RESOURCE EFFICIENT INFERENCE SERVING WITH SLO GUARANTEE

Vom Fachbereich Informatik der Technischen Universität Darmstadt genehmigte

Dissertation

zur Erlangung des akademischen Grades eines Doctor rerum naturalium (Dr. rer. nat.).

Eingereicht von
Kamran Razavi, M.Sc.

Erstreferent: Prof. Dr. Lin Wang
Korreferent: Prof. Dr. Max Mühlhäuser
Tag der Einreichung: 12.08.2024
Tag der Disputation: 23.09.2024

Fachbereich Informatik
Telecooperation Lab
Technische Universität Darmstadt
Hochschulkennziffer D17
Darmstadt 2024

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Kamran Razavi: *Resource Efficient Inference Serving With SLO Guarantee:*

Darmstadt, Technische Universität Darmstadt

Tag der Einreichung: 12.08.2024
Tag der Disputation: 23.09.2024

URN: urn:nbn:de:tuda-tuprints-286159
URI: https://tuprints.ulb.tu-darmstadt.de/id/eprint/28615
Jahr der Veröffentlichung der Dissertation auf TUprints: 2024

Urheberrechtlich Geschützt
In Copyright: https://rightsstatements.org/page/InC/1.0/

ABSTRACT

Deep Learning (DL) has gained popularity in various online applications, including intelligent personal assistants, image and speech recognition, and question interpretation. Serving applications composed of DL models resource efficiently is challenging because of the computational intensity of the DL models with multiple layers and a large number of parameters, stringent Service Level Objective (SLO) (e.g., end-to-end serving latency requirements), and data dependency between the DL models. Efficiently managing computing resources becomes even more challenging in the presence of dynamic workloads.

The primary focus of this thesis is the following overarching question: How can we design, implement, and deploy resource-efficient DL-based inference serving systems while ensuring a guaranteed SLO under both predictable and unpredictable workloads? In response to this question, this thesis presents four contributions aimed at enhancing the resource efficiency of DL-based applications and enabling the serving of DL models on nontraditional computing resources, such as programmable switches.

First, we improve the multi-model serving system efficiency by increasing the system utilization in a work named, *FA2*: Fast, Accurate Autoscaling for Serving Deep Learning Inference with SLA Guarantees. *FA2* introduces a graph-based model to capture the resource allocation and batch size joint configurations and data dependency in DL inference serving systems and presents a horizontal-scaling-based resource allocation algorithm leveraging graph transformation and dynamic programming.

Second, to guarantee the SLO while accounting for the dynamic conditions on the communication network between the user device and the serving system, we use the responsiveness benefit of vertical scaling and propose a new work titled *Sponge*: Inference Serving with Dynamic SLOs Using In-Place Vertical Scaling. *Sponge* employs in-place vertical scaling mechanisms to instantaneously adjust computing resources based on the demand, strategically reorders incoming re-

quests to prioritize those with the most constrained remaining latency budgets, and utilizes dynamic batching techniques to increase the utilization of the system. Furthermore, to enhance user satisfaction, we introduce model variants—different DL models with varying cost, accuracy, and latency properties for the same request—by using vertical scaling and changing the DL model and the computing resources in a work named *IPA*: Inference Pipeline Adaptation to Achieve High Accuracy and Cost-Efficiency. *IPA* introduces a multi-model accuracy metric to calculate the end-to-end accuracy by defining a new accuracy metric and using multiplication on this metric to approximate the overall accuracy of the application.

Third, we explore how to achieve both responsiveness and resource efficiency. When the workload becomes unpredictable, the SLO violation can increase in the DL inference serving system. In-place vertical scaling can respond quickly to dramatic workload changes, but it is not as resource-efficient as horizontal scaling. We explore the use of both horizontal and vertical scaling simultaneously in a paper called *Biscale*: Integrating Horizontal and Vertical Scaling for Inference Serving Systems. In *Biscale*, we analyze the effect of both scaling mechanisms in terms of serving speed up using Amdahl's law and address three key questions of why, how, and when to switch between horizontal and vertical scaling mechanisms to guarantee the SLO and optimize the computing resources.

Finally, we investigate the feasibility of serving DL models in programmable network devices for network security tasks like intrusion detection. Since programmable network devices are responsible for moving the data (including inference requests), serving inference requests directly on the programmable network devices accelerates the computation and eliminates the need for external computing resources. To achieve this, we design a novel method to divide a DL model into multiple parts and assign each part, along with its specific computing requirements, to a set of programmable network devices. We further train an intrusion detection DL model and utilize this approach in a paper titled *NetNN*: Neural Intrusion Detection System in Programmable Networks.

In conclusion, this thesis comprehensively addresses the challenges regarding the main research question, providing innovative solutions to enhance the resource efficiency of DL inference serving systems while ensuring stringent SLO guarantees under various workloads.

ZUSAMMENFASSUNG

Deep Learning (DL) hat in verschiedenen Online-Anwendungen an Popularität gewonnen, darunter intelligente persönliche Assistenten, Bild- und Spracherkennung und Frageninterpretation. Die ressourceneffiziente Bereitstellung von Anwendungen, die aus DL-Modellen bestehen, ist aufgrund der Rechenintensität der DL-Modelle mit mehreren Schichten und einer großen Anzahl von Parametern, strengen Service Level Objectives (SLO) (z. B. End-to-End-Latenzanforderungen) und der Datenabhängigkeit zwischen den DL-Modellen eine Herausforderung. Die effiziente Verwaltung von Rechenressourcen wird noch schwieriger, wenn dynamische Arbeitslasten vorhanden sind.

Das Hauptaugenmerk dieser Arbeit liegt auf der folgenden übergreifenden Frage: Wie können wir ressourceneffiziente DL-basierte Inferenzservicesysteme entwerfen, implementieren und einsetzen und dabei eine garantierte SLO sowohl unter vorhersehbaren als auch unvorhersehbaren Arbeitslasten gewährleisten? Als Antwort auf diese Frage werden in dieser Arbeit vier Beiträge vorgestellt, die darauf abzielen, die Ressourceneffizienz DL-basierter Anwendungen zu verbessern und die Bereitstellung von DL-Modellen auf nicht-traditionellen Rechenressourcen, wie z.B. programmierbaren Switches, zu ermöglichen.

Erstens verbessern wir die Effizienz von Multi-Modell-Serving-Systemen durch Erhöhung der Systemauslastung in einer Arbeit namens *FA2*: Schnelle, genaue Autoskalierung für die Bereitstellung von Deep Learning-Inferenzen mit SLA-Garantien. *FA2* führt ein graphenbasiertes Modell ein, um die Ressourcenzuweisung und die Stapelgröße gemeinsamer Konfigurationen und die Datenabhängigkeit in DL-Inferenz-Servicing-Systemen zu erfassen, und stellt einen auf horizontaler Skalierung basierenden Ressourcenzuweisungsalgorithmus vor, der Graphentransformation und dynamische Programmierung nutzt.

Zweitens, um die SLO zu garantieren und gleichzeitig die dynamischen Bedingungen im Kommunikationsnetzwerk zwischen dem Benutzergerät und dem Serving-System zu berücksichtigen, nutzen wir den Vorteil der vertikalen Skalierung und schlagen eine neue Arbeit

mit dem Titel *Sponge* vor: Inference Serving mit dynamischen SLOs mit vertikaler Skalierung vor Ort. *Sponge* verwendet Mechanismen zur vertikalen Skalierung an Ort und Stelle, um Rechenressourcen sofort an die Nachfrage anzupassen, ordnet eingehende Anfragen strategisch neu an, um diejenigen mit den am stärksten eingeschränkten verbleibenden Latenzbudgets zu priorisieren, und nutzt dynamische Stapelverarbeitungstechniken, um die Auslastung des Systems zu erhöhen. Um die Benutzerzufriedenheit zu erhöhen, führen wir außerdem Modellvarianten ein - verschiedene DL-Modelle mit unterschiedlichen Kosten-, Genauigkeits- und Latenz-Eigenschaften für dieselbe Anfrage -, indem wir die vertikale Skalierung nutzen und das DL-Modell und die Rechenressourcen in einer Arbeit namens *IPA* ändern: Inference Pipeline Adaptation to Achieve High Accuracy and Cost-Efficiency. *IPA* führt eine Multi-Modell-Genauigkeitsmetrik ein, um die End-to-End-Genauigkeit zu berechnen, indem eine neue Genauigkeitsmetrik definiert und die Multiplikation dieser Metrik verwendet wird, um die Gesamtgenauigkeit der Anwendung zu approximieren.

Drittens untersuchen wir, wie man sowohl Reaktionsfähigkeit als auch Ressourceneffizienz erreichen kann. Wenn die Arbeitslast unvorhersehbar wird, kann die SLO-Verletzung im DL Inference Serving System zunehmen. Die vertikale Skalierung an Ort und Stelle kann schnell auf dramatische Änderungen der Arbeitslast reagieren, ist aber nicht so ressourceneffizient wie die horizontale Skalierung. Wir untersuchen die gleichzeitige Verwendung von horizontaler und vertikaler Skalierung in einem Papier mit dem Titel *Biscale*: Integrating Horizontal and Vertical Scaling for Inference Serving Systems. In *Biscale* analysieren wir die Auswirkungen beider Skalierungsmechanismen in Bezug auf die Serving-Geschwindigkeit unter Verwendung des Amdahl-Gesetzes und befassen uns mit den drei Schlüsselfragen, warum, wie und wann zwischen horizontalen und vertikalen Skalierungsmechanismen gewechselt werden sollte, um die SLO zu garantieren und die Rechenressourcen zu optimieren.

Schließlich untersuchen wir die Machbarkeit der Bereitstellung von DL-Modellen in programmierbaren Netzwerkgeräten für Netzwerksicherheitsaufgaben wie die Erkennung von Eindringlingen. Da programmierbare Netzwerkgeräte für die Übertragung der Daten (einschließlich der Inferenzanfragen) verantwortlich sind, beschleunigt die Bereitstellung von Inferenzanfragen direkt auf den programmierbaren Netzwerkgeräten die Berechnung und eliminiert den Bedarf an ex-

ternen Rechenressourcen. Um dies zu erreichen, entwickeln wir eine neuartige Methode, um ein DL-Modell in mehrere Teile aufzuteilen und jeden Teil zusammen mit seinen spezifischen Rechenanforderungen einer Reihe von programmierbaren Netzwerkgeräten zuzuweisen. Darüber hinaus trainieren wir ein DL-Modell zur Erkennung von Eindringlingen und verwenden diesen Ansatz in einem Papier mit dem Titel *NetNN*: Neural Intrusion Detection System in programmierbaren Netzwerken.

Zusammenfassend lässt sich sagen, dass diese Arbeit die Herausforderungen in Bezug auf die Hauptforschungsfrage umfassend adressiert und innovative Lösungen bietet, um die Ressourceneffizienz von DL-Inferenzsystemen zu verbessern und gleichzeitig strenge SLO-Garantien unter verschiedenen Arbeitslasten zu gewährleisten.

ACKNOWLEDGMENTS

I want to take this page and thank the people who were there for me during the PhD journey.

Lin, you were not just an academic supervisor but someone who I could talk to and rely on in hard times. Getting a PhD is difficult, but your advice and support softened it to a great extent. Cheers to you.

Max, you have been a great mentor by offering invaluable advice and providing opportunities for me to grow both academically and personally. Thanks for that.

With my colleagues, friends, and family, this journey became a much more enjoyable experience. Uwe and Florian, thanks for being amazing officemates. Thanks to Niko, Thomas, Leon, Simon, and other TK members, who would have spared their time whenever I was in their office for discussions or fun. George, Vinod, Mehran, Saeid, Alireza, Shayan, and Mo, it was a great pleasure collaborating with you.

Kai, your presence was just enough to make me calm during the defense day. Mom and Dad, thanks for your support from day one of my life. Kaveh, thanks for being a source of academic inspiration. Raha, thanks for making discussions about research fun. Niyayesh, it is always nice to talk about science to someone in Iran. Maaike and Shaun, thanks for bringing joy whenever there is an opportunity.

And, Nafise, supporting my decision to pursue a PhD, moving to a different continent to make it happen, and being always there for me are just a few things you did. Thanks for being in my life.

CONTENTS

I Synopsis

1 Introduction 1
- 1.1 Research Questions . . . 3
  - 1.1.1 Resource Efficiency . . . 4
  - 1.1.2 Responsiveness and Accuracy . . . 6
  - 1.1.3 Unpredictable Workloads . . . 7
  - 1.1.4 In-Network DL Intrusion Detection . . . 8
- 1.2 Contributions . . . 9
- 1.3 Outline . . . 10

2 State of the Art 11
- 2.1 Resource Scaling in Inference Serving . . . 11
- 2.2 Inference Offloading to Network Devices . . . 17

3 Thesis Organization and Contributions 21
- 3.1 Resource Efficiency . . . 21
  - 3.1.1 Research Approach . . . 22
  - 3.1.2 Main Findings . . . 24
- 3.2 Responsiveness and Accuracy . . . 25
  - 3.2.1 Research Approach . . . 27
  - 3.2.2 Main Findings . . . 28
- 3.3 Unpredictable Workloads . . . 29
  - 3.3.1 Research Approach . . . 30
  - 3.3.2 Main Findings . . . 31
- 3.4 In-network DL Intrusion Detection . . . 32
  - 3.4.1 Research Approach . . . 33
  - 3.4.2 Main Findings . . . 34
- 3.5 Summary . . . 35

4 Conclusion and Outlook 37
- 4.1 Summary of Achievements . . . 37
  - 4.1.1 Resource Efficiency . . . 37
  - 4.1.2 Responsiveness and Accuracy . . . 38
  - 4.1.3 Unpredictable Workloads . . . 38
  - 4.1.4 In-Network DL Intrusion Detection . . . 39
  - 4.1.5 Open Source Inference Serving . . . 39
- 4.2 Future Work . . . 40

4.2.1 Heterogeneous Hardware Framework 40
4.2.2 Inference Pipeline Accuracy 40
4.2.3 In-Network Mapper 41

II Publications

P1 FA2 55
P1.1 Introduction 56
P1.2 Background and Motivation 58
P1.2.1 Deep Learning Inference Serving 58
P1.2.2 Resource Autoscaling 59
P1.2.3 Autoscaling Challenges in DL Inference Serving 61
P1.3 FA2 System Design 63
P1.4 Autoscaling Problem and Algorithm 65
P1.4.1 DNN Performance Modeling 66
P1.4.2 Queuing Delay Modeling 66
P1.4.3 Problem Formulation 67
P1.4.4 Graph Transformation 69
P1.4.5 Scaling with Dynamic Programming 71
P1.5 Implementation 72
P1.6 Evaluation 73
P1.6.1 Experimental Setup 74
P1.6.2 End-to-End Performance 79
P1.6.3 Resource Efficiency 82
P1.6.4 Impact of Optimized Batch Sizes 83
P1.6.5 Performance of FA2 on GPUs 85
P1.7 Related Work 86
P1.8 Conclusion 88
P1.9 Acknowledgments 89
P2 Sponge 97
P2.1 Introduction 97
P2.2 Motivation 100
P2.2.1 Dynamic SLO 100
P2.2.2 Autoscaling Challenges 103
P2.3 System Design 104
P2.3.1 Overview 104
P2.3.2 Performance Model 105
P2.3.3 Problem Formulation 107
P2.3.4 Solution 108
P2.4 Preliminary Evaluation 109
P2.5 Related Work 110
P2.6 Conclusion & Future Work 112

P2.7 Acknowledgments 112
P3 IPA 115
P3.1 Introduction 116
P3.2 Background and Motivation 119
P3.2.1 Search Space 119
P3.2.2 Configuration Space 121
P3.2.3 Inference Pipelines 122
P3.3 System Design 123
P3.4 Problem Formulation 126
P3.4.1 Accuracy Definition over Pipeline 126
P3.4.2 Profiler 128
P3.4.3 Optimization Formulation 130
P3.4.4 Gurobi Solver 133
P3.4.5 Dropping 135
P3.5 Evaluation 135
P3.5.1 Experimental Setup 135
P3.5.2 End-to-End Evaluation 139
P3.5.3 IPA Scalability 143
P3.5.4 IPA Adaptability 145
P3.5.5 IPA Predictor 145
P3.6 Related Works 146
P3.7 Conclusion and Future Works 147
P3.8 Acknowledgments 148
P3.9 Appendix 149
P3.10 Pipelines Stages Specifications 149
P3.11 Constant Multipliers Values 154
P3.12 Alternate Inference Pipeline Accuracy Definition 154
P4 Biscale 167
P4.1 Introduction 168
P4.2 Background and Motivation 171
P4.3 Biscale 174
P4.3.1 Challenges 174
P4.3.2 System Design 175
P4.3.3 Assumptions and Limitations 177
P4.4 Autoscaling Problem 178
P4.4.1 Performance Profiling 178
P4.4.2 Queue 179
P4.4.3 Problem Formulation 180
P4.4.4 Optimizer 181
P4.5 Transition 184
P4.5.1 Vertical Scaling to Horizontal Scaling 185

P4.5.2 Horizontal Scaling to Vertical Scaling 188
P4.6 Experimental Evaluation 189
P4.6.1 End-to-End Performance 192
P4.6.2 Intra and Inter Parallelism 198
P4.6.3 Request Dropping Effect 198
P4.7 Related Work 200
P4.8 Conclusion 201
P4.9 Acknowledgments 202
P5 Distributed DNN Serving 209
P5.1 Introduction 209
P5.2 A Case Study with mini-AlexNet 211
P5.3 Challenges and Discussion 216
P5.4 Acknowledgments 217
P6 NetNN 221
P6.1 Introduction 221
P6.2 Motivation 224
P6.2.1 Feature Extraction for Intrusion Detection . . . 224
P6.2.2 Our Goal 226
P6.2.3 Intrusion Detection on Programmable Switches 226
P6.3 System Design 228
P6.3.1 System Overview 228
P6.3.2 Mapper 229
P6.3.3 Neural Network Executor 232
P6.3.4 Packet Generator 234
P6.4 Evaluation 236
P6.4.1 Resource Usage 236
P6.4.2 Impact of Parameters 236
P6.5 Related Work 238
P6.6 Conclusion 238
P6.7 Acknowledgments 239

LIST OF FIGURES

Figure 2.1 An inference serving system with multiple DL models and execution paths. 12

Figure 2.2 Horizontal Scaling (left) and Vertical Scaling (right). Horizontal scaling adjusts the number of instances for a DL model where each instance comes with a base resource configuration and distributes the workload to multiple instances, while vertical scaling changes the computing resources of the available instance(s). 13

Figure 2.3 Protocol Independent Switch Architecture (PISA). 18

Figure P1.1 The execution graph of an example modern intelligent application taking images as input. Depending on the object recognition result, different execution paths will be taken for each inference request. 58

Figure P1.2 Performance comparison between horizontal and vertical scaling policies with respect to throughput and latency on the object detection (OBJD) model in Table P1.2. Each data point represents the result obtained with a batch size in the range of [1,9] respectively. 60

Figure P1.3 Number of instances required by different autoscaling policies. InferLine neglects the combinatorial nature of the models and iterates on all models one by one and GrandSLAm uses a predefined stack allocation for DNN models, both resulting in need of extra resources (3x and 2x, respectively) compared with the optimal resources provided by Gruboi. 62

Figure P1.4 An overview of the FA2 architecture. The monitoring service collects the metric data from the DL inference serving system. The optimizer makes scaling decisions for the DNNs. The controller enforces the scaling decisions by configuring the inference serving system. 63
Figure P1.5 Graph transformation example: (left) the original dataflow graph, (right) the transformed graph containing only egress aggregators. . . . 69
Figure P1.6 Inference latency distribution for the considered DNNs under varying batch sizes on CPU (an instance equipped with one CPU core). . . 74
Figure P1.7 Inference latency distribution for the considered DNNs under varying batch sizes on GPU (on an instance equipped with 10% of the total GPU processing units specified using CUDA MPS). 75
Figure P1.8 Workloads used in evaluation: two steady synthetic traces following Poisson distributions, a fluctuating synthetic trace with a spike in the middle, a random workload following a normal distribution, and two realistic traces from the Azure FaaS function invocation traces. 79
Figure P1.9 CPU utilization of five DNN instances for all approaches under the steady workload at a given time-window. 80
Figure P1.10 Resource usage and SLA violation over time under fluctuating (left) and random (right) workloads. 81
Figure P1.11 Resource usage and SLA violation over time under two (left and right) Azure FaaS workload traces. 81
Figure P1.12 Comparison of resource consumption under varying request rates on CPUs. FA2 shows clear advantages over all the existing approaches and is comparable to the theoretical optimum. . . . 83
Figure P1.13 End-to-end latency distribution for the three execution paths in the application under different scaling approaches on CPUs. FA2 outperforms all the existing approaches to a large extent with SLA violations less than 1%. 84

Figure P1.14 Comparison of resource consumption under varying request rates on GPUs. FA2 shows clear advantages over all the existing approaches and is comparable to the theoretical optimum. . . . 85
Figure P1.15 End-to-end latency distribution for the six execution paths in the application under different scaling approaches on GPUs. FA2 outperforms all the existing approaches to a large extent with SLA violations around 1%. 86
Figure P2.1 Bandwidth measurements in 4G networks provided by [20]. The bandwidth varies from 0.5MB/s to 7MB/s in a 10-minute range (top figure). The below figure demonstrates the remaining SLO for processing when the user sends a 100 KB, a 200 KB, or a 500 KB image over the same network's bandwidth. 101
Figure P2.2 An overview of the Sponge architecture. The monitoring service collects metric data from the DL model. The queue prioritizes requests according to the EDF policy. The scaler is responsible for determining vertical scaling and batch size decisions for the DL model and adjusting the system accordingly. 104
Figure P2.3 Latency vs. different CPU core allocations and batch sizes using real and predicted for the YOLOv5n and ResNet18 DL models. 106
Figure P2.4 SLO violations and allocated CPU cores. 111
Figure P3.1 IPA provides a tunable framework for adjusting the system based on two contradictory cost and accuracy objectives. 117
Figure P3.2 Performance difference across ResNet Family models for a batch size of one and one CPU core allocation. 120
Figure P3.3 Impact of configuration knobs, batching indirectly affects the cost, e.g., decreasing the throughput will affect the IPA to more scaling and increase in the cost. 121
Figure P3.4 IPA system design. It consists of an offline phase for model profiling and an online phase for adaptive inference serving. 124

Figure P3.5 Switching between different configurations under (a) low and (b) high loads. 126
Figure P3.6 Representative pipelines used in this work. . . 134
Figure P3.7 Representative tested load patterns from the Twitter trace[5], showing LSTM predictions. . 135
Figure P3.8 Performance analysis of the Video pipeline. . . 137
Figure P3.9 Performance analysis of the Audio-qa pipeline. 138
Figure P3.10 Performance analysis of the Audio-sent pipeline.139
Figure P3.11 Performance analysis of the Sum-qa pipeline. . 140
Figure P3.12 Performance analysis of the NLP pipeline. . . 141
Figure P3.13 Decision time of Gurobi optimizer for IPA formulation with respect to the number of models and tasks on the inference graph. 142
Figure P3.14 Comparison of IPA results for different tradeoffs between accuracy and cost objectives, IPA can navigate effectively between the two cost and accuracy objectives. 143
Figure P3.15 End-to-end latency distribution for the five tested inference pipelines under different approaches. IPA can achieve latency close to the FA2-low with light model variants, and only RIM is achieving better latency at the expense of high resource over-provisioning. 143
Figure P3.16 Effect of using predictor on reducing SLA violations on bursty workload, IPA LSTM can reduce SLA violations up to 10x with the same resource usage. Also, using baseline predictors with perfect knowledge about the future reveals potential room for reduction of SLA with better predictors. 144
Figure P3.17 Performance analysis of the Video pipeline. . . 155
Figure P3.18 Performance analysis of the Sum-qa pipeline. . 156
Figure P4.1 Vertical scaling vs. horizontal scaling reaction time in case of workload bursts. Horizontal scaling is not responsive. The gray area indicates SLO violation. 172
Figure P4.2 Using vertical and horizontal scaling jointly to absorb bursts and reduce operational costs. Vertical scaling provides responsiveness, while horizontal scaling provides cost efficiency. The gray area indicates SLO violation. 172

Figure P4.3 An overview of the Biscale architecture. The executor receives the requests and processes them. The monitor service collects the metric data from the stages. The optimizer makes both vertical and horizontal scaling decisions for the DL models. The Adapter enforces the scaling decisions by configuring the queues and the executor. 175
Figure P4.4 Transition states between vertical and horizontal scaling strategies. 185
Figure P4.5 LSTM inference result. 188
Figure P4.6 Performance profile of Translator and Classifier DL models with different CPU and batch configurations. 192
Figure P4.7 End-to-end comparison on the video monitoring pipeline. Biscale reduces the SLO violation to roughly 0.1% by using both scaling mechanisms jointly. Horizontal and vertical scaling mechanisms violate 2.4% and 39.3% of the requests' SLO, respectively. 194
Figure P4.8 End-to-end comparison on the audio sentiment analysis pipeline. Biscale reduces the SLO violation to less than 0.5%. Horizontal scaling violates 6.5% of the requests, while vertical scaling violates most of the requests, reaching over 72% of the total SLO violation. 196
Figure P4.9 End-to-end comparison on the natural language processing pipeline. Biscale reduces the SLO violation to less than 3.8%. Horizontal scaling violates 50.7% of the requests SLO, and vertical scaling violates 96.7% of the requests SLO. . . 197
Figure P4.10 The effect of Intra and Inter parallelism parameters on the YOLOv5n and ResNet18 models with 4 CPU cores. Inter parallelism parameter does not affect the processing latency when there is one batch at a time in the DL model. . 199
Figure P4.11 The effect of different dropping strategies. The 1xSLO dropping strategy helps reduce the total number of SLO violations. 200
Figure P5.1 An overview of in-network DNN serving. . . . 211
Figure P5.2 The spine/leaf network architecture. 213

Figure P6.1 The DNN architecture used in NetNN with two Conv1D layers (Conv2D with the second dimension as one), followed by a flatten and three dense layers. 225
Figure P6.2 Comparison of NetBeacon, Random, and NetNN for both packet and flow classification. 225
Figure P6.3 An overview of the NetNN architecture. 227
Figure P6.4 The workings of NetNN Mapper 230
Figure P6.5 The target network architecture. 234
Figure P6.6 Inference points. 237
Figure P6.7 Input representation. 237

LIST OF TABLES

Table 2.1 Efficiency: Does this work use horizontal scaling for the highest resource efficiency? Responsiveness: Does this approach use in-place vertical scaling to respond quickly to the changes in the workload? Accuracy: Does this system use model variants as a way of vertical scaling to increase accuracy? 15
Table 2.2 In-network ML approaches. DT: Decision Tree, RF: Random Forest, XGB: XGBoost, SVM: Support Vector Machines, NB: Naive Bayes, KM: K-Means, BDT: Binary Decision Tree, BNN: Binary Neural Network 19
Table P1.1 Notations 65
Table P1.2 DNNs Involved in the Application 76
Table P1.3 All Possible Execution Paths in the Application with Their SLAs When Deployed on CPUs and GPUs 77
Table P2.1 Execution latency (P99) of a ResNet model (human detector) with different CPU cores using different batch sizes while guaranteeing SLO of 1000 ms under the workload of 100 RPS. . . . 102
Table P2.2 Notations 107

Table P3.1 Comparison of IPA with previous works; Pipeline: A chain of models inference or just single model inference? Cost: Whether the work optimizes cost? Accuracy: Does it optimize accuracy? Adaptive: Can it adapt to different accuracy/-cost optimization trade-offs? 117
Table P3.2 Performance difference across ResNet family models under different CPU allocations for a batch size of one, both blue and red core/model configuration can respond to 20 RPS and 75 ms throughput and latency requirements with different accuracy and costs. 120
Table P3.3 Two-stage pipeline tasks options. Latency is in Milliseconds and Cost is the number of physical cores. 123
Table P3.4 Notations 127
Table P3.5 Sample CPU cores base allocation for different YOLO variants under different RPS thresholds (Capped on maximum 32 cores). 129
Table P3.6 per-stage and end-to-end SLA of inference pipelines (in seconds). 136
Table P3.7 Object Detection Task Models 149
Table P3.8 Object Classification Task Models 150
Table P3.9 Audio Task Models 150
Table P3.10 Question Answering Task Models 151
Table P3.11 Summarisation Task Models 151
Table P3.12 Sentiment Analysis Task Models 152
Table P3.13 Language Identification Task Models 153
Table P3.14 Neural Machine Translation Task Models . . . 153
Table P3.15 Pipelines objectives multiplier values 154
Table P4.1 Comparison of Biscale with previous works; Pipeline: A chain of models or just a single model? Resource Efficiency: Does this work use horizontal scaling for the highest resource efficiency? Responsiveness: Does this approach use in-place vertical scaling to respond quickly to the changes in the workload? (∗) uses model-variants as a way of vertical scaling. 173
Table P4.2 Notations 179
Table P4.3 DL Models and Applications 191

Table P5.1 The mini-AlexNet network. It contains seven layers, with over 15 million parameters and 91 million operations requiring less than 16MB of memory to store. 212
Table P6.1 DNN in-switch performance. Memory is in byte. 237

Part I

SYNOPSIS

1 Introduction 1
  1.1 Research Questions 3
  1.2 Contributions 9
  1.3 Outline 10
2 State of the Art 11
  2.1 Resource Scaling in Inference Serving 11
  2.2 Inference Offloading to Network Devices . . 17
3 Thesis Organization and Contributions 21
  3.1 Resource Efficiency 21
  3.2 Responsiveness and Accuracy 25
  3.3 Unpredictable Workloads 29
  3.4 In-network DL Intrusion Detection 32
  3.5 Summary 35
4 Conclusion and Outlook 37
  4.1 Summary of Achievements 37
  4.2 Future Work 40

1

INTRODUCTION

Deep learning (DL) has been at the forefront of many recent advancements in artificial intelligence. DL models have demonstrated extraordinary performance in tasks such as intelligent personal assistant [10, 48], image and speech recognition [3, 12], question interpretation, and text-to-speech/speech-to-text [28, 44]. This extraordinary performance is primarily due to the complex architectures of DL models, which consist of multiple layers [51]. These layers extract features from the data and help the model understand and process complex patterns. These layers are typically composed of neurons that transform the input through weighted connections. During training, these weights are adjusted to reduce errors and improve accuracy [8]. However, the power of these models is not just in their ability to learn from data (training phase) but also in their ability to make predictions or “inferences" on new, unseen data (inference phase), where inference serving systems come into play [11].

Inference serving systems take trained DL models and make them available to applications that need to make predictions. These systems handle the complexities of serving inferences from these DL models, including load balancing, scaling to handle varying workloads, and ensuring low latency responses [2, 37]. Providing low latency responses and dealing with varying workloads are crucial for applications where predictions need to be made in real time, such as in autonomous driving, intrusion detection, and personalized recommendations [9, 16, 17, 28, 63].

One of the primary challenges in building an inference serving system is managing resources efficiently [10, 43–45]. DL models are computationally expensive and demand significant memory and processing power. They are becoming increasingly larger, with the current state-of-the-art models having trillions of parameters [33]. Therefore, inference serving systems need to be able to adjust computing resources quickly in response to changes in the workload to avoid over-provisioning

when the demand is low, but that capacity is available when the demand is high.

In addition to managing resources, inference serving systems must ensure that they meet the desired Service Level Objective (SLO) [10, 34, 35, 48]. These are measures of the quality of service provided by the system, such as the latency of responses (the end-to-end latency), accuracy (the percentage of correct predictions), or the system's throughput (the rate at which the system can process inference requests over a given period). Meeting these requirements is crucial for ensuring that the applications using the DL models can function correctly and provide a good user experience.

