* for each configuration vm_i as follows:

$$cpr_i = \frac{cost_i}{n_req_i}$$

Let i^* denote the index of the VM configuration with the minimum cost per request, i.e., $cpr_{i^*} = \min_{i=1...m}\{cpr_i\}$. Clearly, given an estimated workload y' from the output of Equation (IV.5), an optimal rational solution will deploy $\alpha^* = \frac{y'}{n_req_{i^*}}$ VMs of configuration vm_{i^*} and incurs a total cost:

$$total_cost^* = \frac{y'}{n_req_{i^*}} \cdot cost_{i^*} \tag{IV.6}$$

To find the optimal integral solution is unfortunately NP-hard (via a simple reduction from the knapsack or the subset sum problem). However, Equation (IV.6) nevertheless serves as

a lower bound on the optimal total cost.

To solve the integral problem, our greedy algorithm also chooses a single configuration vm_{i^*} that has the minimum cost per request while breaking ties by selecting the configuration with a smaller deployment cost. Thus, it deploys $\alpha = \lceil \frac{y'}{n_req_{i^*}} \rceil$ VMs of configuration vm_{i^*} for serving y' requests, and incurs a total cost that satisfies:

$$\begin{aligned} total_cost &= \lceil \frac{y'}{n_req_{i^*}} \rceil \cdot cost_{i^*} \\ &< \left(\frac{y'}{n_req_{i^*}} + 1 \right) \cdot cost_{i^*} \\ &= total_cost^* + cost_{i^*} \end{aligned} \tag{IV.7}$$

Equation (IV.7) shows that the total cost of the greedy algorithm is no more than the optimal cost plus an additive factor $cost_{i^*}$. When serving a large number of requests, the incurred cost is expected to be close to the optimal. Furthermore, the algorithm always deploys VMs from the same configuration regardless of the number of requests to be served. This makes it an attractive solution for handling dynamic workload variations without switching between different VM configurations. The complete algorithm is illustrated in Algorithm 3.

IV.4.5 Resource Provisioner

Resource provisioning is a critical process running inside the Barista resource manager. Its main objective is to scale the resources proactively to handle the dynamic workload. Barista resource provisioner intelligently frees acquired resources for latency-sensitive services when the predicted workload is low, i.e., horizontal scaling down. And when the workload increases, the resource provisioner acquires new resources while taking into account already freed resources. We use two sets, κ and ψ, to indicate VMs that are used for latency-sensitive prediction services. The VMs in κ denote resources that are actively used for serving latency-sensitive prediction services while the VMs in ψ are those that have been freed by resource provisioner.

Algorithm 3: Resource Estimation

```
1  Initialize: i* ← 0, cpr* ← ∞, cost* ← ∞, n_req* ← 0
2  for i = 1 to m do
3      t_{p_i} ← getExecutionTime(vm_i, model)
4      mem_i ← getMemory(vm_i)
5      cost_i ← getCost(vm_i)
6      if mem_i ≥ min_mem then
7          n_req_i ← ⌊λ / t_{p_i}⌋
8          cpr_i = cost_i / n_req_i                    ▷ Cost per request
9          if cpr_i < cpr* then
10             i* ← i
11             cpr* ← cpr_i
12             n_req* ← n_req_i
13             cost* ← cost_i
14         else if cpr_i = cpr* & cost_i < cost* then
15             i* ← i
16             n_req* ← n_req_i
17             cost* ← cost_i
18         end
19     end
20 end
21 Deploy α ← ⌈y' / n_req*⌉ VMs of configuration vm_{i*}
```

Algorithm 4: Resource Provisioning

1. Initialize: $Flag \leftarrow True$, $\alpha \leftarrow 0$, $n_req_{i^*} \leftarrow 0$, $i^* \leftarrow 0$, $params \leftarrow \{model, \lambda, min_mem, mem_i, p_i, cost_i, \forall i = 1 \ldots m\}$, $\kappa \leftarrow \emptyset$, $\psi \leftarrow \emptyset$, H_{cdl}, H_{mld}, H_{exp}
2. **while** $True$ **do**
3. $\quad t \leftarrow$ GetCurrentTime()
4. $\quad y' \leftarrow$ GetForecast(t, t'_{setup})
5. $\quad$ **if** $Flag$ **then**
6. $\quad\quad i^*, n_req_{i^*} \leftarrow$ ResourceEstimation$(params)$
7. $\quad\quad Flag \leftarrow False$
8. $\quad$ **end**
9. $\quad \alpha \leftarrow \lceil \frac{y'}{n_req_{i^*}} \rceil$
10. $\quad C^{\kappa}_{exp}, C^{\psi}_{exp} \leftarrow$ GetExpireVMCount$(t + t'_{setup}, \kappa, \psi)$
11. $\quad \delta \leftarrow \alpha - (|\kappa| - C^{\kappa}_{exp})$
12. $\quad$ **if** $\delta > 0$ **then**
13. $\quad\quad \delta_{scaled} \leftarrow \min\left(|\psi| - C^{\psi}_{exp}, \delta\right)$
14. $\quad\quad \delta_{new} \leftarrow \max\left(0, \delta - \delta_{scaled}\right)$
15. $\quad\quad$ **for** $i = 1$ *to* δ_{new} **do**
16. $\quad\quad\quad IP \leftarrow$ DeployVM(i^*)
17. $\quad\quad\quad \kappa$.add(IP)
18. $\quad\quad\quad H_{cdl}[t + t_{vm}] = (IP, model)$
19. $\quad\quad\quad H_{mld}[t + t_{vm} + t_{cd}] = (IP, model)$
20. $\quad\quad\quad H_{exp}[t + \tau_{vm}] = (IP, model)$
21. $\quad\quad$ **end**
22. $\quad\quad$ HScaleUp$(\delta_{scaled}, \kappa, \psi)$
23. $\quad$ **else**
24. $\quad\quad$ HScaleDown(δ, ψ, κ)
25. $\quad$ **end**
26. $\quad C \leftarrow H_{cdl}[t]$
27. $\quad M \leftarrow H_{mld}[t]$
28. $\quad E \leftarrow H_{exp}[t]$
29. $\quad$ **foreach** $c \in C$ **do**
30. $\quad\quad$ DownloadContainer$(c.IP, c.model)$
31. $\quad$ **end**
32. $\quad$ **foreach** $m \in M$ **do**
33. $\quad\quad$ LoadModel$(m.IP, m.model)$
34. $\quad$ **end**
35. $\quad$ **foreach** $e \in E$ **do**
36. $\quad\quad$ ModelUnload$(e.IP, e.model)$
37. $\quad\quad$ TerminateVM$(e.IP)$
38. $\quad$ **end**
39. $\quad$ LoadBalancerUpdate()
40. $\quad$ WakeUpAtNextTick()
41. **end**

Resource provisioner is implemented as a daemon process that is invoked at a fixed interval of time, as shown in Algorithm 4. On each invocation, the resource manager obtains a workload forecast t'_{setup} timesteps into the future [Line 4], where t'_{setup} is the accumulation of infrastructure provisioning time (t_{setup}) and forecasting time ($t_{forecast}$). The required number of VMs (i.e., α) can be calculated based on the forecasted workload, as indicated in Algorithm 3. The heuristic presented in Section IV.4.4 depends upon the SLO and cost per request; the best VM configuration will remain fixed as long as these two factors are unaltered. Thus, the best VM configuration index and the maximum number of requests served per VM are calculated once and stored in variables i^* and n_req^*, respectively [Lines 5-8]. After obtaining the required number of VMs α [Line 9], the difference between α and $|\kappa|$ is calculated to find the net number of VMs needed at timestep $t + t'_{setup}$. For this difference, C^{κ}_{exp} (the number of VMs that will expire at time $t + t'_{setup}$) is subtracted from $|\kappa|$ to compensate for the VMs that will become unavailable due to lease expiration. The final difference value is referred to as δ [Lines 10-11].

A positive value of δ implies more VMs will be required at $t + t'_{setup}$. This requirement is fulfilled by: ① re-acquiring δ_{scaled} freed VMs, and ② spawning δ_{new} new VMs, such that $\delta = \delta_{scaled} + \delta_{new}$ [Lines 13-22]. A negative value of δ signifies the abundance of VMs at timestep $t + t'_{setup}$ which leads to freeing VMs [Lines 23-25]. Here, *HScaleUp* and *HScaleDown* functions scale the resources by re-acquiring δ_{scaled} VMs from κ to ψ and freeing δ VMs from κ and ψ, respectively. New VMs are spawned by scheduling container download, model loading and lease expiration events at timestamps $t + t_{vm}$, $t + t_{vm} + t_{cd}$ and $t + \tau_{vm}$, respectively [Lines 16-20]. Here, H_{cdl}, H_{mld} and H_{exp} are used to register events related to downloading containers, loading models and terminating VMs as key-value pairs, where the timestamps are the keys and the tuple $(IP, model)$ is the value. Apart from scheduling VMs for the future, resource provisioner also initiates the routines for downloading containers and loading models from the previous forecasts at the current timestamp t, followed by terminating VMs whose leases expire [Lines 29-38]. In the end,

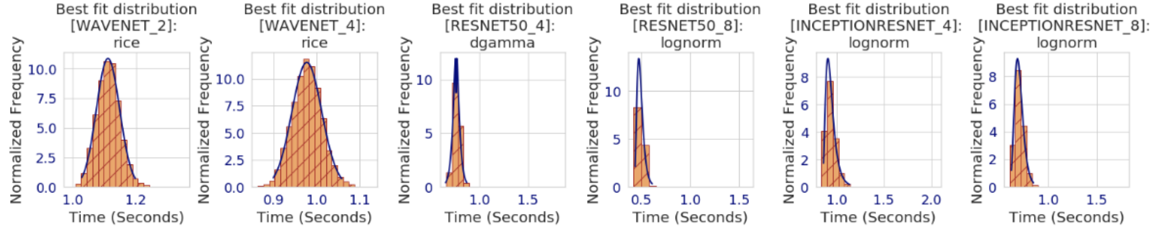


Figure IV.6: Top ranked distribution that describes the variation in the sample data. The distribution (blue) is plotted on top of the histograms (orange) of observations.

resource provisioner updates load balancer for newly deployed VMs, and sleeps till the next tick [Lines 39-40].

We vertically scale down the number of cores of a particular container if we meet the SLO with some threshold margin, and share the cores with the batch jobs. Vertical scaling is also helpful in allocating more resources for sudden workload surges. Moreover, if the resource estimator over-estimates the resources and whenever our prediction services can de-allocate cores while maintaining the SLO, Barista frees up cores, and if we miss any SLO, Barista will increase the number of cores immediately if more cores are available. We de-allocate one core at a time to minimize latency miss, and we double the number of cores (within maximum core limits of the VM) for the prediction service if there is any SLO miss.

IV.5 Evaluation

We now empirically validate the Barista framework.

IV.5.1 Experiment Setup

Our testbed comprises an OpenStack Cloud Management system running on a cluster of AMD Opteron 2300 (Gen 3 Class Opteron) physical machines. We emulated the VM configurations as per the Amazon EC2 pricing model[3] [102]. We used the AWS pricing

[3] We considered only forty-seven different configurations with two, four, and eight cores. Any GPU-based or SSD-based configuration was not considered.

model to emulate our pricing model[4]. We employed DockerSwarm [103] as our container management service on top of the VMs. HAProxy (http://www.haproxy.org) was used as our frontend and backend load balancers. We built a NodeJS-based frontend web application to relay the query to the backend predictive analytics service.

IV.5.2 Predicting Execution Time of Predictive Analytics Services

In this study, we considered two different kinds of predictive analytics applications: image recognition and speech recognition based on six different pre-trained deep learning models: Xception, VGG16, InceptionV3, Resnet50, InceptionResnetV2, and Wavenet (see Figure IV.1). All these models were profiled on OpenStack VMs of variable numbers of VCPU cores. Based on the data generated from profiling these models, the best distribution was estimated from a list of available probability distributions. All the empirical distributions closely resembled the hypothesized distributions. The best-fit distribution for Wavenet service on two and four cores, Resnet50 service on four and eight cores, and InceptionResnetV2 service on four and eight cores as a sample shown in Figure IV.6. Based on the best-fit distribution, we calculate 95th percentile latency for each service.

IV.5.3 Workload Forecasting

Two different time-series datasets are used to emulate a realistic workload for predictive analytics services. The first dataset is collected and published by NYC Taxi and Limousine Commission [104]. We processed the data to extract the number of cab requests generated every minute based on pick-up and drop-off dates and times [104]. This dataset is an appropriate workload for a speech recognition component in a ride-sharing application to request a ride. The second dataset contains data on the number and types of vehicles that entered from each entry point on the toll section of the Thruway with their exit points [105]. This data can be used to represent a real-world workload of an image recognition based

[4] We did not run the experiments on Amazon cloud because of monetary constraints.

predictive analytics service that aims to automatically detect the license plate number of the entering or leaving car from a toll plaza. We processed the data to extract the total number of cars entering a toll plaza every minute.

A total of 10,000 data-points from each dataset was used in our study. We utilized 6000, 500, 2500 data points for training, validation, and testing the Prophet-based Forecaster model in both datasets, respectively. We performed *hyper-parameter tuning* for Fourier series order, N, by iterating over five different values, [10, 15, 20, 25, 30], and with different sizes of training window,W [4000, 5000, 6000]. Out of 15 possibilities, the configurations with $(N=30, W=6000)$ and $(N=20, W=6000)$ produced the least absolute percentage error (95th percentile) for first and second data sets respectively. The mean absolute error and absolute percentage error (95th percentile) for first and second datasets are (27.66, 29%), and (27.84, 30.26%), respectively.

In Barista, we extended Prophet with a machine learning-based compensator to adjust the forecast based on the last five prediction errors. We determined 'five' errors to be considered based on empirical results. Apart from five past prediction errors, the recent forecast of Prophet along with upper and lower estimation bounds are used as features for learning the model. We used the data points of Prophet (3000 points) to train the compensator model, and another 1000 points were used to test and analyze our hybrid approach with Prophet. We used H2O's implementation of AutoML framework [101] to identify the best family of the learning algorithm and tune hyperparameters. *XGBoost based gradient boosted trees* outperformed other machine learning models such as Neural Networks, Random forest model, and selected as the best model. The training, cross-validation, and testing mean absolute errors for first and second datasets are (12.65, 15.10, 21.26), (12.24, 15.13, 22.65), respectively. The forecasting results of Barista and Prophet, along with the actual workloads, are shown in Figure IV.7 and IV.8.

It is visible from the figure that the Barista prediction curve closely resembles the actual workload and predicts the sudden burst of requests with reasonable accuracy as compared

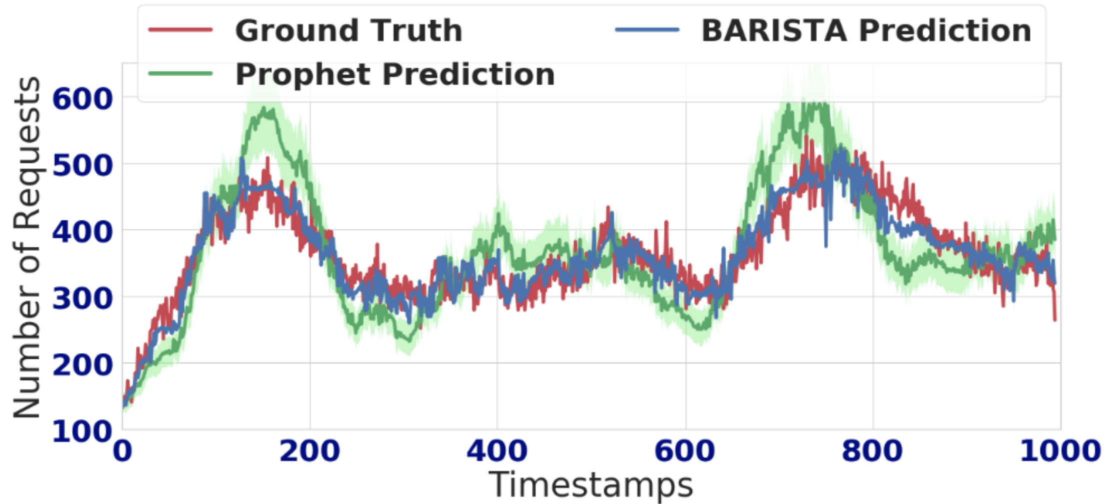


Figure IV.7: Performance comparison of Barista (blue) and Prophet(green) along with ground truth (First Dataset) (red)

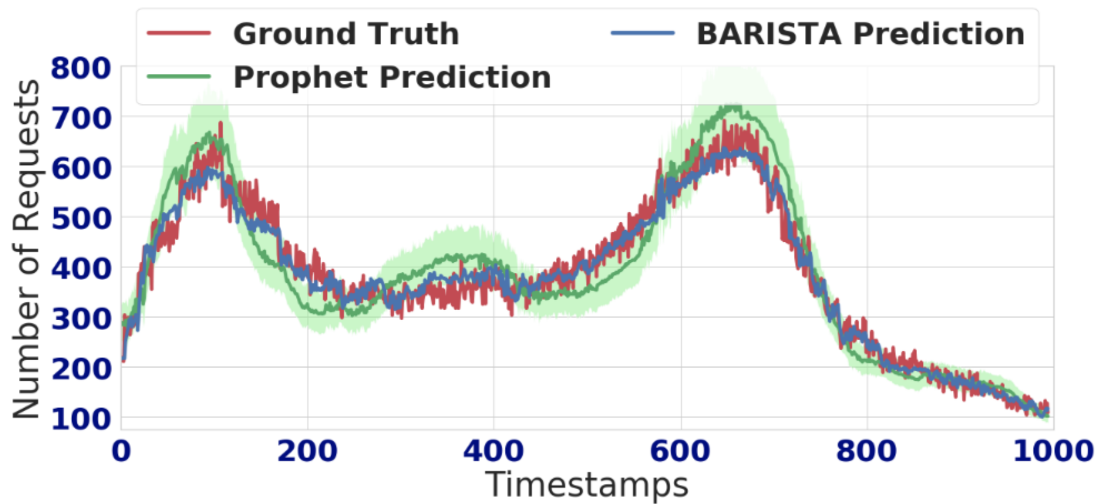


Figure IV.8: Performance comparison of Barista (blue) and Prophet(green) along with ground truth (Second Dataset) (red)

to Prophet, which often lags and leads. Figure IV.9 and IV.10 shows the cumulative percentage error distribution of both approaches on the test set of thousand points. Barista outperforms Prophet by 37 and 46% in first and second data sets, respectively.