Modern inference serving systems employ several techniques to manage resources while satisfying the requirements:

1. **Resource Autoscaling**. Resource autoscaling refers to dynamically changing the number of instances, called horizontal scaling, or the computing power of instances, named vertical scaling, of a DL model in response to the changes in the workload and SLOs [18, 48]. In this dissertation, we consider autoscaling computing resources since the monetary expense from cloud providers is primarily proportional to the allocated computing resources [15].

2. **Batching**. Batching is a technique where several requests are grouped and processed simultaneously. This method enhances the system's efficiency by improving the system's throughput by handling multiple requests at once, allowing the system to process more requests per unit of time, thereby increasing the overall resource utilization [4, 10].

3. **Model Switching**. Model variants are DL models with different accuracy, latency, and resource consumption for the same DL task. Prioritizing any property (accuracy, latency, or resource consumption) affects other properties of the same DL model and directly impacts the whole inference serving system regarding accuracy, latency, and resource cost.

4. **Request Reordering**. Request reordering improves system performance by strategically ordering the processing of tasks to maximize resource utilization, where requests (with the least

remaining latency budget) are prioritized to maximize system utilization and user satisfaction [28].

5. **Offloading**. Inference offloading is the process of transferring the inference computations from a local device to more powerful remote processing units [4, 58]. Programmable network devices, primarily used for network management simplification, have been recently explored for application-specific computations, known as in-network computing [46, 61, 62]. Network devices are suitable candidates for DL-based network security tasks since they have high performance (high throughput, low latency) for packet processing and are strategically located inside the network. However, they have limited memory (tens of hundreds of megabytes) and limited processing operations (lacking support for floating point operations, as well as integer division and multiplication).

None of the above techniques, when considered in isolation, is capable of deploying DL models that meet the requirements of modern applications. Moreover, these techniques, when applied together, can have complex interactions and may even conflict with each other. Therefore, we need DL inference serving systems that reconcile and integrate these techniques and provide holistic solutions.

1.1 RESEARCH QUESTIONS

We are now aware of DL inference serving systems requirements where the responses must be delivered within a tight SLO, the more accurate model variants provide higher quality results but also may harm the SLO, different resource scaling mechanisms, and the challenges regarding offloading DL models to the programmable network devices. This leads us to ask the following overarching research question and investigate the possible solutions in this thesis:

How to design, implement, and deploy resource-efficient DL inference serving systems with SLO guarantee?

To answer the above research question, we investigate by asking the following four sub-questions and aim to find answers to these sub-questions.

Q_1: How can computing resources be optimally allocated to DL models in a multi-model serving system to guarantee the SLO?

Q_2: How can DL models be effectively deployed to provide fast response to system dynamics and improve overall inference accuracy?

Q_3: How must multi-model serving systems respond to unpredictable workload changes to achieve resource efficiency while minimizing the SLO violation rate?

Q_4: How can non-conventional computing resources be leveraged to improve the serving system efficiency?

1.1.1 *Resource Efficiency*

To answer Q_1, we use horizontal scaling to allocate resources optimally to multi-model serving systems while guaranteeing the SLO. Existing work has focused on horizontal scaling in inference serving systems due to its superior ability to handle increasing workloads and cost-efficiency [10, 44]. However, the state-of-the-art designs of resource scalers primarily rely on separating the batch size from the decision-making process of autoscaling, employing simple heuristics to solve the autoscaling problem, which results in sub-optimized resource allocations for such systems.

Allocating resources optimally to DL models in multi-model serving systems by considering the batch size and the resource allocation jointly poses new challenges that need to be addressed. First, in a multi-model serving system, one DL model's output serves as the next DL model's input. The models for the application form a directed acyclic graph (DAG), where the nodes are the models, and the links indicate the flow of data and the order of the models' execution. The DAG structure allows the provisioning of each model to be done independently. However, in the DAG structure, data dependency happens between different models, meaning that changing a model's configuration may affect the downstream models. For example, increasing the number of instances of a DL model in response to the increased workload results in an increased arrival rate of the downstream models. When the arrival rate of a (downstream) model is changed, the num-

ber of instances of the DL model may need to be changed to support the incoming workload. Therefore, all the models in the application should be considered when one of the model's configurations is going to be changed to avoid under or over-provisioning of resources in the application.

Next, inference requests can be batched to increase the system's performance. Instead of processing requests one by one, requests can be grouped and processed together to leverage the parallel processing capabilities of the DL model and the hardware. Batch sizes can be dynamically changed based on the current situation of the DL model, such as the workload intensity and the allocated processing power to the DL model. Dynamically changing batch sizes can enhance the system's efficiency by improving its throughput through the simultaneous handling of multiple requests. This allows the system to process more requests per unit of time, thus increasing the overall resource utilization. However, this method harms the request processing and queuing latencies since the first request in the batch must wait for the arrival of other requests to form a batch, and larger batches take longer to process. Moreover, changing the batch size of a single model results in changes to both its throughput and latency, which impacts downstream models due to their interdependencies. It is crucial to understand how inference processing latency is affected by DL models under dynamic batch sizes.

Finally, applications with different objectives require different SLOs. An autonomous driving application requires an end-to-end latency of a few tens of milliseconds [52], while an intelligent personal assistant application allows a deadline of a few hundred milliseconds or even seconds [10, 44]. These deadline differences result in having multiple latency requirements in the system. This requirement, in addition to the batch size and the data dependency of the DL models in the system, increases the solution space for finding the optimal resource allocation exponentially [43]. Hence, exploring all possible solutions to find the optimal solution in real time is hard. Therefore, having approaches to provide solutions as close as possible to the optimal solutions is of the most desire to avoid over-provisioning or under-provisioning.

By considering the above challenges, after analyzing the horizontal scalers in different domains such as inference serving, stream processing, and microservices, we propose a novel horizontal scaler that considers both batch size and resource allocation jointly with dif-

ferent SLOs for different applications to optimize resources while guaranteeing the SLO, to answer the first research question.

1.1.2 *Responsiveness and Accuracy*

The network bandwidth fluctuates in mobile environments where inference requests have to be sent from a mobile device to an inference server over a non-stable network, e.g., WiFi or 4G/5G [34]. On the other hand, the SLO is mainly defined from end to end, including the variable network time required for transferring user requests and input data [23, 56]. Ignoring the network latency, inference serving systems might lack sufficient time to process requests adequately, leading to violations of the SLOs. To account for this issue, the application on the server side needs to consider network latency and compensate for the reduced inference latency budget by adjusting its configurations.

Vertical scaling can be used to adjust the configurations (i.e., computing resources or the model variant) of the DL models. Unlike horizontal scaling that uses a base configuration (i.e., both computing resources and the model variant), other affecting parameters, such as batch sizes, are directly affected by changing those configurations. Therefore, it becomes crucial to understand how inference processing latency is affected by DL models under different computing resource and batch size configurations.

In-place vertical scaling changes the computing resources of an instance of a running DL model in-spot without the cold start to enable responsiveness [24]. This feature can be used to capture the network bandwidth fluctuations. The main challenge of in-place vertical scaling is: How many cores or fractions of a core are sufficient to support the workload for the DL model being executed on CPUs?

Moreover, model variants can be used to change the tradeoff between accuracy, latency, and resource cost of the inference requests. We use vertical scaling (with cold start) to change not only the computing resources of the instances of the DL model but also a different model variant to increase user satisfaction. However, finding relationships between the three conflicting model variant properties (i.e., accuracy, latency, and resource cost) in multi-model serving is challenging.

We address Q_2 by proposing two new inference serving systems: the first one considers the unstable communication networks by using the combination of in-place vertical scaling, dynamic batching, and request reordering to guarantee the requests' SLO, and the second one incorporates the model variants into the vertical scaling with cold start to increase the inference accuracy.

1.1.3 *Unpredictable Workloads*

As discussed above, in-place vertical scaling provides responsiveness when more computing resources can be added to cut down the inference processing latency. Conversely, horizontal scaling is beneficial due to its superior ability to handle increasing workloads and resource efficiency. Therefore, a natural follow-up to the previous research questions would be how to leverage in-place vertical scaling under unpredictable workloads to enable responsiveness and how to leverage the benefits of horizontal scaling to achieve resource efficiency when the serving system does not require the expensive responsiveness that comes with the in-place vertical scaling.

Combining these two scaling mechanisms is challenging since both of them change the computing resources of the DL models. Horizontal scaling changes the number of instances of the DL models with the same base configurations. In contrast, vertical scaling changes the base configurations, making the use of both scaling mechanisms simultaneously impractical. Thus, the question of which scaling mechanism must be used should be answered based on the workload situation. Consequently, the workload situation must be clarified, i.e., is the workload stable enough to start saving resources, or is responsiveness still the required criteria to guarantee the SLO?

Furthermore, if multiple instances are needed to support the workload, how much computing resources should be allocated for each instance of the same DL model? Should they all have the same base configuration, or does a diverse set of computing configurations provide more resource efficiency and responsiveness?

To address the above challenges regarding Q_3, we introduce a new autoscaler that enables responsiveness in response to sudden changes in the workload by using in-place vertical scaling and increases resource

efficiency when the workload becomes stable by using horizontal scaling.

1.1.4 *In-Network DL Intrusion Detection*

As DL progresses, the significance and complexity of inference serving systems are growing in parallel, making DL inference serving highly computation intensive.

Programmable network devices enhance application components by leveraging their high throughput, low latency processing capabilities, and strategic on-path placement. Specially, intrusion detection on programmable network devices (e.g., based on decision trees) has become popular due to the increasing demand for real time network security [6, 32, 61–63]. DL-based intrusion detection can extract features from data, learn intricate relationships between input and output variables, and find new patterns without human intervention. Performing DL-based intrusion detection directly in those devices reduces processing overhead and network traffic, leading to faster and more efficient detection.

However, programmable network devices must use simple packet processing instructions in their data plane programs to ensure line rate processing, which results in imposing a fixed processing time, limiting the number and complexity of operations, such as floating-point operations, loops, and even integer multiplication/division. Actions associated with the memory of the programmable network devices are restricted to simple operations like additions and bit shifts. Additionally, the memory architecture of those devices imposes constraints on data structures. Moreover, the stateful memory available is limited to tens of megabytes, and accessing all the available registers can be challenging due to access restrictions within the processing stages.

To tackle Q_4, we study the intersection of DL models for intrusion detection and in-network computing. We analyze the feasibility of deploying DL models within network devices to leverage their unique capabilities without requiring hardware modifications. Our proposed approach consists of three key steps: first, we partition the DL model and map its components across a set of programmable network devices; second, we emulate the inference execution flow using network

packets; and finally, we issue precise mathematical execution instructions to these devices to perform the inference.

1.2 CONTRIBUTIONS

This thesis aims to advance state-of-the-art resource efficiency in inference serving systems. The primary focus is introducing novel techniques and algorithms to guarantee inference execution latency while reducing the operational cost of serving inference requests. This thesis comprises six research papers (five peer-reviewed and published and one currently under submission) between 2022 and 2024. Four papers address the challenges of inference serving resource efficiency on general-purpose computing resources such as CPUs and GPUs, while the other two papers focus on enabling in-network inference serving. Finally, To support developers and increase community engagement, we have open-sourced four of our publications implementations (*Sponge*, *IPA*, *NetNN*, and *Biscale*). *FA2* algorithm is used for evaluating both *IPA* and *Biscale* and *Distributed Serving* does not have an implementation.

The results of our contributions can be summarized as follows:

- Resource Efficiency: An autoscaling mechanism based on horizontal scaling, *FA2*, that guarantees the requests SLO and increases the system's utilization (hence, reducing the required amount of computing resources). *FA2* uses the system's DAG to encapsulate the resource allocation problem into an optimization problem and solves it using graph transformation and dynamic programming.

- Responsiveness and Accuracy: A new in-place vertical resource scaler, *Sponge*, adjusts the DL model's computational resource based on the requests' remaining latency budget (thus avoiding SLO violation). *Sponge* calculates the remaining request's latency budget caused by the dynamism in the network bandwidth and provides a greedy algorithm to change the CPU resources of the DL model instantly. Furthermore, we design a new resource scaler, *IPA*, that uses model switching to reconcile DL applications' latency, accuracy, and computational cost properties. *IPA* guarantees SLOs by encapsulating the conflicting properties

(latency, accuracy, and computational cost) into a mathematical model and solving it using an optimizer, consequently increasing inference accuracy and reducing operational costs.

- Unpredictable Workloads: A novel resource scaler that uses horizontal and vertical scaling jointly, named *Biscale*. After analyzing the advantages and disadvantages of each autoscaling mechanism, *Biscale* leverages the responsiveness of vertical scaling to absorb the sudden changes in the workload and then transitions to horizontal scaling to save resources.

- In-Network DL Intrusion Detection: A novel distributed in-network DL inference framework for fast in-network inference execution. *NetNN* and *Distributed Serving* enable in-network DL intrusion detection inference by mapping the weights of the DL model to a set of network devices, mimicking the DL inference dataflow with network packet routing, and instructing network devices to execute DL model computations.

1.3 OUTLINE

This thesis consists of two parts. Part i contains the synopsis, and Part ii contains the cumulative publications. Chapter 2 of Part i introduces the general state-of-the-art inference serving systems (excluding the contributions of this thesis). Chapter 3 introduces each contribution and its primary results. Finally, Chapter 4 provides a conclusion of the thesis and outlines future work. Part ii is the cumulative part of this thesis and introduces each publication in its original form with minor edits for readability. Chapters P1-P6 each contains one publication. Chapter P4 has been submitted to 15th ACM Symposium on Cloud Computing (SoCC'24) and has been published to arXiv under a different paper title (A Tale of Two Scales: Reconciling Horizontal and Vertical Scaling for Inference Serving Systems) and a different system name (*Themis*).

2 STATE OF THE ART

Inference serving systems primarily consist of multiple DL models encapsulated within microservices [10, 28]. These microservices are orchestrated in a graph-like structure, where the output of one DL model acts as the input for the next downstream DL model, as depicted in Figure 2.1.

The microservice-based architecture for inference serving allows for complex workflows and data transformations, enabling the system to handle complex tasks efficiently by leveraging the strengths of each DL model in the sequence. Furthermore, the microservice-based architecture enables independent deployment, scaling, and management of each DL model, allowing resources to be allocated precisely where needed. When the demand for a particular DL model increases, only the corresponding DL model needs to be scaled up rather than the entire system. This granularity in scaling enhances resource utilization and cost efficiency, as it prevents over-provisioning and ensures that computational power is directed to the most critical components of the system.

The following sections discuss the main resource scaling mechanisms, namely horizontal and vertical scaling, and their advantages and disadvantages. After that, we talk about the challenges regarding in-network DL inference serving.

2.1 RESOURCE SCALING IN INFERENCE SERVING

Resource scaling can be achieved through horizontal and vertical scaling techniques, as shown in Figure 2.2.

Horizontal Scaling: Horizontal scaling, also known as scaling in/out, involves removing/adding more DL model instances to distribute the load. This method effectively manages increased workloads by

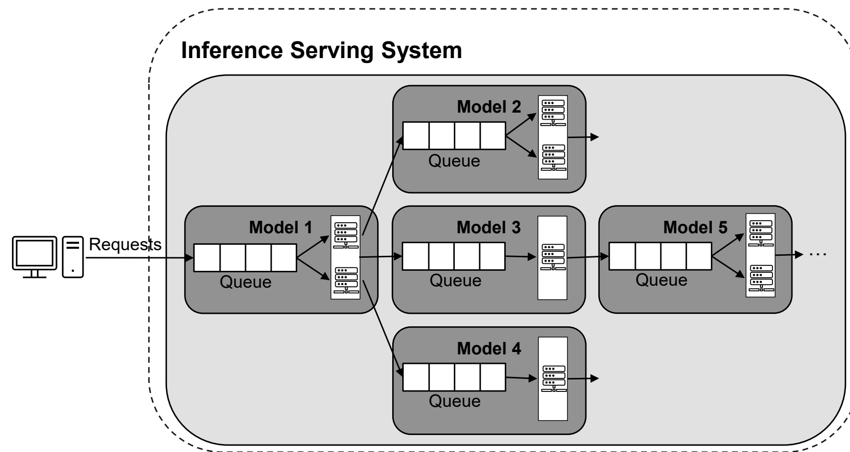


Figure 2.1: An inference serving system with multiple DL models and execution paths.

distributing incoming requests across multiple instances, enhancing the system's capacity to handle higher workloads without compromising performance. By deploying multiple instances of a DL model, the system can balance incoming requests across these instances, reducing the risk of bottlenecks. Moreover, distributing the workload across multiple instances allows the system to dynamically adjust resource allocation based on the current demand, ensuring that resources are neither under-provisioned nor wasted, thus maximizing efficiency and effectiveness.

A notable challenge associated with horizontal scaling in DL inference systems is the issue of cold start [10, 21, 43, 45]. When new instances are added to handle increased demand, they often need to initialize and load the DL models into memory, which can be time consuming. This initialization delay, known as cold start, can significantly impact the system's ability to respond promptly to sudden changes in the workload. Cold starts are particularly problematic for DL models that require substantial computational resources and time to load large library dependencies or substantial model data into memory. During this period, the system may experience reduced performance and increased latency, potentially leading to a suboptimal user experience caused by violating the SLO. To mitigate the effects of cold start, strategies such as pre-warming instances [58], maintaining a pool of ready-to-use instances [34], or lightweight model variants can be employed [60]. These approaches aim to minimize instance startup time and ensure that new instances can quickly start serving requests. However, their responsiveness comes with the reduced quality of the

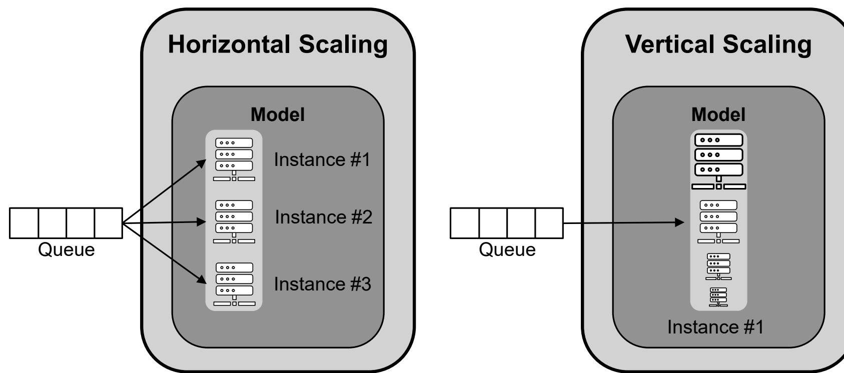


Figure 2.2: Horizontal Scaling (left) and Vertical Scaling (right). Horizontal scaling adjusts the number of instances for a DL model where each instance comes with a base resource configuration and distributes the workload to multiple instances, while vertical scaling changes the computing resources of the available instance(s).

generated results (lightweight model variants) or the extra operational costs (keeping extra DL models running).

Vertical Scaling: Vertical scaling, or scaling up/down, involves increasing/decreasing the computational resources of existing instances, such as adding more CPU cores. Vertical scaling is mainly associated with cold start, meaning that the instance needs to be restarted again. During the restart, the computing resources of the instances can be adjusted to allow using model variants. This approach enhances the capacity of instances to handle more workloads

Vertical scaling is more straightforward to implement initially since it does not require architectural changes or the management of multiple instances and implications like load balancing. Additionally, vertical scaling allows for increased computing power of an instance, which can reduce the end-to-end processing latency of the current DL model or enable the use of a different model variant to optimize accuracy-latency performance tradeoffs. Moreover, in-place vertical scaling allows changing the computing resources of the instances on the spot, increasing or decreasing the latency performance of the running DL models, which can be beneficial to reduce the SLO violation in case of a surge of inference requests.

One of the primary limitations is the finite capacity for upgrades of an instance. Each instance has a maximum upper limit for CPU, beyond which it cannot be scaled up further. This constraint can become a bottleneck, especially as the demands on the DL model grow or

in cases of model variant switching. Another challenge in vertical scaling is the cost and latency tradeoffs. Due to the sequential nature of data pre-processing, the internal architecture of the DL model, and post-processing, not all components of a DL model can benefit from additional computing resources. These stages, including those that must be executed sequentially, limit the overall scalability and resource allocation efficiency of vertical scaling.

Hybrid Scaling: In practice, a combination of both scaling techniques is desirable to maximize efficiency by employing horizontal scaling to distribute the load across multiple instances and vertical scaling to absorb the sudden changes in the workload. Vertical scaling is particularly effective at absorbing sudden changes in the workload, as it allows for rapidly adding computational power to existing instances. This can significantly reduce inference times and improve throughput for the DL models during unexpected surges in demand. Horizontal scaling is beneficial when the workload becomes stable, as it enables the system to bring up multiple instances with minimal computing resources and distribute the load across them, avoiding resource waste caused by the sequential nature of the DL model. This hybrid approach allows microservice-based DL inference systems to maintain high performance and cost-effectiveness.

One significant challenge is the increased complexity of system architecture and management. Hybrid scaling requires sophisticated orchestration to dynamically adjust both the number of instances (horizontal scaling) and the resources of each instance (vertical scaling). Coordinating resource allocations of both techniques with the transitions needed between the two scaling approaches in real-time to ensure optimal performance without over-provisioning or underutilizing resources can be complex.

Table 2.1 summarizes the state-of-the-art approaches, excluding our works in this dissertation, that use either horizontal scaling for resource efficiency or vertical scaling for responsiveness in response to changes in the system. Additionally, we describe each approach and point out its strengths and limitations.

INFaaS [43] gathers user preferences regarding accuracy, cost, or performance and dynamically provides the most accurate model variant that meets the specified latency and cost requirements based on the workload. However, they do not consider the challenges regarding hor-

Table 2.1: Efficiency: Does this work use horizontal scaling for the highest resource efficiency? Responsiveness: Does this approach use in-place vertical scaling to respond quickly to the changes in the workload? Accuracy: Does this system use model variants as a way of vertical scaling to increase accuracy?

System	Multi-models	Efficiency	Responsiveness	Accuracy
INFaaS [43]	✗	✓	✗	✓
Rim [22]	✓	✗	✗	✓
InferLine [10]	✓	✓	✗	✗
GrandSLAm [28]	✓	✗	✗	✗
Scrooge [21]	✓	✓	✗	✗
Cocktail [19]	✗	✗	✓	✓
InfAdapter [45]	✗	✓	✗	✓
Llama [44]	✓	✓	✗	✗
Model Switch [60]	✗	✗	✗	✓
Jellyfish [34]	✗	✗	✓	✓
Nexus [47]	✓	✓	✗	✗
MArk [59]	✗	✓	✗	✗

izontal scaling and data dependency caused by having multi-models in their approach.

Rim [22] is a DL-based video and audio processing management system that leverages model variants to increase accuracy while guaranteeing the application's desired latency. However, they do not consider horizontal scaling when the workload exceeds the available DL models' capacity.

InferLine [10] facilitates autoscaling by initially establishing a feasible system configuration and subsequently optimizing it by adjusting each DL model's hardware and batch size. This approach aims to enhance system performance and resource utilization through tailored configurations for different workloads. However, the heuristic method for selecting configurations for each model often results in significant resource underutilization. This is because the heuristic may only partially capture the complexities and dynamic nature of workloads, leading to suboptimal allocation and efficiency in resource usage.

GrandSLAm [28] is a microservice-based inference framework designed to enhance throughput while guaranteeing the SLO. It provides a robust solution for managing microservices to maintain high performance and reliability. However, GrandSLAm does not address

the challenge of autoscaling, either horizontal or vertical. This limitation means that while it increases the system throughput, it lacks mechanisms for dynamically adjusting resources in response to fluctuating workloads, which is crucial for maintaining efficiency and cost effectiveness.

Scrooge [21] proposes a scheduler for inference serving systems that decouples the resource placement and resource allocations that dynamically re-adapt to the changes in the workload. However, they do not consider the vertical scaling and the challenges regarding the accuracy, latency, and cost of multi variants.

Cocktail [19] introduces an ensemble learning approach aimed at reducing costs while maintaining the desired latency and accuracy. By leveraging ensembling to maximize accuracy, Cocktail ensures high-quality predictions. However, this method involves sending each request to multiple multi variants, which can be costly. The need to process requests across several multi variants increases resource consumption and operational expenses, potentially offsetting the cost-saving benefits that Cocktail aims to achieve. Also, they do not consider pipelines in their approach.

InfAdapter [45] leverages multiple model variants for the same task to increase the model's overall accuracy. They also proactively use horizontal scaling to react to changes in the workload. However, they do not consider pipelines in their workload adaptation mechanism.

Llama [44] is a specialized pipeline configuration system designed for video inference applications. It aims to minimize the computational costs by dynamically changing the hardware type, ensuring efficient video processing. However, they do not consider vertical scaling and the accuracy of the results.

Model Switch [60] dynamically switches between lightweight and heavier DL models in response to workload adaptation. It transitions to a less accurate DL model to respond to sudden workload surges efficiently, thereby maintaining responsiveness and performance under varying load conditions. However, they do not consider horizontal scaling and data dependency in their workload adaptation mechanism.

Jellyfish [34] is a GPU-based inference serving system that dynamically switches to different model variants in response to network bandwidth changes to guarantee the accuracy and latency requirements. However,

they do not consider horizontal scaling, and their work supports a single model.

Nexus [48] addresses the autoscaling problem by primarily targeting scenarios involving simple tree-like dataflow graphs, simplifying the complexity of managing dependencies and data flow within the system. Additionally, Nexus allocates a significant portion of the SLO to queuing delays, reserving half the time for these delays. While this approach ensures that queuing delays do not excessively impact performance, it also leads to reduced resource utilization.

MArk [59] proposes autoscaling policies based on request batching and workload prediction and enhances resource utilization and efficiency across diverse computational environments. However, MArk works on a single model and does not consider the accuracy of the predictions or vertical scaling.

While existing works have advanced the resource efficiency, responsiveness, and accuracy of DL inference serving systems, they still fail to address challenges regarding the joint batch size and resource allocation problem to increase the resource efficiency, vertical scaling for multi-model serving with responsiveness and inference accuracy improvements, and a joint horizontal and vertical resource scaler.

2.2 INFERENCE OFFLOADING TO NETWORK DEVICES

Programmable data planes have become a promising solution for offloading a variety of tasks in the last few years with the rise in demand for high-speed and low-latency data processing. As a result, new opportunities have emerged for network applications in multiple domains [25, 26, 53, 57]. Especially, there has been a growing interest in exploring the potential of in-network computing to enhance the performance of machine learning tasks [30, 31, 46]. Further attempts have been made to leverage programmable switches and SmartNICs to execute per-packet network security tasks like intrusion detection, including decision trees, support vector machines, and binary neural networks, thereby facilitating data-plane packet classification [49, 50, 55].

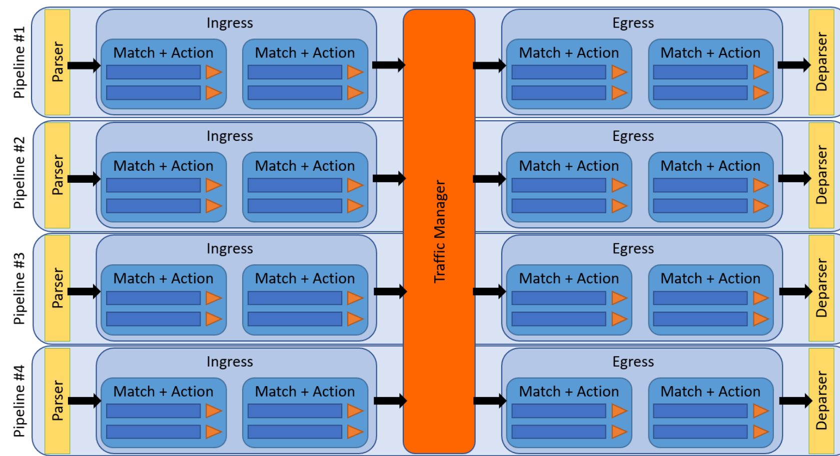


Figure 2.3: Protocol Independent Switch Architecture (PISA).

Modern programmable switches use the Protocol Independent Switch Architecture (PISA) as depicted in Figure 2.3. When a packet enters the PISA switch, it undergoes a series of stages in the processing pipeline. Each stage is capable of performing specific packet processing operations instructed by the P4 programming language [7]. The packet flows from one stage to the next, and at each stage, the processing parts within that stage process the packet based on their configured functionality.

Despite their flexibility and programmability, programmable switches do have certain hardware limitations that can impact their performance and capabilities; 1) Processing constraint: To ensure line-rate processing in programmable switches, data plane programs (e.g., written in P4) must use simple packet processing instructions. Each pipeline stage in the switch imposes a fixed processing time on packets, limiting the number and complexity of operations performed per stage. Consequently, complex operations such as floating-point calculations, loops, and integer multiplication/division are typically unsupported. 2) Memory constraint: The memory architecture of programmable switches further constrains the data structures used in P4 programs. The available stateful memory in programmable switches is limited, typically in tens of megabytes. Accessing all available registers can also be challenging, as registers within one stage of the processing pipeline cannot be accessed from different stages. This architectural limitation necessitates careful planning and optimizing memory usage to ensure efficient and effective packet processing within the constrained environment.

Table 2.2: In-network ML approaches. DT: Decision Tree, RF: Random Forest, XGB: XGBoost, SVM: Support Vector Machines, NB: Naive Bayes, KM: K-Means, BDT: Binary Decision Tree, BNN: Binary Neural Network

System	Task	ML Algorithms	Device
IIsy [61]	Packet/Flow Classification	DT, RF, XGB, SVM, NB, KM	Tofino FPGA
FlowLens [6]	Flow Classification	DT, RF	Tofino
Mousika [54]	Data Classification	BDT	Tofino
NetBeacon [63]	Packet/Flow Classification	DT, RF	Tofino
N3IC [49]	Traffic Analysis	BNN	SmartNIC

Table 2.2 summarizes the state-of-the-art in-network inference serving deployed directly on the switch, excluding our works in this dissertation. Additionally, we describe each approach and point out their strengths.

IIsy [61] can deploy various classification algorithms without hardware modifications, including decision trees, random forest, XGBoost, K-means, support vector machines, and Naïve Bayes. The framework includes a control plane component that maps a trained model to the switch hardware by converting model parameters into match-action table entries, allowing model parameter changes without altering the P4 program.

FlowLens [6] leverages programmable switches to enhance machine learning-based network security applications. By collecting features of packet distributions directly on the switches, FlowLens enables real-time classification of network flows. This integration allows for efficient and immediate network traffic analysis, improving the detection and mitigation of security threats while minimizing latency.

Mousika [54] deploys binary decision trees into the data plane using much fewer resources than other approaches by employing a top-down knowledge distillation architecture to translate complex machine learning models into simpler binary decision trees, resulting in fewer classification rules.

NetBeacon [63] proposes a multi-phase sequential model architecture for dynamic analysis of packets/flows. By analyzing packets at multiple stages, the model adapts to changes in flow state, enhancing overall classification accuracy and reliability.

N3IC [49] proposes a system that uses binary neural networks for in-network traffic analysis. By deploying binary neural networks in the network, they reduce the effect of the computing and memory constraints, reduce the inference latency, and increase the throughput of the traffic analysis.

Despite these advancements, the deployment of DL models, characterized by hundreds of thousands of weights, across a network of programmable network devices for user applications remains an unaddressed challenge due to the main difficulties of complex neural computations, communication between layers, feature preparation, and the size of the DL model.

3 THESIS ORGANIZATION AND CONTRIBUTIONS

In the previous chapters, we provided an overview of the state-of-the-art and highlighted important research gaps in DL inference serving systems. This chapter summarizes the contributions of this thesis toward filling these research gaps.

3.1 RESOURCE EFFICIENCY

The first contribution of this thesis uses horizontal scaling to address Q_1: *How can computing resources be optimally allocated to DL models in a multi-model serving system to guarantee the SLO?*

In our paper, *FA2* [41], we analyze the relationship between processing latency and allocated resources. Then, we propose a mathematical approach to capture the joint batch size and resource allocation problem in inference serving systems and leverage graph transformation and dynamic programming to accurately calculate each DL model instance number and batch size in a single shot. The remainder of this section consists of the contribution statement, the applied research approach, and our main findings.

This contribution is based on the following publication:

Kamran Razavi, Manisha Luthra, Boris Koldehofe, Max Mühlhäuser, and Lin Wang. "FA2: Fast, accurate autoscaling for serving deep learning inference with SLA guarantees." In: *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2022, pp. 146–159

Contribution Statement: I led the idea generation, established and coordinated the interdisciplinary and international group of experts, and wrote the majority of the paper. All co-authors

helped with critiques and comments on the concept design and participated in creating the publication.

3.1.1 *Research Approach*

This contribution aims to enhance the horizontal autoscaling mechanisms for DL inference serving systems. Resource autoscaling in inference serving systems concerns dynamically adjusting the resources of each DL model in the system based on the changes in the workload to improve resource efficiency while guaranteeing service-level agreements (SLAs). Resource autoscaling, in general, is a non-trivial problem, and the unique properties of DL inference systems (data dependency, batch size, and execution path uncertainty), while guaranteeing SLAs, make the resource autoscaling problem even more challenging.

We address the resource autoscaling problem in inference serving systems by first comprehensively analyzing autoscalers in different domains, such as microservices and stream processing, and then introducing a new horizontal autoscaler that decides the batch size and the number of instances of each DL model in the system. Furthermore, we provide an autoscaling problem formulation with the inference serving properties encapsulated in an integer program (IP). We then provide a close-to-optimal solution using a two-step approach to solve the IP: graph transformation and dynamic programming.

There can be multiple applications with different SLAs in the system, for which DL model sharing can be beneficial for resource efficiency. However, prioritizing one application SLA over the other may violate the latter SLA. To avoid such conflicts, we propose the following solution: There should be at most one either egress aggregator or ingress aggregator per DL model. Either of these options removes the conflicts. In the case of having multiple egress/ingress aggregators per DL model, the question of which egress/ingress to keep arises. To address this issue, we define a new metric, called SHARING DEGREE, that captures the number of execution paths passing through the current DL models. We then remove the egress/ingress with the lowest SHARING DEGREE and continue the same procedure until there is no more conflict in the system. We choose to remove the egress/ingress

with the lowest SHARING DEGREE since removing an egress/ingress breaks each associated application into two independent applications. However, the original application SLA should also be divided into two to make both new applications fully independent. For this, we define a new metric called INT, which denotes the intensity of the processing latency of each application. The INT metric is calculated for each application segment by calculating the average processing latency of that segment with different batch sizes over the whole application. Graph transformation gets the system's DAG and transforms it to have exactly one ingress/egress per DL model using the SHARING DEGREE and INT metrics before sending it to the dynamic programming part. The last step of graph transformation is to sort the DL models topologically (topological sort) to indicate which DL models have successors.

We use dynamic programming on the topologically sorted transformed DAG to solve the autoscaling problem. We consider two cases: First, we consider the DL models without successors, meaning the ones that are the last DL models for processing the inference request. For this, we calculate all possible configurations by iterating over the time, t, from one to the SLA incrementally and calculate the optimal configuration (batch size and number of instances) for the given time. Second, we consider the DL models with successors. Using the topologically sorted DAG, we get the immediate DL models with successors with optimal configurations. Similarly, we iterate over t, from one to the application SLA, one by one, and allocate t to the DL model with successors and $SLA - t$ to the DL model without successors. We then calculate the optimal configuration for the current DL model with the given time t and use the calculated optimal configuration with $SLA - t$ in case one. We follow the same procedure until we find the optimal configurations for all the DL models in the system.

The above calculations require estimating each DL model's queuing and processing latency. For estimating the queuing latency, we consider the worst-case queuing scenario, where the arriving request must wait for the arrival of the last request to form the predefined batch size (b), meaning that it needs to wait $\frac{b-1}{\lambda}$, where λ is the arrival rate.

Unlike state-of-the-art approaches which use a linear relationship between batch size and processing latency for inference processing latency estimation, we use a second-order quadratic polynomial, $d(b) = \alpha b^2 + \beta b + \gamma$ with b being the batch size and α, β, and γ being

the fitted parameters, since it gives a lower mean squared error, resulting in higher estimation accuracy. In the next section, we discuss the impact of *FA2* on resource optimization by first comparing it to other state-of-the-art approaches and, second, evaluating its components.

3.1.2 *Main Findings*

To show the capabilities of *FA2* in handling different workload scenarios and hardware types, we evaluate FA2 on six workloads: Four synthetics and two real-world workloads on CPUs and GPUs. We use four servers, each equipped with Core i9-9980x or 10940x CPU and an NVIDIA RTX 2080 GPU, supporting up to 60 one-core CPU instances and 40 10%-GPU instances. We consider ten DL models in three domains: computer vision, natural language processing, and audio recognition. These DL models form six applications with varying SLAs ranging from 3020ms to 25710ms. To compare *FA2* with the other state-of-the-art autoscaler in inference serving systems [10] and stream processing [27] and microservices [28] areas, we use four metrics: Processing and queuing delay, SLA violation rate, resource consumption, and instance utilization.

We execute each DL model 25000 times using five different batch sizes (1, 2, 4, 8, 16). We then use the 99th percentile as the reference point and feed the result to the quadratic formulation to find the fitted parameters. We use the fitted parameters in the optimization to estimate and analyze the processing delay.

Under steady workloads, the state-of-the-art approaches demonstrate SLA violation of less than 1%, while *FA2* needs much less resources due to the optimized batch sizes for all the DL models in the system. The optimized batch size can be confirmed by the resource utilization of the *FA2*, where the resources are not underutilized for small batches and are not stalled by waiting for unnecessarily large batch sizes.

When there is a change in the workload due to fluctuating, random, or real-world workloads, all approaches detect and provide new configurations using the same time interval to have a fair comparison. When scaling up, new instances require a few seconds to initialize before they can begin serving inference requests. This initialization period is shorter with *FA2* compared to other approaches [10, 27, 28], which

need more instances and experience higher management overheads. This overhead slows down the scaling process due to performance interference caused by co-locating multiple instances on the same physical machine.

Finally, we assess the resource efficiency of *FA2* and the number of instances needed to serve requests under static workloads varying from 6 (one request per application) to 60 (the maximum capacity supported by the hardware infrastructure). *FA2*, on average, saves 19% when using CPUs (up to 62%) and saves 25% resources (up to 54%) when serving requests on GPUs while guaranteeing 99th SLAs of all applications. Moreover, the algorithms in *FA2* perform close to optimal in terms of finding the optimal resource allocation, matching over 96.8% of the results the optimizer provides. Further details are discussed in detail in Chapter P1.

3.2 RESPONSIVENESS AND ACCURACY

The second contribution of this thesis uses vertical scaling to investigate Q2: *How can DL models be effectively deployed to provide fast response to system dynamics and improve overall inference accuracy?*.

The first paper, *Sponge* [39], considers the dynamism in the network bandwidth, e.g., 4G/5G or WiFi, and adjusts the computing resources of a DL model using in-place vertical scaling in response to the sudden changes in the network bandwidth. *Sponge* uses an integer program to incorporate the communication latency caused by a non-stable network and provides a greedy approach to find an optimal CPU resource allocation for the DL model. The second paper, *IPA* [14], studies the challenges of model variants in terms of reconciling accuracy, latency, and resource cost in inference serving systems and proposes a multi-objective adaptation system to reconcile these three main conflicting properties. After the contribution statements, this section introduces the research approach and our main results.

This contribution is based on the following publications:

Kamran Razavi, Saeid Ghafouri, Max Mühlhäuser, Pooyan Jamshidi, and Lin Wang. *Sponge: Inference Serving with Dynamic SLOs Using In-Place Vertical Scaling*. 2024

Contribution Statement: I led the idea generation, established and coordinated the interdisciplinary and international group of experts, and wrote the majority of the paper. *Saeid Ghafouri* contributed to the implementation and evaluation. All co-authors helped with critiques and comments on the concept design and participated in creating the publication.

Saeid Ghafouri, Kamran Razavi, Mehran Salmani, Alireza Sanaee, Tania Lorido Botran, Lin Wang, Joseph Doyle, and Pooyan Jamshidi. *[Solution] IPA: Inference Pipeline Adaptation to achieve high accuracy and cost-efficiency*. 2024

Contribution Statement: As the second author, I helped to shape the initial idea of multivariate inference serving and design the problem formulation. I also contributed to the implementation by adapting the *FA2* components (queuing design, inter-intra parallelisms, and the Gurobi optimizer solver) to fit the needs of *Sponge*. I helped with restructuring the text, designing the evaluation and evaluation figures, and editing all of the papers (scientific and writing proofreading). Furthermore, I helped the first author prepare the response to the reviewer letter JSys October submission by helping with ideas and the scientific and writing proofreading of the letter. The first author, *Saeid Ghafouri*, led the idea generation, did all the implementation, conducted all the evaluations, formulated the problem, and was the main contributor to all of the sections of the paper. His contributions also include writing the paper and preparing the revisions. He also coordinated the interdisciplinary and international group of experts. This paper has previously appeared as the primary chapter of the first author's PhD thesis entitled "Machine Learning in Container Orchestration Systems: Applications and Deployment" defended at the Queen Mary University of London on 15/3/2024 under the supervision of Dr. Joseph Doyle and the main credit of *Sponge* remains with the first author. All co-authors helped with critiques and comments on the concept design and participated in creating the publication.

3.2.1 *Research Approach*

This contribution aims to investigate when to change resources in place and when to use model variants to increase inference accuracy while guaranteeing the SLO. In the first part of this contribution, we consider changes in the network bandwidth, e.g., 4G/5G or WiFi, caused by the user's movement. When the network bandwidth changes, the time that the request takes to reach a remote inference server varies (known as the communication cost), resulting in varying remaining latency budgets on the server side. To capture the dynamism in the requirement latency, we propose a new autoscaling mechanism, *Sponge*, using in-place vertical scaling, request reordering, and dynamic batching. In-place vertical scaling changes the computing resources of the current running DL model without bringing them down and respawning them (no cold start). Request reordering prioritizes requests with lower remaining latency budgets, and dynamic batching increases the system performance by utilizing the resources more. Nevertheless, the literature lacks performance modeling of inference execution latency under varying CPU core allocations and batch sizes. To fill the gap, we provide a mathematical performance modeling by incorporating the linear relationship of the processing latency with the batch size into the inverse relationship of the processing latency with CPU core allocations. Next, leveraging the performance modeling and a greedy algorithm, we explore all possible CPU core and batch size allocations to identify the optimal configuration for serving requests with varying remaining latency requirements. In the next chapter, we discuss the effectiveness of *Sponge*, the first inference serving system using in-place vertical scaling. We focus on its ability to guarantee the SLO by integrating communication latency to absorb changes caused by fluctuations in the network bandwidth.

In the second part of this contribution, we investigate when and how to switch to a different model variant for serving inference requests. Model variants can perform the same inference task while offering varying levels of accuracy, latency, and computational cost. The key challenge lies in balancing these properties, especially when the SLO must be guaranteed. A more accurate DL model increases the processing latency, which in turn reduces the number of requests it can handle per second. Consequently, more instances are needed to support the same number of requests. Therefore, inference serving systems must

consider all three pillars at the same time to optimize the resource allocation cost or accuracy while guaranteeing the desired latency. To achieve this, we first introduce a new metric to calculate the overall multi-model application accuracy. We then integrate this metric into a mathematical model to optimize costs while adhering to the predefined latency budget. We consider two approaches to define a new accuracy metric: First, we sort the model variants based on the accuracy from one to the number of model variants and then aggregate the chosen model variants' number. Second, we use the accuracy metric defined by the model variant provider and multiply the chosen model variants' accuracy to derive the overall application accuracy. In the first approach, we allocate zero to the lowest accurate model variant and one to the highest accurate model variant. The remaining model variants get a number between zero and one proportionally aligned with their ordering. Then, the application accuracy will be the summation of the chosen model variants. However, this approach disproportionately favors more accurate model variants. For instance, if one model variant has 78% accuracy and another variant has 69%, the first one gets scaling of one and the second one gets zero while their difference is just 9% in accuracy. The second approach, in terms of the definition of application accuracy, suffers from diverse metrics from different domains. For example, the accuracy metric in computer vision DL models is usually in percentage, while in natural language processing DL models, is the Word Error Rate. To unify the accuracy metrics, we must delve into each domain and map each domain's metric into a unified metric. The second approach is preferred because it maintains the integrity of the individual model accuracies, avoiding the disproportionate weighting issue that can arise in the first approach. For ease of interpretation, we use the percentile metric as our application accuracy metric and create a multi-objective optimization problem that optimizes the application accuracy or computing cost based on the user's preference. Finally, we use an optimizer to solve the multi-objective optimization and apply the results directly to the system.

3.2.2 *Main Findings*

We use one physical machine with the Kubernetes in-place vertical scaling feature [24] to evaluate *Sponge*. We design a workload generator

that generates 20 requests per second asynchronously with predefined latency requirements, similar to a real-world network bandwidth dataset [20]. Our evaluation shows that the performance model provides a realistic estimation of processing latency given CPU core and batch size allocations using two different DL models. Furthermore, *Sponge* reduces resource consumption by over 20% while guaranteeing over 99.6% of the requests when compared to statically assigned CPU cores (over-provisioning). Compared to another autoscaler [5], *Sponge* reduces latency violation by over 50% when the network bandwidth becomes limited. Further details are discussed in detail in Chapter P2.

We use six physical machines to evaluate *IPA*, each equipped with Intel(R) Xeon(R) Gold 6240R. We use eight different DL models, with twenty-nine model variants, forming five applications in three domains, and we analyze the applications using four real-world workloads. Our end-to-end performance evaluation demonstrates that *IPA* enhances the end-to-end application accuracy by up to 21% while having a comparable computing cost with the other state-of-the-art approach [22]. Moreover, to demonstrate the scalability of *IPA*, we progressively increase the workload, prompting the system to scale up to over 500 CPU cores. This enables us to conduct a production-level experiment that closely mirrors real-world demands, all while maintaining critical performance requirements such as latency and accuracy. Further details are discussed in detail in Chapter P3.

3.3 UNPREDICTABLE WORKLOADS

The third contribution tackles the challenges of using horizontal and vertical scaling mechanisms jointly in inference serving systems and addresses Q3: *How must multi-model serving systems respond to unpredictable workload changes to achieve resource efficiency while minimizing the SLO violation rate?*.

Horizontal scaling offers a superior cost-efficiency, while vertical scaling provides responsiveness. Consequently, a key consideration is determining the conditions for using each scaling mechanism. We propose *Biscale* [42] that leverages the cost-efficiency of horizontal scaling when the workload is stable and the responsiveness of in-place vertical scaling to absorb sudden changes in the workload. *Biscale* achieves cost-efficiency and responsiveness by proposing an integer program

formulation to capture the resource allocation problem and solve it according to the workload situation. After the contribution statements, this section introduces the research approach and our main results.

This contribution is based on the following publication:

Kamran Razavi, Mehran Salmani, Max Mühlhäuser, Boris Koldehofe, and Lin Wang. *A Tale of Two Scales: Reconciling Horizontal and Vertical Scaling for Inference Serving Systems*. 2024. arXiv: 2407.14843 [cs.DC]

Contribution Statement: I led the idea generation, established and coordinated the interdisciplinary and international group of experts, and wrote the majority of the paper. *Mehran Salmani* contributed to the idea generation, system design, implementation, and evaluation. All co-authors helped with critiques and comments on the concept design and participated in creating the publication.

3.3.1 *Research Approach*

In this contribution, we investigate the challenges regarding DL inference serving systems that use both vertical scaling and horizontal scaling mechanisms jointly. Horizontal scaling enables workload distribution by changing the number of instances of DL models, hence processing more requests per second. On the other hand, vertical scaling changes the computing resources of a DL model's instance, resulting in changes in both queuing and processing latencies of requests. Therefore, the question of which one to use arises in the presence of both scaling mechanisms.

In the previous work, *FA2*, we extensively studied the effect of horizontal scaling on applications. In *Biscale*, we analyze vertical scaling effects on applications with multiple DL models under different workloads and configurations. Similar to horizontal scaling, data dependency and batch size factors play important roles in responsiveness and cost of serving inference requests in vertical scaling. Changing these factors affects the downstream DL models' processing, queuing latencies, and workload. Unlike horizontal scaling, vertical scaling is constrained by

hardware limitations. It is not possible to allocate more computing resources to an instance than what is available on the physical hardware where that instance resides.

To effectively address the challenges of joint scaling, it is crucial to first understand WHY we should transition between vertical scaling and horizontal scaling. Second, we must examine HOW to transition between these two scaling mechanisms. Lastly, we must identify WHEN to transition between horizontal scaling and vertical scaling.

To answer the WHY challenge, we use Amdahl's performance law and prove that horizontal scaling provides better resource efficiency, leading to lower costs. Therefore, we need to switch to horizontal scaling whenever there is no need for responsiveness. To answer the HOW question, we again use Amdahl's performance law to show that it is more resource-efficient to scale all the DL model's instances to the same computing resources. Finally, to answer the WHEN question, we design a long short-term memory (LSTM) model to predict the next 10-second maximum workload and use it to identify whether the system is stable or not. We feed the predicted and current workloads to the horizontal scaling algorithm to check whether the current and future configurations are the same. If the configurations are the same, we switch to horizontal scaling to save resources.

We design a mathematical formulation for the joint resource scaling mechanisms and solve it based on whether the system requires responsiveness or cost-efficiency. We use dynamic programming to solve the mathematical formulation by iterating over the DL models in the application with different batch sizes and CPU core allocations and applying the new configurations to the system.

3.3.2 *Main Findings*

We evaluate *Biscale* using two physical machines with seven different DL models in computer vision, audio recognition, and natural language processing domains. With the DL models, we form three pipelines with different tasks and different SLOs and evaluate the created pipelines under a real-world workload. Our evaluations demonstrate that *Biscale* reduces the SLO violation by over $10\times$ on all pipelines compared to just using one of the scaling mechanisms

for resource allocation. Furthermore, *Biscale* reduces costs when the workload becomes stable, showing *Biscale*'s potential to provide responsive and cost-effective inference serving solutions. Further details are discussed in detail in Chapter P4.

3.4 IN-NETWORK DL INTRUSION DETECTION

The fourth contribution of this thesis addresses the in-network DL inference challenges of Q_4: *How can non-conventional computing resources be leveraged to improve the serving system efficiency?*.

In our papers, *Distributed Serving* [40] and *NetNN* [38], after understanding the challenges in regards to the execution of DL inference serving in programmable network devices, we propose an in-network DL intrusion detection system for fast, end-to-end inference serving by distributing a DL model across a network of programmable network devices. For this purpose, we follow a three-step approach: DL model mapping to a set of programmable network devices, generating packets mimicking the DL model execution, and instructing programmable network devices to perform the DL model tasks. The remainder of this section consists of the contribution statement, the applied research approach, and our main findings.

This contribution is based on the following publications:

Kamran Razavi, George Karlos, Vinod Nigade, Max Mühlhäuser, and Lin Wang. "Distributed DNN serving in the network data plane." In: *EuroP4*. 2022

Contribution Statement: I led the idea generation, established and coordinated the interdisciplinary and international group of experts, and wrote the majority of the paper. All co-authors helped with critiques and comments on the concept design and participated in creating the publication.

Kamran Razavi, Shayan Davari Fard, George Karlos, Vinod Nigade, Max Mühlhäuser, and Lin Wang. *NetNN: Neural Intrusion Detection System in Programmable Networks*. 2024. arXiv: 2406.19990 [cs.CR]

Contribution Statement: I led the idea generation, established and coordinated the interdisciplinary and international group of experts, and wrote the majority of the paper. All co-authors helped with critiques and comments on the concept design and participated in creating the publication.

Note: *NetNN* has been awarded the Second Best Paper at the 29th IEEE Symposium on Computers and Communications (ISCC).

3.4.1 *Research Approach*

This contribution aims to enable DL inference serving on programmable network devices residing inside the networks. To achieve this, we first need to understand the limitations of the programmable network devices for DL inference serving and then propose approaches to resolve these limitations to leverage the high-performance capabilities of such devices. Unlike advanced machine learning model execution, such as serving DL models that rely on matrix multiplication with a tremendous number of weights requiring floating point operations, programmable network devices are designed primarily for packet processing and forwarding. Consequently, they are equipped with limited memory, primarily for storing routing tables, and they support only basic arithmetic operations, such as bit shifting and integer aggregation. This fundamental difference highlights the challenge of adapting such devices for serving complex DL models.

To tackle the memory and processing limitations, we propose using a two-tier or three-tier Clos topology combined with the three-step approach. First, we outline a method for mapping a DL model into a set of programmable network devices. Second, we detail the process for generating and routing packets to emulate the DL model's execution flow. Finally, we describe how mathematical computations must be executed on these devices to effectively perform inference, ensuring efficient and scalable operation despite the inherent constraints of the hardware.

For the mapping part, we propose to divide the DL model based on the number of weighted layers, assigning each layer's neurons and associated weights to a single programmable network device. We also

map the layers to the lowest-tier programmable network devices in the multi-tier Clos topology to avoid stragglers in the synchronization. We distribute the neurons within the same layer across multiple available programmable network device pipelines, as they do not require intercommunication within the layer. This approach enables parallelism by allowing neurons in the same layer to operate concurrently across multiple pipelines. For the packet generation and routing, we must create network packets similar to the DL model's execution dataflow. Once the processing on the current layer is finished, we generate as many network packets as necessary to encapsulate the input or intermediate data into the packet headers and emit them to the next programmable network device with the next layer. Lastly, we need to address the computing challenges regarding the on-device execution. As we discussed above, programmable network devices may not be able to perform multiplication or floating point operations. Therefore, we use DL models with integer weights (quantized DL models) and decompose the multiplication operation into bit shifting and aggregation. With these techniques, we can perform the necessary mathematical executions to get an inference.

Finally, to demonstrate the feasibility of getting inference in the network, we design a new quantized DL model with an accuracy of 97% for intrusion detection. Accurate intrusion detection requires statistics, such as the standard deviation of the inter-arrival time and the length of the packets. However, calculating standard deviation requires multiplication and division, which may not be available in programmable network devices. Therefore, to avoid the feature calculation needed for getting accurate inference on the programmable network devices, we use the first 68 bytes of packets from the same flow (covering the maximum UDP and IP header size) and use the hardware-provided minimum and maximum timestamps of these packets to train the DL model. We use the trained DL model and the three-step approach to enable in-network DL intrusion detection.

3.4.2 *Main Findings*

In the first paper of this contribution, *Distributed Serving*, we demonstrate that, in theory, in-network DL inference serving can reduce inference execution latency of a real-world object detection DL model

by over 50% compared to traditional CPU-based inference. Moreover, this approach eliminates the need for dedicated inference servers by leveraging programmable switches capable of delivering a throughput of 12.8 Tbps. Further details are discussed in detail in Chapter P5.

In the second paper of this contribution, *NetNN*, we implement a preliminary version of the quantized intrusion detection DL model using the P4 language [7] with the behavioral model, bmv2 [36], which serves as a P4 target. We create a network consisting of eighteen switches. The first switch is responsible for generating the features, and the last one finalizes the inference result. The other sixteen switches serve as four layers with four pipelines, denoted as switches in the emulation. Since we execute the quantized DL model on bmv2, and the software emulation on bmv2 lacks the time precision needed for the end-to-end performance measurements, we instead focus our analysis on the packet generation process, the computational operations required per packet and the memory consumption of each layer, which is distributed across four switches. Our evaluation shows that convolutional layers require significantly more operations per packet than dense layers. As a result, having additional switches is advantageous for distributing these computations. While dense layers require more memory than convolutional layers to store their weights, this demand remains within the memory capacity of a programmable switch. Further details are discussed in detail in Chapter P6.

3.5 SUMMARY

In summary, this thesis made four contributions spread across six publications One of the publications (*NetNN*) was awarded the BEST PAPER AWARD at the conference (second place).

We improved DL inference serving resource efficiency in the following areas:

i. We proposed a new horizontal resource scaler to increase system utilization and reduce operational costs.

ii. We proposed a new in-place vertical resource scaler to absorb the network dynamism while guaranteeing the SLO. Furthermore,

we designed a new resource scaler to use model variants to reduce further the operational cost of serving requests.

iii. We analyzed the challenges of using vertical scaling and horizontal scaling jointly. We then proposed a new two-step autoscaler that uses either scaling mechanism based on the workload.

iv. We investigated the challenges regarding in-network DL intrusion detection. We proposed a three-step mapping approach to enable complex DL models serving inside a set of programmable network devices

4 CONCLUSION AND OUTLOOK

Deep learning serving has become an essential component in various modern applications. Numerous factors, often conflicting, must be considered to efficiently allocate resources to deep learning models while ensuring that the SLO is met. These conflicting factors include the selection of resource allocation strategies, the trade-off between accuracy and latency, and the decision of where to deploy the deep learning models.

To achieve our goal of *Resource Efficiency of Inference Serving with SLO Guarantee*, we divided the main research question, *How to design, implement, and deploy resource-efficient DL inference serving systems with SLO guarantee?*, into four sub-questions, each addressed in a separate chapter of this dissertation.

4.1 SUMMARY OF ACHIEVEMENTS

4.1.1 *Resource Efficiency*

FA2 provides a significant resource consumption improvement over existing horizontal-based resource scalers in DL inference serving that require SLO guarantees. *FA2* encapsulates the resource scaling problem into an integer program and adjusts the number of instances and batch sizes of all the DL models in the system in a single shot. By solving the resource scaling problem using graph transformation and dynamic programming, *FA2* reduces the resource consumption by over half in some scenarios on both CPUs and GPUs under real-world workloads. Moreover, approaches in *FA2* can be applied to other domains with predictable performance models in their cores.

4.1.2 *Responsiveness and Accuracy*

Sponge enables guaranteeing SLO by considering the dynamic network bandwidth caused by non-stable networks, e.g., 4G/5G or WiFi, using in-place vertical scaling, dynamic batching, and request reordering. To enable this, *Sponge* provides a performance modeling concerning the inverse relationship of inference latency to the number of allocated CPU cores and linear relationship of inference latency to the batch sizes and designs a new in-place vertical resource scaler based on the performance model. A prototype of *Sponge* demonstrates that it can reduce the SLO violation by over 50% compared to a state-of-the-art horizontal resource scaler.

IPA increases inference accuracy by designing a new resource scaler that uses model variants and considers the trade-offs between the execution latency, accuracy, and computational cost of such models. *IPA* then proposes a new pipeline accuracy metric to calculate the overall accuracy of a pipeline composed of multiple models. Extensive evaluations show that *IPA* increases the pipeline accuracy by switching between different model variants while guaranteeing the SLO.

4.1.3 *Unpredictable Workloads*

Biscale proposes a new resource scaler design that uses vertical scaling to absorb sudden changes in the workload and transitions to horizontal scaling to save resources when the workload becomes stable. *Biscale* uses a two-stage resource scaling strategy: it starts with in-place vertical scaling to manage workload surges and then switches to horizontal scaling for resource efficiency once the workload stabilizes. The system then profiles DL model latency, calculates queuing delays, and uses dynamic programming algorithms to optimally allocate resources and balance horizontal and vertical scaling based on the workload. Evaluations with real-world workload traces show over $10\times$ reduction in the SLO violation compared to state-of-the-art horizontal or vertical resource scaling methods while maintaining resource efficiency when the workload is stable.

4.1.4 *In-Network DL Intrusion Detection*

Distribute Serving and *NetNN* enable in-network DL inference by using the multi-tier Clos network architecture and a three-step design. They propose mapping a DL model to a set of programmable network devices, mimicking the DL inference execution flow similar to how packets flow in the network, and directing the programmable network devices to carry out the necessary computations for getting an inference. Furthermore, *NetNN* proposes a new DL model that advances in-network intrusion detection and implements it on a network emulator.

4.1.5 *Open Source Inference Serving*

In addition to our analytical achievements, we have developed and presented four frameworks to aid others in research on inference serving systems. All four frameworks are shared with the community and can be accessed upon request.

IPA adaptable framework leverages state-of-the-art technologies such as Kubernetes [13] and Grafana [29] and runs on a public cloud provider (Chameleon Cloud [1]) to provide a realistic production environment.

Sponge framework uses the experimental branch of Kubernetes [24] on a single machine since the in-place vertical scaling feature is not yet in the official releases, allowing researchers to experiment with inference serving on the soon-to-be-available feature of a dominant container orchestrator.

Biscale creates a multi-machines Kubernetes cluster and reconciles a system's in-place vertical scaling and horizontal scaling mechanisms.

NetNN P4 source code can be used to implement hardware-specific P4 code to allow DL inference execution on a set of network devices. Moreover, the network architecture proposed in *NetNN* is implemented on a network emulator that can be directly used or modified based on the DL model's architecture.

4.2 FUTURE WORK

Our research uncovered several new and remaining challenges within the domain.

4.2.1 *Heterogeneous Hardware Framework*

As discussed in *FA2*, different hardware components, e.g., CPUs and GPUs, have varying capabilities and performance characteristics, making it challenging to allocate resources optimally. Moreover, there is no unified library to serve and monitor inference requests on all types of hardware, making it challenging to develop and maintain inference serving frameworks. Furthermore, a heterogeneous hardware environment brings new challenges in data security, such as how to ensure data and execution isolation and how to secure data transmission across multiple hardware components. A generic inference serving framework will be valuable for using each hardware characteristic and serve inference based on the user and the system provider requirements.

4.2.2 *Inference Pipeline Accuracy*

As we discussed in *IPA*, multiplying the accuracy of individual models reflects the combined probability of each model correctly processing its input and passing correct results to the next model. It captures the cumulative effect of each model's performance on the overall system accuracy. However, we are assuming that each DL model's error is independent of others, which, in reality, errors might propagate through the application, and the performance of one DL model could affect the performance of subsequent models, resulting in a wrong accuracy estimation. On top of that, model variants can be used for the same inference request, adding an extra level of complexity to calculating the multi-model applications' accuracy. We must further investigate how to calculate multi-model accuracy accurately to reduce inference errors and increase user satisfaction.

4.2.3 *In-Network Mapper*

As we demonstrated in *NetNN*, in-network DL inference has the potential to enable real-time low-latency DL inference within the network. Nevertheless, achieving hardware-specific DL inference serving requires addressing challenges regarding not only the resource and processing limitations of the network devices but also the internal architecture of DL models that we want to serve in the network. Some DL models have complex relationships between their internal calculations, making it challenging to implement on network devices with fast-forwarding packet processing. Therefore, having a model-to-hardware mapper that maps any DL model into any network hardware will be beneficial.

REFERENCES

[1] *A configurable experimental environment for large-scale edge to cloud research*. https://www.chameleoncloud.org/.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. "Tensorflow: A system for large-scale machine learning." In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2016, pp. 265–283.

[3] Fawad Ahmad, Hang Qiu, Ray Eells, Fan Bai, and Ramesh Govindan. "CarMap: Fast 3D Feature Map Updates for Automobiles." In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2020, pp. 1063–1081.

[4] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. "Batch: Machine learning inference serving on serverless platforms with adaptive batching." In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2020, pp. 1–15.

[5] The Kubernetes Authors. *Kubernetes Vertical Pod Autoscaling*. https://cloud.google.com/kubernetes-engine/docs/concepts/verticalpodautoscaler/. Accessed on 30.01.2024. 2024.

[6] Diogo Barradas, Nuno Santos, Luís Rodrigues, Salvatore Signorello, Fernando MV Ramos, and André Madeira. "FlowLens: Enabling Efficient Flow Classification for ML-based Network Security Applications." In: *NDSS*. 2021.