IV.5.4 Resources Selection and Provision

Barista makes the resource selection and provisioning based on algorithms described in section IV.4.4 and IV.4.5. We evaluated our resource estimator and provisioner at different

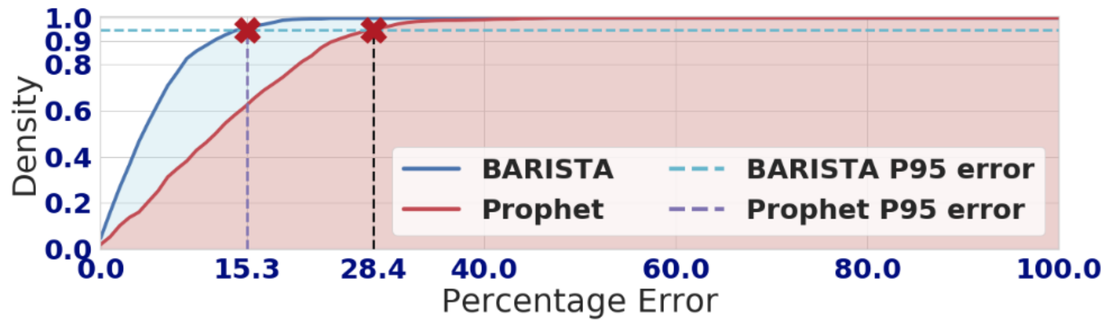


Figure IV.9: Cumulative Absolute Percentage Error Distribution of First Dataset

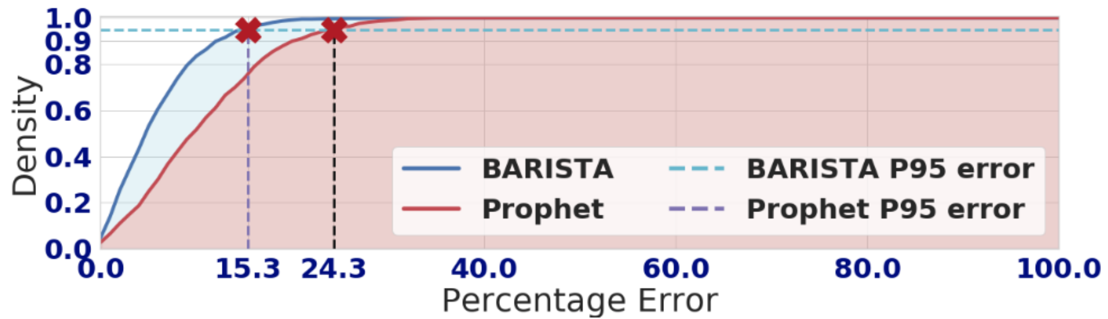


Figure IV.10: Cumulative Absolute Percentage Error Distribution of Second Dataset

time points in the life cycle of prediction services. We uniformly distributed the workload traces from one minute to five seconds for our experiment. We met the target SLO (two seconds and 1.5 seconds respectively) 99% of the time over 12000 seconds, as shown in Figure IV.12 and IV.13 for the workload datasets. However, the SLO (two seconds) compliance rate marginally dropped to 97%, as shown in Figure IV.14.

Barista not only guarantees SLO compliance but also minimizes running and management costs by intelligently selecting the VM type and provision these VMs. Even if assigning more cores reduces the running time of the prediction services, selecting VMs with the highest number of cores is not always the best option. As mentioned before in Section IV.4.4, we consider the price of the Amazon EC2 instances for solving the resource estimation problem. Barista selected VM configuration depends on SLO, cost, and estimated execution time of the prediction service. We considered VM expiration time on an hourly basis (instance hour as an example scenario) and emulated the prices of VMs accordingly. For *configuration 1*, we considered t3.2xlarge VMs, for *configuration 2*, we

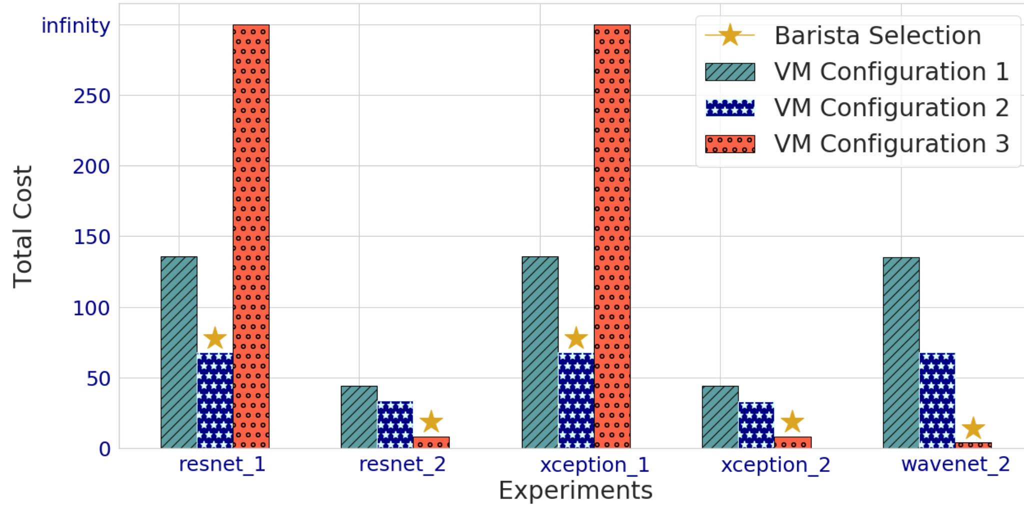


Figure IV.11: Cost comparison between multiple VM configurations (Cost infinity means the VM is infeasible option, it cannot serve the request within the SLO bound)

considered t3.xlarge VMs, and for *configuration 3*, we considered t3.small VMs(as our min_mem constraint is 2 GB). We solved the optimization problem for given SLO bound to select the VM type; we considered this VM type as one of the configurations for our experiment. Then we considered the other two VMtypes of the same VM group(Here group means Amazon EC2 instance types groups e.g., general-purpose(t3, compute-intensive(c4-5 group), etc.) with different core capacities. In the figure IV.11, we have shown the total cost for hosting the backend VMs for 10 hours(600 mins), while guaranteeing the SLO bound for the workload traces [104, 105]. Because of the cost difference of different VM configurations, selection of the most powerful VM type is not a good option always; as shown here, more number of low-powerful VM configurations can be better. This happens because prediction services are stateless, and it utilizes all the cores available in that VM/machine, and these jobs are always served sequentially. We observed Barista could perform $50-95\%$ better than the naive approach.

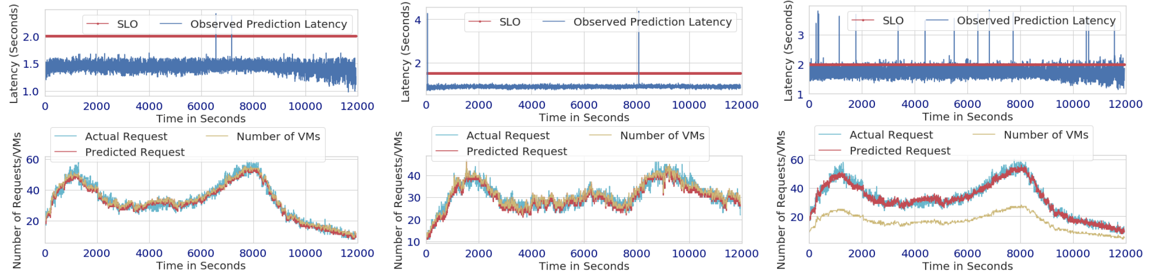


Figure IV.12: The upper image shows how we guaranteed 2 seconds SLO for Resnet Prediction service and the experienced latency by selecting Barista selected VM configuration on toll dataset, Lower image shows the actual request rate, predicted request rate, and number of allocated VMs (t3.small (2cores)).

Figure IV.13: The upper image shows how we guaranteed 1.5 seconds SLO for Wavenet Prediction service and the experienced latency by selecting Barista selected VM configuration on taxi dataset, Lower image shows the actual request rate, predicted request rate, and number of allocated VMs (t3.small (2cores)).

Figure IV.14: The upper image shows how we guaranteed 2 seconds SLO for Xception Prediction service and the experienced latency by selecting Barista selected VM configuration on toll dataset, Lower image shows the actual request rate, predicted request rate, and number of allocated VMs(t3.xlarge (4cores)).

Figure IV.15: Barista Performance Results on selected VM configuration as backend

IV.5.5 Reactive Vertical Scaling for Model Correction

We monitored the latency of the services at every five seconds and take decisions accordingly based on the monitored latency and the SLO bound. We considered a deployment scenario where prediction services can run with other low-priority co-located services, so our goal here to demonstrate that we can vertically allocate and de-allocate CPU cores by monitoring the SLO.

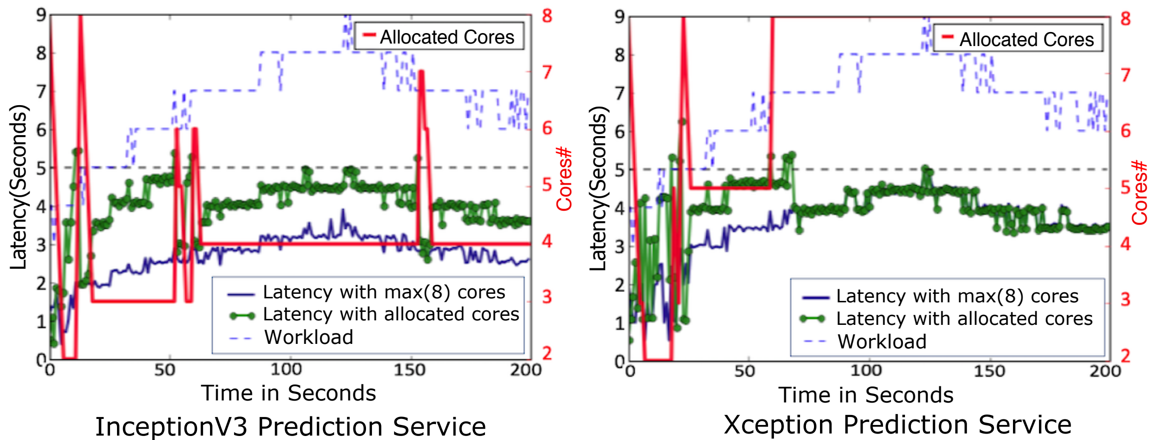


Figure IV.16: Vertical Scaling to allocate the number of CPU cores(red line) while maintaining the SLO bound of 5 seconds. The blue dotted line shows the workload pattern, and solid navy blue line shows the latency of the prediction services if run on maximum allocated cores on a VM of 8 cores. The green line shows the latency if we dynamically (de)-allocate the cores.

Figure IV.16 shows how over-provisioning for two prediction services (Xception and

InceptionV3) could be handled reactively by adjusting the CPU cores at runtime. Using our reactive approach for vertical scaling, we saved approximately 15% and 30% of CPU shares of an eight cores OpenStack VM for Xception and InceptionV3, respectively, on a particular workload trace. Our reactive approach also achieves over 98% of the SLO hits while optimizing the CPU shares significantly. We also observed similar behaviors for other prediction services, thus demonstrating the capabilities of Barista to make the model correction if there is any resource over-provisioning due to over-estimated time-series prediction.

IV.6 Conclusion

IV.6.1 Summary

Predictive analytics services based on deep learning pre-trained models can be hosted using serverless computing paradigm due to their stateless nature. However, meeting their service level objectives, i.e., bounded response times and bounded hosting costs, is a hard problem because workloads on these services can fluctuate, and the state of infrastructure can result in different performance characteristics. To resolve these challenges, this chapter describes Barista, which is a dynamic resource management framework providing horizontal and vertical auto-scaling of containers based on predicted service workloads. Selecting an optimal cloud configuration is an NP-hard problem; hence we proposed a heuristic to select proper cloud configuration, which is minimizing the cost while maintaining the prediction latency bound (SLO).

IV.6.2 Discussions

The Barista approach can broadly apply to other compute-intensive and parallelizable simulation services where the model needs to be loaded in memory first, and the behavior of the simulation determined based on users' request and user-specified SLO. In this

chapter, we did not consider running different co-located prediction services together on the same machine, where workload patterns for each prediction service can be different. Vertical scaling the different prediction services at the same time on the same machine is the dimension of future work.

CHAPTER V

DEEP-EDGE: AN EFFICIENT FRAMEWORK FOR DEEP LEARNING MODEL UPDATE ON HETEROGENEOUS EDGE

V.1 Introduction

V.1.1 Emerging Trends

The past decade has seen substantial progress in *Deep Learning* (DL) [67], particularly *Deep Neural Networks* (DNNs), leading to its widespread adoption in various domains, such as medicine [106], geology [107] and vehicular navigation [108]. With the advent of the Internet of Things, intelligent edge devices gather and analyze data streams continuously based on trained DL models. Traditionally, the model updating process happens in the cloud where the data collected from the edge of the network is transferred to the cloud data-centers. Once the models are trained, these prediction models are seamlessly integrated into the edge-based applications to predict outcomes based on new input data. These DL models perform exceedingly well in terms of accuracy on the trained and validation data.

V.1.2 Challenges and State-of-the-Art Solutions

Despite advances in DL technology, the predictive analytics-based application that is composed of a deep learning component can experience a reduction in accuracy over time due to changes in the input data distribution [13]. This phenomenon is referred to as *Concept Drift* [12]. In order to overcome model staleness and incorporate changes due to input data streams, *continual learning* [109]has been used to periodically refine the static models by re-training the existing model using the recent data. Figure V.1 shows the lifecycle of a machine learning model in production, where an inference API hosts the DL model.

Recent data, along with the predicted and actual labels are stored in a data store that is fed to the model update process based on a user-defined trigger to replace the stale model with an updated one.

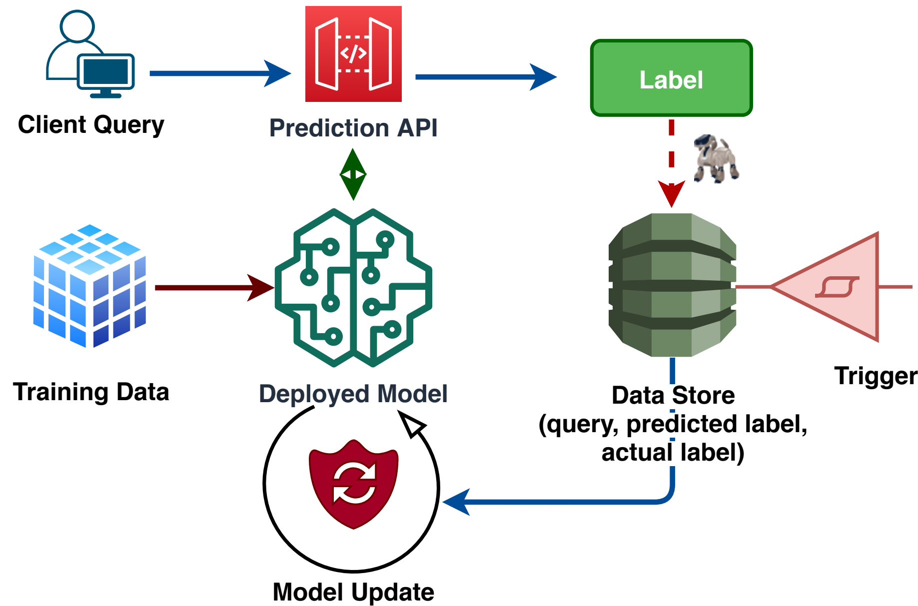


Figure V.1: Life Cycle of Machine Learning Task

However, re-training of the DL model is a highly resource-intensive and time-consuming task even with parallelization techniques. Various distributed DL frameworks, such as Tensorflow [110], MXNET [111], Ray [112], have been developed to reduce the training time by distributing the training workload among multiple machines (cluster) consisting of one or more Graphics Processing Units (GPUs) or Application Specific Integrated Circuits (ASICs) such as Tensor Processing Units (TPUs). With its resource-rich environment and elastic capabilities, the cloud provides the right platform for the DL model re-training. In this context, distributed training of DL models using powerful GPU clusters in cloud data centers has been studied extensively in the literature [113, 114, 115]. Distributed training of *Deep Neural Networks (DNNs)* is performed across multiple worker machines with many possibilities for parallelization. The predominant methods of parallelization are (1) *data-parallelism*, where each worker is responsible for training on a shard of the data set; (2) *model-parallelism*, where each worker is responsible for training on a shard of the model;

and (3) a hybrid approach of the two. In this chapter, we mainly focus on data-parallel distributed training.

However, the availability of powerful and reliable GPU-accelerated edge AI products, such as NVIDIA's Jetson family [116] (TX2, Nano, Xavier) and Google's edge TPU [117] (Coral), has made continual learning viable using edge devices. In particular, the edge devices are suitable for performing the model update due to the following reasons:

① The computational power of edge devices is sufficient for updating small to medium-sized DNNs (up to 150 million parameters), such as VGG, Resnet, Inception, Mobilenet, Densenet.

② The duration of the model update task is far less than the initial model training time, as fewer full data iterations (epochs) are required.

③ Performing model update at the edge avoids costly data transfers to the cloud.

④ Using edge devices for the model update also handles data privacy concerns and reduces data security threats.

Edge computing has been discussed primarily in the literature to run prediction tasks for DL applications to achieve low latency and bandwidth savings [118, 119, 11]. In this chapter, we study methodologies to update DL models on edge devices in a distributed manner to reduce training time and to increase throughput. Most prior works on distributed learning (be it in the cloud or edge) assume a homogeneous cluster, where workers possess similar computation and network capacity [120]. However, most of the approaches do not apply to edge clusters due to the higher level of heterogeneity among the edge devices. The heterogeneity can be a result of the different physical characteristics of the devices, such as the number of processors, CUDA cores, memory, etc. or due to the workload associated with the devices.

Furthermore, besides training jobs, the edge devices are also assigned to perform latency-critical jobs, such as prediction or inference from the incoming image requests. These latency requirements are captured by Service-Level Objectives (SLOs), which should be met by the hosting platform. However, co-habiting DL training jobs in the background, along with such latency-critical applications, may violate their SLOs. Hence, the *sensitivity* and *degree of interference* between the different types of co-located jobs [73, 20] also need to be investigated. Due to these challenges, there is a need to develop a custom resource manager for DL model update workloads at the edge by considering the timing constraints of the background applications, the computational capabilities, and workload of the individual edge devices along with the structure and characteristics of the DL jobs.

V.1.3 Overview of Technical Contributions

In this chapter, we propose *Deep-Edge*, a custom resource management framework for DL model update jobs to minimize the model update time by distributing the re-training workloads among a set of heterogeneous edge devices while adhering to the timing and latency constraints of the background tasks. We focus on data-parallel distributed training based on the centralized parameter server architecture. Specifically, we make the following contributions:

① We define unified monitoring, profiling, and deployment framework for model update tasks at the edge.

② We build accurate performance and interference models for DL model update task and latency-critical background tasks by profiling them under various system metrics, such as CPU, GPU, Memory utilization, etc.

③ We formulate an optimization problem that incorporates the edge node selection and workload distribution decisions to minimize the overall model update time.

④ We present a polynomial-time heuristic solution based on the timing constraints, the performance, and interference models of the model update and background tasks.

⑤ We show the efficacy of the framework by evaluating the accuracy of the proposed solution using a real-world DL model update task based on the Caltech dataset and an edge AI cluster testbed.

V.1.4 Organization of the Chapter

The rest of the chapter is organized as follows: Section V.2 provides a brief introduction to DL and presents a survey of existing solutions in the literature and compares them with Deep-Edge; Section V.3 presents the motivation behind the problem formulation and solution of Deep-Edge; Section V.4 describes the problem formulation; Section V.5 discusses the design and implementation of *Deep-Edge*; Section V.6 evaluates the *Deep-Edge* framework using a prototypical case study; and inally, Section V.7 presents the concluding remarks by alluding towards future directions.

V.2 Background and Related Work

This section provides a literature survey along the dimensions of distributed deep learning, performance modeling, and resource interference, all of which are critical for the success of Deep-Edge.

V.2.1 Deep Learning Model Training

Deep Learning(DL) methods aim to learn the representation of data with multiple levels of transformation of the neural network model that is composed of multiple processing layers [67]. The training process is resource-intensive, and it may speed up by parallelization of the training in multiple devices, which can be either GPU or CPU. These devices can be placed on a single machine or across multiple machines. In the literature, multiple

approaches have been studied to distribute the deep learning training in a cluster of machines, and multiple optimization techniques have been proposed to speed up the training task [121, 113, 114, 115, 120].

V.2.1.1 Distributed Deep Learning - Data Parallelism

As mentioned earlier (Section V.1.2), the two predominant methods for parallelizing deep learning training is data parallelism and model parallelism. In this chapter, we mainly focus on the data parallelism approach; hence, we will discuss the data parallelism techniques onwards.

In data parallelism, each worker runs an identical replica of the DL model, but with the non-overlapping shard of the input data, which are further sliced up into *batches*. The processing of a batch in each worker constitutes a training *step*, which involves: 1. inferring the output and calculating a loss function for each data sample in the batch (*forward pass*); 2. determining the gradients based on the loss function, i.e., changes to be made to the parameters of the DL model (*backward pass*); and 3. updating the parameters of the shared global model after every batch, and then the updated model is used in the next round of computation. When the forward pass, backward pass, and the model update processes are completed on all the data points in the data shard, it is considered as an *epoch*. The process repeats itself until the desired accuracy is achieved. Once the training is completed on all the worker nodes, then the training job is considered as completed. The whole training duration is denoted as job completion time. There are many challenges, and state-of-the-art approaches are mentioned in the literature to update the shared model using these gradients (*parameter synchronization*) [122, 123].

There exists another classification in distributed training based on how the knowledge (parameters) learned by individual workers is shared across the group. Most DL frameworks implement either *centralized* or *decentralized* architecture for storing and sharing the updated parameters of a DL model. In *centralized* architecture, all workers compute

forward and backward passes locally and send the gradients to a central entity, designated as *parameter server*, for updating the parameters based on an optimization algorithm such *Stochastic Gradient Descent (SGD)*. Parameters are then pulled back by each worker to continue the next training step. In *decentralized* architecture, no central entity exists, and the workers exchange among each other after each batch and locally update their gradients. Decentralized architecture is not suitable for the model update at the edge because it incurs higher transfer costs due to the need to broadcast the learned gradients to all the other workers. The parameter server updates the model parameters *synchronously* or *asynchronously* using these gradients [111, 121, 113] as shown in Figure V.2.

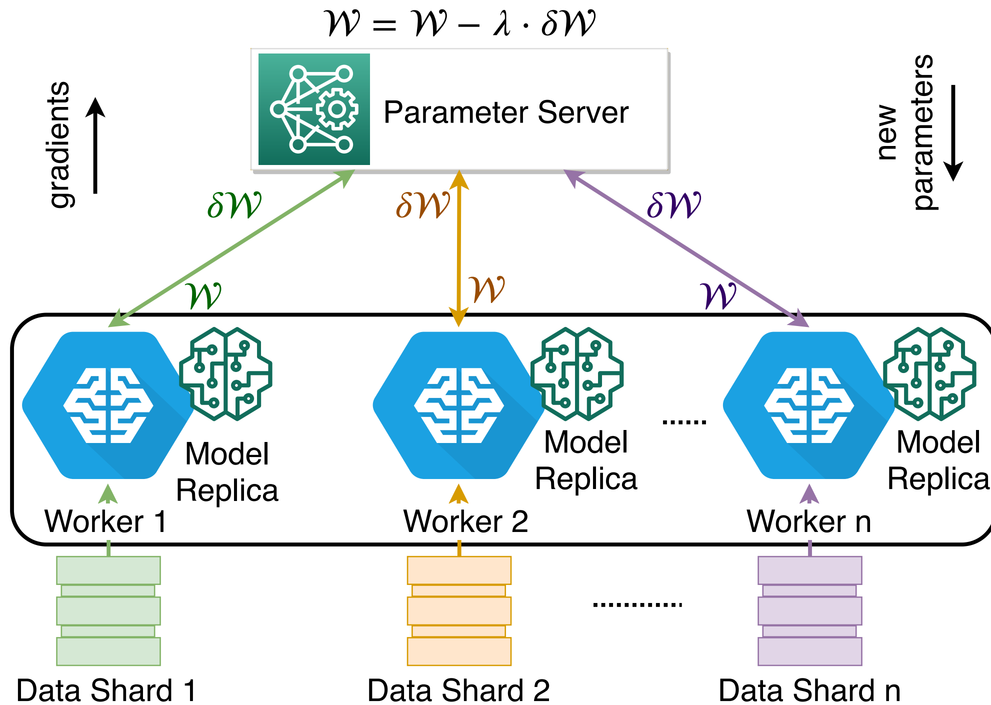


Figure V.2: Parameter Server architecture for distributed DL (data parallel) training

▷ In *synchronous training*, all workers use the same synchronized set of model parameters at the start of every batch. The synchronization cost for the heterogeneous edge cluster may be very high because of the asymmetry between the computation powers of the edge devices and the network partition, straggler problem, or node failure in the cluster. [124, 125].

▷ In *asynchronous training*, the server updates the model parameters whenever it receives the gradients from one worker. Then the worker pulls the updated parameters from servers and starts training on the next batch. It is faster than synchronous training because there is no synchronization overhead. However, in this approach, the workers can get out of sync, and compute their parameters on the stale parameters, which can delay the model convergence [120] and increases the job completion time. In practice, for asynchronous SGD, an upper bound of maximal delay can be placed to guarantee the convergence with some synchronization costs [126].

We assume that there would be a single training job at a time, and we investigate how to distribute the data into the available edge cluster to minimize the job completion time as well as synchronization delay. The data management problem refers to the mapping of training data shards to the processing nodes in the distributed infrastructure. A performance model is required to map resources to training speed. Ernest [74] proposed a methodology to estimate job completion time for tree-based ML jobs by running the experiment on small data and configuration. Optimus [113] also proposed a similar approach to predict the number of remaining iterations for DL jobs, and schedule the resources in distributed systems.

V.2.1.2 Distributed Deep Learning Task Scheduling

There are several approaches in the scientific literature for resource allocation to achieve a variety of objectives in cloud settings such as Borg [127], Coral [128], TetriSched [129], Morpheus [130]. However, the schedulers mentioned above are not designed for DL workloads. There are recent research efforts on GPU sharing for machine learning tasks. Baymax [131] explores GPU sharing as a way to mitigate both queuing delay and resource contention. Following that, Prophet [132] proposes an analytical model to predict the performance of GPU workloads. Gandiva [114] proposes GPU time-sharing in shared GPU clusters through checkpointing at low GPU memory usage of the training job. CROSS-BOW [125] proposed a dynamic task scheduler to automatically tune the number of work-

ers to speed up the training and to use the infrastructure optimally. Optimus [113] also dynamically adjusts the number of workers and parameter servers to minimize the training completion time while achieving the best resource efficiency. SLAQ [133] targets the training quality of experimental ML models instead of models in production. It adopts an online fitting technique similar to Optimus to estimate the training loss of convex algorithms. Dorm [134] uses a utilization-fairness optimizer to schedule jobs. However, these approaches are not applicable for edge clusters as none of them considers resource interference while allocating heterogeneous resources for the DL model update task.

V.2.2 Model Update Strategy

Given that there is a monitoring mechanism to detect such changes after a model is deployed in the production data pipeline, we trigger the model update to incorporate data recency to avoid model staleness. Various methods are proposed in the literature to detect model drift [135]; in this chapter, we are not focusing on how to detect the model drift. We mainly focus on how to efficiently update the model when the change is detected. Continuum [109] proposed two policies – best-effort and cost-aware, to keep the model updated. Based on a prediction model, they estimate the training cost, with which the update controller decides, for each application, when to perform updating based on the application-specified policy. However, they always incorporate the recent data, and the waiting is penalized. In this chapter, we took the update decision if the model is drifted since the last update. Once the threshold is violated, we update the DL model to prevent further degradation.

V.2.3 Resource Interference and Performance Modeling

Co-located applications on the same physical machine conflict over access for shared resources, such as CPU, memory, network devices, and impose varying degrees of stress on the underlying hardware. The contention for these resources causes *resource interference*

on the execution of co-located applications and harms their performance. *Sensitivity* measures the impact of co-located applications on the target application, and *pressure* measures the impact of the target application on the co-located applications [136, 20].

In the literature, resource interference is studied extensively, and different approaches are proposed to understand and quantify performance interference [89, 73, 83]. Performance modeling is essential to predict the impact of interference for co-located applications on different hardware configurations [89, 137, 138, 139]. Paragon [83] presents an interference-aware job scheduler to predict application performance under different stress on different target architectures and server configurations using collaborative filtering. Authors in [140] studied the impact of co-located application performance for a single multi-core machine and developed a piece-wise regression model using cache contention and bandwidth consumption of co-located applications as input features. The ESP project [141] also uses a regression model to predict performance interference for every possible co-location combination. Similarly, Pythia [139] proposed a linear regression model approach for predicting combined resource contention by training on a small fraction of the large configuration space of all possible co-locations. PARTIES [138] proposed a feedback-based controller to dynamically adjust resources between co-scheduled latency-critical applications using fine-grained monitoring and resource partitioning to guarantee the QoS. INDICES [20] proposes an interference aware fog server selection using a gradient boosting based performance model of latency-critical applications. The authors extend the same approach in [142] to offload a latency-critical task between fog and edge devices while considering user mobility. However, the effect of interference on GPU is not well-studied, in Deep-Edge we build a performance model by stressing CPU, IO, memory as well as GPU, and studied the effect of the interference on distributed deep learning job. Moreover, virtualization techniques to isolate the GPU resources for multiple processes is not supported on Tegra family NVIDIA edge devices.

V.3 Motivation

This section focuses on the motivation behind developing the Deep-Edge framework. Here, we describe how a DL model degrades in the presence of new data, and why we should update the existing DL model. We describe the motivation behind distributed training and the impact of heterogeneity and resource interference on the model update task and latency-critical tasks running in the background.

V.3.1 Motivation for Model Update

We use an example of mapping images to steering angle control in the autonomous car with CARLA [143], an autonomous driving simulator, to explain how the model degrades when the initially trained images are synthetically changed with rain ntensities [144]. In our experiments, we used the inbuilt controller to drive the car on a bright sunny day, and collect 5000 front dashboard camera images at five fps (frames per second) along with the steering data for training a neural network model. Using the labeled data, we trained an NVIDIA's DAVE-II CNN model [145] to learn the steering actions. We trained the model for 50 epochs and evaluated its performance on a validation dataset, for which we got a mean square error (MSE) of 0.0078 (lower is better). For evaluating the models' performance for different weather conditions, we chose a single straight road seen during training and then deployed the trained model to perform the steering actions. Figure V.3 shows the absolute error between the actual steering and the model steering predictions for the entire experimental evaluation. For the first 150 simulation steps, we maintain sunny weather, so the absolute error is minimal, and in the next simulation step, we change to rainy weather. With the new rainy condition, the error increases, and here we collect the images for 100 simulation steps with corrected steering value[1] and then triggered the model update. It takes about 10 seconds (50 simulation steps) to get the model updated. At step

[1]We assume that there will be a human-in-the-loop who will correct the steering angle if the DL model predicts wrong.

301, we deploy the updated model, and we see the error goes low compared to the error before it was retrained. Figure V.3 also illustrates that the error continues to be high if the model is not updated.

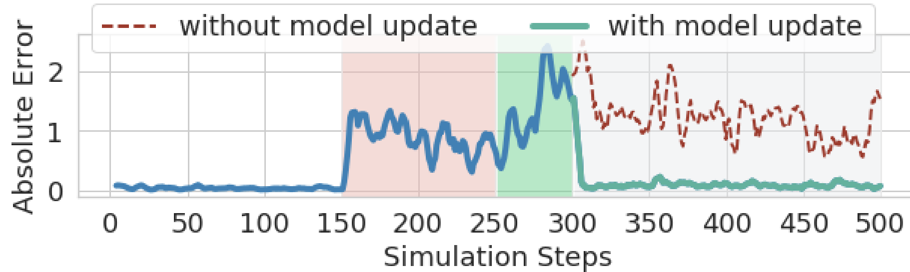


Figure V.3: Absolute error between predicted steering value and the actual steering value (with and without model update)

Similar model degradation was also observed for the DL-based image classifiers when we emulate the training on a small chunk of data from the Caltech dataset [146]. As new data arrives, the model degrades over time, and the experiment motivates the need for an online DL model updating with unseen images to make the model robust.

V.3.2 Motivation for Distributed Training

Our application is based on DL model training on Caltech dataset [146] using MxNet framework [111]. We trained our model on 3855 data points with different resource configurations and batch size.

We did our experiments on Jetson Nano devices on standalone mode and in distributed (asynchronous-centralized) mode. For the first experiment, we ran the experiment in standalone mode with 8 and 16 batch size. As shown in Figure V.4, increasing batch size reduces the per epoch training time, because of less number of updates. For the second set of experiments, we distributed the training data equally (1928 and 1927 data points respectively) on 2 Jetson Nano worker nodes and used the Jetson TX2 device as the parameter server. We changed the batch size among 8, and 16 in all worker nodes, respectively[2]. As

[2]We considered maximum batch size as 16, because Nano is limited to 4GB memory, and bigger batch size than 16 for CalTech DL model exceeds the memory of Nano device. The batch size limit for the TX2 device(8GB memory) is 64.

shown in Figure V.4, training time reduced when we distributed training over a single Jetson Nano device. For the third set of experiments, we distributed the training data equally (1285 points on each node) on 3 Jetson Nano worker nodes and used the Jetson TX2 device as a parameter server. Similarly, we observed significant improvement in training time by distributing the data points on three worker nodes over a single Jetson Nano device; however, the training time suffers significantly because of communication overhead. Therefore, we should consider the communication impact on distributing DL training in multiple machines [111].

Figure V.4: DL model training time (per epoch) for different resource configurations and different batch sizes.

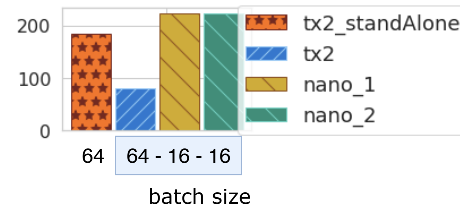


Figure V.5: DL model training time (per epoch) on standAlone TX2 and on heterogeneous cluster with different batch sizes.

V.3.3 Impact of Heterogeneity on Model Update Time

In this experiment, we trained our model on 3855 data points on a single Jetson TX2 device. Then, we distributed the training data equally (1285 points on each node) among 2 Jetson Nano worker nodes and a Jetson TX2 device. We used another Jetson Nano device as a parameter server. We conducted experiments with maximum possible batch size combinations on each device. Typically, an equal amount of data is distributed among the workers in multi-machine training. However, this approach can increase job completion time because of the heterogeneity of the edge devices. Figure V.5 illustrates that distributed training in these experiments was less efficient than doing the same training on the most powerful device, i.e., Jetson TX2. For instance, we observed that TX2 is 30% faster on an average in completing a training *step* than Nano, when we updated a state of the art Inception model [147] using Caltech-256 object category dataset. Figure V.6 shows

the cumulative distribution of time to complete one step, where the average step time for TX2 and Nano are 1.89 and 2.69 secs, respectively. Thus, equal distribution can lead to underutilization of edge resources. In the scenario mentioned above, TX2 was idle for a significant amount of time. Hence, finding the proper ratio of data and map that among the heterogeneous devices is challenging, and this motivates our study to find the near-optimal data ratio to minimize the overall job completion time.

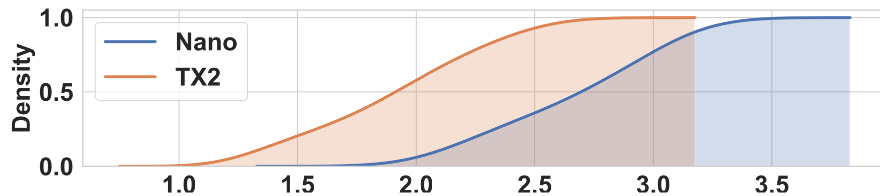


Figure V.6: Variation of step time w.r.t device type

Moreover, the performance of the model update task can be impacted by the state of the node (e.g., CPU, GPU, Memory Utilization) as shown in Figure V.7 where an initial GPU utilization of 88% and 66% increases the average step time by almost 20%. Hence, an intelligent data sharding policy is required, which considers the current state of the workers.

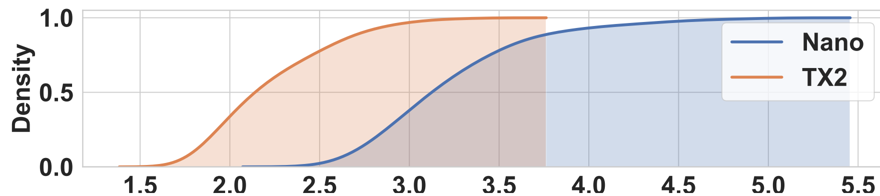


Figure V.7: Increase in step time due to resource contention

V.3.4 Impact of Resource Interference on Background Tasks

The selection of a worker to participate in the model update task depends upon the degree of interference that can be tolerated by the workers' background tasks. Figure V.8 shows the changes in three system metrics (GPU, CPU, and memory utilization) after running a model update task on two workers (TX2 and Nano). The updated system state (defined in terms of the system metrics) can lead to deadline violation of a latency-critical

background task. Moreover, adding more workers can reduce the training throughput, as described in [113]. Hence, a resource scheduler for a DL task needs to find the optimal number of workers, along with the ideal data shards, without violating the SLO constraints of background tasks.

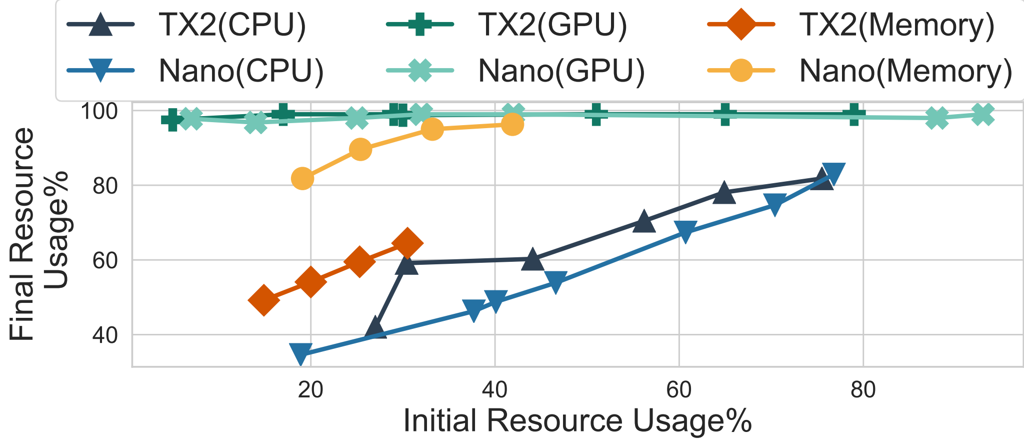


Figure V.8: Initial and Final resource Usage along multiple resource dimensions.