[7] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. "P4: programming protocol-independent packet processors." In: *SIGCOMM Comput. Commun. Rev.* 44.3 (July 2014), pp. 87–95. ISSN: 0146-4833. DOI: 10.1145/2656877.2656890.

[8] Jake Bouvrie. *Notes on convolutional neural networks*. 2006.

[9] Yujeong Choi, Yunseong Kim, and Minsoo Rhu. "Lazy Batching: An SLA-aware batching system for cloud machine learning inference." In: *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2021, pp. 493–506.

[10] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. “InferLine: Latency-Aware Provisioning and Scaling for Prediction Serving Pipelines.” In: *ACM Symposium on Cloud Computing (SoCC)*. 2020, pp. 477–491. DOI: 10.1145/3419111.3421285.

[11] Shi Dong, Ping Wang, and Khushnood Abbas. “A survey on deep learning and its applications.” In: *Computer Science Review* 40 (2021), p. 100379.

[12] Mohamed Elhoseny. “Multi-object detection and tracking (MODT) machine learning model for real-time video surveillance systems.” In: *Circuits, Systems, and Signal Processing* 39.2 (2020), pp. 611–630.

[13] The Linux Foundation. *Kubernetes*. https://kubernetes.io. Accessed on 29.10.2021. 2019.

[14] Saeid Ghafouri, Kamran Razavi, Mehran Salmani, Alireza Sanaee, Tania Lorido Botran, Lin Wang, Joseph Doyle, and Pooyan Jamshidi. *[Solution] IPA: Inference Pipeline Adaptation to achieve high accuracy and cost-efficiency*. 2024.

[15] Google. *CloudRun*. https://cloud.google.com/run. Accessed on 29.10.2021. 2021.

[16] Arpan Gujarati, Sameh Elnikety, Yuxiong He, Kathryn S McKinley, and Björn B Brandenburg. “Swayam: distributed autoscaling to meet slas of machine learning inference services with resource efficiency.” In: *ACM/IFIP/USENIX Middleware Conference*. 2017, pp. 109–120.

[17] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. “Serving DNNs like Clockwork: Performance Predictability from the Bottom Up.” In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2020, pp. 443–462.

[18] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. “Serving DNNs like Clockwork: Performance Predictability from the Bottom Up.” In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 443–462. ISBN: 978-1-939133-19-9.

[19] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R Das. "Cocktail: A multidimensional optimization for model serving in cloud." In: *USENIX NSDI*. 2022, pp. 1041–1057.

[20] J. van der Hooft, S. Petrangeli, T. Wauters, R. Huysegems, P. R. Alface, T. Bostoen, and F. De Turck. "HTTP/2-Based Adaptive Streaming of HEVC Video Over 4G/LTE Networks." In: *IEEE Communications Letters* 20.11 (2016), pp. 2177–2180.

[21] Yitao Hu, Rajrup Ghosh, and Ramesh Govindan. "Scrooge: A cost-effective deep learning inference system." In: *Proceedings of the ACM Symposium on Cloud Computing*. 2021, pp. 624–638.

[22] Yitao Hu, Weiwu Pang, Xiaochen Liu, Rajrup Ghosh, Bongjun Ko, Wei-Han Lee, and Ramesh Govindan. "Rim: Offloading Inference to the Edge." In: *Proceedings of the International Conference on Internet-of-Things Design and Implementation*. 2021, pp. 80–92.

[23] Junxian Huang, Feng Qian, Alexandre Gerber, Z Morley Mao, Subhabrata Sen, and Oliver Spatscheck. "A close examination of performance and power characteristics of 4G LTE networks." In: *Proceedings of the 10th international conference on Mobile systems, applications, and services*. 2012, pp. 225–238.

[24] *In-place Update of Pod Resources*. https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/.

[25] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. "NetChain: Scale-Free Sub-RTT Coordination." In: *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*. Ed. by Sujata Banerjee and Srinivasan Seshan. USENIX Association, 2018, pp. 35–49.

[26] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. "NetCache: Balancing Key-Value Stores with Fast In-Network Caching." In: *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 2017, pp. 121–136. DOI: 10.1145/3132747.3132764.

[27] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. "Three steps is all you need: fast, accurate, automatic scaling decisions for

distributed streaming dataflows." In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2018, pp. 783–798.

[28] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. "Grandslam: Guaranteeing SLAs for jobs in microservices execution frameworks." In: *Proceedings of the Fourteenth EuroSys Conference 2019*. 2019, pp. 1–16.

[29] Grafana Labs. *Grafana*. https://grafana.com/. Accessed on 11.08.2024. 2024.

[30] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael M. Swift. "ATP: In-network Aggregation for Multi-tenant Learning." In: *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*. Ed. by James Mickens and Renata Teixeira. USENIX Association, 2021, pp. 741–761.

[31] Zhaoyi Li, Jiawei Huang, Yijun Li, Aikun Xu, Shengwen Zhou, Jingling Liu, and Jianxin Wang. "A2TP: Aggregator-aware In-network Aggregation for Multi-tenant Learning." In: *Proceedings of the Eighteenth European Conference on Computer Systems*. 2023, pp. 639–653.

[32] Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. "Jaqen: A High-Performance Switch-Native approach for detecting and mitigating volumetric DDoS attacks with programmable switches." In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 3829–3846.

[33] Zixuan Ma, Jiaao He, Jiezhong Qiu, Huanqi Cao, Yuanwei Wang, Zhenbo Sun, Liyan Zheng, Haojie Wang, Shizhi Tang, Tianyu Zheng, et al. "BaGuaLu: targeting brain scale pretrained models with over 37 million cores." In: *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2022, pp. 192–204.

[34] Vinod Nigade, Pablo Bauszat, Henri Bal, and Lin Wang. "Jellyfish: Timely Inference Serving for Dynamic Edge Networks." In: *2022 IEEE Real-Time Systems Symposium (RTSS)*. 2022, pp. 277–290. DOI: 10.1109/RTSS55097.2022.00032.

[35] Vinod Nigade, Lin Wang, and Henri Bal. "Clownfish: Edge and cloud symbiosis for video stream analytics." In: *IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE. 2020, pp. 55–69.

[36] *p4lang/behavioral-model*. original-date: 2015-01-26T21:43:23Z. Aug. 2024.

[37] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. "Pytorch: An imperative style, high-performance deep learning library." In: *Advances in neural information processing systems* 32 (2019).

[38] Kamran Razavi, Shayan Davari Fard, George Karlos, Vinod Nigade, Max Mühlhäuser, and Lin Wang. *NetNN: Neural Intrusion Detection System in Programmable Networks*. 2024. arXiv: 2406.19990 [cs.CR].

[39] Kamran Razavi, Saeid Ghafouri, Max Mühlhäuser, Pooyan Jamshidi, and Lin Wang. *Sponge: Inference Serving with Dynamic SLOs Using In-Place Vertical Scaling*. 2024.

[40] Kamran Razavi, George Karlos, Vinod Nigade, Max Mühlhäuser, and Lin Wang. "Distributed DNN serving in the network data plane." In: *EuroP4*. 2022.

[41] Kamran Razavi, Manisha Luthra, Boris Koldehofe, Max Mühlhäuser, and Lin Wang. "FA2: Fast, accurate autoscaling for serving deep learning inference with SLA guarantees." In: *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2022, pp. 146–159.

[42] Kamran Razavi, Mehran Salmani, Max Mühlhäuser, Boris Koldehofe, and Lin Wang. *A Tale of Two Scales: Reconciling Horizontal and Vertical Scaling for Inference Serving Systems*. 2024. arXiv: 2407.14843 [cs.DC].

[43] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. "INFaaS: Automated Model-less Inference Serving." In: *USENIX Annual Technical Conference (ATC)*. 2021, pp. 397–411.

[44] Francisco Romero, Mark Zhao, Neeraja J Yadwadkar, and Christos Kozyrakis. "Llama: A Heterogeneous & Serverless Framework for Auto-Tuning Video Analytics Pipelines." In: *ACM Symposium on Cloud Computing (SoCC)*. 2021, pp. 1–17.

[45] Mehran Salmani, Saeid Ghafouri, Alireza Sanaee, Kamran Razavi, Max Mühlhäuser, Joseph Doyle, Pooyan Jamshidi, and Mohsen Sharifi. "Reconciling High Accuracy, Cost-Efficiency, and Low Latency of Inference Serving Systems." In: *Proceedings of the 3rd Workshop on Machine Learning and Systems*. 2023, pp. 78–86.

[46] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter Richtárik. "Scaling Distributed Machine Learning with In-Network Aggregation." In: *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*. Ed. by James Mickens and Renata Teixeira. USENIX Association, 2021, pp. 785–808.

[47] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. "Nexus: A GPU cluster engine for accelerating DNN-based video analysis." In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 2019, pp. 322–337.

[48] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. "Nexus: a GPU cluster engine for accelerating DNN-based video analysis." In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*. Ed. by Tim Brecht and Carey Williamson. ACM, 2019, pp. 322–337. DOI: 10.1145/3341301.3359658.

[49] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Gianni Antichi, Paolo Costa, Hamed Haddadi, and Roberto Bifulco. "Re-architecting Traffic Analysis with Neural Network Interface Cards." In: *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, April 4-6, 2022*. USENIX Association, 2022, pp. 513–533.

[50] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, Ishan Gaur, and Kunle Olukotun. "Taurus: a data plane architecture for per-packet ML." In: *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*. Ed. by Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch. ACM, 2022, pp. 1099–1114. DOI: 10.1145/3503222.3507726.

[51] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. "Efficient processing of deep neural networks: A tutorial and survey." In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329.

[52] Marvin Teichmann, Michael Weber, Marius Zoellner, Roberto Cipolla, and Raquel Urtasun. "Multinet: Real-time joint semantic reasoning for autonomous driving." In: *2018 IEEE intelligent vehicles symposium (IV)*. IEEE. 2018, pp. 1013–1020.

[53] Muhammad Tirmazi, Ran Ben Basat, Jiaqi Gao, and Minlan Yu. "Cheetah: Accelerating Database Queries with Switch Pruning." In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. Ed. by David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo. ACM, 2020, pp. 2407–2422. DOI: 10.1145/3318464.3389698.

[54] Guorui Xie, Qing Li, Yutao Dong, Guanglin Duan, Yong Jiang, and Jingpu Duan. "Mousika: Enable general in-network intelligence in programmable switches by knowledge distillation." In: *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE. 2022, pp. 1938–1947.

[55] Zhaoqi Xiong and Noa Zilberman. "Do Switches Dream of Machine Learning?: Toward In-Network Classification." In: *Proceedings of the 18th ACM Workshop on Hot Topics in Networks, HotNets 2019, Princeton, NJ, USA, November 13-15, 2019*. ACM, 2019, pp. 25–33. DOI: 10.1145/3365609.3365864.

[56] Dongzhu Xu, Anfu Zhou, Xinyu Zhang, Guixian Wang, Xi Liu, Congkai An, Yiming Shi, Liang Liu, and Huadong Ma. "Understanding operational 5G: A first measurement study on its coverage, performance and energy consumption." In: *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*. 2020, pp. 479–494.

[57] Siyu Yan, Xiaoliang Wang, Xiaolong Zheng, Yinben Xia, Derui Liu, and Weishan Deng. "ACC: Automatic ECN tuning for high-speed datacenter networks." In: *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 2021, pp. 384–397.

[58] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. "INFless: a native serverless system for low-latency, high-throughput infer-

ence." In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2022, pp. 768–781.

[59] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. "Mark: Exploiting cloud services for cost-effective, SLO-aware machine learning inference serving." In: *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 2019, pp. 1049–1062.

[60] Jeff Zhang, Sameh Elnikety, Shuayb Zarar, Atul Gupta, and Siddharth Garg. "Model-switching: Dealing with fluctuating workloads in machine-learning-as-a-service systems." In: *12th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. 2020.

[61] Changgang Zheng, Zhaoqi Xiong, Thanh T Bui, Siim Kaupmees, Riyad Bensoussane, Antoine Bernabeu, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. "IIsy: Practical in-network classification." In: *arXiv preprint arXiv:2205.08243* (2022).

[62] Changgang Zheng and Noa Zilberman. "Planter: seeding trees within switches." In: *Proceedings of the SIGCOMM'21 Poster and Demo Sessions*. 2021, pp. 12–14.

[63] Guangmeng Zhou, Zhuotao Liu, Chuanpu Fu, Qi Li, and Ke Xu. "An efficient design of intelligent network data plane." In: *32nd USENIX Security Symposium (USENIX Security 23)*. 2023, pp. 6203–6220.

Part II

PUBLICATIONS

P1 FA2 55

P1.1 Introduction 56

P1.2 Background and Motivation 58

P1.3 FA2 System Design 63

P1.4 Autoscaling Problem and Algorithm 65

P1.5 Implementation 72

P1.6 Evaluation 73

P1.7 Related Work 86

P1.8 Conclusion 88

P1.9 Acknowledgments 89

P2 Sponge 97

P2.1 Introduction 97

P2.2 Motivation 100

P2.3 System Design 104

P2.4 Preliminary Evaluation 109

P2.5 Related Work 110

P2.6 Conclusion & Future Work 112

P2.7 Acknowledgments 112

P3 IPA 115

P3.1 Introduction 116

P3.2 Background and Motivation 119

P3.3 System Design 123

P3.4 Problem Formulation 126

P3.5 Evaluation 135

P3.6 Related Works 146

P3.7 Conclusion and Future Works 147

P3.8 Acknowledgments 148

P3.9 Appendix 149

P3.10 Pipelines Stages Specifications 149

P3.11 Constant Multipliers Values 154

P3.12 Alternate Inference Pipeline Accuracy Definition 154

P4 Biscale **167**
- P4.1 Introduction . . . 168
- P4.2 Background and Motivation . . . 171
- P4.3 Biscale . . . 174
- P4.4 Autoscaling Problem . . . 178
- P4.5 Transition . . . 184
- P4.6 Experimental Evaluation . . . 189
- P4.7 Related Work . . . 200
- P4.8 Conclusion . . . 201
- P4.9 Acknowledgments . . . 202

P5 Distributed DNN Serving **209**
- P5.1 Introduction . . . 209
- P5.2 A Case Study with mini-AlexNet . . . 211
- P5.3 Challenges and Discussion . . . 216
- P5.4 Acknowledgments . . . 217

P6 NetNN **221**
- P6.1 Introduction . . . 221
- P6.2 Motivation . . . 224
- P6.3 System Design . . . 228
- P6.4 Evaluation . . . 236
- P6.5 Related Work . . . 238
- P6.6 Conclusion . . . 238
- P6.7 Acknowledgments . . . 239

P1

FA2: FAST, ACCURATE AUTOSCALING FOR SERVING DEEP LEARNING INFERENCE WITH SLA GUARANTEES

ABSTRACT

Deep learning (DL) inference has become an essential building block in modern intelligent applications. Due to the high computational intensity of DL, it is critical to scale DL inference serving systems in response to fluctuating workloads to achieve resource efficiency. Meanwhile, intelligent applications often require strict service level agreements (SLAs), which need to be guaranteed when the system is scaled. The problem is complex and has been tackled only in simple scenarios so far.

This paper describes FA2, a fast and accurate autoscaler concept for DL inference serving systems. In contrast to related works, FA2 adopts a general, contrived two-phase approach. Specifically, it starts by capturing the autoscaling challenges in a comprehensive graph-based model. Then, FA2 applies targeted graph transformation and makes autoscaling decisions with an efficient algorithm based on dynamic programming. We implemented FA2 and built and evaluated a prototype. Compared with state-of-the-art autoscaling solutions, our experiments showed FA2 to achieve significant resource reduction (19% under CPUs and 25% under GPUs, on average) in combination with low SLA violations (less than 1.5%). FA2 performed close to the theoretical optimum, matching exactly the optimal decisions (with the least required resources) in 96.8% of all the cases in our evaluation.

P1.1 INTRODUCTION

With the rapid advancements of deep learning (DL) techniques, DL inference has become a popular component in various modern intelligent applications and services [3, 12, 36, 47]. Some applications involve a single deep neural network (DNN) for inference tasks like object recognition or natural language understanding. Others, such as digital assistant services (e.g., Amazon Alexa), involve a more complex chain of DNNs for inference tasks, including speech recognition, question interpretation, and text-to-speech to serve user requests [12, 62]. Most of these applications are mission-critical or user-interactive, imposing strict service-level agreements (SLAs) on the inference latency, e.g., the end-to-end latency should be bounded by a deadline [8, 36, 47].

One critical concern in provisioning applications with DL inference is on *resource efficiency*. Resource efficiency is essential simply because DNNs typically require intensive computation, which imposes prohibitive costs and stringent deployment constraints when the resources are limited, e.g., in the edge environment [57]. Ideally, the amount of resources assigned to each of the DNNs in an application should be just right, matching the real-time workload (measured in requests per second, RPS) of the application while guaranteeing the SLA. We refer to this problem as "resource autoscaling," which has become a critical challenge in building efficient DL inference serving systems [12, 62].

Resource autoscaling is a non-trivial problem in general and has been heavily explored in various contexts, including stream processing [14, 21, 46], serverless computing [53, 66], and microservices [18, 30, 72, 73]. The unique properties of DL inference serving systems make the situation even worse. In particular, we identify the following factors in DL inference serving systems, which, when combined, add new challenges and significantly exacerbate existing ones: (a) The DNNs for an application are orchestrated with data **dependencies** specified by a dataflow graph. Thus, the resource scaling decision for one DNN may affect that for other (downstream) DNNs due to workload changes [12, 47]. (b) Inference requests may follow **uncertain** execution paths, where, depending on the output of a DNN, requests may be forwarded to different succeeding DNNs (thus different paths) for further processing [62]. (c) DL inference serving systems typically require strict **SLA guarantee** on the end-to-end latency over all possible execution paths specified in the dataflow graph [36, 47]. (d) Request

batching is widely used for DNNs to improve resource utilization by trading processing time for throughput [8, 47, 65]. Thus, the batch size for one DNN may affect the scaling decision of others and also the end-to-end latency. These factors, when combined, make existing autoscalers inapplicable or inefficient for DL inference serving systems, calling for new solutions.

In this paper, we present a comprehensive study of resource autoscaling for DL inference serving systems and present FA2—a fast, accurate resource autoscaler tailored for efficient provisioning of DL inference-based applications. Given an application with a set of DNNs orchestrated with a dataflow graph, FA2 makes collective resource scaling decisions, including both the number of instances and the corresponding batch size for each of the involved DNNs adaptively. Our goal is to minimize the total amount of resources occupied by all the DNNs of an application while ensuring the SLAs of all the execution paths in the application.

To this end, we first present a graph-based model to capture all the aforementioned factors holistically. In particular, we model the processing time and the worst-case queuing delay at all DNNs explicitly and take both into account when calculating the end-to-end delay for all the execution paths. This design choice is critical in guaranteeing SLAs on all the execution paths with dependency and uncertainty. Through targeted graph transformation, we relax the problem and present an efficient resource autoscaling algorithm based on dynamic programming. The proposed algorithm runs fast and generates accurate scaling decisions. In contrast to existing solutions where simple heuristics are used for scaling decision-making [12, 62], our design is principled and achieves almost optimal performance.

In short, this paper makes the following contributions. After presenting the background for DL inference serving systems and identifying the challenges (§P1.2), we

- present the design of FA2, including its overall architecture and components (§P1.3);
- introduce a comprehensive graph-based model to capture the resource scaling problem in DL inference serving systems and present our scaling algorithm based on graph transformation and dynamic programming (§P1.4);

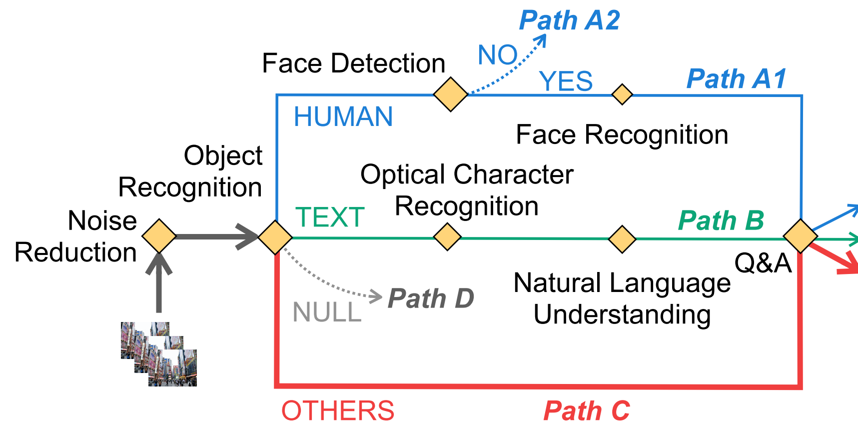


Figure P1.1: The execution graph of an example modern intelligent application taking images as input. Depending on the object recognition result, different execution paths will be taken for each inference request.

- build a system prototype for FA2 (§P1.5) and evaluate it with synthetic and real-world workload traces (§P1.6). Overall, FA2 outperforms the state-of-the-art autoscalers, achieves significant resource reductions (19% under CPUs and 25% under GPUs), while showing only slightly over 1% SLA-violation rates, meaning that the latency at the 99-th percentile can mostly be guaranteed. Our results also reveal that FA2 matches the theoretical optimum scaling decisions in around 96% of the cases.

§P1.7 summarizes related work. §P1.8 draws final conclusions.

P1.2 BACKGROUND AND MOTIVATION

This section presents the background on deep learning (DL) inference and discusses the resource autoscaling problem in DL inference serving systems. We then identify the challenges in efficient autoscaling and motivate a new autoscaler design.

P1.2.1 *Deep Learning Inference Serving*

With the fast advancement of DL techniques, a variety of modern applications such as intelligent personal assistants, augmented reality (AR), and autonomous driving adopt deep neural networks (DNNs)

as a fundamental building block [13, 45, 47]. Typically, DL is used for inference tasks such as object detection and recognition and voice recognition, where input data in the form of images or voice recordings is sent to DNNs, producing predictions for the input. For sophisticated applications, multiple DNNs may be involved in the process where all these DNNs are chained or orchestrated into a complex dataflow graph, represented by a direct acyclic graph (DAG), to process the input data step by step. Each input to the system spawns an *inference request* which needs to be handled by a subset of the DNNs specified in the dataflow graph sequentially [12]. Figure P1.1 depicts the dataflow graph of an example modern intelligent application.

One salient feature of these DL-based applications is being timed critical since they are mostly either user-interactive (e.g., personal assistant and AR) or mission-critical (e.g., autonomous driving) [3, 12, 45, 51]. It is often required that the end-to-end latency in serving each inference request by the system has to meet a certain threshold dictated by the application for the request to be useful. The DL inference serving system provisioning such applications thus needs to provide strict service-level agreements (SLAs) where the tail inference latency has to be bounded by the threshold to guarantee the overall usability of the application [36, 47].

P1.2.2 *Resource Autoscaling*

Resource scaling concerns dynamically adjusting the computing resources assigned to applications according to their changing workload. The goal is to improve resource efficiency while meeting the resource demands of the application. Resource scaling has been extensively explored in cloud-based systems, including stream processing [14, 21, 46], serverless computing [53, 66], and microservices [18, 30, 72, 73].

There are generally two types of resource scaling mechanisms: *horizontal* scaling and *vertical* scaling. Horizontal scaling constructs a base instance (with a fixed amount of resources) built into a virtual machine or container and decides the number of instances required to support the real-time workload (e.g., throughput) at runtime. Vertical scaling adjusts the performance of every single instance by changing the resource allocation for the instance. Most data-intensive computing systems adopt horizontal scaling considering its simplicity [46]. Hori-

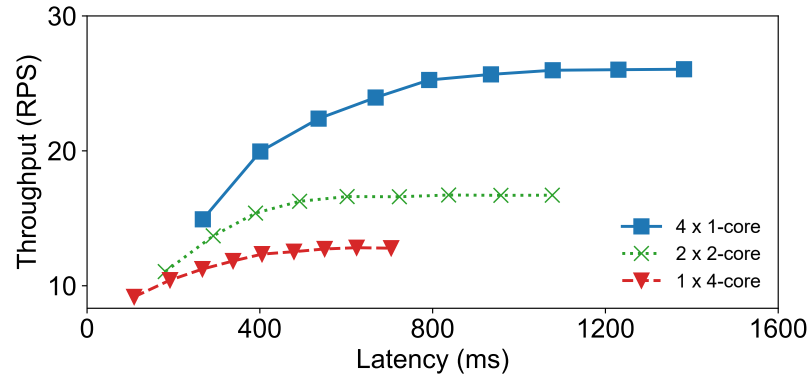


Figure P1.2: Performance comparison between horizontal and vertical scaling policies with respect to throughput and latency on the object detection (OBJD) model in Table P1.2. Each data point represents the result obtained with a batch size in the range of [1,9] respectively.

zontal scaling is also beneficial to DL inference serving, as confirmed in Figure P1.2, where we show the latency-throughput comparison for an object detection model (Inception V2) under three different resource configurations. We observe that given the same total amount of resources (four CPU cores here), the configuration with the smallest instance provides higher throughout under the same latency. This is mainly because, even with batching, DL inference cannot be fully parallelized to take full advantage of the big instances. Therefore, we focus on horizontal scaling in this paper.

Apart from the scaling mechanism, it is critical to answering questions like *when* and *how* to scale, and these are typically handled by a scaling controller known as the resource autoscaler [26, 29, 37, 46]. Through conventional monitoring tools, symptoms of under- and over-provisioning can be detected, and whether to make a change is decided. The resource autoscaler then identifies the causes of the symptoms (e.g., bottleneck or underutilized components) and performs scaling actions accordingly. Usually, the detection of suboptimal provisioning is based on performance metrics such as CPU/memory utilization and backpressure or congestion [21, 29]. Resource autoscalers then make scaling decisions using simple threshold-based heuristics, leveraging control-theoretic models, or adopting complex queuing theory models based on workload prediction [26, 28].

P1.2.3 *Autoscaling Challenges in DL Inference Serving*

Designing an efficient resource autoscaler for DL inference serving systems is non-trivial. In particular, we identify the following challenges, all of which combined distinguish the autoscaling problem in DL inference serving systems from those studied in other systems such as stream processing [46].

- **Dependency:** Modern DL inference systems typically involve multiple DNNs orchestrated with a DAG [12]. The edges in the DAG indicate the data dependencies between the DNNs, leading to the tight coupling of the scaling decisions for different DNNs. Hence, the scaling decisions for all the DNNs need to be coordinated holistically.

- **Uncertainty:** Depending on the output of the preceding DNN, an inference request may follow different execution paths in the dataflow graph [62]. This brings significant uncertainty in the system workload and renders prediction models based on queuing theory inaccurate.

- **SLA guarantee:** DL inference requests need to be processed within a certain amount of time in order to be useful, which refers to SLA guarantees [12, 47, 62]. It is particularly challenging to guarantee SLA in DL inference serving systems since requests following different execution paths may be specified with different SLAs, and due to uncertainty, the type (thus the corresponding SLA) of a request cannot be known a priori.

- **Batching:** DL inference serving systems typically adopt request batching, which is effective in improving resource utilization [8, 13, 65], as shown in Figure P1.2. In essence, we trade off latency for throughput with SLA guaranteed. Changing the batch size of one DNN leads to changes in the throughput and latency, affecting other DNNs due to dependency and SLA guarantee.

Existing autoscaler designs for DL inference serving systems are mainly based on decoupling the batch size and the autoscaling decision-making problems using simple heuristics. A popular approach is called "split-and-conquer" where we split the SLA over the DNNs following a proportional policy and make scaling decisions locally for

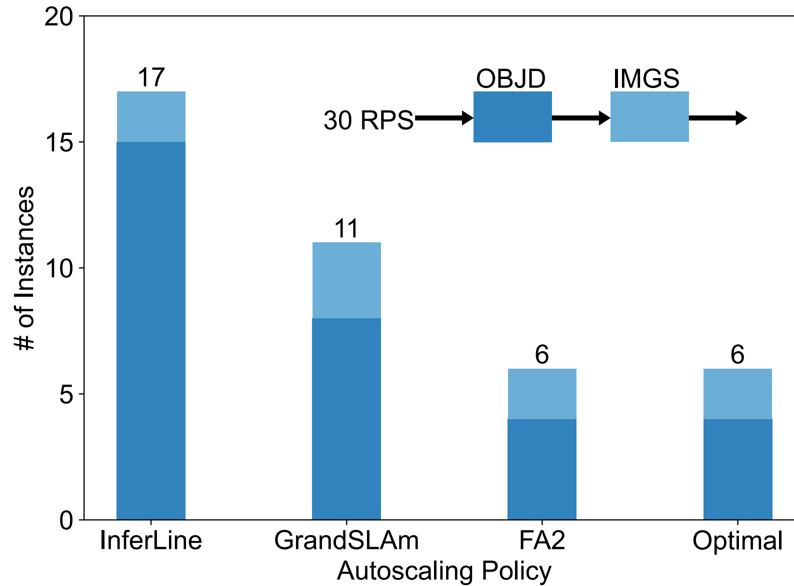


Figure P1.3: Number of instances required by different autoscaling policies. InferLine neglects the combinatorial nature of the models and iterates on all models one by one and GrandSLAm uses a predefined stack allocation for DNN models, both resulting in need of extra resources (3x and 2x, respectively) compared with the optimal resources provided by Gruboi.

each DNN, as done in GrandSLAm [47] and InferLine [12]. Despite convergence issues already noted in stream processing engines [46], such an approach is conservative and can lead to significant resource over-provisioning [62]. To confirm this observation, we perform an experiment using a simple DL inference serving pipeline consisting of two DNNs for object detection (OBJD) and image segmentation (IMGS), respectively. The results are shown in Figure P1.3. With a steady input of 30 RPS, GrandSLAm and InferLine require almost 2× and 3× of the optimal number of instances (obtained by the Gurobi solver), respectively. However, the exploration space for the optimal solution is exponentially large, and thus exhaustive search is not practical. In the rest of this paper, we will adopt a principled approach to tackle the autoscaling problem in DL inference serving systems. We will show how FA2 achieves (almost) optimal scaling decisions while being computationally efficient.

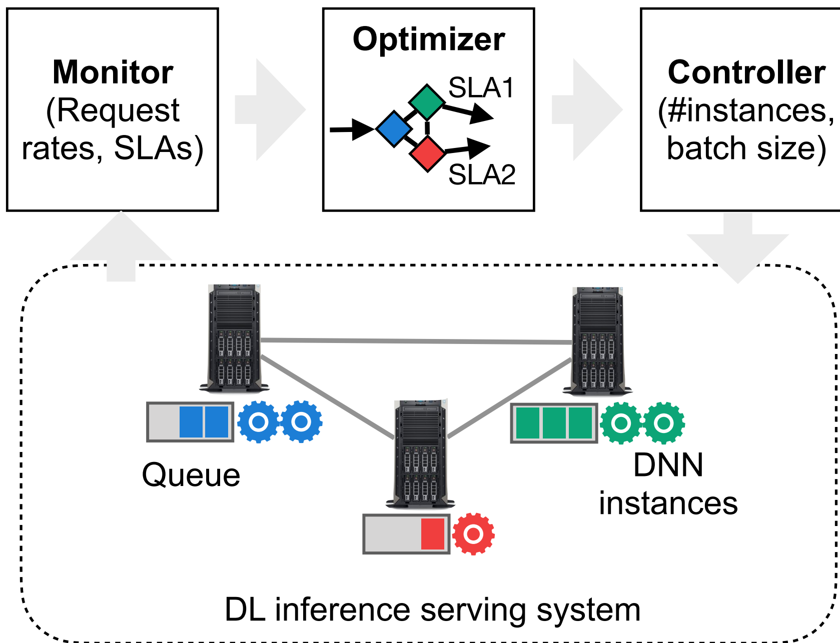


Figure P1.4: An overview of the FA2 architecture. The monitoring service collects the metric data from the DL inference serving system. The optimizer makes scaling decisions for the DNNs. The controller enforces the scaling decisions by configuring the inference serving system.