V.4 Problem Formulation

In this chapter, we consider distributed data-parallel training of DL models using a centralized parameter server architecture with asynchronous training loops. This section models the various costs involved in the DL model update process, and formulates an optimization problem to minimize the overall cost, and states our assumptions.

V.4.1 Cost Models

We consider a set $\mathcal{W} = \{w_1, w_2, \ldots, w_N\}$ of N heterogeneous edge nodes (or workers) that can perform distributed training, and a set $\mathcal{M}$ of data samples for training a DL model. Let $\mathcal{D} = \{d_1, d_2, \ldots, d_N\}$ denote the size of *data shards* among all the workers, i.e., the number of data samples assigned to each worker, such that $\sum_{i=1}^{N} d_i = |\mathcal{M}|$.

V.4.1.1 Data Transfer Cost

All data samples are assumed to be initially stored in a data store $ds \in DS$, where DS is the set of data stores, which need to be transferred to each edge worker for training. Let $transfer_i^{ds}$ denote the cost of transferring a single data sample from data store ds to edge node w_i. Thus, the total transfer cost for node w_i is given by $Transfer_i^{ds} = d_i \cdot transfer_i^{ds}$.

V.4.1.2 Initialization Cost

After receiving the data samples, each edge node w_i incurs a one-time initialization cost, denoted by $Initialize_i$, before the training begins. This cost is DL framework dependent and mainly consists of data pre-processing (un-packing), loading the DL model, setting up the logical DL cluster, etc.

V.4.1.3 Training Cost

The data shards at each worker w_i are further divided into *batches* of size b_i, and let $\mathcal{B} = \{b_1, b_2, \dots, b_N\}$ denote the set of batch sizes for all workers. The cost to process each batch includes the time to do forward propagation (for computing the loss function) and the time to do backward propagation (for computing the gradients). Let $forward_i$ denotes the forward propagation time for one data sample on worker w_i, and let $backward_i$ denote the backward propagation time, which is typically incurred once per batch and is not related to the size of the batch. Given a batch size b_i, the per-sample compute time on worker w_i is then given by $t_i^{compute} = forward_i + backward_i / b_i$.

After processing each batch, each worker w_i pushes the gradients to a centralized parameter server ps for update, and then pulls the updated parameters before continuing to train on the next batch. Let $push_i^{ps}$, $update^{ps}$ and $pull_i^{ps}$ denote the time to push, update and pull the parameters, respectively. Then, the update time is given by the sum of these three times, i.e., $t_i^{update} = push_i^{ps} + update^{ps} + pull_i^{ps}$. Since each worker w_i has $B_i \approx d_i / b_i$ batches, the total time to process all the data samples on the worker, called an *epoch*, is

given by:

$$epochTime_i = B_i\Big(b_i \cdot t_i^{compute} + t_i^{update}\Big).$$

Note that the per-sample compute time $t_i^{compute}$ to perform forward and backward propagation depends on the computing capability of the individual worker as well as the background tasks running on the worker. Further, the update time t_i^{update} to perform push, update, and pull on each worker is also not fixed. It depends on the batch size, the total number of deployed workers as well as the state of the workers, and the parameter server.

V.4.1.4 Total Cost

The total cost includes the data transfer and initialization costs for all workers, followed by the asynchronous training and update costs from different workers.

As the set of workers is assumed to be heterogeneous, some of them may not be deployed for training (e.g., due to high data transfer cost or low computational capability). Typically, having more workers will reduce the workload of each participating worker; hence the epoch time may be reduced. However, it may also increase the update time (i.e., t_i^{update}) due to resource contentions caused by different workers trying to update the parameters at the same time.

Let $\gamma_i \in \{0,1\}$ denote a binary variable indicating if worker w_i will be deployed for training or not, i.e., $\gamma_i = 1$ if $d_i > 0$ and $\gamma_i = 0$ if $d_i = 0$. Then, for all workers to complete a specified number of epoches, denoted by $numEpoch$, the total cost of distributed training can be expressed as:

$$\begin{aligned} Total_cost = {} & \max_i\{Transfer_i^s\} + \max_i\{Initialize_i \cdot \gamma_i\} \\ & + \max_i\{epochTime_i\} \cdot numEpoch. \end{aligned}$$

V.4.2 Optimization Problem

The goal of *Deep-Edge* is to minimize the overall cost of distributed training on a set of heterogeneous edge nodes by choosing a data sharding scheme $\mathcal{D}$, the batch sizes $\mathcal{B}$, as well as the number of deployed workers while subject to some system and performance constraints. The following states the optimization problem:

$$
\begin{aligned}
\text{minimize} \quad & Total_cost \\
\text{subject to} \quad & b_{\min} \leq b_i \leq b_{\max},\ \forall i && \text{(V.1)} \\
& 0 \leq d_i \leq |\mathcal{M}|,\ \forall i && \text{(V.2)} \\
& \textstyle\sum_i d_i = |\mathcal{M}| && \text{(V.3)} \\
& pressure_i^a \leq \delta^a,\ \forall a \in backApp_i, \forall i && \text{(V.4)}
\end{aligned}
$$

Constraint (V.1) requires the batch size to be within the range of minimum and maximum system-specific batch size, which could be determined by the DL model or the device's memory constraint. Constraints (V.2) and (V.3) require that each worker receives a portion of the data samples, and altogether they cover the entire set of data samples. Finally, Constraint (V.4) requires that, for each worker w_i, the pressure to its set of background applications, $backApp_i$, due to running the distributed training job on the same device, should be contained to be within an application-specific threshold δ_a for each application $a \in backApp_i$ in order not to violate the SLO of the application. The estimation of the pressure function to a background task, and the sensitivity function for the training job will be discussed further in Section V.5.

As the objective function (i.e., $Total_cost$) and the pressure constraint in the above optimization problem have a complex, non-linear relationship with the decision variables (i.e., $\mathcal{D}, \mathcal{B}$), it cannot be expressed analytically. Therefore, the problem cannot be solved using standard solvers and/or analytical techniques. Therefore, we aim at designing efficient

heuristic solutions, which will be described in Section V.5.3.2.

V.4.3 Assumptions

We assume that the user specifies the maximum number of epochs (i.e., $numEpoch$) required for the training of the DL model, which is independent of the configuration of the workers. The user also provides the model trigger condition, and the model is only updated whenever the trigger condition is received. As the DL model is usually updated on a large amount of data, we further assume that the one-time transfer cost (i.e., $Transfer_i^s$) and initialization cost (i.e., $Initialize_i$) are negligible compared to the total training time. Finally, we assume that the location of the parameter server(s) is given, and we do not need to select a node as a parameter server based on some specific criteria.

V.5 Design and Implementation of Deep-Edge

This section presents the design and implementation details of *Deep-Edge* by describing the architecture model, various components of the framework, and its modes of operation. We also explain our solution approach for minimizing the job completion of the DL model update in heterogeneous edge clusters under background stress.

V.5.1 Architecture Model of Deep-Edge

The architecture model consists of $\mathcal{K}$ edge nodes, out of which $\mathcal{N}$ are workers, $\mathcal{O}$ are data sources, and $\mathcal{P}$ are parameter servers, and they are represented by disjoint sets $\mathcal{W}$, $\mathcal{DS}$ and $\mathcal{PS}$, respectively. These edge nodes form a local area network and are connected via a layer 2 Ethernet switch, as shown in Figure V.9. As mentioned in the previous section, the nodes in $\mathcal{W} \cup \mathcal{PS}$ forms the DL model update cluster, where the worker nodes perform the actual re-training and the parameter server nodes act as a central repository for model parameters. The remaining nodes are belonging to set $\mathcal{DS}$ store data samples for the model

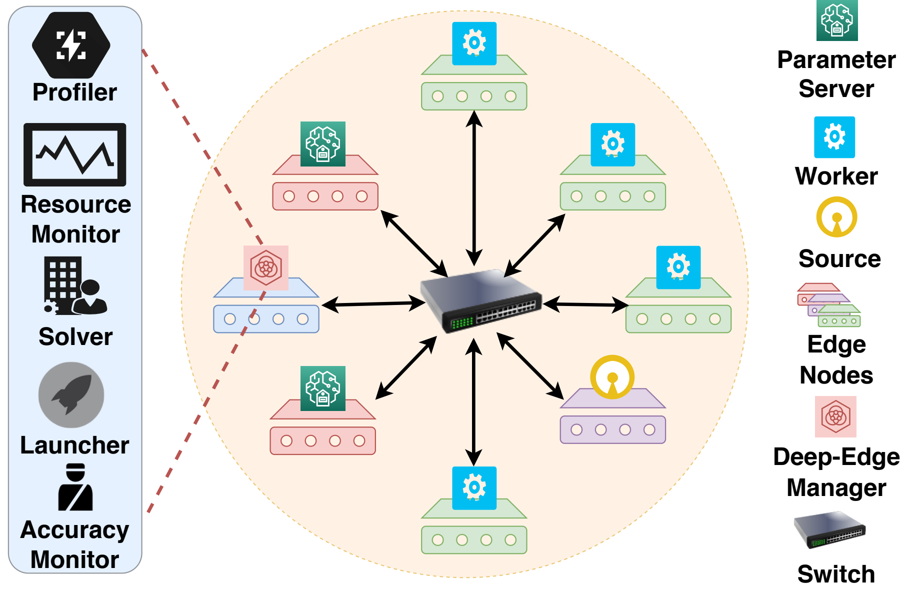


Figure V.9: Deep-Edge architecture

update task.

One of the nodes in $\mathcal{DS}$ also acts as a *Deep-Edge Manager* (DEM), which implements the *Deep-Edge* framework. DEM is a collection of components that enables profiling, resource scheduling and runtime monitoring of the DL cluster. The components are hosted as REST endpoints as `http://<ip>:<port>/<endpt_name>?<endpt_args>`, where `ip` is the IP address of the host, `port` is the port associated with the DEM, and `endpt_name`, `endpt_args` are the name and input arguments of the endpoint. The following subsections describe the different components and modes of operation associated with the DEM, as shown in Figure V.10.

V.5.2 Components of Deep-Edge Manager

The DEM consists of five components, which together provide a unified solution for profiling, scheduling, and monitoring the DL model update tasks.

① **Profiler:** *Deep-Edge* uses a data-driven approach to estimate $t^{compute}$, t^{update} and *pressure* experienced by background applications. This component allows both latency-critical and model update tasks to be profiled against stress points along the dimen-

sions of CPU, GPU, and memory utilization. The Profiler accepts different system metrics and stress points as the input arguments. *Deep-Edge* uses CPU, Memory, Disk I/O stressors from the well-known library Stress-ng [148] and the GPU load stressing application is based on the NVIDIA Cuda-10 library.

② **Solver:** This component implements the scheduling strategy to identify the candidate workers and their respective data shards. The Solver takes the number of data samples, the current state of the edge cluster along with the performance and interference models as inputs, and outputs a data sharding scheme. The scheduling strategy is explained in more detail in Section V.5.3.2.

③ **Resource Monitor:** The Resource Monitor maintains a map of active workers in the edge cluster and periodically monitors the ongoing model update tasks. This component is responsible for re-triggering the model update task in response to worker node failure.

④ **Launcher:** This component is responsible for launching applications (DL & stressing) on the worker and parameter server nodes. It accepts three different kinds of arguments: a) Stressor arguments; b) DL arguments; and c) Logging arguments. The Stressor arguments include parameters for the different stressing applications. The DL arguments include machine learning specific arguments, such as batch size, number of epochs, optimizer, etc., and the Logging arguments include the file path for creating log files. It launches the DL model update on the designated worker nodes with the assigned parameter server.

⑤ **Accuracy Monitor:** Accuracy monitor observes the progress of the training job after each epoch. *Deep-Edge* allows dynamic stopping of the model update task by tracking the accuracy of the validation set.

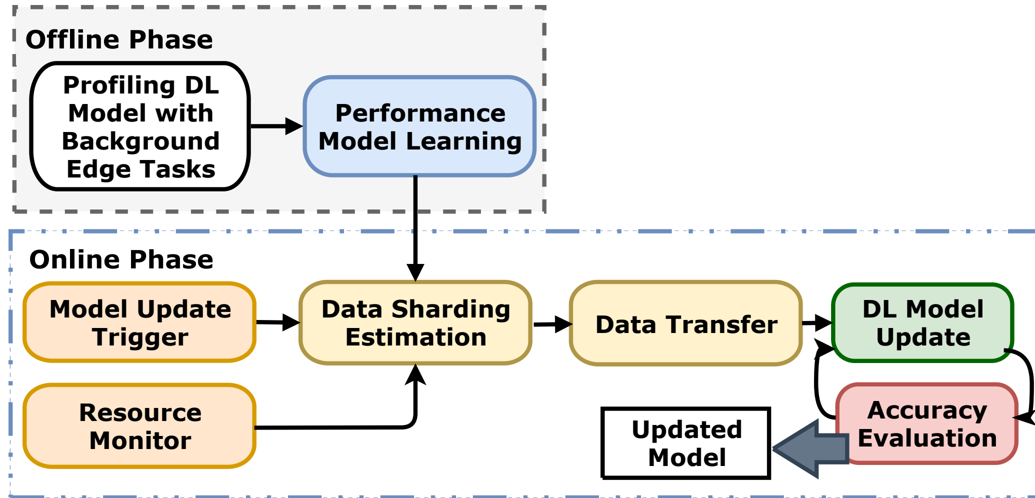


Figure V.10: Modes of operation

V.5.3 Modes of Operation

The operation of DEM can be categorized into two modes, *offline* and *online*, as illustrated in Figure V.10. In the *offline* or *design* mode, we build machine learning models to predict the execution time of the DL model update task and latency-critical background tasks. In the *online* mode, appropriate workers, along with batch size distribution and data shards, are calculated based on the developed performance models such that model update time is minimized while adhering to SLO constraints imposed by the background applications. The *online* mode also handles worker failures by trying to restart the DL model update task. The details of the features, as mentioned above of *Deep-Edge*, are provided in the following.

V.5.3.1 Performance and Interference Modeling

Deep-Edge uses a data-driven approach for modeling the performance of DL model update tasks on each node as well as the interference experienced by latency-critical background tasks. The performance of the DL model update task is measured by the time to complete one epoch, *epochTime*, which is the function of per-sample compute time,

$t^{compute}$, and update time, t^{update}, as defined in the optimization problem. Here, $t^{compute}$ depends upon the node type, node state, and batch size. We describe node state as a vector of system metrics containing CPU, GPU, and memory utilization. However, t^{update} depends not only on the node state but also on the complete batch distribution, state of the parameter server, and the number of workers. We define two functions, `EstComputeTime` and `EstUpdateTime`, to model the relation between the state of the DL cluster and $t^{compute}$ and t^{update} as shown in Equations (V.5) and (V.6) below, where $\mathcal{X}_i$ and b_i represent the state and batch size associated with worker $w_i \in \mathcal{W}$, $\mathcal{B}$ is the batch size distribution and $\mathcal{X}_{ps}$ is the state of the parameter server.

$$t_i^{compute} = EstComputeTime(\mathcal{X}_i, b_i) \tag{V.5}$$

$$t_i^{update} = EstUpdateTime(\mathcal{X}_i, \mathcal{B}, \mathcal{X}_{ps}, |\mathcal{W}|) \tag{V.6}$$

In order to create an interference profile of the DL model update task, we model interference as performance degradation experienced by the background applications. As shown in Figure V.12, we use a two-step approach to quantify performance degradation, i.e., increase in execution time. In the first step, we model the effects of running the DL model update task on a node whose state is described in Equation (V.7). The function, `EstState`, gives the relation between the initial state of the node, $\mathcal{X}^{initial}$, and its new state, $\mathcal{X}^{new}$ while executing the DL model update task. Since the system metrics can vary during the execution of the DL model update task, we use the 95th percentile value statistic. The second step involves learning the performance degradation, i.e., pressure to the background task, as a function of the new node state, defined by `EstExecTime` as shown in Equation (V.8).

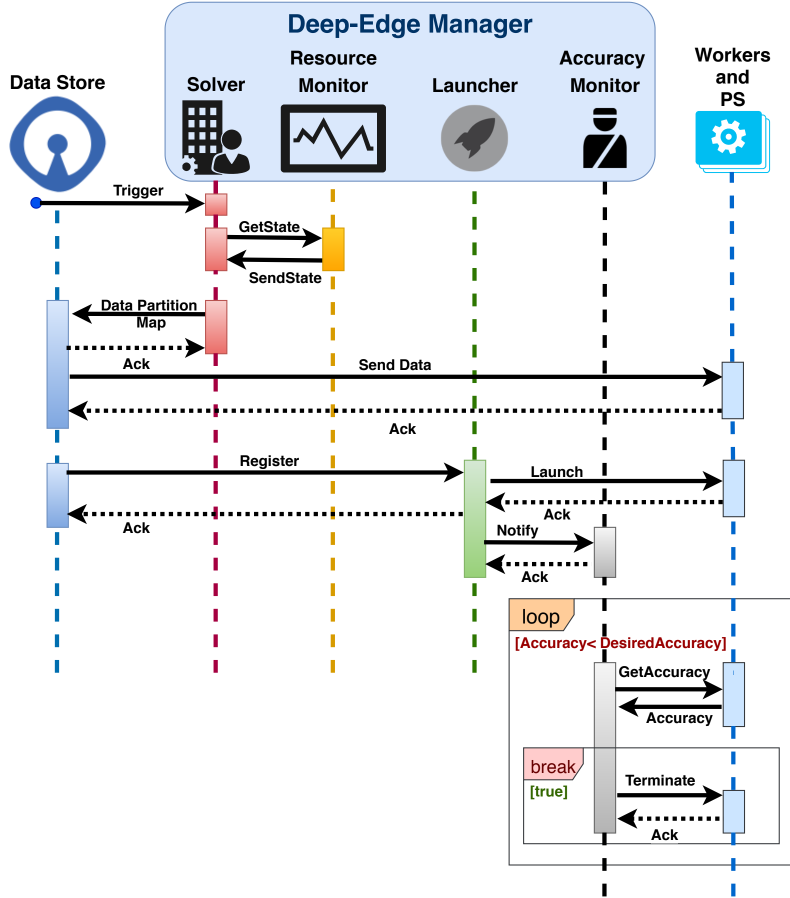


Figure V.11: Event Sequence Diagram of Data Sharding and Resource Scheduling

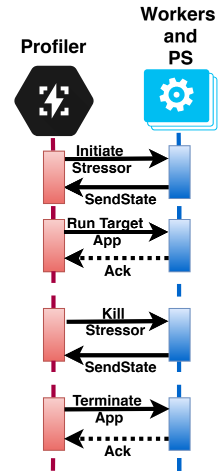


Figure V.12: Event Sequence Diagram of Profiling the background tasks along with DL model update task

$$\mathcal{X}_i^{new} = EstState(\mathcal{X}_i^{initial}) \tag{V.7}$$

$$pressure_i^a = EstExecTime(\mathcal{X}_i^{new}) \tag{V.8}$$

The models, as mentioned above, are learned by first performing sensitivity analysis to understand the importance/influence of the prospective features. Based on the candidate features set obtained after sensitivity analysis, regression models are learned. In *Deep-Edge*, we use H2O's AutoML framework [149] to find the best hyperparameter tuned algorithm.