P1.3 FA2 SYSTEM DESIGN

In this section, we describe the design of FA2—fast, accurate autoscaling for DL inference serving with SLA guarantee. We first provide an overview of the system architecture and then discuss the system's major components.

System overview. An overview of the FA2 architecture is depicted in Figure P1.4. Our system consists of three main components (monitor, optimizer, and controller). The *monitor* keeps monitoring statistics about the distribution of request arrivals, i.e., the average number of requests that have followed each execution path in the execution graph. The *optimizer* takes the application execution graph with specified SLAs over execution paths as well as the request distribution from the monitor as input and makes scaling decisions (i.e., number of instances and batch size for every DNN in the execution graph) by solving an optimization problem. Afterward, the *controller* informs the DL inference serving system and reconfigures the system according to the scaling decision. The DL inference serving system deploys the

DNN instances, which serve requests from a central queue preceding all the DNN instances of the same type at each server. The system runs periodically to adapt to request rate changes as well as possible request distribution drifts. Similar to DS2 [46] for stream processing (which is not applicable to DL inference serving due to the challenges we have identified), FA2 aims to make holistic scaling decisions for all the DNNs in a single shot and satisfies the SASO properties (i.e., stability, accuracy, short settling time, and not overshooting) in control theory [40].

Monitor. The monitor pulls two types of metrics from the DL inference serving system in a predefined time interval: (a) the sequence of DNNs (i.e., the execution path) each inference request has traversed while in the system, and (b) the end-to-end processing time of each inference request following the execution path. The former is used to calculate the average number of requests that have followed a specific execution path in the past, while the latter is used to decide if a scaling operation is necessary depending on the ratio of SLA violations for the served requests.

Optimizer. The optimizer aims to generate the scaling decision—the optimal configuration (including the number of instances per DNN and the corresponding batch size for each DNN) to achieve the highest resource efficiency while respecting all the SLAs in the system using the metrics reported from the monitor. To this end, the optimizer first builds a graph-based model incorporating both the request processing and queuing delays. Then, it performs graph transformation to simplify the graph-based model and provides an efficient algorithm based on dynamic programming to calculate the scaling decision. We will elaborate them in §P1.4. Finally, the optimizer passes the scaling decision to the controller to enforce the system configuration in the system.

Controller. The controller is responsible for reconfiguring the system according to the scaling decision generated by the optimizer. This reconfiguration includes the batch size and the number of instances for each DNN in the inference serving system. To this end, the controller first compares the new configuration from the optimizer with the current system configuration. If both configurations are the same, no reconfiguration will be needed. Otherwise, the controller sends the new batch size information to the queues at each DNN and brings up/down DNN instances based on the difference between the two

Table P1.1: Notations

Symbol	Description
G	Application's dataflow graph (a DAG)
S	Set of registered DNNs
s	A DNN model from set S
P	Set of possible execution paths
p	An execution path in set P
n_s	Number of instances for DNN s
b_s	Batch size of DNN s
Int_s	Computation intensity of DNN s
$d_s(b_s)$	Processing time for DNN s with batch size b_s
$q_s(b_s)$	Max. queuing time at s with batch size b_s
$l_s(b_s)$	Total time spent at DNN s, i.e., $d_s(b_s) + q_s(b_s)$
$h_s(b_s)$	Throughput of s with batch size b_s
SLA_p	Service-level agreement for path $p \in P$
λ_p	Request rate for execution path $p \in P$
λ_s	Request rate at DNN s
$OPT(s,t)$	Optimal resources consumed by s and all its successors under time budget t

configurations. Note that if the only change in the new configuration is the batch size, the system can be reconfigured immediately without any delay.

P1.4 AUTOSCALING PROBLEM AND ALGORITHM

In this section, we focus on the optimization problem that the optimizer needs to solve to produce accurate scaling decisions. In particular, we adopt a principled approach and model the system comprehensively. We first model the processing time of DNNs and then provide a model for the worst-case queuing delay at the queue preceding the DNN instances of the same type. With these models, we formulate the autoscaling problem with an integer program. We then propose an efficient algorithm to solve the problem based on graph transformation and dynamic programming. Table P1.1 summarizes the notations we use in the paper.

P1.4.1 *DNN Performance Modeling*

To facilitate decision making at the optimizer, FA2 requires to know the performance, i.e., throughput $h(b)$ and latency $d(b)$ with respect to the batch size b, of each DNN instance. Prior work has demonstrated that the performance of DL models is quite predictable, especially of those for deep learning inference [13, 36, 47]. We follow the same line and use profiling data and robust regressions [20] to build models for all the DNNs in the system. Other more sophisticated performance modeling methods [44] can also be employed. Such performance models can be built offline and be reused throughout the lifecycle of the DNNs as long as the size of the DNN instance stays the same, which is true since only horizontal scaling will be considered.

In contrast to existing work [13, 47] which suggests a linear relationship between batch size and latency, we apply some slight changes that improve the prediction accuracy considering that a larger batch size could potentially better utilize non-shareable resources such as CPU caches other than the computing units. In particular, we use a second-order quadratic polynomial $d(b) = \alpha b^2 + \beta b + \gamma$ for latency prediction under a given amount of resources, where α, β, and γ are parameters and they will be fitted with profiling data. The throughput of a DNN instance is directly given by $h(b) = b/d(b)$. Our evaluation with 100K inferences for each DNN latency profiling (see Figure P1.6) confirms that the quadratic model is more accurate than a linear model with a much smaller mean squared error.

P1.4.2 *Queuing Delay Modeling*

The queuing delay at a given DNN can be affected by the following factors: the average request arrival rate λ reported by the monitor, the batch size b, processing time of the DNN instance $d(b)$, and the number of DNN instances n. For a request at a specific DNN, the worst-case queuing delay can be captured in two scenarios: (a) Assume a DNN instance is idle and waiting to serve requests. When a request arrives, it has to wait for another $b - 1$ requests to come until a batch can be formed and served by the DNN instance. In this case, the first request has to be queued for $(b - 1)/\lambda$ time before it can be processed. (b) Assume a DNN instance has been assigned a batch for

processing, and following a round-robin policy, the next batch for the same instance containing the $(nb+1)$-th through $(nb+b)$-th requests have arrived. Thus, the batch needs to wait for the DNN instance to be freed from processing the previous batch before it can be processed, even after the arrival of the last request of the new batch for the DNN instance, with a queuing time of $d(b) - (nb+1)/\lambda$ (the worst case happens to the first request in the new batch). By combining the above two cases, the worst-case queuing latency at a DNN with batch size b and n instances is given by

$$q(b,n) = \max\left(\frac{b-1}{\lambda}, d(b) - \frac{nb+1}{\lambda}\right). \tag{P1.1}$$

Meanwhile, the total throughput $n \cdot h(b)$ of all the n instances for the same DNN has to match the arrival rate, i.e., $n \cdot h(b) = \lambda$. With some simple manipulation we can obtain $d(b) = nb/\lambda$. Plugging this equation in the above queuing latency formula, we observe that the second term is always negative. Therefore, the worst-case queuing delay can be simplified as

$$q(b) = \frac{b-1}{\lambda}. \tag{P1.2}$$

We will use this equation for modeling the worst-case queuing delay in the problem formulation and our algorithm.

P1.4.3 *Problem Formulation*

Based on the DNN performance model and the queuing delay model we have introduced, we now provide a formal description for the resource autoscaling problem. We denote the dataflow graph of the application by $G = (S, E)$ where node-set S represents the set of DNNs and edge-set E represents the data dependencies between the DNNs. On the dataflow graph, the application specifies a set of execution paths denoted by set P. For each path $p \in P$, $S_p \subseteq S$ denotes the set of DNNs on p, and SLA_p denotes the SLA specified on the end-to-end latency of this path. The aggregate request arrival rate for the application is denoted by λ, which may change over time. Due to the uncertainty property of DL inference serving systems as discussed in Section P1.2, SLA-aware request scheduling (e.g., priority-based scheduling or earliest deadline first) at a DNN is not possible since it

is unknown which execution path will be taken by each request before the request leaves the system.

The monitoring system continuously reports to the controller the execution path that has been taken by each of the inference requests in a given period in the past. From such information, we derive the average number of requests served by each of the execution paths $p \in P$, denoted by λ_p. To ensure system stability, the aggregate throughput of all the instances for a DNN should be no less than the expected request rate, i.e., for any DNN $s \in S$, $h_s(b_s) \cdot n_s \geq \sum_{s \in p : p \in P} \lambda_p$. Such a constraint ensures that all the DNNs are sufficiently provisioned. As a result, queuing of inference requests at each DNN will be under control.

The optimization problem is to decide n_s and b_s for all $s \in S$ such that under a given workload, none of the SLAs specified by the execution paths are violated. The goal is to minimize the aggregate amount of resources used for all the DNNs. The problem can be formulated with the following integer program (IP):

$$
\begin{aligned}
\min \quad & \sum_{s \in S} n_s + \delta(\sum_{s \in S} b_s) \\
\text{subject to} \quad & \sum_{s \in S_p} d_s(b_s) + q_s(b_s) \leq SLA_p, \forall p \in P, \\
& h_s(b_s) \cdot n_s \geq \sum_{s \in p : p \in P} \lambda_p, \forall s \in S, \\
& b_s, n_s \in \mathbb{Z}^+, \forall s \in S.
\end{aligned} \tag{P1.3}
$$

In the objective function, in addition to the total number of instances, we introduce a small penalty term $\delta(\cdot)$ on the total batch sizes. This penalty ensures we will use the minimal possible batch sizes under the optimal number of instances due to the fact that a larger batch size without enough queries in the system not only increases the processing and queuing delay (which can be valuable for other DNNs) but also does not increase the system's utilization. The first constraint ensures that all the SLAs will be satisfied. We omit the network latency as we assume high-throughput, low-latency network links are available in data centers. Our model is capable of incorporating network latency. The second constraint enforces the stability of the system. The number of instances and the batch size should both be positive integers, as shown in the last line. The objective is to minimize the resources, i.e., the total number of DNN instances used by the application. Each DNN can have a different batch size and number of instances. Thus, the

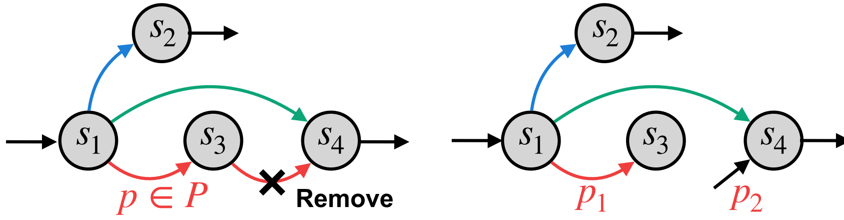


Figure P1.5: Graph transformation example: (left) the original dataflow graph, (right) the transformed graph containing only egress aggregators.

solution space for the IP increases exponentially for every new DNN added to the system, making it hard to explore the space to find the optimal solution exhaustively. For example, for just 10 models with the maximum batch size of 128 and the maximum number of instances of 8, the feasible space could be as large as $(128 \times 8)^{10} = 1024^{10} \approx 10^{30}$). To address this issue, we provide an efficient algorithm to solve the IP based on graph transformation and dynamic programming, as detailed in the following sections.

P1.4.4 *Graph Transformation*

Our autoscaling algorithm consists of two parts: graph transformation and dynamic programming. We now describe why and how we transform the graph model and elaborate on the dynamic programming design in the next section.

The autoscaling problem is challenging due to the dataflow dependency, combined with SLA guarantee and batching, as discussed in Section P1.2. This problem is illustrated in Figure P1.5 (left), where two execution paths (green and red-colored edges) share DNNs s_1 and s_4 in a fork-join fashion, leading to a deadlock situation in scaling these DNNs if we look at these DNNs one by one. In this example, if we decide the instance numbers and batch sizes for s_1 and s_4 from the green-colored path, the configurations may be sub-optimal or even infeasible for the same DL models in the red-colored path with a different SLA and more DNNs on the path. More specifically, if an application's dataflow graph meets the following condition, the problem can be solved efficiently: The dataflow graph does not contain both ingress and egress aggregators at more than one DNN. Here, an ingress aggregator represents a DNN that receives requests from multi-

ple preceding DNNs, and an egress aggregator represents a DNN that sends requests to multiple succeeding DNNs. If we can avoid having both types of aggregators in the dataflow graph, we can eliminate the dependency issue explained in the above example. This property significantly reduces the complexity of the autoscaling problem. Without loss of generality, we choose to avoid ingress aggregators.

To ensure the above property, we apply graph transformations to remove a subset of edges from the dataflow graph. To this end, we define a new metric called *sharing degree* for each edge, which captures the number of execution paths on which the edge is being used. The calculation of the sharing degree for each edge can be done by traversing through all the execution paths. After that, we loop through all the ingress aggregator DNNs and pick the edge with the lowest sharing degree to remove. Since removing the edge from the dataflow graph requires splitting the execution paths that use this edge to be partitioned into two parts, choosing the ones with the lowest sharing degree leads to the least number of execution paths we need to partition. To partition an execution path, we need to split the SLA specified for that execution path into two parts, and then the two path segments can be treated as entirely independent execution paths with their own SLAs. We decide to split the SLA proportionally for the two path segments based on a metric called *intensity* (denoted by Int), which characterizes how intensive the computation is for the DNNs on a path segment. Other optimizations considering the queuing delay of the DNNs can also be applied. We compute the intensity of a DNN by averaging the processing time of the DNN over a set of batch sizes (here, we use batch sizes in [1,16] as the processing time with a larger batch size may violate SLAs solely). For a path p to be split, we assume p_1 is the first segment before the edge to be removed, and p_2 is the other segment. The SLA for the first segment can thus be calculated as

$$SLA_{p_1} = \frac{\sum_{s \in p_1} Int_s}{\sum_{s \in p} Int_s} \cdot SLA_p. \tag{P1.4}$$

The SLA for p_2 can be computed analogously. We repeat the above procedure until no ingress aggregators can be found in the dataflow graph.

Finally, the number of requests to a DNN is directly calculated as the sum of its preceding DNNs on the original dataflow graph (not the one after transformation). To this end, we adopt a topological sort

algorithm [9] to sort all the DNNs in the original dataflow graph and calculate the workload for each execution path in the transformed graph. The new graph with the workload information now can be handled with dynamic programming to obtain the optimal scaling decision, as we will explain in the next section.

P1.4.5 *Scaling with Dynamic Programming*

Now, we focus on how to solve the resource scaling problem with dynamic programming on the transformed graph. We denote by $OPT(s,t)$ the optimal solution, i.e., the minimum resource consumption, when we consider only DNN s and all its successors in the transformed graph, given a time budget of t. We consider two cases here:

Case 1: A DNN without successors. In this case, time t can be allocated to the DNN entirely, and the optimal solution is achieved with the maximum batch size that can be handled within time t:

$$OPT(s,t) = \min_{b:l_s(b)\leq t} \left(\lceil \frac{\lambda_s}{h_s(b)} \rceil \right). \tag{P1.5}$$

Case 2: A DNN with successors. Let us denote by $M \subset S$ the set of successors of s. In this case, time t can be split into two parts: one part for the current DNN s and the other for all its succeeding path segments. The optimal solution consists of the resources consumed by s plus the sum of resources consumed by DNNs on all the succeeding paths of s. For each DNN s we denote by r_s the maximum time that can be spent on DNN, which is calculated as the minimum of the SLAs of paths that s is on, i.e.,

$$r_s = \min_{p:s\in p \wedge p\in P} SLA_p. \tag{P1.6}$$

The optimal solution from DNN s given time budget t can be obtained following the recursive function:

$$OPT(s,t) = \min_{b:l_s(b)\leq r_s} \left(\lceil \frac{\lambda_s}{h_s(b)} \rceil + \sum_{m\in M} OPT(m,t_m) \right) \tag{P1.7}$$

where $t_m = \min\{SLA_p, t - l_s(b)\}$ if m is the first DNN on path p and $t' = t - l_s(b)$ otherwise. Assuming s_1 is the first DNN in the transformed graph after the topological sort, we add an artificial node s_0 preceding s_1 in the graph. The optimal resource consumption by

the system is thus given by $OPT(s_0, t_{max})$ where $t_{max} = \max_{p \in P} SLA_p$ denotes the maximum time any execution path on the graph can spend.

The above recursive function leads to an algorithm based on dynamic programming, which leverages the optimal substructure of the problem and avoids redundant computation. The pseudo-code for our algorithm based on dynamic programming is listed in Algorithm 1. We define a matrix dp containing tuples of $(instance_number, batch_size)$. We initialize dp in line 1. Then, we iterate over all the DNNs in a reversed order of the topologically sorted dataflow graph, the possible time budget, and the batch size. For each DNN under the given time budget and batch size, we compute its processing time and queuing time to see if the time budget is possible. If so, we apply fill the dp matrix with Equation P1.5 (line 9) and Equation P1.7 (lines 10–14). Finally, we use backtrack in the filled dp matrix to obtain the optimal number of instances and batch size for each DNN. The time complexity of the proposed algorithm is dominated by the dynamic programming part, where we need to iterate over multiple dimensions to fill in the DP table. This time is calculated as $O(|S| \cdot t_{max} \cdot b_{max} \cdot |S|)$ where $|S|$ is the number of DNNs, t_{max} is the number of time slots (of 1 ms length) conditioned by the largest SLA, b_{max} is the maximum possible batch size. Also, the dominant data space needed in Algorithm 1 is for the DP table, an array of size $|S| \cdot t_{max}$ holding two-tuples of integers.

P1.5 IMPLEMENTATION

We implemented FA2, including the monitor, optimizer, and the controller in Python. The source code of the FA2 runtime framework is available at [59]. The DNNs are implemented with TensorFlow [2] which is an open-source framework for machine learning. Each DNN instance is built with a Docker container [16] with a pre-specified amount of resources running the target DNN inside the container. FA2 leverages Kubernetes [23] to orchestrate and manage the containers in the system. In particular, FA2 sends control commands to Kubernetes for scaling in/out the DNNs and reconfiguring the batch size for the instances of each DNN.

Each DNN is built with two components: (a) a central queue in front of all the instances for the DNN, and (b) a set of DNN instances (the

Algorithm 1: Dynamic Programming

```
input  : graph G (after transformation), {SLA_p : p ∈ P}, {λ_s : s ∈ S}
output : OPT(s_0, t_max)
1  dp ← [|S|][t_max](∞,0) // (instance_number, batch_size)
2  for s in reversed(S) do
3      for t in [1, t_max] do
4          for b in [1, b_max] do
5              q_s ← (b_s − 1)/λ_s, p_s ← d_s(b_s), l_s ← p_s + q_s
6              if l_s > t then
7                  break
8              else if s has no successors then
9                  dp[s][t] ← (⌈λ_s/(b_s · ⌊t/p_s⌋⌉, b_s)
10             sum ← ⌈λ_s/(b_s · ⌊t/p_s⌋⌉
11             for p in P do
12                 for m in p do
13                     if (s,m) ∈ p and t − l_s <= SLA_p then
14                         sum ← dp[m][t − l_s].in + sum
15             if sum < dp[s][t].in then
16                 dp[s][t] ← (sum, b)
```

number of instances is given by FA2) running in Docker containers which fetch requests from the central queue for processing. The central queue holds pending requests and composes batches to feed the DNN instances according to the batch size configuration given by FA2 for the DNN. The central queue sends batched requests to the DNN instances using a round-robin policy [6]. The interactions between the queue and the DNN instances and the communication between different DNNs are handled by gRPC [34].

P1.6 EVALUATION

In this section, we perform comprehensive experiments to demonstrate the effectiveness of FA2 in real-world applications with various realistic workloads. All experiments are performed based on the aforementioned system implementation.

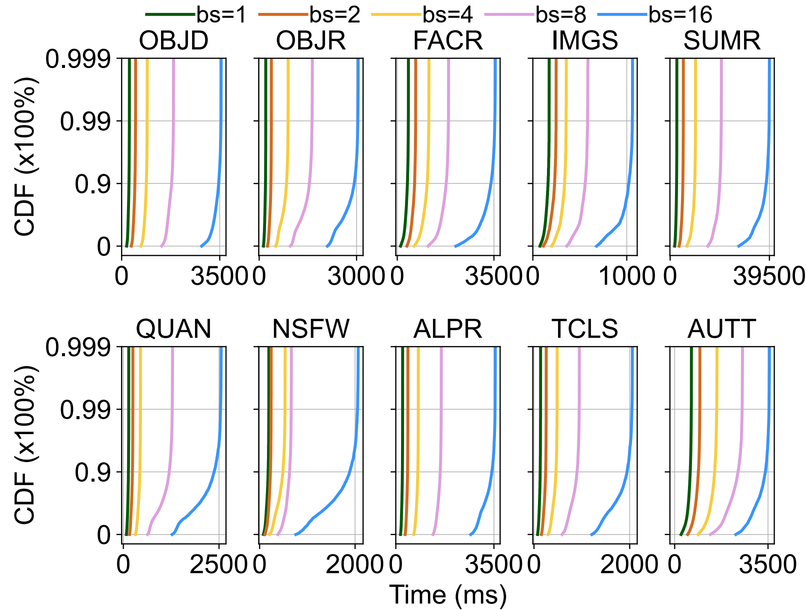


Figure P1.6: Inference latency distribution for the considered DNNs under varying batch sizes on CPU (an instance equipped with one CPU core).

P1.6.1 *Experimental Setup*

We now describe the setup for our experiments, including the application and its constituting DNNs, the SLAs, the evaluation metrics, the baselines, and the workloads.

Hardware. We deploy FA2 on a testbed consisting of four servers where two have Core i9-9980x CPUs, and the other two have Core i9-10940x CPUs. Each server is equipped with an NVIDIA RTX2080 GPU. The servers can support up to 60 one-core CPU instances and 40 GPU instances, each taking 10% of the GPU share via CUDA MPS. The servers run the Ubuntu 18.04 operating system and are interconnected with a stable private Ethernet network (1Gbps). We evaluate FA2 on both the CPU and GPU.

The Application and DNNs. To compare FA2 with existing solutions in realistic environments, we use DNNs in the computer vision, natural language processing, and audio recognition domains, which are also heavily used in other DL inference serving systems [12, 47, 61, 62]. We consider an application comprised of the DNNs listed in Table P1.2. Each instance encapsulates the DNN with pre-specified resources: 1

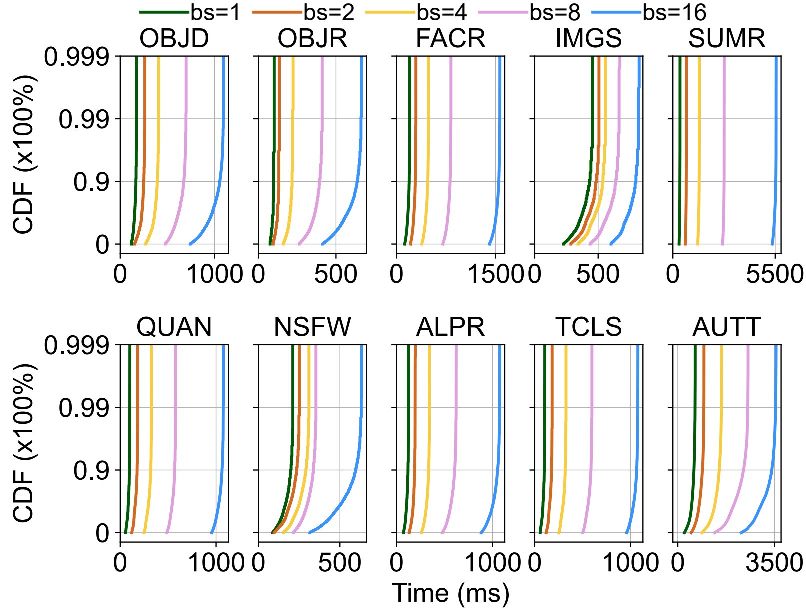


Figure P1.7: Inference latency distribution for the considered DNNs under varying batch sizes on GPU (on an instance equipped with 10% of the total GPU processing units specified using CUDA MPS).

CPU core or 10% GPU share, determined based on the observation from Figure 2. The application contains multiple possible execution paths over the involved DNNs as specified in Table P1.3. Note that FA2 does not assume the path for a request is known a priori—the next hop of a request is revealed only after the processing is done at a DNN. We profile all the DNNs to obtain the throughput and latency to build the performance model for the DNN processing with varying batch sizes. Figure P1.6 and Figure P1.7 show the distribution of the inference latency over more than 250K data points on CPU and GPU, respectively. We use the data points at the 99-th percentile as the reference numbers, as also done in [47], to build the performance models following the methodology described in Section P1.4.1.

SLAs. SLAs are commitments between service providers and users and are typically defined based on the processing time of the involved DNNs. To set the application SLAs realistically, we calculate the tail processing time (proved to be highly predictable [36]) required by each of the execution paths under batch size $b = 1$ and multiply that time by a factor of five following a similar methodology described in [35]. This SLA setup serves as a suggestion, and FA2 can work with other SLA setups. The actual SLA depends on the application and can also be part of the service provider's pricing scheme, where

Table P1.2: DNNs Involved in the Application

Task	Abbreviation	DNN Model
Object Detection	OBJD	Inception V2
Object Recognition	OBJR	ResNet50
Not Safe For Work	NSFW	MobileNet V2
Car Recognition	ALPR	SSD
Face Recognition	FACR	ResNet50
Image Segmentation	IMGS	MobileNet V2
Question Answering	QUAN	DistilBERT
Text Summarization	SUMR	BART
Text Classification	TCLS	DistilBERT
Audio To Text	AUTT	Wav2Vec2

different prices will be offered under different SLAs. In the execution paths in Table P1.3, the SLAs varying from 3020ms−25710ms and 960ms−4815ms, for the CPU and the GPU cases, respectively.

Table P1.3: All Possible Execution Paths in the Application with Their SLAs When Deployed on CPUs and GPUs

Execution Path	**SLA-CPU (ms)**	**SLA-GPU (ms)**	**Description**
OBJD→ALPR→QUAN	3020	1205	Provides answers to queries regarding a car in an image
OBJD→NSFW→FACR	3505	1670	Detects and recognizes a human if the image is safe for work
OBJD→OBJR→IMGS	3095	960	Detects, classifies an object, and provides segmentation of the object
SUMR→QUAN	13580	2200	Summarizes texts and provides answers in the texts
AUTT→QUAN	12245	3005	Converts audios to texts then performs question answering on them
AUTT→SUMR→TCLS	25710	4815	Performs text classification on summarized texts from audios

Evaluation metrics. We consider the following four types of metrics in our evaluation. (a) *Processing and queuing delay*: We use these data to demonstrate the effectiveness of FA2 in predicting the processing and queuing delay of DNNs. (b) *SLA violation ratio*: We measure the end-to-end latency of every inference request and check whether the request is processed within its SLA. (c) *Resource consumption*: We use the total number of DNN instances to denote the resource consumption where instances are homogeneous (1 core in the CPU case and 10% GPU power set with CUDA MPS). (d) *Instance utilization*: We collect the CPU utilization of each DNN instance captured in 100ms intervals.

Baselines. We compare FA2 with the recently proposed inference serving frameworks, namely GrandSLAm [47] and InferLine [12], and a state-of-the-art stream processing autoscaler called DS2 [46]. GrandSLAm maximizes the system's throughput by using dynamic batching and request reordering while guaranteeing SLAs. We consider the same resource configuration as produced by FA2 for GrandSLAm and examine its SLA violation. InferLine uses a greedy approach to find the minimum cost by choosing the most affordable configuration among different hardware and increasing the batch size for DNNs while not violating any SLAs. DS2 is an autoscaler that uses the true processing time and output rate to scale up/down stream processing operators based on the workload. DS2 does not consider request batching, so we use a fixed batch size $b = 1$. Finally, we compare FA2 with the optimal solutions generated by the Gurobi solver [39] to show how close FA2 performs to the optimal.

Workloads. To evaluate FA2 under varying workload conditions, we develop a workload generator that generates requests at different rates (following different traces) and distributes these requests to the application paths uniformly at random. Figure P1.8 shows the six workload trace types we use for evaluating FA2: two *steady* traces where the request rate follows a Poisson distribution (also used in [47, 61]) with an average arrival rate of around 8RPS (low) or 27RPS (high), *fluctuating* trace where the average request rate jumps between 8RPS and 27RPS, a random trace following a normal distribution with mean 15 and standard deviation 5 to have high-variance request rates, and two different *real-world* traces where we use request arrival rates directly from the Microsoft Azure FaaS (MAFS) traces provided in [64]. The Azure traces consist of more than 46K functions, counting invocation counts per minute per function. The invocation counts

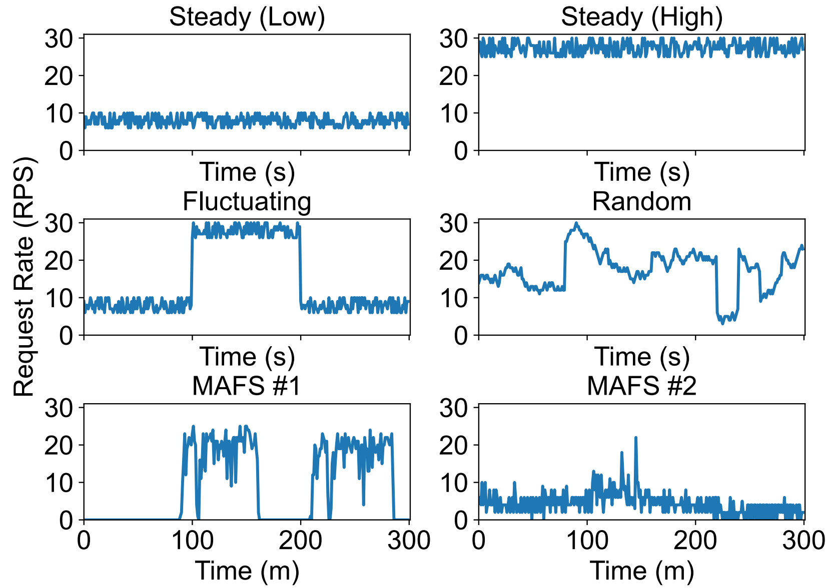


Figure P1.8: Workloads used in evaluation: two steady synthetic traces following Poisson distributions, a fluctuating synthetic trace with a spike in the middle, a random workload following a normal distribution, and two realistic traces from the Azure FaaS function invocation traces.

vary from 0 to over 150K per minute, per function. We take two traces of function innovation counts for this experiment, which our experimental environment can handle with the maximum utilization and have different request arrival patterns.

P1.6.2 *End-to-End Performance*

In this part, we evaluate FA2 and compare its performance with the baselines under different workloads.