V.5.3.2 Resource Scheduling

Figure V.11 shows the sequence of events of a new request of model update. A data store requests to DEM's solver endpoint to trigger the mode: *Model Update*. Upon the receipt of the request, the *Solver* gets the updated states of the workers, i.e., the number of prospective workers and their respective states (system metrics) from the *Resource Monitor*. Then, the *Solver* calculates the candidate worker nodes, data shards, and batch distribution using the scheduling strategy illustrated in Algorithm 5. The data and batch distributions are sent back to the *data store* to initiate data and base model transfer. After successful transmission, the data source registers the task information, such as the number of workers, data source, worker hostnames, data shards, and batch distribution, with the *Launcher*. Then, the *Launcher* sends the acknowledgment back to the data source after successfully starting the DL model update task on selected workers. *Accuracy Monitor* observes the progress of the training job after each epoch, and terminate the DL model update process once it reached the desired accuracy.

The heuristic presented in Algorithm 5 provides an efficient solution to the optimization problem described in Section V.4. The input of the algorithm includes the workers' state map $\mathcal{X} = [\mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_N]$, where $\mathcal{X}_i$ = [$\mathcal{X}_i^{cpu}$, $\mathcal{X}_i^{gpu}$, $\mathcal{X}_i^{mem}$] represents the state of worker $w_i \in \mathcal{W}$, the parameter server's state X_{ps} = [$\mathcal{X}_{ps}^{cpu}$, $\mathcal{X}_{ps}^{gpu}$, $\mathcal{X}_{ps}^{mem}$], number of data samples $\mathcal{M}$, the stopping threshold ε, and the maximum number of iterations τ. The output of the algorithm is the data distribution $\widetilde{\mathcal{D}}$ and the batch size distribution $\widetilde{\mathcal{B}}$, such that ($d_i \in \widetilde{\mathcal{D}}$, $b_i \in \widetilde{\mathcal{B}}$) identifies the data shard and batch size associated with worker $w_i \in \mathcal{W}$. If any worker $w_k \in \mathcal{W}$ is not selected for the model update task, the algorithm will return $d_k = 0$ and $b_k = 0$ for the worker.

The heuristic computes the data and batch distributions in an iterative fashion, where the initial estimate of the size of all data shards is ∞ [Lines 2-3]. Using the initial data distribution and memory utilization of a node, the corresponding batch size b_i is calculated [Line 6] using the function `GetMaxBatchSize`, which enforces adherence of the memory

constraint described in the optimization problem, by providing the maximum batch size given the current memory utilization of the node. With the batch size estimates, $t^{compute}$, t^{update} and t^{total} are calculated for all the workers [Lines 7-9]. Refined data distribution is then calculated based on t^{total} to balance the workloads of all the workers [Line 10]. After calculating the refined data distribution, all workers check for adherence for resource interference constraints [Lines 11-22]. A worker is dropped from the DL cluster [Lines 19-21] if any of its background tasks will experience deadline violations [Lines 14-17]. The cycle [Lines 5-28] repeats until the data distributions in two consecutive iterations are almost the same, i.e., the $\mathcal{L}^2$ norm is less than a threshold, ε or the maximum number of iterations τ have been reached [Lines 24-26].

Then, we calculate the epoch time based on the data shards, which is given by the maximum time taken by any worker to process the data samples assigned [Line 29]. Note that, based on our proportional data sharding scheme, the difference between the epoch times from the different workers should be minimal (only due to rounding-off errors). If the resulting epoch time is better than the best one we have found so far, we remember the configuration as a potential solution [Lines 30-32]. Finally, to explore if better solutions are possible, we calculate the effect of removing the slowest worker (in terms of the total per-sample training time) on the overall epoch time [Lines 33-34]. As fewer workers are now present, this may affect the update time for each remaining worker, which will, in turn, affect the data shards and the epoch time. This process is performed iteratively until removing a worker no longer improves the overall epoch time, in which case the algorithm will eventually terminate [Line 36]. The best one will then give the final data shard size found so far.

V.5.3.3 Fault Tolerance

When a worker node experiences a failure while executing a model update task either due to process crash or node failure, the Resource Monitor in the DEM will detect such

Algorithm 5: Resource Scheduling Heuristic

1 **Initialize**: $min_t^* \leftarrow \infty$, $\widetilde{\mathcal{W}} \leftarrow \phi$, $\widetilde{\mathcal{B}} \leftarrow \phi$, $\widetilde{\mathcal{D}} \leftarrow \phi$, $Iter \leftarrow 0$
2 $d_i = \infty,\ \forall w_i \in \mathcal{W}$;
3 $\mathcal{D} \leftarrow \{d_1, d_2, \ldots, d_N\},\ Iter \leftarrow 0$;
4 **while** *True* **do**
5   **while** *True* **do**
6     $b_i \leftarrow \min(\text{GetMaxBatchSize}(\mathcal{X}_i^{mem}), d_i),\ \forall w_i \in \mathcal{W}$;
7     $t_i^{compute} \leftarrow \text{EstComputeTime}(\mathcal{X}_i, b_i),\ \forall w_i \in \mathcal{W}$;
8     $t_i^{update} \leftarrow \text{EstUpdateTime}(\mathcal{X}_i, \mathcal{X}_{ps}, \mathcal{B}, |\mathcal{W}|), \forall w_i \in \mathcal{W}$;
9     $t_i^{total} \leftarrow t_i^{compute} + t_i^{update} / b_i, \forall w_i \in \mathcal{W}$;
10     $d_i = \frac{|\mathcal{M}|}{t_i^{total} \cdot \sum_{w_i \in \mathcal{W}} \frac{1}{t_i^{total}}}, \forall w_i \in \mathcal{W}$;
11   **for** each $w_i \in W$ **do**
12       $flag \leftarrow$ False
13   **for** each $a \in backApp_i$ **do**
14   **if** $pressure_i^a > \delta_i^a$ **then**
15           $flag \leftarrow$ True;
16   break; ▷ Constraint violated
17   **end**
18   **end**
19   **if** $flag$ **then**
20         $\mathcal{W} \leftarrow \mathcal{W} \setminus w_i, b_i \leftarrow 0, d_i \leftarrow 0$; ▷ Remove worker
21   **end**
22   **end**
23     $\mathcal{D}_{new} \leftarrow \{d_1, d_2, \ldots, d_N\},\ Iter$++;
24   **if** $(||\mathcal{D}_{new} - \mathcal{D}||_2 \leq \varepsilon) \vee (Iter == \tau)$ **then**
25   break;
26   **end**
27     $\mathcal{D} \leftarrow \mathcal{D}_{new}$;
28   **end**
29   $epochTime = \max_{w_i \in \mathcal{W}}(d_i \cdot t_i^{total})$;
30   **if** $epochTime < min_t^*$ **then**
31     $min_t^* \leftarrow epochTime$;
32     $\widetilde{\mathcal{W}} \leftarrow \mathcal{W}$, $\widetilde{\mathcal{B}} \leftarrow \mathcal{B}$, $\widetilde{\mathcal{D}} \leftarrow \mathcal{D}$;
33     $w_k \leftarrow \arg\max_{w_i \in \mathcal{W}}(t_i^{total})$; ▷ Find slowest worker
34     $\mathcal{W} \leftarrow \mathcal{W} \setminus w_k, b_k \leftarrow 0, d_k \leftarrow 0$; ▷ Remove worker
35   **else**
36   break;
37   **end**
38 **end**

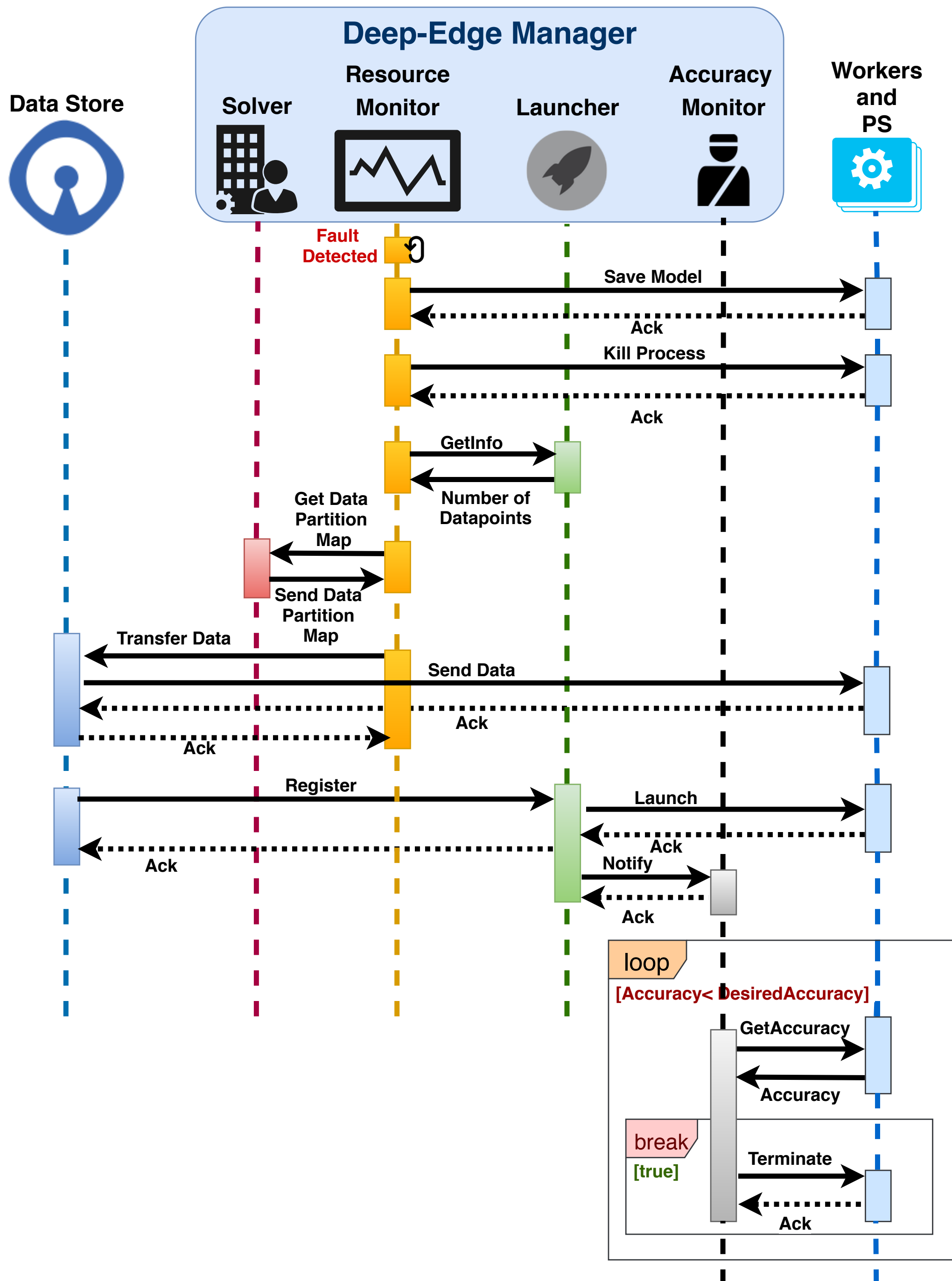


Figure V.13: Event Sequence Diagram of Failure handling

events and trigger the re-launching of the task. The DEM uses a *three-strike rule*, i.e., a worker will not be considered part of the DL cluster if it has experienced at least three interruptions while running a model update task. Figure V.13 highlights the sequence of

events as a result of worker failure. After detecting a worker failure, the *Resource Monitor* requests the available workers to save their current progress and then kills the model update processes on the available workers. After the processes are killed, the *Resource Monitor* gets the DL model update task information such as data shards, batch distribution, worker hostnames, data source, and task id from the *Launcher*. Then, the *Resource Monitor* requests the *Solver* to create the data-sharding map for the data points of the failed node. Once the data-sharding map is created, *Resource Monitor* notifies the appropriate data source to re-trigger the model update task.

V.6 Evaluation

In this section, we present the evaluation results of different phases of *Deep-Edge* framework.

V.6.1 Experiment Setup

V.6.1.1 TestBed

Our test-bed comprises of one *NVIDIA Jetson TX2* (256-core NVIDIA Pascal GPU, 8GB memory, Dual-Core NVIDIA Denver 2 64-Bit CPU and Quad-Core ARM Cortex-A57), three *NVIDIA Jetson Nano* (128-core Maxwell GPU, Quad-Core ARM Cortex-A57, 4GB memory) devices and two *Raspberry Pi 4* (Broadcom BCM2711, Quad-core Cortex-A72 (ARM v8), 4GB memory) devices. These devices are connected by one Gigabit layer 2 switch. One raspberry pi acts as a parameter server while the second acts as a data store and also hosts the DEM.

V.6.1.2 Workloads

The model update task consists of updating a base DNN, Inception [147] with 3855 data samples from Caltech-256 dataset [146] using MXNET [111]. The base Inception model

is created using transfer learning [150], where the last layer of a pre-trained Inception model based on Imagenet [151] replaced by a new layer. We trained the base model (last layer) with 2600 data points to reach an accuracy of 75%. We emulated the model update trigger when the accuracy of the base model decreases below 50% accuracy, which is our assumed tolerance threshold. To emulate real data generation, we continuously feed the base model new 4600 data points, where the model accuracy degraded to 50% accuracy, and we triggered the model update process. We updated our model on 4600 data points of the Caltech dataset, in which 3855 data points are used as training data points, and the rest of the data points are validation set.

The latency-critical task is based on a distributed real-time computer vision application of image reconstruction from multiple video streams where an initial image processing step is performed in parallel on multiple edge devices. The image processing step involves identifying scale and rotation invariant descriptors (features) using Scale Invariant Feature Transform (SIFT) [152]. The latency-critical task constitutes executing SIFT transform on the acquired frame and sending the serialized SIFT features along with the original frame to an image stitching server over a UDP socket every 200 ms. The size and resolution of the acquired frame is 56 KB, 640x320, respectively.

V.6.2 Performance Modeling

We performed sensitivity analysis along the dimensions of GPU, CPU, Memory Utilization, the number of workers, and batch size. The following behaviors are observed:

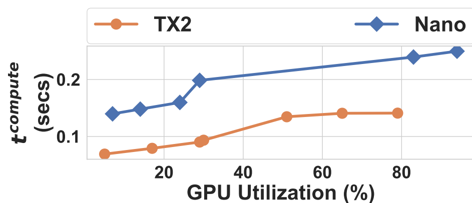


Figure V.14: Stressing GPU increases compute time.

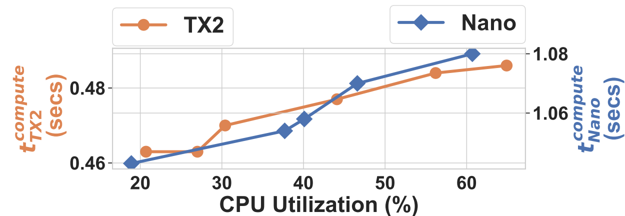


Figure V.15: Stressing CPU increases compute time.

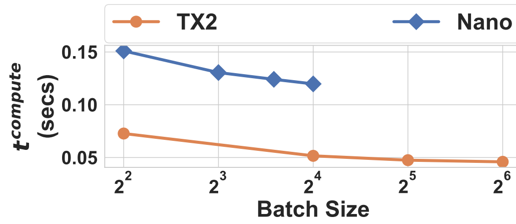


Figure V.16: Increasing Batch size decreases compute time.

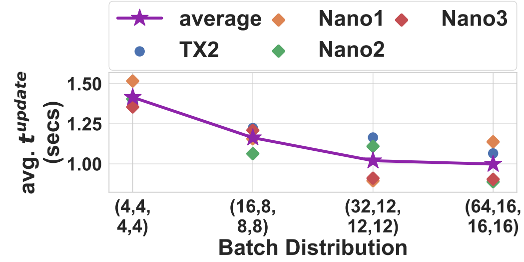


Figure V.17: Increasing Batch size decreases update time.

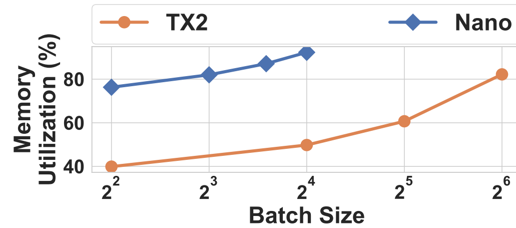


Figure V.18: Increasing the batch size increases the memory footprint.

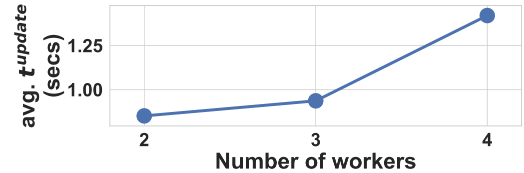


Figure V.19: Increasing workers nodes increase update time.

I) Increasing GPU utilization increases compute time, $t^{compute}$ [Figure V.14].

II) Increasing CPU utilization increases compute time, $t^{compute}$ [Figure V.15].

III) Increasing the batch size of the DL model update process decreases compute time, $t^{compute}$ [Figure V.16].

IV) Increasing the batch size of the DL model update process decreases update time, t^{update} [Figure V.17].

V) Increasing the batch size of the DL model update process increases the memory utilization [Figure V.18].

VI) Increasing the number of workers results in increasing DL model update time, t^{update} [Figure V.19].

VII) Increasing CPU utilization of parameter server, increases the update time, t^{update} [Figure V.20].

VIII) Increasing Memory utilization does not affect the throughput of the model update task. However, insufficient free memory can result in terminating a process by OS.

Based on the above observations, we considered CPU, GPU Utilization, and batch size

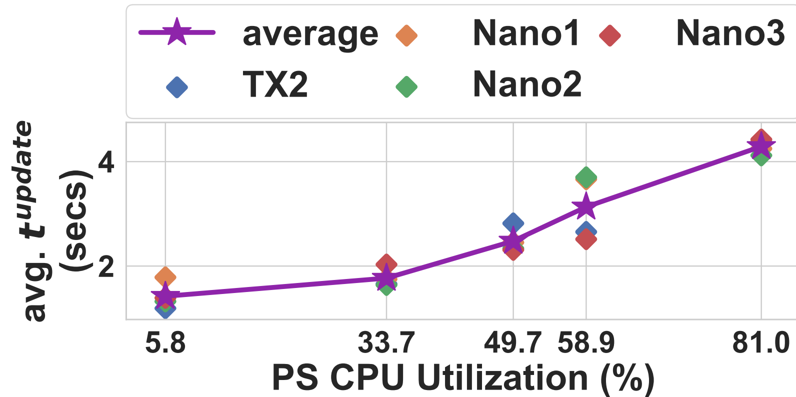


Figure V.20: Stressing CPU of parameter server increases update time.

as candidate features for function `EstComputeTime` and added server CPU utilization, the number of workers along with batch distribution of all nodes to the above list as final features to learn the function `EstUpdateTime`. The function, `EstSatte`, is multiple outputs in nature, i.e., GPU, CPU, Memory, and a separate regressor is learned for each one of them. We extend the label with the feature name to notify the individual regression model, for instance, $\texttt{EstState}^{mem}$ represents the memory regressor. `GetMaxBatchSize` uses $\texttt{EstState}^{mem}$ recursively to identify maximum feasible batch size to run model update job on a node. Finally, `EstExecTime` uses all system metrics of the node as features to predict the time to finish the latency-critical task. We used H2O's AutoML framework to select the best regression algorithm as well as perform the hyperparameter tuning. Table V.1 highlights the number of data points, regression algorithm along with accuracy for all learned functions. As evident from the table, the gradient boosting methods, XGBoost [153], outperformed others with an average MAPE (mean absolute percentage error) of 2.32% overall models.

V.6.3 Resource Scheduling

We compare *Deep-Edge* scheduling policy with fairness based schedulers adopted in many resource managers such as Yarn [154], Mesos [155]. In fairness based schedulers,

Device	Function	Data-points (train/test)	Algorithm	Accuracy (MAPE)
Nano	EstComputeTime	1386/264	XGBoost	0.414 ± 0.35
Nano	EstUpdateTime	1386/264	XGBoost	11.29 ± 9.5
Nano	$\texttt{EstState}^{gpu}$	1386/264	XGBoost	3.06 ± 0.74
Nano	$\texttt{EstState}^{cpu}$	1386/264	XGBoost	8.043 ± 7.45
Nano	$\texttt{EstState}^{mem}$	1386/264	XGBoost	0.82 ± 0.86
Nano	EstExecTime	1386/264	XGBoost	1.73 ± 0.21
TX2	EstComputeTime	378/72	XGBoost	3.84 ± 7.42
TX2	EstUpdateTime	378/72	XGBoost	12.29 ± 10.52
TX2	$\texttt{EstState}^{gpu}$	378/72	XGBoost	3.05 ± 1.35
TX2	$\texttt{EstState}^{cpu}$	378/72	XGBoost	11.61 ± 9.81
TX2	$\texttt{EstState}^{mem}$	378/72	XGBoost	1.01 ± 0.51
TX2	EstExecTime	378/72	XGBoost	1.48 ± 0.12
Pi	$\texttt{EstState}^{cpu}$	630/120	XGBoost	2.32 ± 1.58
Pi	$\texttt{EstState}^{mem}$	630/120	XGBoost	1.70 ± 1.15

Table V.1: Estimator results

the data is divided equally among the worker nodes, where Deep-Edge intelligently shard the data among multiple workers based on the state and type of the worker nodes. We performed 120 random experiments, and in each experiment, all worker nodes are running the SIFT feature detector task along with randomly selected stressors such that the initial state of every node does not violate the deadline of the background application. Figure V.21 highlights the average epoch time observed using *Deep-Edge* and fairness schedulers. On average, the *Deep-Edge* scheduler reduced the epoch time by 1.54*x* times without violating any deadline. A histogram of DL task speedup (%) is highlighted in Figure V.22.

V.6.3.1 Effectiveness of Data Sharding Strategy on Epoch Time

As described in Algorithm 5, we calculated the number of data points for each worker from the whole dataset, and send the respective data points(d_i) to respective workers. In isolation, TX2 devices support the batch size of 64 for the Caltech dataset, whereas Nano

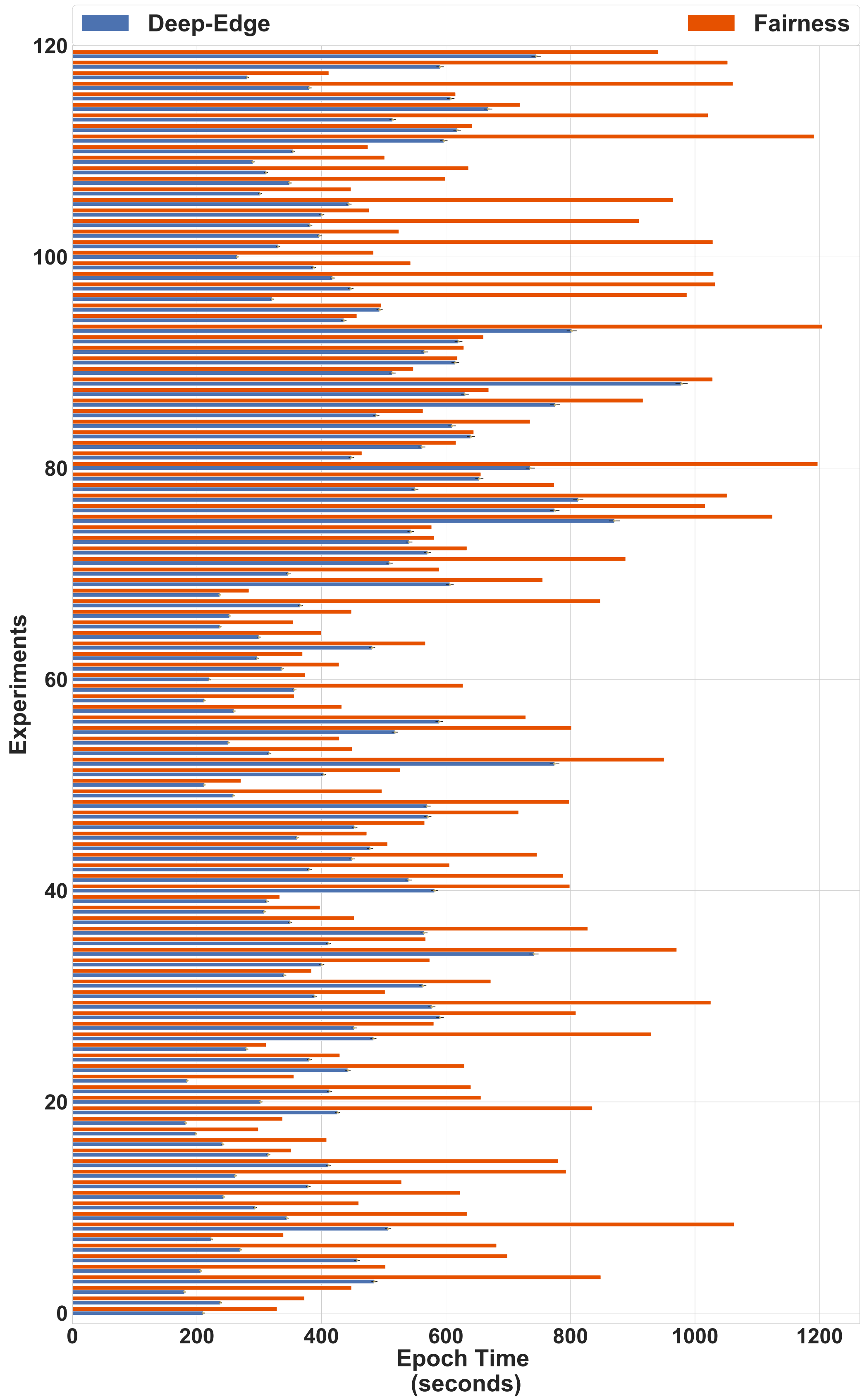


Figure V.21: Epoch time distribution

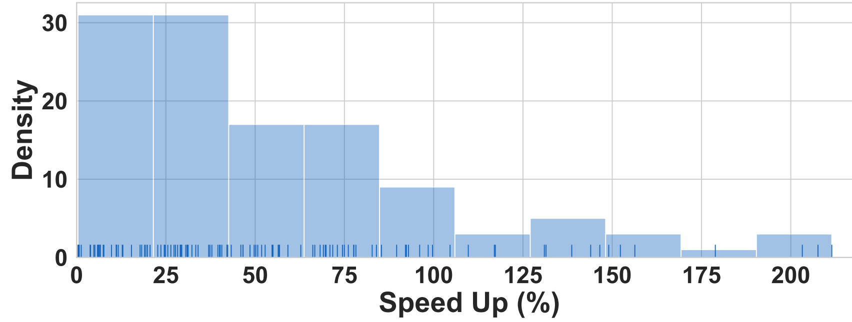


Figure V.22: Speed Up distribution

devices support the batch size of 16 for the Caltech dataset. This is because, TX2 device has 8GB memory, where the Nano device has 4GB memory space. We considered the maximum available batch size because increasing batch size results decrease in the compute and update time. For our experiments, we had one TX2 and three Nano devices. Based on our performance estimation algorithm, we divided 1893 data points for the one TX2 device and 654 data points for the three Nano devices. The epoch time for the experiment in the heterogeneous cluster was around 121 seconds, as shown in the Figure V.23(left). After that, our Algorithm 5 removes the slowest worker node and calculates the epoch time. Here, as shown in Figure V.23(right), removing one worker increases the epoch time, and it becomes around 148 seconds. Then, we compared the epoch time with the best performing standalone device. In this case, if we run the experiment on a standalone TX2 device with 3855 data points, the epoch time would be around 184 seconds. Therefore, based on these experiments, we considered one TX2 and three Nano devices cluster setup for the model update task because, in that setup, the epoch time was minimum, as shown in Figure V.23.

In our second case study, we ran GPU intensive tasks as background co-located applications. In this experiment, the GPU utilization in TX2 was 32% due to background co-located applications. In the first Nano device, GPU utilization was 42%, and in the other two Nano devices, the GPU utilization was 24% due to background co-located applications. With this background GPU utilization, we run the DL model update task in the heterogeneous cluster. Using the Algorithm 5, we calculate the number of data points for

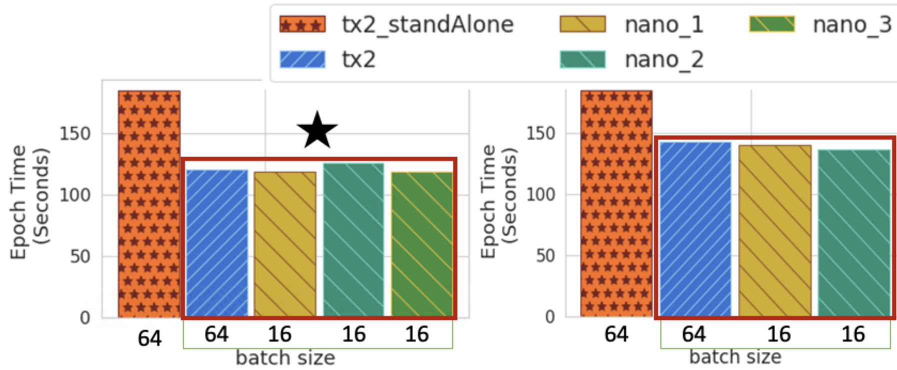


Figure V.23: Epoch time with different cluster setup.

each worker, so that the epoch time remains the same. Based on our performance estimation algorithm, we divided 1631 data points for the one TX2 device, 691 data points for the first Nano device, 769 data points for the second Nano device, and 764 data points for third Nano device. As shown in the Figure V.24(left), the epoch time for all the devices is around 154 seconds in the four worker node cluster setup.

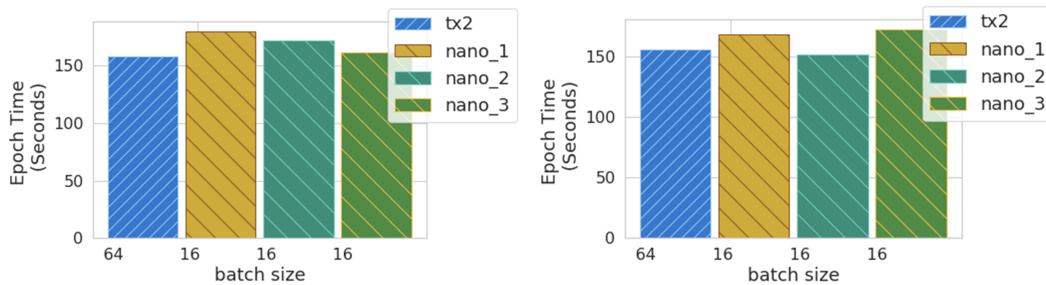


Figure V.24: EpochTime when GPU is occupied by co-located application

Similarly, we changed the background co-located tasks to change the background GPU utilizations. The GPU utilization in TX2 was 28% due to background co-located applications. In the first Nano device, GPU utilization was 25%, and in the other two Nano devices, the GPU utilization was 24% due to background co-located applications. Based on our performance estimation algorithm, we divided 1677 data points for the one TX2 device, 694 data points for the first Nano device, 735 data points for the second Nano device, and 749 data points for third Nano device. As shown in the Figure V.24(right), the epoch time for all the devices is around 155 seconds in the four worker node cluster setup.

V.6.4 Model Convergence

As shown in the Figure V.25 and Figure V.26, the model converges despite different batch combinations and different computing power of the edge devices. As we bound the epoch time of each worker node within a minimal threshold, the synchronization delay is minimally bounded. Hence, the asynchronous parameter synchronization does not stale any worker model, and the overall DL model converges.

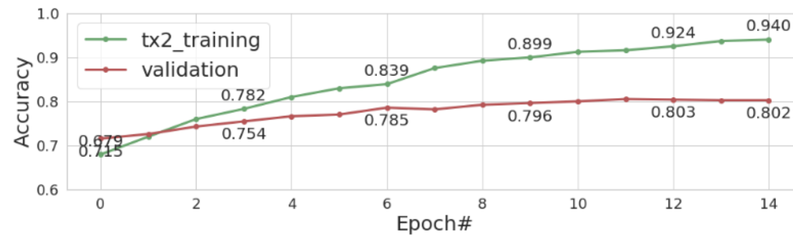


Figure V.25: Model convergence with batch combination of 64,16,16,16 on cluster of 1tx2 and 3 nanos.

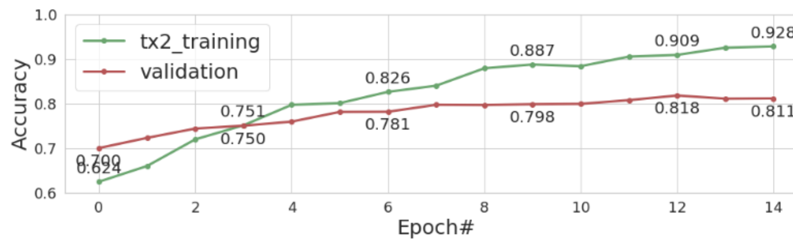


Figure V.26: Model convergence with batch combination of 8,8,8,8 on cluster of 1tx2 and 3 nanos.

V.7 Conclusion

V.7.1 Summary

This chapter presents an interference aware Deep Learning model update platform for the edge devices that minimizes the DL model update time by intelligently distributing data among worker nodes while adhering to the latency constraints of the background applications. We built a performance model to estimate the pressure and sensitivity to run the DL model update with the background latency-critical tasks. We described different components of the framework and also showed the efficacy of the scheduling heuristic by

validating against realistic case study.

V.7.2 Discussions

In the future, we would like to extend this work in three dimensions: 1. Improving performance and interference models by adding more features such as memory and disk bandwidth. 2. Adding support for Multi-Process Service (MPS) based GPU workloads. 3. Including parameter server load balancing as a part of the scheduling problem.

Acknowledgment of Collaboration

I express my sincere appreciation to my colleague, Ajay Dev Chhokra, for his collaboration in this chapter. The Deep-Edge chapter is divided into four subareas: 1. Deep Edge framework design and implementation, 2. Sensitivity Analysis of the DL model update task, 3. Performance Modeling and Evaluation, and 4. Data Sharding Algorithm Design and Evaluation. I would like to acknowledge Ajay's efforts towards developing the Deep-Edge framework. Moreover, he also created a proof of concept and demonstrated that by building the performance model for this work, which is a distinguishable work in this chapter. I would also like to appreciate his effort and useful suggestions for understanding and developing ideas towards formulating the problem and developing the data sharding algorithm. This chapter would not have been possible without his hard work and constructive ideas.

CHAPTER VI

SUMMARY OF RESEARCH CONTRIBUTIONS

In the doctoral research on 'Algorithms and Techniques for Automating Deployment and Efficient Management of Large-Scale Distributed Data Analytics Services,' the contributions are four-fold.

① Contribution 1: Automation of Machine Learning(ML) model development and deployment

② Contribution 2: Automation of Infrastructure and Application Provisioning

③ Contribution 3: Proactive Resource Management to handle the dynamic workload

④ Contribution 4: Efficient and Interference-aware Strategy for continual ML, especially Deep Learning model update on heterogeneous edge cluster

The holistic and intelligent framework for Automating Deployment and Management of Data Analytics Services across Distributed Systems is available in https://github.com/doc-vu/Stratum.git.

VI.1 Summary of Model-Driven Approach to Automate ML Model Development, Deployment and Dissemination

The advent of the Internet of Things (IoT) and microservices-based architectures have enabled a variety of smart distributed applications where IoT devices gather, transform and analyze data in high volumes and velocity. Many of these applications require real-time and robust predictive analytics, which requires executing the stream processing workflows closer to the source of the data as well as dynamic resource management decision making. Moreover, predictive analytics requires the developer to build reliable and robust

Machine Learning (ML) models which in turn requires them to conduct feature engineering, parameter search, and tuning, and ML model selections, all of which are not only time-consuming but developers often lack the needed expertise. The proliferation of ML libraries and frameworks, data ingestion tools, stream and batch processing engines, visualization techniques, and a variety of available hardware platforms further exacerbates these problems. To overcome these daunting challenges faced by IoT smart application developers, we present an end-to-end, holistic IoT data analytics solution called *Stratum* for data analytics applications' full lifecycle management. Stratum uses serverless computing principles thereby shifting the deployment and runtime resource management responsibilities away from the developer while offering them an event-driven ML-as-a-service capability for inference jobs that can opportunistically exploit the edge-fog-cloud resource spectrum to ensure their needed Quality of Service(QoS). Stratum provides users with an intuitive, declarative mechanism based on the principles of model-driven engineering to specify their application needs, which are then transformed via generative programming principles to automate the application lifecycle management. The chapter II describes the Stratum architecture highlighting the problems it resolves and demonstrates its capabilities using real-world case studies.

VI.2 Summary of Model-Driven Approach to Automate Infrastructure and Application Provisioning

Users of cloud platforms often must expend significant manual efforts in the deployment and orchestration of their services on cloud platforms due primarily to having to deal with the high variabilities in the configuration options for virtualized environment setup and meeting the software dependencies for each service. Despite the emergence of many DevOps cloud automation and orchestration tools, users must still rely on specifying low-level scripting details for service deployment and management using Infrastructure-as-Code (IAC). Using these tools required domain expertise along with a steep learning curve.