P1.6.2.1 *Steady Workloads*

Under steady workloads, the number of DNN instances remains constant, and the system is stable. We notice that the SLA violation rate is less than 1% for all the approaches under such a situation. However, FA2 needs less resources to serve the requests compared with the baselines (discussed in Section P1.6.3), which can be explained by the CPU

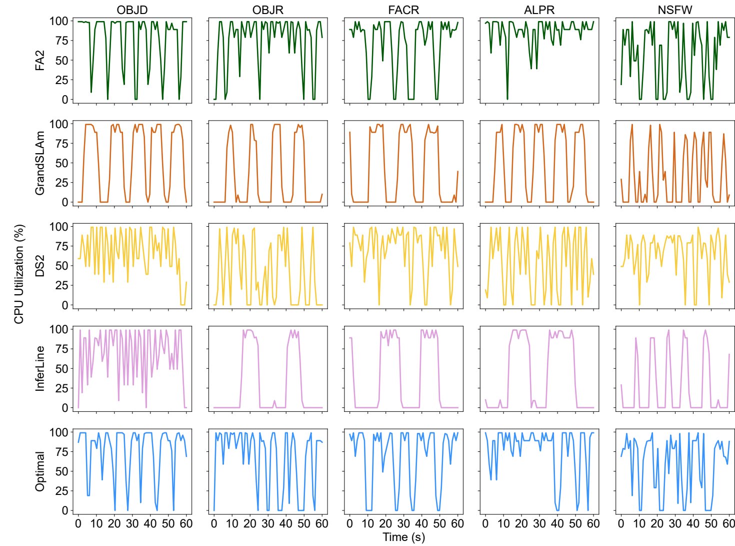


Figure P1.9: CPU utilization of five DNN instances for all approaches under the steady workload at a given time-window.

utilization statistics. As illustrated in Figure P1.9, with GrandSLAm and InferLine, the DNN instances need to wait for request arrivals to create the maximum possible batch at each DNN, resulting in considerable CPU idle time, leading to resource waste. DS2 does not leverage the batching technique, which also leads to CPU under-utilization. With generally lower CPU utilization, the baseline approaches require more computing resources than what is needed (shown as *Optimal* in the figure). FA2 overcomes these issues by making holistic scaling decisions where the number of instances and the batch size for each DNN are determined jointly.

P1.6.2.2 *Fluctuating and Random Workloads*

When the request rate fluctuates, the system needs to adapt by providing a new set of configurations for all the DNNs. To this end, FA2 collects metrics in 10-second intervals and decides new configurations for the next interval.

The adaptation interval of FA2 cannot be further reduced due to the following system overheads: (a) FA2 optimizer needs around 500ms to make a new adaptation decision, (b) the system needs up to 6s to bring up new instances (Kubernetes pod cold-start time) in case of scaling out, and (c) the runtime system needs a few seconds to drain the queues and stabilize the system. One possible solution to this

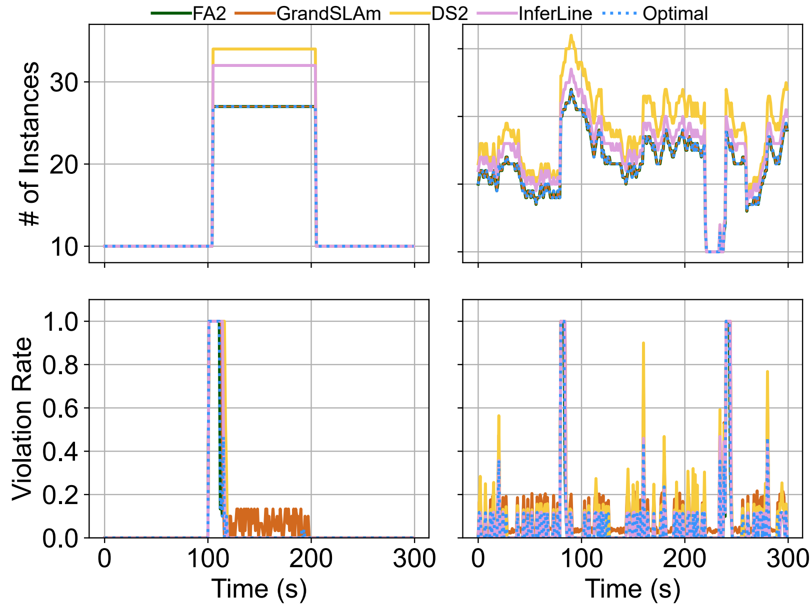


Figure P1.10: Resource usage and SLA violation over time under fluctuating (left) and random (right) workloads.

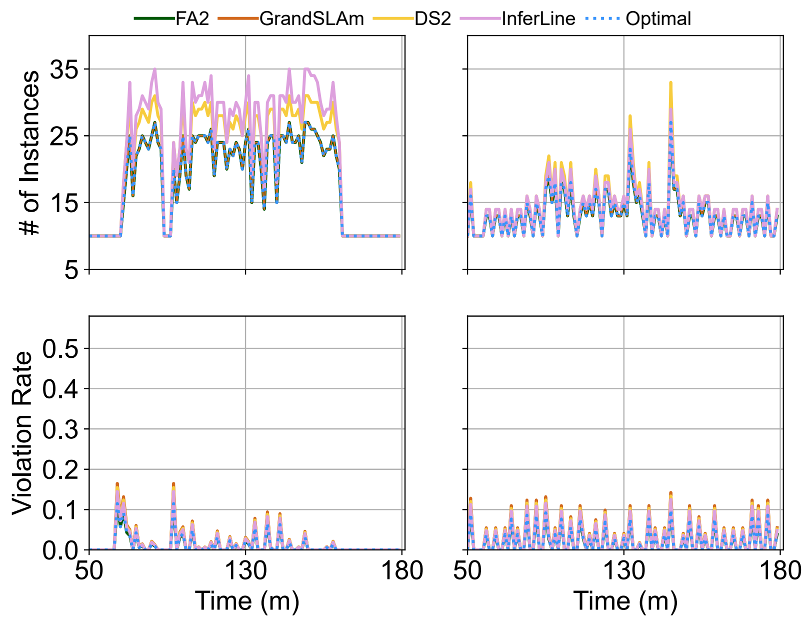


Figure P1.11: Resource usage and SLA violation over time under two (left and right) Azure FaaS workload traces.

limitation is to use a workload predictor to make autoscaling decisions proactively. To have a fair comparison, we set the same adaptation interval for the baselines. Moreover, we use the same scaling decisions produced by FA2 in GrandSLAm as their approach does not decide the number of instances for each DNN.

Figure P1.10 shows the number of instances required for the fluctuating (left) and random (right) workloads in each approach and the corresponding SLA violation rate. When a change in the arrival rate is detected (after around three seconds, as shown in Figure P1.10), the framework provides the new configuration to the Kubernetes cluster, which brings up/down containers for the concerned DNNs. Before the instances can start serving requests, a few seconds delay is observed as the instances need to be initiated. After that, the framework starts to serve the upcoming requests, and the system becomes stable again gradually. This procedure takes longer for the baseline approaches as the number of instances to add to the system is higher than that is in FA2 and the management overhead (of Kubernetes) slows down the scaling-out process of the instances due to performance interference caused by that the instances are co-located on the same physical machine, resulting in a higher SLA violation rate (up to 4% more). Overall, the results from the fluctuating and random workloads are consistent, except that a higher SLA violation rate is observed due to the higher workload variation.

P1.6.2.3 *Real-World Workload*

We follow the same procedure described above and use five hours of two of the real-world FaaS traces provided by Microsoft Azure. Figure P1.11 depicts the required number of instances and the SLA violation rate of all the approaches. As the number of queries suddenly increases or decreases, the framework captures the changes within a few seconds and adapts the system accordingly. Similar to the case with the fluctuating and random workloads, FA2 needs up to 14.6% less computational resources when compared with DS2 and InferLine while reducing the SLA violation by 4.8% compared with DS2 and GrandSLAm.

P1.6.3 *Resource Efficiency*

We assess the resource efficiency of FA2 compared with the baselines under a steady workload with varying request rates, i.e., from 6 to 60. Figure P1.12 depicts the required number of instances for each approach under each request arrival rate. InferLine tries to reduce the

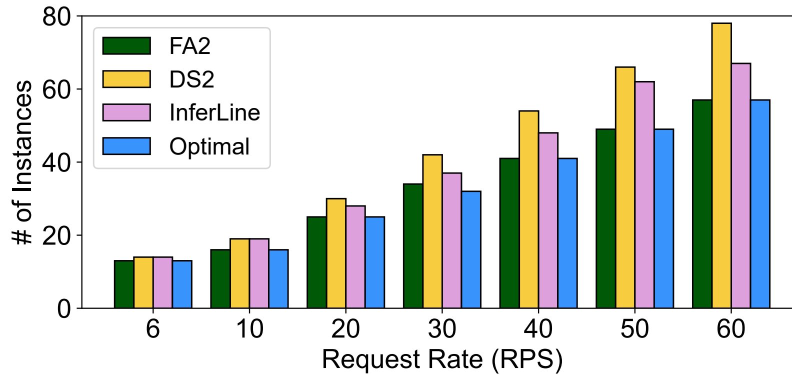


Figure P1.12: Comparison of resource consumption under varying request rates on CPUs. FA2 shows clear advantages over all the existing approaches and is comparable to the theoretical optimum.

number of instances by maximizing local batch sizes one by one until there is no room to increase the batch size of any DNN. This approach needs on average 19% (and up to 48%) more computational resources when compared with FA2 as the InferLine approach does not consider coordinating the scaling decisions of the DNNs on the same execution path. DS2 does not leverage batching, leading to overall low resource utilization. Consequently, the DS2 approach requires on average 26% (and up to 62%) more computational resources when compared with FA2. Also, FA2 performs close to the optimal (produced by Gurobi solver) where the match to optimal decisions is over 96.8%. This proves that FA2 is effective in improving the resource efficiency of DL inference serving systems.

P1.6.4 *Impact of Optimized Batch Sizes*

We evaluate FA2 under static workloads and fixed computational resources and compare it with the baselines and the optimal solution produced by Gurobi. We feed the workload to the Gurobi solver to obtain the optimal number of instances for each workload and apply the results to all the approaches, including FA2. We also consider that a request is dropped if its latency has exceeded 3× the SLA to avoid constant queue overflow. We run the experiment for each approach for 1200 seconds and collect the results when the system is stable.

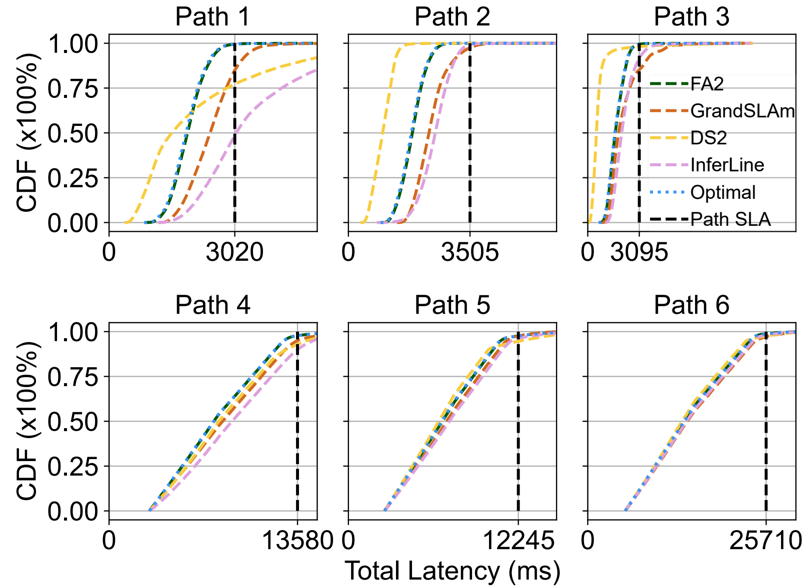


Figure P1.13: End-to-end latency distribution for the three execution paths in the application under different scaling approaches on CPUs. FA2 outperforms all the existing approaches to a large extent with SLA violations less than 1%.

Figure P1.13 shows the CDF of the end-to-end latency of all the approaches under fixed resources and static workload with more than 300K queries over time. DS2 does not consider request batching, leading to the least number of violations (less than 17% among the other baseline approaches. However, due to the lack of resources, over 32% of the requests are dropped. GrandSLAm adopts the "split-and-conquer" approach with a static strategy for SLA partitioning, resulting in poor adaptivity to the changes in workload distribution over the execution paths. InferLine tries to maximize the system's throughput via local optimization, where it iterates over the DNNs to increase the batch size of each DNN until there is no possibility for further batch size increases. While resulting in numerous SLA violations, the request dropout rate is the minimum among all the baselines. FA2 overcomes all these issues by considering the current workload and dynamic SLA allocations to all DNNs in a holistic manner. Moreover, FA2 servers all the requests (0% dropout) with an SLA violation rate as low as 1.4%, matching the optimal solution.

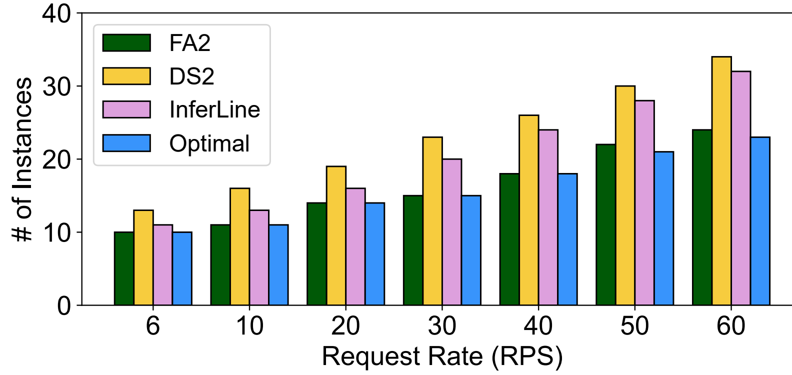


Figure P1.14: Comparison of resource consumption under varying request rates on GPUs. FA2 shows clear advantages over all the existing approaches and is comparable to the theoretical optimum.

P1.6.5 *Performance of FA2 on GPUs*

This section evaluates the performance of FA2 under GPU resources and different workloads and compares its performance with the baseline approaches. We use CUDA Multi-Process Service (MPS) [10] to share GPU processing power across multiple instances, as also used in [15] recently. Each instance receives a subset of the available processing units on the GPU enforced by CUDA MPS. We allocate 10% of the GPU processing units (similar to [15]) and memory to each instance. We show how FA2 saves GPU resources and achieves a better SLA guarantee when compared with the baseline approaches.

Figure P1.14 shows the average required number of instances for all the DNNs under the given workload with GPU. FA2 saves on average 25% (up to 50%) and 31% (up to 54%) GPU resources compared to InferLine, and DS2 respectively. The results differ from the CPU evaluations as the GPU instance has much better performance (higher throughput, lower latency) than the CPU instance, thus being more sensitive to the SLA division policy. FA2 achieves near-optimal SLA division, but the heuristics-based baselines are poor in this respect, thus amplifying the inefficiencies of these baselines.

Figure P1.15 illustrates the CDF of the end-to-end latency in all approaches under static GPU resource allocations and steady workloads with more than 12K queries per approach. In static GPU resource allocations, the number of instances for each DNN remains the same

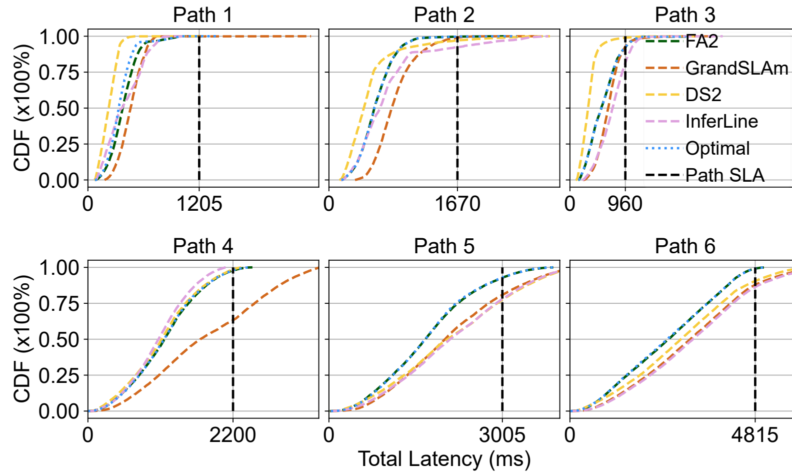


Figure P1.15: End-to-end latency distribution for the six execution paths in the application under different scaling approaches on GPUs. FA2 outperforms all the existing approaches to a large extent with SLA violations around 1%.

during the experiment, but the batch size is different for different approaches. The results confirm that FA2 can be effectively applied on different computing platforms without obvious performance deviations.

P1.7 RELATED WORK

This section discusses resource autoscaling in four related research areas: (a) inference serving, (b) microservices, (c) stream processing, and (d) serverless computing. We note that there are autoscaling studies in batch processing systems (e.g., MapReduce) [58, 71]. However, these systems differ significantly from inference serving systems in that the tasks are transient, and the task dependency is simple and deterministic (e.g., two stages as in MapReduce) in such systems.

Autoscaling for inference serving. InferLine [12] enables autoscaling by starting from a feasible system configuration and then optimizing the configuration by adapting the hardware and batch size for each of the DNNs. InferLine does not account for the conditional execution, and the heuristic approach in choosing the configuration for each DNN leads to considerable resource underutilization. Clockwork [36] focuses on inference serving with SLA guarantees without considering

DNN dependency and conditional execution. Nexus [65] is addressing a similar autoscaling problem. However, Nexus focuses mainly on (a) simple tree-like dataflow graph while FA2 does not assume the graph structure and can work with arbitrary graphs, (b) applications specified with a single SLA while FA2 targets consolidated applications with different SLAs, (c) GPU-level resource allocation, while FA2 allows fine-grained GPU resource allocation via CUDA MPS, and (d) allocating half of the application SLAs to queuing delays leading to reduced GPUs utilization, while FA2 carefully allocates the exact amount of time for each DNN's queue.

Microservice autoscaling. Existing microservice autoscaling mechanisms are mostly rule-based heuristics [22, 33, 49, 67, 68, 72] or meta-heuristics [7, 24, 70]. For microservices, request processing typically follows a deterministic dataflow-graph, instead of one with conditional execution paths as in inference serving systems. Machine learning methods are also employed for workload prediction in microservice systems for better autoscaling [4, 11]. Recently, DAGOR [73] provided overloading detection and collaborative load shedding in microservice systems. ATOM [30] is a model-driven microservice autoscaler based on layered queuing networks [60], but it does not consider request batching and end-to-end SLA guarantees. GrandSLAm is a microservice management framework focusing on improving throughput while guaranteeing application SLAs [47]. However, GrandSLAm does not deal with the autoscaling issue.

Autoscaling for stream processing. A large body of work has been dedicated to scaling operators in stream processing systems [1, 5, 14, 19, 21, 25, 29, 37, 42, 46, 52, 54, 55, 69]. Most of them rely on coarse-grained metrics such as the CPU/memory utilization, system throughput, queue size, and/or back-pressure and apply threshold-based policies for scaling in/out operators. Some alternative solutions such as DRS [25] or Nephele SPE [52] adopt queuing theory models to characterize the system. DS2 focuses on estimating the true processing and output rates of individual dataflow operators and figures out the scaling decisions for all the operators in one shot [46]. Scaling DL inference differs from scaling stream processing operators in that a DL application may contain multiple (conditional) execution paths, each with a specific SLA requirement [27]. While there exist stream processing systems that provide SLA guarantees [14, 52], none of them consider conditional execution while guaranteeing a set of SLAs over

multiple execution paths. Therefore, none of the existing autoscaling solutions for stream processing systems can be directly applied to elastic DL inference serving.

Autoscaling for serverless computing. Serverless computing is a widely adopted paradigm to provide autoscaling in cloud environments [45]. There exist numerous serverless platforms in academia (e.g., [38, 41, 45, 48, 66]), industry (e.g., [31, 56, 63]) as well as open-source solutions (e.g., [17, 43, 50]). Largely, autoscaling is achieved using three approaches: (a) request-based, (b) concurrency value-based, and (c) metric-based [53]. Most industry providers use a request-based scaling approach in which the cloud resources are scaled up when there are more requests for executing functions, while the resources are scaled-down otherwise. However, this approach does not fulfill SLAs. The second approach executes the function concurrently on a given number of instances, and when this value is reached, the resources are scaled down [32, 48]. Finally, most of the open-source platforms such as OpenFaaS [17] and Kubeless [50] use a metric-based scaling approach. This approach aims to maintain metrics such as latency, throughput, and CPU usage within a predefined threshold. However, this approach has the worst delay in adapting to fluctuating workloads. Overall, these solutions lack data dependency and conditional execution support, as typically seen in inference serving systems.

P1.8 CONCLUSION

In this paper, we presented FA2, a fast, accurate resource autoscaler tailored for efficiently provisioning DL inference serving while guaranteeing service-level agreements on the end-to-end latency. FA2 leverages a graph-based model to capture the resource scaling problem and makes resource scaling decisions for all the DNNs in a holistic manner. Evaluation results based on an actual system prototype and real-world workload traces show that FA2 improves the overall resource utilization significantly compared with the state-of-the-art resource scaling solutions. Overall, FA2 is able to match the optimal theoretical decisions almost always. Although we only tested with CPUs and GPUs, our design of FA2 is generally applicable to other mixed setups. In the future, we plan to extend FA2 to more heterogeneous environments

(including CPUs, GPUs, TPUs, and other accelerators) by introducing decision variables for hardware type for each DNN.

P1.9 ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers and our anonymous shepherd for their valuable comments and suggestions. This work has been funded by the German Research Foundation (DFG) within the Collaborative Research Center (CRC) 1053 MAKI.

REFERENCES

[1] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. "The Design of the Borealis Stream Processing Engine." In: *Biennial Conference on Innovative Data Systems Research (CIDR)*. 2005.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. "Tensorflow: A system for large-scale machine learning." In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2016, pp. 265–283.

[3] Fawad Ahmad, Hang Qiu, Ray Eells, Fan Bai, and Ramesh Govindan. "CarMap: Fast 3D Feature Map Updates for Automobiles." In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2020, pp. 1063–1081.

[4] Hanieh Alipour and Yan Liu. "Online machine learning for cloud resource provisioning of microservice backend systems." In: *IEEE International Conference on Big Data (BigData)*. 2017, pp. 2433–2441.

[5] Muhammad Bilal and Marco Canini. "Towards automatic parameter tuning of stream processing systems." In: *ACM Symposium on Cloud Computing (SoCC)*. 2017.

[6] T Brisco. *RFC1794: DNS support for load balancing*. 1995.

[7] Tao Chen and Rami Bahsoon. "Self-Adaptive Trade-off Decision Making for Autoscaling Cloud-Based Services." In: *IEEE Transactions on Services Computing* 10.4 (2017), pp. 618–632.

[8] Yujeong Choi, Yunseong Kim, and Minsoo Rhu. "Lazy Batching: An SLA-aware batching system for cloud machine learning inference." In: *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2021, pp. 493–506.

[9] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[10] NVIDIA Corporation. *CUDA MPS*. https://docs.nvidia.com/deploy/mps/index.html. Accessed on 29.10.2021. 2021.

[11] Nathan Cruz Coulson, Stelios Sotiriadis, and Nik Bessis. "Adaptive Microservice Scaling for Elastic Applications." In: *IEEE Internet of Things Journal* preprint (2020), pp. 1–1. ISSN: 72-2541. DOI: 10.1109/JIOT.2020.2964405.

[12] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. "InferLine: Latency-Aware Provisioning and Scaling for Prediction Serving Pipelines." In: *ACM Symposium on Cloud Computing (SoCC)*. 2020, pp. 477–491. DOI: 10.1145/3419111.3421285.

[13] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. "Clipper: A {Low-Latency} Online Prediction Serving System." In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2017, pp. 613–627.

[14] Tathagata Das, Yuan Zhong, Ion Stoica, and Scott Shenker. "Adaptive stream processing using dynamic batch sizing." In: *ACM Symposium on Cloud Computing (SoCC)*. 2014, pp. 1–13.

[15] Aditya Dhakal, Sameer G. Kulkarni, and K. K. Ramakrishnan. "GSLICE: Controlled Spatial Sharing of GPUs for a Scalable Inference Platform." In: *ACM Symposium on Cloud Computing (SoCC)*. 2020, pp. 492–506. DOI: 10.1145/3419111.3421284.

[16] *Docker Inc*. https://www.docker.com. Accessed on 29.10.2021.

[17] Alex Ellis. *OpenFaaS*. https://www.openfaas.com. Accessed on 29.10.2021. 2019.

[18] Maria Fazio, Antonio Celesti, Rajiv Ranjan, Chang Liu, Lydia Chen, and Massimo Villari. "Open issues in scheduling microservices in the cloud." In: *IEEE Cloud Computing* 3.5 (2016), pp. 81–88.

[19] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter R. Pietzuch. "Integrating scale out and fault tolerance in stream processing using operator state management." In: *ACM International Conference on Management of Data (SIGMOD)*. 2013.

[20] Martin A. Fischler and Robert C. Bolles. "Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography." In: *Communications of the ACM* 24.6 (1981), pp. 381–395.

[21] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. "Dhalion: self-regulating stream processing in heron." In: *Very Large Data Bases (PVLDB)* 10.12 (2017), pp. 1825–1836.

[22] Luca Florio and Elisabetta Di Nitto. "Gru: An Approach to Introduce Decentralized Autonomic Behavior in Microservices Architectures." In: *IEEE International Conference on Autonomic Computing (ICAC)*. 2016.

[23] The Linux Foundation. *Kubernetes*. https://kubernetes.io. Accessed on 29.10.2021. 2019.

[24] Sören Frey, Florian Fittkau, and Wilhelm Hasselbring. "Search-based genetic optimization for deployment and reconfiguration of software in the cloud." In: *IEEE International Conference on Software Engineering (ICSE)*. 2013.

[25] Tom Z. J. Fu, Jianbing Ding, Richard T. B. Ma, Marianne Winslett, Yin Yang, and Zhenjie Zhang. "DRS: Auto-Scaling for Real-Time Stream Analytics." In: *IEEE/ACM Transactions on Networking* 25.6 (2017), pp. 3338–3352.

[26] Xinwei Fu, Talha Ghaffar, James C. Davis, and Dongyoon Lee. "EdgeWise: A Better Stream Processing Engine for the Edge." In: *USENIX Annual Technical Conference (ATC)*. 2019, pp. 929–946.

[27] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. "Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices." In: *ACM International Conference on*

Architectural Support for Programming Languages and Operating Systems (ASPLOS). 2019, pp. 19–33.

[28] Anshul Gandhi, Parijat Dube, Alexei Karve, Andrzej Kochut, and Li Zhang. "Adaptive, model-driven autoscaling for cloud applications." In: *International Conference on Autonomic Computing (ICAC)*. 2014, pp. 57–64.

[29] Bugra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. "Elastic Scaling for Data Stream Processing." In: *IEEE Transactions on Parallel and Distributed Systems* 25.6 (2014), pp. 1447–1463.

[30] Alim Ul Gias, Giuliano Casale, and Murray Woodside. "ATOM: Model-driven autoscaling for microservices." In: *IEEE International Conference on Distributed Computing Systems (ICDCS)*. 2019, pp. 1994–2004.

[31] Google. *Google Cloud Functions*. https://cloud.google.com/functions. Accessed on 29.10.2021. 2019.

[32] Google. *CloudRun*. https://cloud.google.com/run. Accessed on 29.10.2021. 2021.

[33] Manuel Gotin, Felix Lösch, Robert Heinrich, and Ralf Reussner. "Investigating performance metrics for scaling microservices in cloudiot-environments." In: *ACM/SPEC International Conference on Performance Engineering (ICPE)*. 2018, pp. 157–167.

[34] *gRPC*. https://grpc.io. Accessed on 29.10.2021.

[35] Arpan Gujarati, Sameh Elnikety, Yuxiong He, Kathryn S McKinley, and Björn B Brandenburg. "Swayam: distributed autoscaling to meet slas of machine learning inference services with resource efficiency." In: *ACM/IFIP/USENIX Middleware Conference*. 2017, pp. 109–120.

[36] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. "Serving DNNs like Clockwork: Performance Predictability from the Bottom Up." In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2020, pp. 443–462.

[37] Vincenzo Gulisano, Ricardo Jiménez-Peris, Marta Patiño-Martínez, Claudio Soriente, and Patrick Valduriez. "StreamCloud: An Elastic and Scalable Data Streaming System." In: *IEEE Transactions on Parallel and Distributed Systems* 23.12 (2012), pp. 2351–2365.

[38] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Mahmut Taylan Kandemir, Bhuvan Urgaonkar, George Kesidis, and Chita Das. "Spock: Exploiting Serverless Functions for SLO and Cost Aware Resource Procurement in Public Cloud." In: *IEEE International Conference on Cloud Computing (CLOUD)*. 2019, pp. 199–208. DOI: 10.1109/CLOUD.2019.00043.

[39] *Gurobi*. https://www.gurobi.com. Accessed on 29.10.2021.

[40] Joseph L. Hellerstein, Yixin Diao, Sujay S. Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems*. Wiley, 2004. ISBN: 978-0-47126637-2.

[41] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. "Serverless Computation with OpenLambda." In: *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*. 2016.

[42] Moritz Hoffmann, Andrea Lattuada, Frank McSherry, Vasiliki Kalavri, John Liagouris, and Timothy Roscoe. "Megaphone: Latency-conscious state migration for distributed streaming dataflows." In: *Very Large Data Bases (PVLDB)* 12.9 (2019), pp. 1002–1015.

[43] IBM. *IBM OpenWhisk*. https://developer.ibm.com/open/projects/openwhisk. Accessed on 29.10.2021. 2019.

[44] Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. "Performance modeling for cloud microservice applications." In: *ACM/SPEC International Conference on Performance Engineering (ICPE)*. 2019, pp. 25–32.

[45] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. "Occupy the Cloud: Distributed Computing for the 99%." In: *ACM Symposium on Cloud Computing (SoCC)*. 2017, pp. 445–451.

[46] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. "Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows." In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2018, pp. 783–798.

[47] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. “GrandSLAm: Guaranteeing SLAs for Jobs in Microservices Execution Frameworks.” In: *ACM European Conference on Computer Systems (EuroSys)*. 2019, pp. 1–16. ISBN: 978-1-4503-6281-8. DOI: 10.1145/3302424.3303958.

[48] Nima Kaviani, Dmitriy Kalinin, and Michael Maximilien. “Towards Serverless as Commodity: a case of Knative.” In: *International Workshop on Serverless Computing*. 2019, pp. 13–18.

[49] Hamzeh Khazaei, Rajsimman Ravichandiran, Byungchul Park, Hadi Bannazadeh, Ali Tizghadam, and Alberto Leon-Garcia. “Elascale: autoscaling and monitoring as a service.” In: *ACM Annual International Conference on Computer Science and Software Engineering (CASCON)*. 2017.

[50] Kubeless. *Kubeless*. https://kubeless.io. Accessed on 29.10.2021. 2019.

[51] Luyang Liu, Hongyu Li, and Marco Gruteser. “Edge Assisted Real-Time Object Detection for Mobile Augmented Reality.” In: *ACM Annual International Conference On Mobile Computing And Networking (MobiCom)*. 2019, pp. 1–16. ISBN: 978-1-4503-6169-9. DOI: 10.1145/3300061.3300116.

[52] Björn Lohrmann, Peter Janacik, and Odej Kao. “Elastic stream processing with latency guarantees.” In: *IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2015, pp. 399–410.

[53] Nima Mahmoudi and Hamzeh Khazaei. “Performance modeling of serverless computing platforms.” In: *IEEE Transactions on Cloud Computing* (2020), pp. 1–15.

[54] Luo Mai, Kai Zeng, Rahul Potharaju, Le Xu, Steve Suh, Shivaram Venkataraman, Paolo Costa, Terry Kim, Saravanam Muthukrishnan, Vamsi Kuppa, Sudheer Dhulipalla, and Sriram Rao. “Chi: A Scalable and Programmable Control Plane for Distributed Stream Processing Systems.” In: *Very Large Data Bases (PVLDB)* 11.10 (2018), pp. 1303–1316.

[55] Gabriele Mencagli, Patrizio Dazzi, and Nicolò Tonci. “Spinstreams: a static optimization tool for data stream processing applications.” In: *Proceedings of the 19th International Middleware Conference*. 2018, pp. 66–79.

[56] Microsoft. *Microsoft Azure Functions*. https://azure.microsoft.com/en-us/services/functions. Accessed on 29.10.2021. 2019.