To address these challenges in a tool-and-technology agnostic manner, which helps promote interoperability and portability of services hosted across cloud platforms, we present a GUI based cloud automation and orchestration framework called CloudCAMP. It incorporates domain-specific modeling so that the specifications and dependencies imposed by the cloud platform and application architecture can be specified at an intuitive, higher level of abstraction without the need for domain expertise using Model-Driven Engineering(MDE) paradigm. CloudCAMP transforms the partial specifications into deployable Infrastructure-as-Code (IAC) using the Transformational-Generative paradigm and by leveraging an extensible and reusable knowledge base. The auto-generated IAC can be handled by existing tools to provision the services components automatically. We validate our approach quantitatively by showing a comparative study of savings in manual and scripting efforts versus using CloudCAMP.

VI.3 Summary of Proactive Resource Management Strategy

Pre-trained deep learning models are increasingly being used to offer a variety of compute-intensive predictive analytics services such as fitness tracking, speech and image recognition. The stateless and highly parallelizable nature of deep learning models makes them well-suited for serverless computing paradigm. However, making effective resource management decisions for these services is a hard problem due to the dynamic workloads and diverse set of available resource configurations that have their deployment and management costs. To address these challenges, we present a distributed and scalable deep-learning prediction serving system called Barista and make the following contributions. First, we present a fast and effective methodology for forecasting workloads by identifying various trends. Second, we formulate an optimization problem to minimize the total cost incurred while ensuring bounded prediction latency with reasonable accuracy. Third, we propose an efficient heuristic to identify suitable compute resource configurations. Fourth, we propose an intelligent agent to allocate and manage the compute resources by horizontal and vertical

scaling to maintain the required prediction latency. Finally, using representative real-world workloads for urban transportation service, we demonstrate and validate the capabilities of Barista.

VI.4 Summary to Interference-aware continual ML/DL Model Update Strategy on Heterogeneous Edge

Deep Learning (DL) model-based AI services are increasingly offered in a variety of predictive analytics services such as computer vision, natural language processing, speech recognition. However, the quality of the DL models can degrade over time due to changes in the input data distribution, thereby requiring periodic model updates. Although cloud data-centers can meet the computational requirements of the resource-intensive and time-consuming model update task, transferring data from the edge devices to the cloud incurs a significant cost in terms of network bandwidth and are prone to data privacy issues. With the advent of GPU-enabled edge devices, the DL model update can be performed at the edge in a distributed manner using multiple connected edge devices. However, efficiently utilizing the edge resources for the model update is a hard problem due to the heterogeneity among the edge devices and the resource interference caused by the co-location of the DL model update task with latency-critical tasks running in the background. To overcome these challenges, we present Deep-Edge, a load and interference aware, fault-tolerant resource management framework for performing model update at the edge based on distributed training. The chapter V makes the following contributions. First, it provides a unified framework for monitoring, profiling, and deploying the DL model update tasks on heterogeneous edge devices. Second, it presents an optimization problem to minimize the total model update cost in the heterogeneous edge environment while guaranteeing that no latency-critical applications experience deadline violations. Third, it proposes an efficient heuristic to identify suitable compute resource configurations and a data sharding strategy to distribute the data among the available resources by building an interference-aware

performance model. Finally, we present empirical results to validate the efficacy of the framework by presenting a real-world DL model update case-study based on the Caltech dataset and an edge AI cluster testbed.

VI.5 List of Publications

JOURNAL PUBLICATIONS

1. Yogesh Barve, Prithviraj Patil, ***Anirban Bhattacharjee***, and Aniruddha Gokhale. "Pads: Design and implementation of a cloud-based, immersive learning environment for distributed systems algorithms." IEEE Transactions on Emerging Topics in Computing 6, no. 1 (2017): 20-31.
2. Goncalo Martins, Arul Moondra, Abhishek Dubey, ***Anirban Bhattacharjee***, and Xenofon Koutsoukos. "Computation and communication evaluation of an authentication mechanism for time-triggered networked control systems." Sensors 16, no. 8 (2016): 1166.

CONFERENCE PUBLICATIONS

1. ***Anirban Bhattacharjee****, Ajay Dev Chhokra*, Hongyang Sun , Shashank Shekhar, Aniruddha Gokhale, Gabor Karsai and Abhishek Dubey. "Deep-Edge: An Efficient Framework for Deep Learning Model Update on Heterogeneous Edge" , In 2020 IEEE International Conference on Fog and Edge Computing (ICFEC), IEEE, 2020 – Under Review
2. ***Anirban Bhattacharjee***, Yogesh Barve, Shweta Khare, Shunxing Bao, Zhuangwei Kang, Aniruddha Gokhale, and Thomas Damiano. "STRATUM: A BigData-as-a-Service for Lifecycle Management of IoT Analytics Applications", In 2019 IEEE International Conference on Big Data (Big Data), pp. 1607-1612. IEEE, 2019.

3. Shweta Khare, Hongyang Sun, Julien Gascon-Samson, Kaiwen Zhang, Aniruddha Gokhale, Yogesh Barve, ***Anirban Bhattacharjee***, and Xenofon Koutsoukos. "Linearize, predict and place: minimizing the makespan for edge-based stream processing of directed acyclic graphs." In Proceedings of the 4th ACM/IEEE Symposium on Edge Computing, pp. 1-14. 2019.

4. ***Anirban Bhattacharjee***, Ajay Dev Chhokra, Zhuangwei Kang, Hongyang Sun and Aniruddha Gokhale, "BARISTA: Efficient and Scalable Deep Learning Prediction Serving using Serverless Computing." In 2019 IEEE International Conference on Cloud Engineering (IC2E), pp. 23-33. IEEE, 2019.

5. Yogesh Barve, Shashank Shekhar, Ajay Dev Chhokra, Shweta Khare, ***Anirban Bhattacharjee***, Zhuangwei Kang, Hongyang Sun and Aniruddha Gokhale, "FECBench: A Holistic Interference-aware Approach for Application Performance Modeling." In 2019 IEEE International Conference on Cloud Engineering (IC2E), pp. 211-221. IEEE, 2019.

6. ***Anirban Bhattacharjee***, Barve, Yogesh, Shweta Khare, Shunxing Bao, Aniruddha Gokhale, and Thomas Damiano. "Stratum: A Serverless Framework for the Lifecycle Management of Machine Learning-based Data Analytics Tasks." In 2019 USENIX Conference on Operational Machine Learning (OpML 19), pp. 59-61. 2019.

7. Yogesh Barve, Shashank Shekhar, Shweta Khare, ***Anirban Bhattacharjee***, and Aniruddha Gokhale. "UPSARA: A Model-Driven Approach for Performance Analysis of Cloud-Hosted Applications." In 2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC), pp. 1-10. IEEE, 2018.

8. ***Anirban Bhattacharjee***, Yogesh Barve, Aniruddha Gokhale, and Takayuki Kuroda. "A Model-Driven Approach to Automate the Deployment and Management of Cloud Services." In 2018 IEEE/ACM International Conference on Utility and Cloud Com-

puting Companion (UCC Companion), pp. 109-114. IEEE, 2018.

9. Shashank Shekhar, Hamzah Abdel-Aziz, ***Anirban Bhattacharjee***, Aniruddha Gokhale, and Xenofon Koutsoukos. "Performance interference-aware vertical elasticity for cloud-hosted latency-sensitive applications." In 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), pp. 82-89. IEEE, 2018.

10. ***Anirban Bhattacharjee***, Yogesh Barve, Aniruddha Gokhale, and Takayuki Kuroda. "(WIP) CloudCAMP: Automating the Deployment and Management of Cloud Services." In 2018 IEEE International Conference on Services Computing (SCC), pp. 237-240. IEEE, 2018.

11. Shashank Shekhar, Ajay Dev Chhokra, ***Anirban Bhattacharjee***, Guillaume Aupy, and Aniruddha Gokhale. "INDICES: exploiting edge resources for performance-aware cloud-hosted services." In 2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC), pp. 75-80. IEEE, 2017.

12. Goncalo Martins, ***Anirban Bhattacharjee***, Abhishek Dubey, and Xenofon D. Koutsoukos. "Performance evaluation of an authentication mechanism in time-triggered networked control systems." In 2014 7th International Symposium on Resilient Control Systems (ISRCS), pp. 1-6. IEEE, 2014.

WORKSHOPS AND POSTERS

1. Aniruddha Gokhale, Yogesh Barve, ***Anirban Bhattacharjee***, and Shweta Khare, "Software - defined and Programmable CPS/IoT-OS: Architecting the Next Generation of CPS/IoT Operating Systems", in 1st International Workshop on Next-Generation Operating Systems for Cyber-Physical Systems (NGOSCPS 2019 Position Papers)

2. Yogesh Barve, Shashank Shekhar, Ajay Dev Chhokra, Shweta Khare, ***Anirban Bhattacharjee***, and Aniruddha Gokhale. "Poster: Fecbench: An extensible framework for pinpointing sources of performance interference in the cloud-edge resource spectrum." In 2018 IEEE/ACM Symposium on Edge Computing (SEC), pp. 331-333. IEEE, 2018.

3. Yogesh Barve, Shashank Shekhar, Ajay Dev Chhokra, Shweta Khare, ***Anirban Bhattacharjee***, and Aniruddha Gokhale, "Demo Paper: FECBench A Framework for Measuring and Analyzing Performance Interference Effects for Latency-Sensitive Applications", RTSS Works Demo Session of the 39th IEEE Realtime Systems Symposium (RTSS), Dec 1114, 2018.

TUTORIALS AND TALKS

1. ***Anirban Bhattacharjee***, Yogesh Barve, Shweta Khare, and Aniruddha Gokhale, "Investigating Dynamic Resource Management Solutions for Cloud Infrastructures using Chameleon Cloud," 2nd Chameleon Users Meeting, Feb 2019.

2. Aniruddha Gokhale, Yogesh Barve, ***Anirban Bhattacharjee***, and Travis Brummett, "Experiences using Chameleon in a Cloud Computing Course," 2nd Chameleon Users Meeting, Feb 2019.

3. Shashank Shekhar, Yogesh Barve, Shweta Khare, ***Anirban Bhattacharjee***, and Aniruddha Gokhale, "FECBench: An Extensible Framework for Pinpointing Sources of Performance Interference in Cloud-to-Edge hosted Applications," Tutorial at IEEE International Conference on Cloud Engineering (IC2E), Apr 2018.

4. Yogesh Barve, ***Anirban Bhattacharjee***, and Aniruddha Gokhale, "PADS A Model Driven Engineering Framework for Learning Distributed Systems Algorithms," Tutorial at the ACM/IEEE 20th International Conference on Model-driven Engineering Languages and Systems (MODELS), Sept 1722, 2017.

DOCTORAL SYMPOSIUM

1. ***Anirban Bhattacharjee***. "MDE-based Automated Provisioning and Management of Cloud Applications." In MODELS (Satellite Events). 2017.

BIBLIOGRAPHY

[1] D. Delen and H. Demirkan, “Data, information and analytics as services,” 2013.

[2] J. L. Berral-García, “A quick view on current techniques and machine learning algorithms for big data analytics,” in *2016 18th international conference on transparent optical networks (ICTON)*. IEEE, 2016, pp. 1–4.

[3] G. Ananthanarayanan, P. Bahl, P. Bodík, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha, “Real-time video analytics: The killer app for edge computing,” *computer*, vol. 50, no. 10, pp. 58–67, 2017.

[4] G. Martins, A. Bhattacharjee, A. Dubey, and X. D. Koutsoukos, “Performance evaluation of an authentication mechanism in time-triggered networked control systems,” in *2014 7th International Symposium on Resilient Control Systems (ISRCS)*. IEEE, 2014, pp. 1–6.

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

[6] A. Bhattacharjee, “Mde-based automated provisioning and management of cloud applications.” in *MODELS (Satellite Events)*, 2017, pp. 480–483.

[7] A. Bhattacharjee, Y. Barve, A. Gokhale, and T. Kuroda, “A model-driven approach to automate the deployment and management of cloud services,” in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE, 2018, pp. 109–114.

[8] ——, “(wip) cloudcamp: Automating the deployment and management of cloud services,” in *2018 IEEE International Conference on Services Computing (SCC)*. IEEE, 2018, pp. 237–240.

[9] A. Bhattacharjee, Y. Barve, S. Khare, S. Bao, A. Gokhale, and T. Damiano, "Stratum: A serverless framework for the lifecycle management of machine learning-based data analytics tasks," in *2019 {USENIX} Conference on Operational Machine Learning (OpML 19)*, 2019, pp. 59–61.

[10] A. Bhattacharjee, Y. Barve, S. Khare, S. Bao, Z. Kang, A. Gokhale, and T. Damiano, "Stratum: A bigdata-as-a-service for lifecycle management of iot analytics applications," in *IEEE International Conference on Big Data (Big Data)*. IEEE, 2019, pp. 1607–1612.

[11] A. Bhattacharjee, A. D. Chhokra, Z. Kang, H. Sun, A. Gokhale, and G. Karsai, "Barista: Efficient and scalable serverless serving system for deep learning prediction services," in *2019 IEEE International Conference on Cloud Engineering (IC2E)*, June 2019, pp. 23–33.

[12] J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia, "A survey on concept drift adaptation," *ACM computing surveys (CSUR)*, vol. 46, no. 4, p. 44, 2014.

[13] X. He, H. Zhang, M.-Y. Kan, and T.-S. Chua, "Fast matrix factorization for online recommendation with implicit feedback," in *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval*. ACM, 2016, pp. 549–558.

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

[15] D. C. Schmidt, "Model-driven engineering," *COMPUTER-IEEE COMPUTER SOCIETY-*, vol. 39, no. 2, p. 25, 2006.

[16] E. Bisong, "Kubeflow and kubeflow pipelines," in *Building Machine Learning and Deep Learning Models on Google Cloud Platform*. Springer, 2019, pp. 671–685.

[17] F. Tekiner and J. A. Keane, "Big data framework," in *Systems, Man, and Cybernetics (SMC), 2013 IEEE International Conference on*. IEEE, 2013, pp. 1494–1499.

[18] S. Shekhar, H. Abdel-Aziz, A. Bhattacharjee, A. Gokhale, and X. Koutsoukos, "Performance interference-aware vertical elasticity for cloud-hosted latency-sensitive applications," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 2018, pp. 82–89.

[19] S. Khare, H. Sun, J. Gascon-Samson, K. Zhang, A. Gokhale, Y. Barve, A. Bhattacharjee, and X. Koutsoukos, "Linearize, predict and place: minimizing the makespan for edge-based stream processing of directed acyclic graphs," in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, 2019, pp. 1–14.

[20] S. Shekhar, A. D. Chhokra, A. Bhattacharjee, G. Aupy, and A. Gokhale, "Indices: exploiting edge resources for performance-aware cloud-hosted services," in *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*. IEEE, 2017, pp. 75–80.

[21] Y. Barve, S. Shekhar, A. Chhokra, S. Khare, A. Bhattacharjee, and A. Gokhale, "Fecbench: An extensible framework for pinpointing sources of performance interference in the cloud-edge resource spectrum," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2018, pp. 331–333.

[22] H. Gupta, A. Vahid Dastjerdi, S. K. Ghosh, and R. Buyya, "ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments," *Software: Practice and Experience*, vol. 47, no. 9, pp. 1275–1296, 2017.

[23] P. Ravindra, A. Khochare, S. P. Reddy, S. Sharma, P. Varshney, and Y. Simmhan, "Echo: An adaptive orchestration platform for hybrid dataflows across cloud and edge," in *International Conference on Service-Oriented Computing*. Springer, 2017, pp. 395–410.

[24] E. Al-Masri, "Enhancing the microservices architecture for the internet of things," in *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 2018, pp. 5119–5125.

[25] T. Li, J. Zhong, J. Liu, W. Wu, and C. Zhang, "Ease. ml: Towards multi-tenant resource sharing for machine learning workloads," *Proceedings of the VLDB Endowment*, vol. 11, no. 5, pp. 607–620, 2018.

[26] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley, "Google vizier: A service for black-box optimization," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2017, pp. 1487–1495.

[27] D. Baylor, E. Breck, H.-T. Cheng, N. Fiedel, C. Y. Foo, Z. Haque, S. Haykal, M. Ispir, V. Jain, L. Koc *et al.*, "Tfx: A tensorflow-based production-scale machine learning platform," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2017, pp. 1387–1395.

[28] (2017) Meet michelangelo: Uber's machine learning platform. [Online]. Available: https://eng.uber.com/michelangelo/

[29] M. Ma, H. P. Ansari, D. Chao, S. Adya, S. Akle, Y. Qin, D. Gimnicher, and D. Walsh, "Democratizing production-scale distributed deep learning," *arXiv preprint arXiv:1811.00143*, 2018.

[30] W. Wang, J. Gao, M. Zhang, S. Wang, G. Chen, T. K. Ng, B. C. Ooi, J. Shao, and

M. Reyad, "Rafiki: machine learning as an analytics service system," *Proceedings of the VLDB Endowment*, vol. 12, no. 2, pp. 128–140, 2018.

[31] M. Zaharia, A. Chen, A. Davidson, A. Ghodsi, S. A. Hong, A. Konwinski, S. Murching, T. Nykodym, P. Ogilvie, M. Parkhe *et al.*, "Accelerating the machine learning lifecycle with mlflow," *Data Engineering*, p. 39, 2018.

[32] D. Crankshaw, G.-E. Sela, C. Zumar, X. Mo, J. E. Gonzalez, I. Stoica, and A. Tumanov, "Inferline: Ml inference pipeline composition framework," *arXiv preprint arXiv:1812.01776*, 2018.

[33] Y. Lee, A. Scolari, B.-G. Chun, M. Weimer, and M. Interlandi, "From the edge to the cloud: Model serving in ml .net," *Data Engineering*, p. 46, 2018.

[34] R. Chard, Z. Li, K. Chard, L. Ward, Y. Babuji, A. Woodard, S. Tuecke, B. Blaiszik, M. J. Franklin, and I. Foster, "Dlhub: Model and data serving for science," *arXiv preprint arXiv:1811.11213*, 2018.

[35] S. Zhao, M. Talasila, G. Jacobson, C. Borcea, S. A. Aftab, and J. F. Murray, "Packaging and sharing machine learning models via the acumos ai open platform," in *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2018, pp. 841–846.

[36] R. Di Cosmo, A. Eiche, J. Mauro, S. Zacchiroli, G. Zavattaro, and J. Zwolakowski, "Automatic deployment of services in the cloud with aeolus blender," in *Service-Oriented Computing*. Springer, 2015, pp. 397–411.

[37] N. Huber, F. Brosig, S. Spinner, S. Kounev, and M. Bahr, "Model-based self-aware performance and resource management using the descartes modeling language," *IEEE Transactions on Software Engineering*, 2016.

[38] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless computing: An investigation of factors influencing microservice performance," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2018, pp. 159–169.

[39] G. Granchelli, M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, and A. Di Salle, "Towards recovering the software architecture of microservice-based systems," in *Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on*. IEEE, 2017, pp. 46–53.

[40] K. Kumar, S. Sinha, and P. Manupriya, "D-pnr: Deep license plate number recognition," in *Proceedings of 2nd International Conference on Computer Vision & Image Processing*. Springer, 2018, pp. 37–46.

[41] K. M. M. Thein, "Apache kafka: Next generation distributed messaging system," *International Journal of Scientific Engineering and Technology Research*, vol. 3, no. 47, pp. 9478–9483, 2014.

[42] Y. D. Barve, P. Patil, A. Bhattacharjee, and A. Gokhale, "Pads: Design and implementation of a cloud-based, immersive learning environment for distributed systems algorithms," *IEEE Transactions on Emerging Topics in Computing*, vol. 6, no. 1, pp. 20–31, 2018.

[43] M. Maróti, R. Kereskényi, T. Kecskés, P. Völgyesi, and A. Lédeczi, "Online collaborative environment for designing complex computational systems," *Procedia Computer Science*, vol. 29, pp. 2432–2441, 2014.

[44] M. Claesen and B. De Moor, "Hyperparameter search in machine learning," *arXiv preprint arXiv:1502.02127*, 2015.

[45] (2018) Collectd - the system statistics collection daemon. [Online]. Available: https://collectd.org/

[46] (2018) Nvidia system management interface. [Online]. Available: https://developer.nvidia.com/nvidia-system-management-interface/

[47] (2018) Messaging that just works — rabbitmq. [Online]. Available: https://www.rabbitmq.com/

[48] (2018) Influxdb - time series database. [Online]. Available: https://www.influxdata.com/time-series-platform/influxdb/

[49] Y. Barve, P. Patil, A. Bhattacharjee, and A. Gokhale, "Pads: Design and implementation of a cloud-based, immersive learning environment for distributed systems algorithms," *IEEE Transactions on Emerging Topics in Computing*, 2017.

[50] OASIS, "Topology and orchestration specification for cloud applications," http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.pdf, 2013, oASIS Standard.

[51] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, and J. Wettinger, "Integrated cloud application provisioning: interconnecting service-centric and script-centric management technologies," in *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer, 2013, pp. 130–148.

[52] J. Humble and J. Molesky, "Why enterprises must adopt devops to enable continuous delivery," *Cutter IT Journal*, vol. 24, no. 8, p. 6, 2011.

[53] J. Carrasco, J. Cubo, F. Durán, and E. Pimentel, "Bidimensional cross-cloud management with tosca and brooklyn," in *Cloud Computing (CLOUD), 2016 IEEE 9th International Conference on*. IEEE, 2016.

[54] T. Eilam, M. Elder, A. V. Konstantinou, and E. Snible, "Pattern-based composite application deployment," in *12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops*. IEEE, 2011, pp. 217–224.

[55] H. Lu, M. Shtern, B. Simmons, M. Smit, and M. Litoiu, "Pattern-based deployment service for next generation clouds," in *Services (SERVICES), 2013 IEEE Ninth World Congress on*. IEEE, 2013, pp. 464–471.

[56] K. Képes, U. Breitenbücher, and F. Leymann, "The sepade system: Packaging entire xaas layers for automatically deploying and managing applications," *month*, 2017.

[57] L. Leite, C. E. Moreira, D. Cordeiro, M. A. Gerosa, and F. Kon, "Deploying large-scale service compositions on the cloud with the choreos enactment engine," in *Network Computing and Applications (NCA), 2014 IEEE 13th International Symposium on*. IEEE, 2014, pp. 121–128.

[58] D. Ardagna, E. Di Nitto, G. Casale, D. Petcu, P. Mohagheghi, S. Mosser, P. Matthews, A. Gericke, C. Ballagny, F. D'Andria *et al.*, "Modaclouds: A model-driven approach for the design and execution of applications on multiple clouds," in *Proceedings of the 4th International Workshop on Modeling in Software Engineering*. IEEE Press, 2012, pp. 50–56.

[59] S. Narain, G. Levin, S. Malik, and V. Kaul, "Declarative infrastructure configuration synthesis and debugging," *Journal of Network and Systems Management*, vol. 16, no. 3, pp. 235–258, 2008.

[60] E. Torlak and D. Jackson, "Kodkod: A relational model finder," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2007, pp. 632–647.

[61] J. Fischer, R. Majumdar, and S. Esmaeilsabzali, "Engage: a deployment management system," in *ACM SIGPLAN Notices*, vol. 47, no. 6. ACM, 2012, pp. 263–274.

[62] R. Di Cosmo, M. Lienhardt, R. Treinen, S. Zacchiroli, J. Zwolakowski, A. Eiche, and A. Agahi, "Automated synthesis and deployment of cloud applications," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 211–222.

[63] T. A. Lascu, J. Mauro, and G. Zavattaro, "A planning tool supporting the deployment of cloud applications," in *Tools with Artificial Intelligence (ICTAI), 2013 IEEE 25th International Conference on*. IEEE, 2013, pp. 213–220.

[64] P. Hirmer, U. Breitenbücher, T. Binz, F. Leymann *et al.*, "Automatic topology completion of tosca-based cloud applications." in *GI-Jahrestagung*, 2014, pp. 247–258.

[65] U. Breitenbucher, T. Binz, K. Képes, O. Kopp, F. Leymann, and J. Wettinger, "Combining declarative and imperative cloud application provisioning based on tosca," in *Cloud Engineering (IC2E), 2014 IEEE International Conference on*. IEEE, 2014, pp. 87–96.

[66] I. Giannakopoulos, N. Papailiou, C. Mantas, I. Konstantinou, D. Tsoumakos, and N. Koziris, "Celar: automated application elasticity platform," in *Big Data (Big Data), 2014 IEEE International Conference on*. IEEE, 2014, pp. 23–25.

[67] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[68] D. Bahdanau, J. Chorowski, D. Serdyuk, P. Brakel, and Y. Bengio, "End-to-end attention-based large vocabulary speech recognition," in *Acoustics, Speech and Signal Processing (ICASSP), 2016 IEEE International Conference on*. IEEE, 2016, pp. 4945–4949.

[69] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Autonomic vertical elasticity of docker containers with elasticdocker," in *Cloud Computing (CLOUD), 2017 IEEE 10th International Conference on*. IEEE, 2017, pp. 472–479.

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

[71] A. Gujarati, S. Elnikety, Y. He, K. S. McKinley, and B. B. Brandenburg, "Swayam: distributed autoscaling to meet slas of machine learning inference services with resource efficiency," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. ACM, 2017, pp. 109–120.

[72] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics." in *NSDI*, vol. 2, 2017, pp. 4–2.

[73] C. Delimitrou and C. Kozyrakis, "Quasar: resource-efficient and qos-aware cluster management," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 127–144, 2014.

[74] S. Venkataraman, Z. Yang, M. J. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics." in *NSDI*, 2016, pp. 363–378.

[75] S. K. Barker and P. Shenoy, "Empirical evaluation of latency-sensitive application performance in the cloud," in *Proceedings of the first annual ACM SIGMM conference on Multimedia systems*. ACM, 2010, pp. 35–46.

[76] Y. Ukidave, X. Li, and D. Kaeli, "Mystic: Predictive scheduling for gpu based cloud servers using machine learning," in *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE, 2016, pp. 353–362.

[77] Y. Barve, S. Shekhar, S. Khare, A. Bhattacharjee, and A. Gokhale, "Upsara: A model-driven approach for performance analysis of cloud-hosted applications," in *2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 2018, pp. 1–10.

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

[79] "AWS lambda," https://aws.amazon.com/serverless/, 2018.

[80] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, and S. Wagner, "Opentosca–a runtime for tosca-based cloud applications," in *International Conference on Service-Oriented Computing*. Springer, 2013, pp. 692–695.

[81] A. Aldhalaan and D. A. Menascé, "Near-optimal allocation of vms from iaas providers by saas providers," in *Cloud and Autonomic Computing (ICCAC), 2015 International Conference on*. IEEE, 2015, pp. 228–231.

[82] A. Brogi and S. Forti, "Qos-aware deployment of iot applications through the fog," *IEEE Internet of Things Journal*, vol. 4, no. 5, pp. 1185–1192, 2017.

[83] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *ACM SIGPLAN Notices*, vol. 48, no. 4. ACM, 2013, pp. 77–88.

[84] E. B. Lakew, C. Klein, F. Hernandez-Rodriguez, and E. Elmroth, "Towards faster response time models for vertical elasticity," in *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*. IEEE Computer Society, 2014, pp. 560–565.

[85] E. Kalyvianaki, T. Charalambous, and S. Hand, "Adaptive resource provisioning for virtualized servers using kalman filters," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 9, no. 2, p. 10, 2014.

[86] N. R. Herbst, N. Huber, S. Kounev, and E. Amrehn, "Self-adaptive workload classification and forecasting for proactive resource provisioning," *Concurrency and computation: practice and experience*, vol. 26, no. 12, pp. 2053–2078, 2014.

[87] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes, "Agile: Elastic distributed resource scaling for infrastructure-as-a-service." in *ICAC*, vol. 13, 2013, pp. 69–82.

[88] N. Vasić, D. Novaković, S. Miučin, D. Kostić, and R. Bianchini, "Dejavu: accelerating resource allocation in virtualized environments," in *ACM SIGARCH computer architecture news*, vol. 40, no. 1. ACM, 2012, pp. 423–436.

[89] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013, pp. 607–618.

[90] A. Y. Nikravesh, S. A. Ajila, and C.-H. Lung, "An autonomic prediction suite for cloud resource provisioning," *Journal of Cloud Computing*, vol. 6, no. 1, p. 3, 2017.

[91] S. Islam, J. Keung, K. Lee, and A. Liu, "Empirical prediction models for adaptive resource provisioning in the cloud," *Future Generation Computer Systems*, vol. 28, no. 1, pp. 155–162, 2012.

[92] M. B. Sheikh, U. F. Minhas, O. Z. Khan, A. Aboulnaga, P. Poupart, and D. J. Taylor, "A bayesian approach to online performance modeling for database appliances using gaussian models," in *Proceedings of the 8th ACM international conference on Autonomic computing*. ACM, 2011, pp. 121–130.

[93] S. J. Taylor and B. Letham, "Forecasting at scale," *The American Statistician*, no. just-accepted, 2017.

[94] R. N. Calheiros, E. Masoumi, R. Ranjan, and R. Buyya, "Workload prediction using arima model and its impact on cloud applications qos," *IEEE Transactions on Cloud Computing*, vol. 3, no. 4, pp. 449–458, 2015.

[95] N. Roy, A. Dubey, and A. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*. IEEE, 2011, pp. 500–507.

[96] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of grid computing*, vol. 12, no. 4, pp. 559–592, 2014.

[97] R. Wilcox, "Kolmogorov–smirnov test," *Encyclopedia of biostatistics*, 2005.

[98] H. G. Tucker, "A generalization of the glivenko-cantelli theorem," *Ann. Math. Statist.*, vol. 30, no. 3, pp. 828–830, 09 1959. [Online]. Available: https://doi.org/10.1214/aoms/1177706212

[99] T. Hastie and R. Tibshirani, "Generalized additive models: some applications," *Journal of the American Statistical Association*, vol. 82, no. 398, pp. 371–386, 1987.

[100] A. C. Harvey and N. Shephard, "Structural time series models," *Handbook of statistics*, vol. 11, no. 10, pp. 261–302, 1993.

[101] (2018) H2OAutoML. http://docs.h2o.ai/h2o/latest-stable/h2o-docs/_sources/automl.rst.txt.

[102] (2018) Amazon ec2 instances. https://aws.amazon.com/ec2/pricing/.

[103] (2018) Docker swarm. https://docs.docker.com/engine/swarm/.

[104] (2018) NYC taxi and limousine commission. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page.

[105] (2018) NYC thruway. https://catalog.data.gov/dataset/nys-thruway-origin-and-destination-points-for-all-vehicles-15-minute-intervals-latest-full.

[106] F. Milletari, N. Navab, and S.-A. Ahmadi, "V-net: Fully convolutional neural networks for volumetric medical image segmentation," in *2016 Fourth International Conference on 3D Vision (3DV)*. IEEE, 2016, pp. 565–571.

[107] L. Huang, X. Dong, and T. E. Clee, "A scalable deep learning platform for identifying geologic features from seismic attributes," *The Leading Edge*, vol. 36, no. 3, pp. 249–256, 2017.

[108] C. Chen, A. Seff, A. Kornhauser, and J. Xiao, "Deepdriving: Learning affordance for direct perception in autonomous driving," in *Proceedings of the IEEE International Conference on Computer Vision*, 2015, pp. 2722–2730.

[109] H. Tian, M. Yu, and W. Wang, "Continuum: A platform for cost-aware, low-latency continual learning," in *Proceedings of the ACM Symposium on Cloud Computing*, 2018, pp. 26–40.

[110] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283.

[111] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[112] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan *et al.*, "Ray: A distributed framework for emerging {AI} applications," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 561–577.

[113] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: an efficient dynamic resource scheduler for deep learning clusters," in *Proceedings of the Thirteenth EuroSys Conference*. ACM, 2018, p. 3.

[114] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang *et al.*, "Gandiva: Introspective cluster scheduling for

deep learning," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 595–610.

[115] M. Jeon, S. Venkataraman, A. Phanishayee, J. Qian, W. Xiao, and F. Yang, "Analysis of large-scale multi-tenant gpu clusters for dnn training workloads," in *2019 USENIX Annual Technical Conference ({USENIX}{ATC} 19)*, 2019.

[116] "Embedded Systems Developer Kits & Modules from NVIDIA Jetson." [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/

[117] "Products | Coral." [Online]. Available: https://coral.ai/products/

[118] Z. Zhao, K. M. Barijough, and A. Gerstlauer, "Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2348–2359, 2018.

[119] L. Zhou, H. Wen, R. Teodorescu, and D. H. Du, "Distributing deep neural networks with containerized partitions at the edge," in *2nd {USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 19)*, 2019.

[120] J. Jiang, B. Cui, C. Zhang, and L. Yu, "Heterogeneity-aware distributed parameter servers," in *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 463–478.

[121] H. Cui, H. Zhang, G. R. Ganger, P. B. Gibbons, and E. P. Xing, "Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server," in *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016, p. 4.

[122] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu, "Petuum: A new platform for distributed machine learning on big data," *IEEE Transactions on Big Data*, vol. 1, no. 2, pp. 49–67, 2015.

[123] A. Sergeev and M. Del Balso, "Horovod: fast and easy distributed deep learning in tensorflow," *arXiv preprint arXiv:1802.05799*, 2018.

[124] J. Chen and X. Ran, "Deep learning with edge computing: A review," *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1655–1674, 2019.

[125] A. Koliousis, P. Watcharapichat, M. Weidlich, L. Mai, P. Costa, and P. Pietzuch, "Crossbow: Scaling deep learning with small batch sizes on multi-gpu servers," *arXiv preprint arXiv:1901.02244*, 2019.

[126] T. Ben-Nun and T. Hoefler, "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis," *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, p. 65, 2019.

[127] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems*, 2015, pp. 1–17.

[128] V. Jalaparti, P. Bodik, I. Menache, S. Rao, K. Makarychev, and M. Caesar, "Network-aware scheduling for data-parallel jobs: Plan when you can," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 407–420, 2015.

[129] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger, "Tetrisched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters," in *Proceedings of the Eleventh European Conference on Computer Systems*, 2016, pp. 1–16.

[130] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, Í. Goiri, S. Krishnan, J. Kulkarni *et al.*, "Morpheus: Towards automated slos for enterprise clusters," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 117–134.

[131] Q. Chen, H. Yang, J. Mars, and L. Tang, "Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 681–696, 2016.

[132] Q. Chen, H. Yang, M. Guo, R. S. Kannan, J. Mars, and L. Tang, "Prophet: Precise qos prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017, pp. 17–32.

[133] H. Zhang, L. Stafman, A. Or, and M. J. Freedman, "Slaq: quality-driven scheduling for distributed machine learning," in *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 2017, pp. 390–404.

[134] P. Sun, Y. Wen, N. B. D. Ta, and S. Yan, "Towards distributed machine learning in shared clusters: A dynamically-partitioned approach," in *2017 IEEE International Conference on Smart Computing (SMARTCOMP)*. IEEE, 2017, pp. 1–6.

[135] E. Aleksandrova, C. Anagnostopoulos, and K. Kolomvatsos, "Machine learning model updates in edge computing: An optimal stopping theory approach," in *2019 18th International Symposium on Parallel and Distributed Computing (ISPDC)*. IEEE, 2019, pp. 1–8.

[136] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in

Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture. ACM, 2011, pp. 248–259.

[137] Y. D. Barve, S. Shekhar, A. Chhokra, S. Khare, A. Bhattacharjee, Z. Kang, H. Sun, and A. Gokhale, "Fecbench: A holistic interference-aware approach for application performance modeling," in *2019 IEEE International Conference on Cloud Engineering (IC2E)*, June 2019, pp. 211–221.

[138] S. Chen, C. Delimitrou, and J. F. Martínez, "Parties: Qos-aware resource partitioning for multiple interactive services," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2019, pp. 107–120.

[139] R. Xu, S. Mitra, J. Rahman, P. Bai, B. Zhou, G. Bronevetsky, and S. Bagchi, "Pythia: Improving datacenter utilization via precise contention prediction for multiple co-located workloads," in *Proceedings of the 19th International Middleware Conference*. ACM, 2018, pp. 146–160.

[140] J. Zhao, X. Feng, H. Cui, Y. Yan, J. Xue, and W. Yang, "An empirical model for predicting cross-core performance interference on multicore processors," in *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE, 2013, pp. 201–212.

[141] N. Mishra, J. D. Lafferty, and H. Hoffmann, "Esp: A machine learning approach to predicting application interference," in *2017 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, 2017, pp. 125–134.

[142] S. Shekhar, A. Chhokra, H. Sun, A. Gokhale, A. Dubey, and X. Koutsoukos, "Urmila: A performance and mobility-aware fog/edge resource management middleware," in *2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, 2019, pp. 118–125.

[143] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "Carla: An open urban driving simulator," *arXiv preprint arXiv:1711.03938*, 2017.

[144] Y. Tian, K. Pei, S. Jana, and B. Ray, "DeepTest: Automated testing of deep-neural-network-driven autonomous cars," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 303–314.

[145] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang *et al.*, "End to end learning for self-driving cars," *arXiv preprint arXiv:1604.07316*, 2016.

[146] L. Fei-Fei, R. Fergus, and P. Perona, "Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories," in *2004 conference on computer vision and pattern recognition workshop*. IEEE, 2004, pp. 178–178.

[147] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.

[148] "cking/stress-ng.git - Unnamed repository; edit this file 'description' to name the repository." [Online]. Available: https://kernel.ubuntu.com/git/cking/stress-ng.git/

[149] "Home - Open Source Leader in AI and ML." [Online]. Available: https://www.h2o.ai/

[150] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?" in *Advances in neural information processing systems*, 2014, pp. 3320–3328.

[151] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpa-

thy, A. Khosla, M. Bernstein *et al.*, "Imagenet large scale visual recognition challenge," *International journal of computer vision*, vol. 115, no. 3, pp. 211–252, 2015.

[152] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

[153] "GitHub - dmlc/xgboost: Scalable, Portable and Distributed Gradient Boosting (GBDT, GBRT or GBM) Library, for Python, R, Java, Scala, C++ and more. Runs on single machine, Hadoop, Spark, Flink and DataFlow." [Online]. Available: https://github.com/dmlc/xgboost

[154] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*, 2013, pp. 1–16.

[155] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI*, vol. 11, no. 2011, 2011, pp. 22–22.