[57] Vinod Nigade, Lin Wang, and Henri Bal. "Clownfish: Edge and cloud symbiosis for video stream analytics." In: *IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE. 2020, pp. 55–69.

[58] Balaji Palanisamy, Aameek Singh, and Ling Liu. "Cost-effective resource provisioning for mapreduce in a cloud." In: *IEEE Transactions on Parallel and Distributed Systems* 26.5 (2014), pp. 1265–1279.

[59] Kamran Razavi and Lin Wang. *FA2 Source Code*. Accessed on 3.3.2022.

[60] Jerome A. Rolia and Kenneth C. Sevcik. "The Method of Layers." In: *IEEE Trans. Software Eng.* 21.8 (1995), pp. 689–700.

[61] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. "INFaaS: Automated Model-less Inference Serving." In: *USENIX Annual Technical Conference (ATC)*. 2021, pp. 397–411.

[62] Francisco Romero, Mark Zhao, Neeraja J Yadwadkar, and Christos Kozyrakis. "Llama: A Heterogeneous & Serverless Framework for Auto-Tuning Video Analytics Pipelines." In: *ACM Symposium on Cloud Computing (SoCC)*. 2021, pp. 1–17.

[63] Amazon Web Services. *AWS Lambda*. https://aws.amazon.com/lambda. Accessed on 29.10.2021. 2019.

[64] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider." In: *USENIX Annual Technical Conference (ATC)*. 2020, pp. 205–218. ISBN: 978-1-939133-14-4.

[65] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. "Nexus: a GPU cluster engine for accelerating DNN-based video analysis." In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*. Ed. by Tim Brecht and Carey Williamson. ACM, 2019, pp. 322–337. DOI: 10.1145/3341301.3359658.

[66] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. "Cloudburst: Stateful Functions-as-a-Service." In: *Very Large Data Bases (PVLDB)* 13.12 (July 2020), pp. 2438–2452. ISSN: 2150-8097. DOI: 10.14778/3407790.3407836.

[67] Akshitha Sriraman and Thomas F. Wenisch. "¯Tune: Auto-Tuned Threading for OLDI Microservices." In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2018.

[68] Giovanni Toffetti, Sandro Brunner, Martin Blöchlinger, Josef Spillner, and Thomas Michael Bohnert. "Self-managing cloud-native applications: Design, implementation, and experience." In: *Future Generation Computer Systems* 72 (2017), pp. 165–179.

[69] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J. Franklin, Benjamin Recht, and Ion Stoica. "Drizzle: Fast and Adaptable Stream Processing at Scale." In: *ACM Symposium on Operating Systems Principles (SOSP)*. 2017.

[70] Chad Verbowski, Ed Thayer, Paolo Costa, Hugh Leather, and Björn Franke. "Right-Sizing Server Capacity Headroom for Global Online Services." In: *IEEE International Conference on Distributed Computing Systems (ICDCS)*. 2018, pp. 645–659.

[71] Xuezhi Zeng, Saurabh Garg, Zhenyu Wen, Peter Strazdins, Lizhe Wang, and Rajiv Ranjan. "Sla-aware scheduling of map-reduce applications on public clouds." In: *IEEE International Conference on High Performance Computing and Communications (HPCC)*. IEEE. 2016, pp. 655–662.

[72] Tianlei Zheng, Xi Zheng, Yuqun Zhang, Yao Deng, ErXi Dong, Rui Zhang, and Xiao Liu. "SmartVM: a SLA-aware microservice deployment framework." In: *World Wide Web* 22.1 (2019), pp. 275–293.

[73] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. "Overload control for scaling wechat microservices." In: *ACM Symposium on Cloud Computing (SoCC)*. 2018, pp. 149–161.

P2

SPONGE: INFERENCE SERVING WITH DYNAMIC SLOS USING IN-PLACE VERTICAL SCALING

ABSTRACT

Mobile and IoT applications increasingly adopt deep learning inference to provide intelligence. Inference requests are typically sent to a cloud infrastructure over a wireless network that is highly variable, leading to the challenge of dynamic Service Level Objectives (SLOs) at the request level.

This paper presents Sponge, a novel deep learning inference serving system that maximizes resource efficiency while guaranteeing dynamic SLOs. Sponge achieves its goal by applying in-place vertical scaling, dynamic batching, and request reordering. Specifically, we introduce an Integer Programming formulation to capture the resource allocation problem, providing a mathematical model of the relationship between latency, batch size, and resources. We demonstrate the potential of Sponge through a prototype implementation and preliminary experiments and discuss future works.

P2.1 INTRODUCTION

Within the domain of mobile and IoT applications, cloud-based Deep Learning (DL) inference plays an important role, with user satisfaction and resource efficiency serving as key performance indicators [1, 26, 27]. Since most DL-powered applications involve user interaction, they must comply with strict requirements on the inference latency, a.k.a. meeting the Service Level Objectives (SLOs) of the inference request.

On the other hand, the resources needed to provision such a DL inference serving system should be minimized to reduce the cost [10, 17, 18, 24, 29, 32].

SLOs are comprehensively defined from end to end, with the variable network time required for transferring user requests and input data introducing dynamic time budgets for serving inference requests. Therefore, when setting expectations for mobile and IoT applications, it is important to define SLOs that cover both the network and computing aspects from start to finish. Ignoring the time it takes for information to travel through the network, inference serving systems may find themselves with not enough time to handle requests properly, resulting in SLO violation. Hence, resource allocation must consider a variety of time budgets of a single user using the same application. Managing this dynamism poses a critical challenge for inference serving systems, where the effective handling of diverse SLOs and the consideration of fluctuating network conditions are imperative to ensure the fulfillment of end-to-end SLOs.

Existing inference serving systems mostly consider only the inference part with static SLOs, i.e., all requests have the same SLOs when they reach computing units. Their horizontal scaling-based approach cannot incorporate diverse SLOs at the request level [10, 12, 18]. For example, FA2 [29] adjusts the number of minimum-resource instances to achieve the highest resource efficiency (throughput). Moreover, bringing new instances in horizontal scaling ties with the cold-start issue (a few seconds [15, 30]), which cannot cope with the dynamically changing network conditions. Jellyfish [28], on the other hand, aims to guarantee end-to-end SLOs while achieving high inference accuracy by using pre-loaded model-switching and trading accuracy for latency, which may not always be possible for all applications.

We propose a new system, Sponge, aiming to address this research gap. Our main insight is that the combination of in-place vertical scaling, dynamic batching, and request reordering is a powerful tool to combat request-level dynamism. In particular, the new in-place vertical scaling feature of Kubernetes [3] allows developers to resize CPU/memory resources allocated to containers without restarting them, eliminating the cold-start issues of vertical scaling, while request reordering allows for requests with a lower remaining time budget to be processed earlier. At the same time, dynamic batching increases the system utilization to further reduce the needed comput-

ing resources. We formulate the problem and propose a method for inference serving with dynamic SLOs. Sponge relies on three adaptation strategies to capture per-request dynamic SLOs: ❶ in-place vertical scaling to change the computing resources of DL models in spot, ❷ request reordering to prioritize close-to-deadline requests, and ❸ dynamic batching to increase the utilization of the DL models. More specifically, Sponge achieves dynamic SLOs guarantee and high resource utilization by first providing a mathematical relation between vertical scaling with batching and processing latency of the DL model using historical data and then designing a request-based mathematical modeling of the entire framework to guarantee SLOs of all requests while minimizing the resources. Furthermore, we propose a simple algorithm for small cases to iterate over all possible configurations and find optimal resource and batch size allocations. The preliminary experimental results show a reduction in over 15× of the SLO violation compared to the existing approaches.

Sponge currently does not consider pipelines of DL models. Complex applications such as intelligent virtual assistants consist of multiple DL models, coordinated with a Directed Acyclic Graph (DAG), collaboratively generating a meaningful output. Such applications require a more intricate solution due to data dependencies among DL models, resulting in a strong coupling of scaling decisions for different DL models. Furthermore, vertical scaling sustains workloads to some extent due to the DL model parallelization level and availability of computing resources in a sine node. Therefore, multiple instances of the same DL model (horizontal scaling) may need to reside in different computing nodes to support the incoming workload. We consider these directions as future works of Sponge.

This paper contributes by discussing the challenges of dynamic SLOs on DL inference serving systems. Then, we

- present the design of Sponge, a new DL inference serving system for dynamic SLOs based on the idea of in-place vertical scaling, request reordering, and dynamic batching.

- provide an Integer Programming formulation to encapsulate the problem of dynamic SLOs by introducing a mathematical modeling of the relation between latency, batch, and CPU in inference serving systems.

- build a prototype system for Sponge [1] and evaluate it using 4G/LTE bandwidth logs datasets. Sponge reduces the SLO violation by over 15× compared to a horizontal state-of-the-art autoscaler.

P2.2 MOTIVATION

In this section, we first discuss the challenges raised by variable networks and then identify the challenges in efficient in-place vertical scaling.

P2.2.1 *Dynamic SLO*

Fluctuations in network bandwidths, e.g., caused by user mobility, are inevitable [8, 14], as illustrated in Figure P2.1 (top). This variability influences the transmission overhead associated with sending data across the network for remote processing, leading to a reduction in the time budget available for server-side deployed services, as depicted in Figure P2.1 (bottom). Consequently, service providers are compelled to account for network latency to ensure compliance with the end-to-end latency requirements specified in the SLO.

We use a simple human detection model trained on the ResNet architecture to motivate this work, where it detects a human in an image of 200 KB while the requests are being sent on a dynamically changing network (e.g., 4G) under a static workload of 100 requests per second with the SLO of 1000 ms (similar conditions to Figure P2.1). Table P2.1 shows the execution latency of the model with different allocated CPU cores and batch sizes when considering the SLO. For calculating the required number of instances per instance type (different core numbers), we divide the incoming workload by the throughput of an instance. Following the approach in FA2 [29], where they use one-core instances, we need five instances to process a batch of 2 requests per 97 ms, which means that we can process a batch of 10 requests, or 20 requests per second (RPS), over a 1000 ms SLO. This approach works perfectly if the network is static. However, if the network latency takes up to half of the SLO, FA2 will drop all the requests, as there is no

1 https://github.com/saeid93/sponge

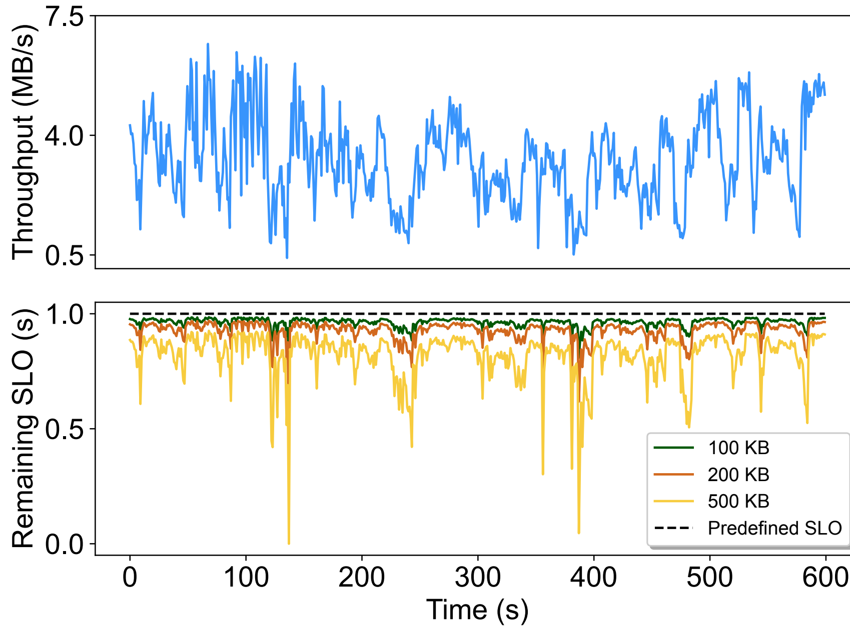


Figure P2.1: Bandwidth measurements in 4G networks provided by [20]. The bandwidth varies from 0.5MB/s to 7MB/s in a 10-minute range (top figure). The below figure demonstrates the remaining SLO for processing when the user sends a 100 KB, a 200 KB, or a 500 KB image over the same network's bandwidth.

possible solution in their approach with one-core instances, even with the smallest batch size. Furthermore, even if the network latency takes just 40 ms, the system needs to bring up a new instance to avoid dropping requests or violating the SLO, meaning that the system will suffer from the cold start of a new instance until the system stabilizes again. Alternatively, meeting the SLO in the context of a dynamically changing network bandwidth could have been achieved through the dynamic modification of computing resources within the instance (in-place vertical scaling). In the same scenario, if we had up to 600 ms of network delay, we could still serve the requests without violating or dropping any request by changing the instance core from 1 core to 8 cores with a batch size of 4. InfAdapter [32] employs profiling data to determine CPU core allocation for DL models. For instance, under a workload of 100 RPS, the model's computing resources and batch size remain static. However, when faced with changes in the SLO, it switches to a different model variant with predefined CPU core allocation, encountering similar challenges as FA2 (cold start and static CPU core allocation).

Cores	Batch	Latency (ms)	Throughput (RPS)	Total Cores
1	1	55	$18 \times 6 = 108$	$1 \times 6 = 6$
1	2	97	$20 \times 5 = 100$	$1 \times 5 = 5$
2	4	94	$40 \times 3 = 120$	$2 \times 3 = 6$
4	8	92	$80 \times 2 = 160$	$2 \times 4 = 8$
8	4	37	$108 \times 1 = 108$	$1 \times 8 = 8$
8	8	62	$128 \times 1 = 128$	$1 \times 8 = 8$

Table P2.1: Execution latency (P99) of a ResNet model (human detector) with different CPU cores using different batch sizes while guaranteeing SLO of 1000 ms under the workload of 100 RPS.

P2.2.2 *Autoscaling Challenges*

Creating an effective in-place vertical scaling system for DL inference serving is a complex task. Precisely, we pinpoint the following challenges, which collectively differentiate the scaling problem in DL inference serving systems from those examined in other systems.

Dynamic SLO at the request level. In wireless networks conditions can change over time. This can be due to various factors, such as changes in network traffic, hardware performance, signal strength, and resource availability [23, 36]. These factors can cause variable delays in network transmission for inference requests, leading to requests with dynamic SLOs. Accommodating dynamic SLOs at the request level requires fine-grained control over resource allocation to ensure each request meets its SLO. This level of granularity is challenging to achieve with vertical scaling since changing the resources to guarantee one request SLO affects all the requests' processing latency in the system.

Batch size. DL inference serving systems commonly utilize request batching to enhance resource efficiency [9–11, 33]. More precisely, batching can increase throughput as more tasks or requests can be processed in a given amount of time. Furthermore, batching can help meet latency constraints with dynamic batching policies, where batch sizes are determined online, during runtime, depending on the latency constraints of each application [24, 29]. However, it is important to note that large and small batch sizes can have drawbacks if not properly managed. Large batch sizes can critically violate the latency of many requests within a batch, while small batch sizes could cause excessive queuing and may not exploit potential opportunities for increased throughput.

In the next section, we provide an in-place vertical-based autoscaler to capture the discussed challenges by first discussing how to reconcile vertical scaling and batch size in the context of inference serving systems, and second, providing a mathematical formulation to mimic the autoscaling problem with the consideration of dynamic SLOs.

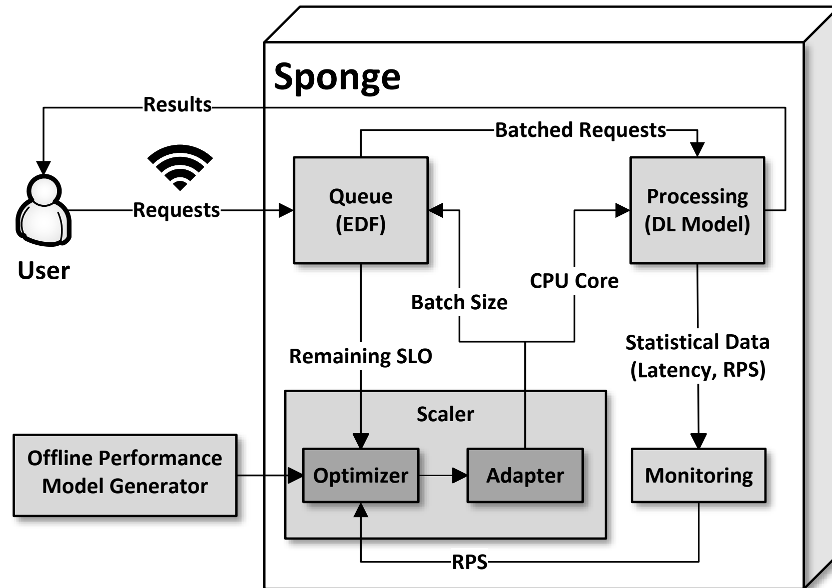


Figure P2.2: An overview of the Sponge architecture. The monitoring service collects metric data from the DL model. The queue prioritizes requests according to the EDF policy. The scaler is responsible for determining vertical scaling and batch size decisions for the DL model and adjusting the system accordingly.

P2.3 SYSTEM DESIGN

This section provides our solution for inference serving systems with dynamic SLOs. Our goal is to use minimal resources to provision the DL model with in-place vertical scaling, request reordering, and dynamic batch sizing while guaranteeing all the requests' SLO.

P2.3.1 *Overview*

Sponge consists of four components as is shown in Figure P2.2:

Monitoring. The monitoring component uses Prometheus [7] to observe the incoming workload to the system. It will monitor the workload destined for the model on a predefined time interval. Additionally, it receives the end-to-end request latency from the processing component to calculate the SLO violation rate and the accuracy of the performance model.

Queuing. The queuing component receives the request from the user, reorders the request based on the remaining SLO (Earliest Deadline First (EDF)), and creates a batch with the given batch size from the solver. In addition, it sends the set of requests with their communication latency to the optimizer.

Processing. The processing component has the computing power to execute inferences. It receives batches from the queue, processes them, and sends them to the user. Furthermore, it sends the statistical data (queuing latency and processing latency) to the monitoring component.

Scaler. The scaler component first aims to find the vertical scaling CPU cores and batch size decisions to achieve the highest resource efficiency while respecting all the request SLOs in the system by using the workload (reported by the monitoring component) and the remaining SLOs of all the requests after being reordered by the queuing component in the optimizer. Next, its adapter part adjusts the system by sending a signal to the processing component with the new CPU core allocation and a signal to the queueing component with the new batch size configuration.

P2.3.2 *Performance Model*

For effective decision-making within the solver, Sponge needs knowledge of the performance metrics, specifically the throughput $h(b,c)$ and latency $d(b,c)$, associated with the DL model. Previous research has indicated that the performance of DL inference tends to be highly predictable [11, 18, 24, 35]. We follow the same line and use profiling data and robust regressions [13] to build a model for any given DL model. GrandSLAm [11, 24] suggests a linear relationship between batch size and latency, that is, $l(b,c) = \alpha_1 \times b + \beta_1$, and FA2 [29] suggests a second-order quadratic polynomial for a lower total MSE. However, none of the above works consider changes in the computational resources (e.g., number of CPU cores) of the DL models. For simplicity, we use the linear relation in the current work.

To have a relation between latency and CPU, we use Amdahl's law [2] for latency prediction under a given batch size:

$$L(b,c) = \frac{\alpha_2}{c} + \beta_2 \quad \text{(P2.1)}$$

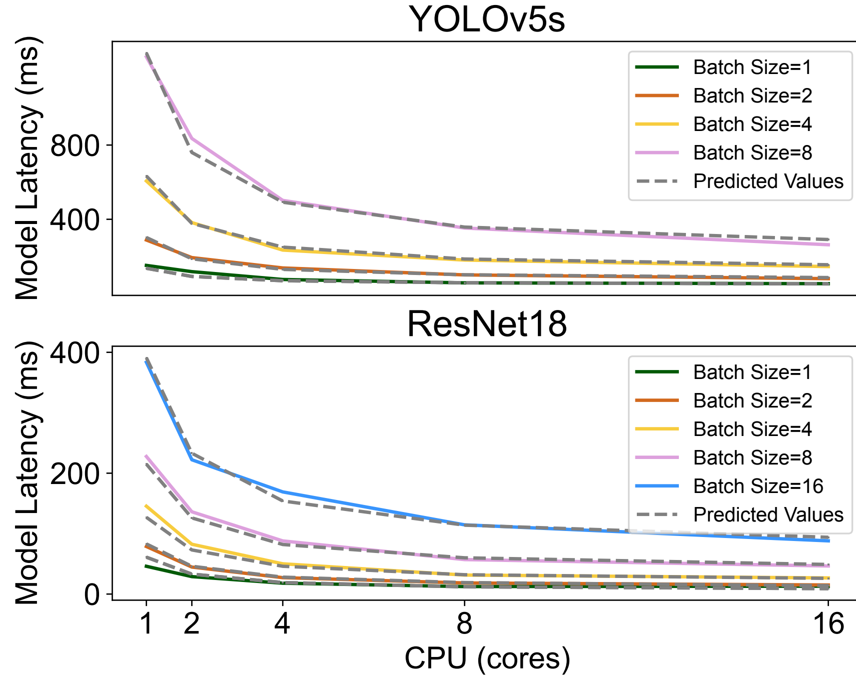


Figure P2.3: Latency vs. different CPU core allocations and batch sizes using real and predicted for the YOLOv5n and ResNet18 DL models.

Equation P2.1 states an inverse relation between the number of CPU cores and latency if the model can use additional CPU cores, which is the case in ML models.

On the other hand, the linear relation of batch size and latency suggests that α_1 and β_1 have inverse relations with CPU cores, e.g., $\alpha_1 = \gamma_1/c + \delta_1$ and $\beta_1 = \epsilon_1/c + \eta_1$ (otherwise, $l(b,c)$ would become linear in Equation P2.1). Therefore, to incorporate computational resources into batch/latency profiling, we combine the linear relation of batch/latency and the inverse relation of CPU/latency as follows.

$$
\begin{aligned}
l(b,c) &= (\frac{\gamma_1}{c} + \delta_1) \times b + \frac{\epsilon_1}{c} + \eta_1 \\
&= \frac{\gamma_1 \times b}{c} + \frac{\epsilon_1}{c} + \delta_1 \times b + \eta_1
\end{aligned}
\tag{P2.2}
$$

Our preliminary evaluation with the data sets profiled from ResNet18 and YOLOv5n models used in Figure P2.3 confirms that the latency/CPU/batch model in Equation P2.2 provides a realistic estimation of latency with different CPU cores and batch sizes on different DL models. The throughput of a DL model is directly given as a function of batch size and CPU cores, e.g., $h(b,c) = b/l(b,c)$.

Table P2.2: Notations

Symbol	Description
R	Set of all requests
b	Model's batch size
c	Model's CPU allocation
cl_r	Communication latency associated with $r \in R$
cl_{max}	Highest cl_r in R
SLO	Pre-defined SLO for R
$l(b,c)$	Processing time of a model with allocation core c and batch size b
$q_r(b,c)$	Queuing time of $r \in R$ with allocation core c and batch size b
$h(b,c)$	Throughput of a model with allocation core c and batch size b
λ	Request arrival rate

P2.3.3 *Problem Formulation*

The optimizer generates scaling decisions by solving an optimization problem. Now, we provide a formal formulation for the problem given that the end-to-end latency for a request is the aggregation of the communication latency (the time the request takes to be received by the system from the user device), the queuing (the time the request spends in the queue before being processed), and the processing latencies (inference latency) of the request.

Suppose that we are given a model and a set of requests R with a predefined SLO. Each request $r \in R$ has communication latency cl_r. The arrival rate of the application request is denoted by λ. Due to the instability of the network, as we have already discussed in Section P2.2, we apply the earliest-deadline-first (EDF) queue ($q(b,c)$), similar to GradnSLAm [24], since request reordering prioritizes the processing of requests with lower remaining SLOs due to their more stringent completion deadlines.

Let us denote the number of CPU cores allocated and the batch size of the model by c and b, respectively. In addition, we use $cl_{max} = max(cl_r, r \in R)$ to indicate the highest communication latency in the current requests.

The monitoring system continuously reports to the adapter the average number of requests served by the model in a given period. To ensure the stability of the system, that is, no back pressure should form in the queue, and the throughput of the model should be no less than the expected request rate, that is, $h(b,c) \geq \lambda$. Such a constraint ensures that the model is sufficiently provisioned. As a result, the queuing of requests on the model will be under control.

The optimization problem is to decide c and b for the model such that under the workload λ, none of the request SLOs are violated. The goal is to minimize the amount of resources (CPU cores) used for the model. The problem can be formulated with the following integer program (IP):

$$\begin{aligned} \text{Minimize} \quad & c + \delta \times b \\ \text{subject to} \quad & l(b,c) + q_r(b,c) + \text{cl}_{max} \leq SLO, \quad \forall r \in R \\ & h(b,c) \geq \lambda \\ & b, c \in \mathbb{Z}^+ \end{aligned} \tag{P2.3}$$

In the objective function, in addition to CPU cores, we incorporate an insignificant penalty term δ into the batch size to mitigate unnecessary latencies. The first constraint ensures that all requests for SLOs, including communication latency, will be satisfied. We use the smallest SLO in the current batch for all requests in the same batch because we do not intend to violate any remaining SLO requests. The second and third constraints are designed to maintain system stability, necessitating that the CPU cores and the batch size be constrained to positive integer values. The objective is to minimize the total amount of resources, that is, the total number of CPU cores given to the model. All the notation used is available in Table P2.2.

P2.3.4 *Solution*

With IP and a single model, we use a brute force approach shown in Algorithm 2. We feed the requests with their remaining SLOs to a queue and then reorder them based on the EDF policy (lines 1–2). After finding the maximum communication latency in the set of requests (line 4), we then iterate over all possible batch sizes and CPU core allocations (lines 5–6). Furthermore, we check if the current configuration and all the requests in the subsequent batches will satisfy

Algorithm 2: Optimal CPU and batch size finder

```
   input  : SLO, Set of requests r ∈ R with communication latency,
            Performance model
   output : c, b
1  q ← R
2  Reorder q (EDF policy)
3  n = len(R)
4  Calculate cl_max
5  for c in [1, c_max] do
6      for b in [1, b_max] do
7          Calculate l(b, c)
8          better = True
9          q_r = 0
10         for i in [1, n, b] do
11             if l(b, c) + cl_max + q_r ≥ SLO then
12                 better = False
13                 break
14             q_r = q_r + l(b, c)
15         if better = True then
16             return c, b
```

their remaining SLOs (lines 10–15). Note that there will be a waiting time for the subsequent batches equal to the processing latency of the previous batches, calculated in line 14. Finally, if there is no objection against the current batch size and CPU core allocation configurations (line 15), we send the found configuration to be enforced to the system. The algorithm generates the optimal CPU core allocation with the smaller batch size with the current allocation, since it iterates from 1 to the maximum CPU core and batch size allocations.

P2.4 PRELIMINARY EVALUATION

Sponge is implemented in 6K lines of Python. For evaluation, we use a physical machine from Chameleon Cloud [25] equipped with Intel(R) Xeon(R) Gold 6240R (48 threads). To enable the in-place vertical scaling, we install the experimental branch of minikube [6] since the in-place vertical scaling feature is not yet in the official releases [3].

Baseline. We compare Sponge with a state-of-the-art horizontal autoscaler in inference serving systems, FA2, and static 8-core and 16-core instances. All approaches (including Sponge) use a YOLOv5s [34] with

the performance modeling in Figure P2.3 to detect humans in images. We also set b_{max} and c_{max} to 16 for Sponge as there is no significant gain afterward. For the adaptation period, we set one second same as the network bandwidth interval in the dataset.

Workload generator. In order to assess Sponge in scenarios with dynamic network bandwidth, we design a workload generator that produces requests asynchronously at a fixed rate of 20 RPS with predefined SLOs similar to Figure P2.1. We use gRPC [16] to handle communication between all components of the system, including the workload generator.

Performance evaluation. Figure P2.4 demonstrates the overall performance of Sponge, FA2, and statically assigned CPU cores under a dynamic network bandwidth. Under a given workload and the remaining SLOs, FA2 violates a large number of requests' SLO (roughly 5% and over 50% violation in some severe cases (Time = 1 and 360 in the same Figure) when the bandwidth becomes limited since bringing new instances is tied with the cold startup issue, and FA2 needs roughly 10 seconds to find a new configuration, adjust itself, and stabilize the system. The statically assigned 8-core instance experiences SLO violations after a few seconds due to insufficient computational resources to handle the requests, necessitating a more powerful instance. Conversely, the 16-core instance shows almost no SLO violations, indicating potential over-provisioning of the DL model. Sponge solves the resource waste by dynamically changing the allocated CPU cores in response to the network bandwidth changes and reduces the amount of allocated CPU by over 20% while sacrificing less that 0.3% of SLO violations, compared to statically assigned 16-core instance.

P2.5 RELATED WORK

Inference serving with SLO guarantee. Multiple works have been proposed with SLO guarantees [12, 18, 32, 33]. Model switch [38] switches to a different model architecture in response to workload changes to ensure SLO. GrandSLAm [24] uses dynamic batching and request reordering to increase system throughput with the SLO guarantee. InFaaS [30] gets user preferences about accuracy, cost, or performance and provides a model variant to satisfy the requested SLO. Jellyfish [28] trades accuracy with latency by model switching and

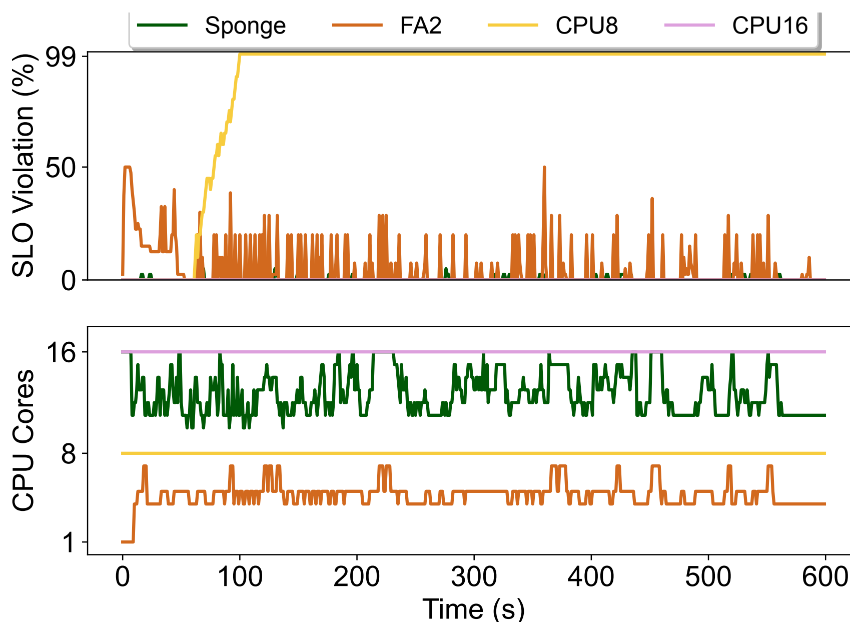


Figure P2.4: SLO violations and allocated CPU cores.

data adaptation to match the input of the model variant to guarantee latency SLO.

Autoscaling in inference serving. Autoscaling in inference serving has been extensively studied [10, 21, 31, 37]. Kubernetes VPA [5] and HPA [4] use threshold-based metrics such as CPU or memory usage to change computing resources or the number of instances of DL-based inference services. Clipper [11] provides an abstraction layer to simplify model deployment across frameworks and uses adaptive batching to increase system throughput. IPA [15] uses model switching and horizontal scaling to increase system accuracy while minimizing computing resources. Cocktail [19] uses a subset of model variants with a weighted scaling policy to ensure low cost, a predefined accuracy, and latency SLOs archived. FA2 [29] uses graph transformation and dynamic programming to design a new horizontal autoscaler to increase system utilization with SLO guarantees.

The mentioned approaches neither consider dynamic networks (wireless and 4G/5G) without changing the model variant that affects other metrics such as cost and accuracy nor use in-place vertical scaling, which Sponge has shown a necessity for state-of-the-art autoscalers to guarantee predefined latency SLO under a dynamic network bandwidth.

P2.6 CONCLUSION & FUTURE WORK

In this work, we presented Sponge, the first inference serving system that uses in-place vertical scaling, request reordering, and dynamic batching with SLO guarantees. The preliminary evaluation shows that Sponge reduces the SLO violation to 0.3% while minimizing the CPU allocation in a dynamic network. We identify the following limitations of Sponge and consider them as future directions:

Model variant. There are variations of the same DL model with different configurations in terms of architecture that are capable of doing similar tasks with different objectives such as accuracy [28, 30, 38]. Incorporating model variants requires careful system design, since the three pillars of accuracy, latency, and CPU allocation (even without vertical scaling) have conflicting relations [32].

Pipeline. Many modern applications are composed of multiple DL models, such as Amazon Alexa, and are usually arranged as a DAG. Generalizing Sponge to support such applications requires a new algorithm design, since there is a data dependency [10, 15, 22, 29] between DL models and finding an optimal resource allocation for individual DL models requires consideration of all models in the system.

Multidimensional scaling. The resource requirements of a DL model can be influenced by the dynamic nature of workloads [17, 37], making them difficult to predict. Vertical scaling can support the incoming workload to a certain degree, meaning that horizontal scaling must be considered if the workload is too much for a single instance of a DL model. The joint optimization of horizontal scaling and vertical scaling mechanisms brings new challenges, such as changing an upstream DL model's processing latency rate (vertical scaling), which affects the input rates on downstream DL models and may require additional instances (horizontal scaling).

P2.7 ACKNOWLEDGMENTS

This work has been supported in part by NSF (Awards 2233873, 2007202, 2038080, and 2107463), Deutsche Forschungsgemeinschaft

(DFG, German Research Foundation) - Project-ID 210487104 - SFB 1053, Roblox Corporation, and Chameleon Cloud.

P3

IPA: INFERENCE PIPELINE ADAPTATION TO ACHIEVE HIGH ACCURACY AND COST-EFFICIENCY

ABSTRACT

Efficiently optimizing multi-model inference pipelines for fast, accurate, and cost-effective inference is a crucial challenge in machine learning production systems, given their tight end-to-end latency requirements. To simplify the exploration of the vast and intricate trade-off space of latency, accuracy, and cost in inference pipelines, providers frequently opt to consider one of them. However, the challenge lies in reconciling latency, accuracy, and cost trade-offs. To address this challenge and propose a solution to efficiently manage model variants in inference pipelines, we present IPA, an online deep learning **I**nference **P**ipeline **A**daptation system that efficiently leverages model variants for each deep learning task. Model variants are different versions of pre-trained models for the same deep learning task with variations in resource requirements, latency, and accuracy. IPA dynamically configures batch size, replication, and model variants to optimize accuracy, minimize costs, and meet user-defined latency Service Level Agreements (SLAs) using Integer Programming. It supports multi-objective settings for achieving different trade-offs between accuracy and cost objectives while remaining adaptable to varying workloads and dynamic traffic patterns. Navigating a wider variety of configurations allows IPA to achieve better trade-offs between cost and accuracy objectives compared to existing methods. Extensive experiments in a Kubernetes implementation with five real-world inference

pipelines demonstrate that IPA improves end-to-end accuracy by up to 21% with a minimal cost increase.

P3.1 INTRODUCTION

Nowadays, companies run some or all of their Machine Learning (ML) pipelines on cloud computing platforms [70]. The efficient deployment of ML models is crucial in contemporary systems where ML inference services consume more than 90% of datacenter resources dedicated to ML workloads [2, 7]. In various critical applications, such as healthcare systems [20], recommendation systems [52], question-answering, and chatbots [11], a range of ML models, including computer vision models [20] and speech models [38], play an essential role. It is imperative to deploy these models cost-effectively while maintaining system performance and scalability.

Automatic resource allocation is a complex problem that requires careful consideration and has been extensively studied in various domains, including stream processing [18, 22, 39], serverless computing [46, 65], and microservices [21, 24, 76, 77]. Static auto-configuration of hardware resources [68], dynamic rightsizing of resources through autoscaling [73], and maximizing utilization with batching [3] are some of the techniques that have been used for resource management of ML models. In addition to efficient resource allocation, *accurate prediction* is another essential factor influencing ML model deployment. In many real-world scenarios, the predictions from these models have significant implications for business [57], industry [15], or human lives [1], and inaccuracies can lead to severe consequences [30, 43]. Hence, ensuring that ML models make accurate predictions is critical to producing reliable and trustworthy outcomes.

ML inference pipelines, as a chain of ML models, will raise several challenges for performance optimization. Unlike individually optimizing each stage, optimizing end-to-end ML inference pipelines will capture the correlation between configuration changes across multiple pipeline steps. Previous works [16, 33, 40, 59] have proposed solutions to address the challenges of efficient autoscaling, batching, and pipeline scheduling not only to consider the above challenges but also to consider the dynamic nature of ML workloads. However, none of

Table P3.1: Comparison of IPA with previous works; Pipeline: A chain of models inference or just single model inference? Cost: Whether the work optimizes cost? Accuracy: Does it optimize accuracy? Adaptive: Can it adapt to different accuracy/cost optimization trade-offs?

System	Pipeline	Cost	Accuracy	Adaptive
Rim [34]	✓	✗	✓	✗
INFaaS [58]	✗	✓	✓	✗
Inferline [16]	✓	✓	✗	✗
GPULet [13]	✓	✓	✗	✗
Llama [59]	✓	✓	✗	✗
FA2 [56]	✓	✓	✗	✗
Model Switch [75]	✗	✗	✓	✗
Scrooge [33]	✓	✓	✗	✗
Nexus [63]	✓	✓	✗	✗
Cocktail [28]	✗	✓	✓	✗
InfAdapter [61]	✗	✓	✓	✓
IPA	✓	✓	✓	✓

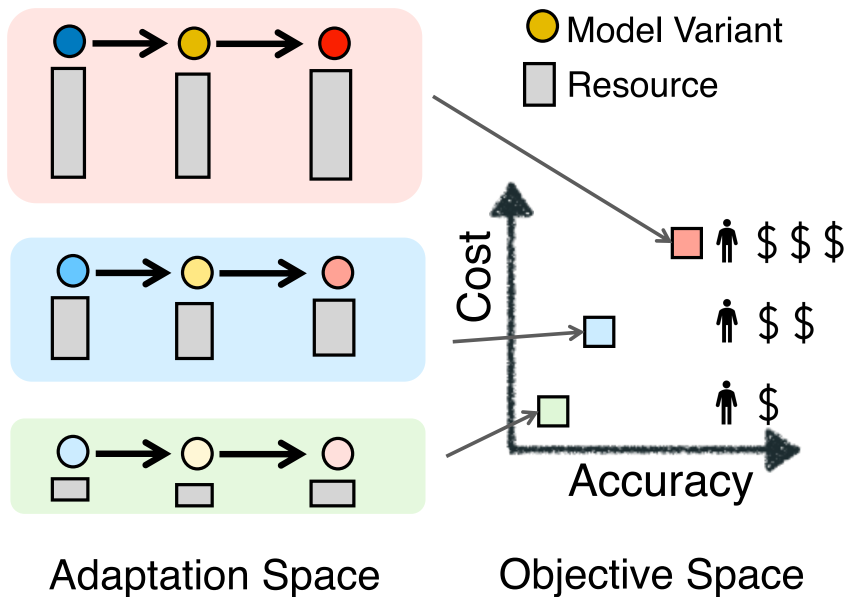


Figure P3.1: IPA provides a tunable framework for adjusting the system based on two contradictory cost and accuracy objectives.

these approaches considers the combined optimization of accuracy and resource allocation across pipelines.

In ML inference pipelines, two adaptation techniques are commonly used: **Autoscaling**, which adjusts resources based on workload, and **Model-switching**, which employs different model variants with different accuracies/latencies to vary resource demands and tasks, allowing finer control over resource allocation and accuracy. The combination of these two techniques has been advocated to achieve more precise adjustments in accuracy and cost trade-offs. Previous autoscaling [60] and model-switching [61, 75] works have argued that using both techniques in conjunction with each other is beneficial in providing more precise adjustments in terms of accuracy and cost trade-offs, providing greater flexibility and efficiency in ML model resource allocation. However, none of the above works has considered optimizing accuracy and cost jointly in a multi-stage pipeline setting. Table P3.1 presents an overview of related inference serving works. Systems with inference *pipeline* serving have often overlooked the presence of multiple model variants for each inference task [16, 33, 40, 56, 59]. The heterogeneity of these model variants presents an opportunity not only to configure the pipeline to meet latency objectives but also to opportunistically select the most suitable model variant to enhance the accuracy of the pipeline output. On the other hand, previous works that have considered accuracy optimization of machine learning services [28, 58, 61, 75] do not support inference pipelines with an end-to-end optimization over all the stages of the inference pipeline.

In this paper, we propose IPA, a system for jointly optimizing accuracy and cost objectives. It can achieve multiple trade-offs between these two conflicting objectives based on the pipeline designer's preference. Figure P3.1 shows the premise of IPA that provides a tunable framework for adapting the inference pipeline to achieve an optimal trade-off between accuracy and cost objectives given constraints and user preference. The choice of models in previous works was limited to using just one pre-selected model variant. IPA can broaden this search space by considering all model variants and dynamically adapting to a suitable choice of models based on the pipeline designer's preference.

The main contributions of this paper are as follows:

- We revisit the resource management design in inference pipelines by incorporating model switching, autoscaling, and pipeline reconfiguration (stage batch size). IPA proposes a new optimization formulation to allow for a more granular trade-off between the accuracy and cost objectives. It is also adaptable based on

the inference pipeline designer's preference for each accuracy and cost objective.

- We propose a new optimization formulation based on the interaction between model switching, replication, and batch size. It can find (1) exact resources to allocate to each stage of the pipeline, (2) variants to decide for each stage, and (3) dependency between stages that enable accurate estimation of demand while guaranteeing end-to-end latency.
- The full implementation of IPA is built on top of Kubernetes. IPA integrates open-source technologies in its stack to facilitate seamless integration into production clusters.
- Experimental results show that IPA can achieve more granular trade-offs between the two contradictory objectives of cost and accuracy. In some scenarios, it was able to provide an improvement of up to 21% in the end-to-end pipeline accuracy metric with a negligible increase in cost.

P3.2 BACKGROUND AND MOTIVATION

This section provides the background of the inference pipeline and discusses the challenges of reconciling cost and accuracy trade-offs.

P3.2.1 *Search Space*

Figure P3.2 illustrates the differences in latency and throughput between different versions of image classification models within the ResNet family. In particular, there exists an inverse relationship between ***throughput*** with both ***latency*** and ***accuracy*** for each variant of the model, considering the same number of ***CPU cores*** and fixed ***batch sizes***. The variability in performance across model variants adds a new dimension to the configuration search space.

Choosing the best configuration among the highlighted parameters is non-trivial and subjective to multiple objectives and constraints, e.g., cost efficiency and Service Level Agreement (SLA) requirements between cloud users and providers. Table P3.2 shows that under

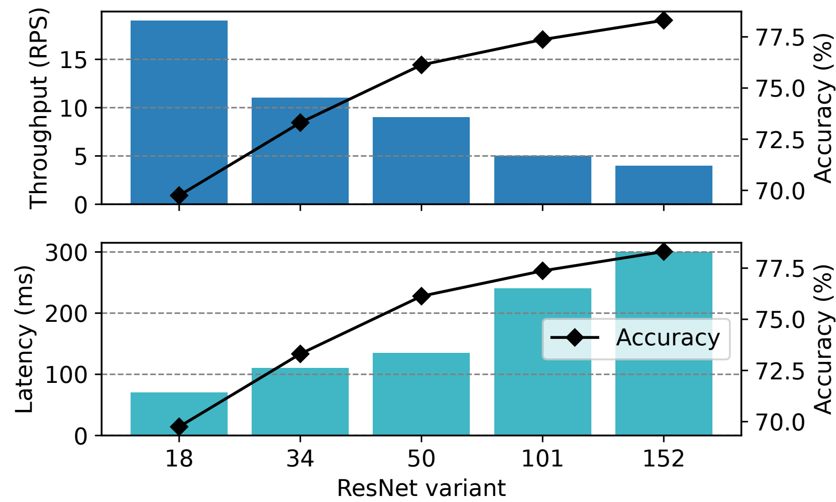


Figure P3.2: Performance difference across ResNet Family models for a batch size of one and one CPU core allocation.

Table P3.2: Performance difference across ResNet family models under different CPU allocations for a batch size of one, both blue and red core/model configuration can respond to 20 RPS and 75 ms throughput and latency requirements with different accuracy and costs.

CPU Cores	ResNet18		ResNet50	
	Latency (ms)	Throughput (RPS)	Latency (ms)	Throughput (RPS)
1	75	20	135	9
4	23	37	57	21
8	14	62	32	29

the same 20 RPS incoming throughput and mutual SLA agreement of 75 ms between the user and the service provider, a user with high accuracy goals will choose a suitable configuration of four core assignments in ResNet50 (marked red). However, a user with lower accuracy demands will choose ResNet18 with one core, which can respond to latency and throughput requirements under lower core allocations (highlighted in blue).

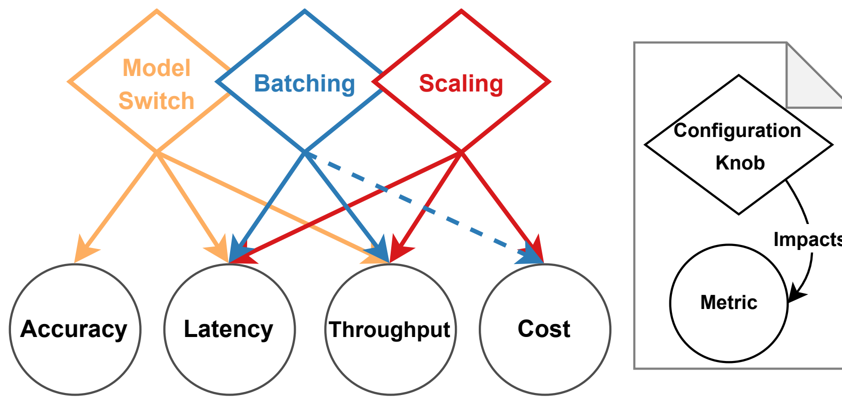


Figure P3.3: Impact of configuration knobs, batching indirectly affects the cost, e.g., decreasing the throughput will affect the IPA to more scaling and increase in the cost.

P3.2.2 *Configuration Space*

Batch Size Neural network structure provides parallel computation capability. Batching multiple requests will leverage the parallelization capability of neural networks and increase the utilization of assigned resources while increasing the total latency. Previous works [3, 16, 33] have shown that there is a relationship between the system utilization and the request latency, resulting in a non-trivial trade-off between maximizing the resource utilization without violating the latency SLA.

Replication There are mainly two resource provisioning techniques: vertical and horizontal scaling of resources. In vertical scaling, the assigned resources are modified, while in horizontal scaling, the number of replicas with the same amount of resources is adjusted to balance performance and cost. Horizontal scaling allows predictable performance using a similar environment [27], while vertical scaling allows a more fine-grained resource allocation to the ML model. We use horizontal scaling for the current work similar to [16, 56].

Variant Selection Previous works [58, 75] have shown that ML models are abundant for a single ML task. This brings the opportunity to abstract away the ML task from the underlying model and opportunistically switch the model based on the performance needs of the system.

Figure P3.3 shows the complex relationship between changing each configuration knob and performance objectives. Changing the batch

size will affect the throughput and latency of each pipeline stage, while changes in the replication factor will directly impact the pipeline deployment cost. Model switching will result in changes both in accuracy and cost as different models have different resource requirements.

P3.2.3 *Inference Pipelines*

Traditional ML applications revolve around the use of a singular deep neural network (DNN) to perform inference tasks, such as identifying objects or understanding natural language. On the contrary, modern ML systems (ML inference pipelines) are more intricate scenarios, such as digital assistant services like Amazon Alexa, where a series of interconnected/chained DNNs (in the form of DAG structures) are used to perform various inference tasks, ranging from speech recognition, question interpretation, question answering, and text-to-speech conversion, all of which contribute to satisfying user queries and requirements [56, 59]. As these systems frequently interact with users, it becomes essential to have a stringent SLA, which in our case is end-to-end latency.

The nonlinear dependency between the three configuration knobs (batch size, replication, and variant selection) and the pipeline variables introduces a complex decision space between multiple conflicting goals.

Figure P3.6(a) depicts one of the evaluated pipelines consisting of two stages, an object detection stage, and an object classifier. A subset of the configuration space is presented in Table P3.3 with different batch sizes, model variants, and deployment resources. We denote cost as the number of replicas $\times$ allocated CPU cores per replica. The chosen configuration at each stage should first support the incoming workload into the pipeline while guaranteeing SLA, e.g., the sum of the latency of both stages should be less than the SLA requirement. Under the arrival rate of 20 RPS (Request Per Second) and the SLA requirement of 600 milliseconds, both combinations of (A1, B1) and (A2, B2) can meet the latency threshold and accommodate the incoming throughput. However, the first combination can support these requirements at the cost of $2 + 2 = 4$ CPU cores with a lower accuracy combination, while the latter can support the same load at the cost of $10 + 3 = 13$ CPU cores and a higher possible accuracy combination.

Table P3.3: Two-stage pipeline tasks options. Latency is in Milliseconds and Cost is the number of physical cores.

Variant	Scale	Batch	Latency	Cost	Accuracy
A1: YOLOv5n	2	1	80	2×1	45.7
A2: YOLOv5m	5	1	347	5×2	64.1
A3: YOLOv5n	2	8	481	2×1	45.7
A4: YOLOv5m	5	8	1654	5×2	64.1
B1: ResNet18	2	1	73	2×1	69.75
B2: ResNet50	3	1	136	3×1	76.13
B3: ResNet18	2	8	383	2×1	69.75
B4: ResNet50	3	8	833	3×1	76.13

Challenge 1 Multiple configurations can satisfy the latency constraints of the inference pipeline. The "optimal" configuration depends on the accuracy and cost goals.

The next challenge is that in inference pipelines, the model selection at an earlier stage of the pipeline will affect the optimal model selection at downstream models, as the latency of a model at an earlier stage will affect the end-to-end latency. Consequently, the options available for the downstream models are more limited. In a similar example of pipeline Figure P3.6(a) and Table P3.3, and under an SLA of 500 ms, choosing a high latency and high accuracy configuration of A2 at the first stage will eliminate the variant B2 from the second stage's option.

Challenge 2 The choice of replication factor, batch size, and model variant is a joint decision in multiple stages of the inference pipeline.

P3.3 SYSTEM DESIGN

This section provides a high-level overview of the main system components in IPA as illustrated in Figure P3.4.

Model Loader Users submit the models they intend to use for each stage of a pipeline. These models should provide a reasonable span of accuracy, latency, and resource footprint trade-offs.

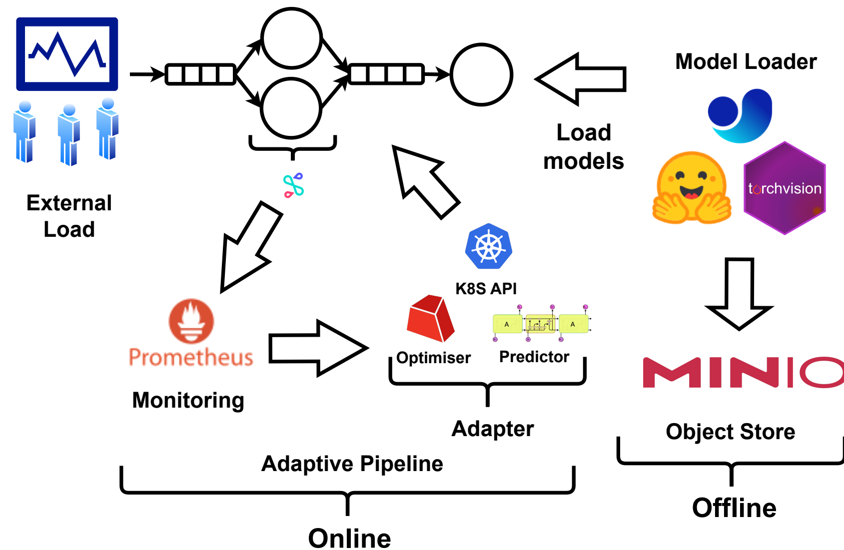


Figure P3.4: IPA system design. It consists of an offline phase for model profiling and an online phase for adaptive inference serving.

Model variants can also be generated using model optimizers such as TensorRT [66], and ONNX graph optimization by using different quantization level of neural networks [23] and Neural Architectural Search methods [12]. After the model submission, the profiler (discussed in Section P3.4.2) will be executed for each model variant and store its latency under multiple batch sizes and resource assignments. Furthermore, as discussed in Section P3.4, the optimizer uses offline profiling to find the optimal solution during execution. Finally, the models are stored in object storage to reduce container creation times. In this work, we have used MinIO object storage [50].

Pipeline System After pipeline deployment through the model loader the pipeline is now ready to operate. Users will send inference requests to the pipeline to go through the models and return the inference predictions. Before models of each stage of the inference pipeline, a centralized load balancer distributes the inference requests between multiple stages of the inference pipeline. A centralized queue is behind each stage of the pipeline. Centralized queues help to have a deterministic queueing behavior and to efficiently model its latency, as described in Section P3.4. The queues of each stage then distribute the batched requests between model replicas. It uses a round-robin policy for load-balancing the batched requests between model replicas. Communications between multiple stages of the pipeline are implemented using gRPC [25]. The load balance between multiple containers of the same stage is achieved using Istio [36] sidecar containers. Each model

container is deployed using Docker containers built from a forked version of MLServer [51] and Seldon Core [62] to implement the gRPC web servers and deployment on Kubernetes.

Monitoring The monitoring daemon uses the highly available time-series database Prometheus [6] underneath to observe incoming load to the system. It will periodically monitor the load destined for the pipeline.

Predictor In our load forecasting process, we employ an LSTM (Long Short-Term Memory), which is a type of recurrent neural network [32]. Our LSTM model is designed to predict the maximum workload for the next 20 seconds based on a time series of loads per second collected from the monitoring component over the past 2 minutes. To train the LSTM model, we utilized the initial two weeks of the Twitter trace dataset [5]. The architecture of our LSTM neural network consists of two layers, a 25-unit LSTM layer followed by a one-unit dense layer serving as the output layer.

Runtime decisions for reconfigurations The profiling data gathered by the profiler provide the latency and throughput of each model variant under different batch sizes. For runtime decision-making, a discrete event simulator uses these profiling data to estimate the end-to-end latency and throughput of the pipeline based on the number of replicas, model variants, and batch sizes at each stage. The predicted pipeline latencies and throughputs are then used by the optimizer in the adapter to determine the optimal configuration in terms of accuracy and cost objectives.

Adapter The Adapter is the auto-configuration module that periodically (1) fetches the incoming load from the monitoring daemon, (2) predicts the next reference load based on the observed historical load using the LSTM module, (3) obtains the optimal configuration in terms of the variant used, batch size, and number of replicas for the next timestep, and finally (4) the new configuration is applied to the pipeline through the Kubernetes Python API [42].

Figure P3.5 shows a snapshot of the system with two available options per model in a video analysis pipeline; the first options are $\{YOLO5l, YOLO5n\}$ and the second are $\{ResNet18, ResNet152\}$. In low loads (a), choosing more accurate models $YOLO5l$ and $ResNet152$ with small batch sizes is preferable to ensure low latency. However,

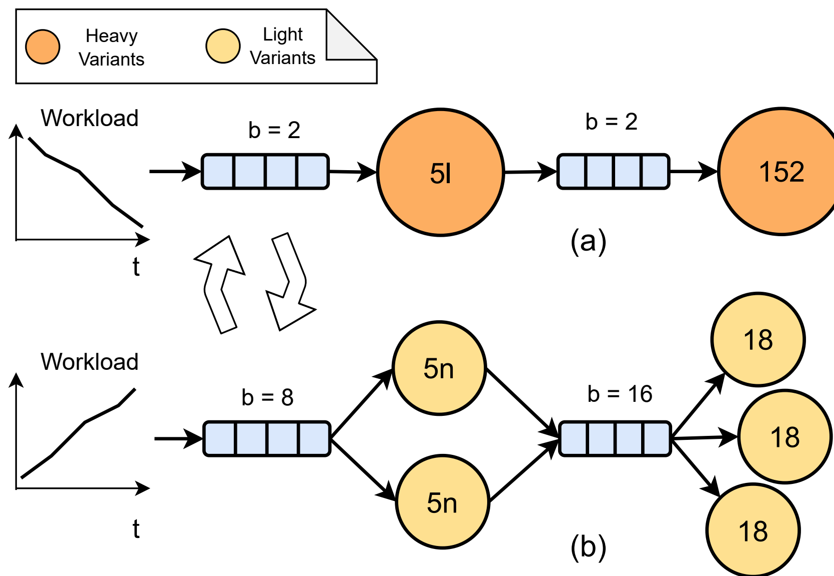


Figure P3.5: Switching between different configurations under (a) low and (b) high loads.

in higher loads, it is preferable to choose lightweight models such as $YOLO5n$ and $ResNet18$ with more replication and larger batch sizes to ensure high throughput for the system.

P3.4 PROBLEM FORMULATION

We now present the details of the optimizer discussed in Section P3.3. Problem formulation requires a robust definition of the accuracy of the inference pipeline and offline latency profiles of the model variants. Sections P3.4.1 and P3.4.2 discuss details of the inference pipeline accuracy definition and profiling methodology, respectively.

P3.4.1 *Accuracy Definition over Pipeline*

To the best of our knowledge, there is only a direct way to compute the end-to-end accuracy of a pipeline if one evaluates the accuracy of the entire pipeline on the labeled data designed for the pipeline. However, for inference pipelines with stages having different semantics (e.g., a speech model connected to a language model) and multiple model options for each stage, finding the accuracy of all the options

Table P3.4: Notations

Symbol	Description
P	Inference pipeline
$s \in P$	Inference pipeline stage
SLA_s	Latency service-level agreement for each stage s
SLA_P	Latency service-level agreement for pipeline P
λ_P	Request arrival rate of pipeline P
M_s	Sets of available model variants for stage s
m	A model variant
b_s	Batch size of stage s
R_m	Resource allocation of variant m
R_s	Resource allocation of stage s
a_m	Accuracy rank of variant m
n_s	Number of replicas of stage s
$I_{s,m}$	Indicator of activeness of variant m in stage s
PAS	Pipeline accuracy score
$q_s(b_s)$	Queuing time of stage s under batch size b
$l_{s,m}(b_s)$	Latency of variant m under batch size b_s
$h_{s,m}(b_s)$	Throughput of variant m under batch size b_s
A_P	End-to-end accuracy of pipeline P
th	Threshold RPS of base allocation of R_m
α	Accuracy objective weight
β	Resource allocation objective weight
δ	Penalty term for batching

by curating handpicked datasets for each pipeline composition is not feasible. In this work, we used a heuristic to rank the inference pipelines. We have only considered linear inference pipelines with one input and one output stage where a set of consecutive models are connected. The accuracy of each model is computed offline and is part of the model's property.

In this work, we do not consider model drifts [47], therefore, we use the per-stage statically computed accuracies to compute the end-to-end inference pipeline accuracy. For models with a qualitative performance measure other than accuracy (e.g., mAP for object detection, 1-WER for speech recognition tasks, and ROUGE for NLP tasks), as long as we have the specification *higher (or lower) means better* in the measure, we can substitute accuracy with that measure. To define a metric over

the accuracy in a pipeline, we use independent stage accuracy to compute the end-to-end accuracy over the entire inference pipeline. With the assumption of independence of errors between different stages of the inference pipeline, we have used *multiplying* of each stage of the inference pipeline accuracy as a heuristic to evaluate the overall accuracy preference between different combinations of models in the inference pipeline. We will refer to this metric for inference pipeline accuracy as **P**ipeline **A**ccuracy **S**core abbreviated as *PAS* metric.

The end-to-end accuracy is typically influenced by the combination of errors at each stage, and the relationship between these errors may not be accurately captured by multiplication. If errors are correlated or there are dependencies between stages, this method might deteriorate the soundness of the proposed inference pipeline's accuracy. However, we argue that through the evaluation of alternative accuracy metrics presented in Appendix P3.12, it becomes evident that until this problem is fully addressed by the machine learning community, the multiplication of individual stage accuracies serves as the most practical measure for approximating end-to-end accuracy.

P3.4.2 *Profiler*

The formulation of the joint cost-accuracy problem needs information on the latency, accuracy, and throughput of each model variant. Previous works have discovered that [27, 33, 56, 58] the latency of a model under a specific resource allocation is predictable based on the incoming batch sizes. The profiler will record the latency on the target hardware for different batch sizes under specific allocations of resources. We found in our experiments that assigning memory beyond a specific value to the models does not impact performance; therefore, we only need to find enough memory allocation for each node, which will be the memory requirement for running the largest batch size.

Finding a minimum allocation to containers is necessary since we have chosen horizontal scaling for workload adaptation in this work. Previous works on CPU evaluation for inference graphs [40, 56] have assigned one core to each container. Adapting a similar approach is not practical in our case, as more resource-intensive models cannot execute inference under given latency requirements with one core per

Table P3.5: Sample CPU cores base allocation for different YOLO variants under different RPS thresholds (Capped on maximum 32 cores).

load	YOLOv5n	YOLOv5s	YOLO5m	YOLOv5l	YOLO5x
5	1	1	4	8	16
10	1	2	8	16	×
15	1	8	16	32	×

container. Therefore, we find a base allocation for each model variant regarding the number of cores that can provide a reasonable base performance. The solver selects the model variants and horizontally scales them with the chosen base allocations.

Following previous works [26, 56], we define the per-stage latency SLA_s as the average latency of all available variants for the task to serve batch size one under the base resource allocation multiplied by 5 as suggested by Swayam [26]. The pipeline SLA_P that is later used in Section P3.4.3 is defined as the sum of per-stage SLA_s, ($SLA_P = \sum_{s \in P} SLA_s$). Table P3.5 provides some of the base CPU allocations under the 5 RPS, 10 RPS, and 15 RPS thresholds for five variants (YOLO5n, YOLO5s, YOLO5m, YOLO5l, and YOLO5x) of the object detection stage. The values in the allocation columns show the minimum number of CPU allocations per container needed to respond to a certain load in the stage SLA_s. We refer to these values as the base resource allocation to a model variant. The allocation of base resources for all stages ($\forall s \in S$) and their corresponding model variants ($\forall m \in M_s$) is the minimum number of resources in terms of CPU cores (Equation P3.1a) that can respond to a certain threshold (Equation P3.1b) and met the predefined base latency requirements per stage SLA_s for the largest batch size in our system (Equation P3.1c). Therefore, the minimum number of resources per model variant can be formulated as follows:

$$\min R_m \tag{P3.1a}$$

$$\text{subject to } th \leq h(m, R_m) \tag{P3.1b}$$

$$l_m(\max(b_s)) \leq SLA_s, \tag{P3.1c}$$

The value th in Equation P3.1b is a hyperparameter of the IPA solver, and its values are empirically found for each pipeline. The hyperparameters used in IPA are reported in Appendix P3.10. In summary, Equation P3.1a is used to find the base allocation for each model

variant where the threshold th is fixed and R_m is the resource requirements of model variant m. Therefore, the base resource allocation for all models can be found statistically. In the same Object Detector task with different model variants, as shown in Table P3.5, we choose the first configuration row, which supports the highest RPS with the minimum resource allocation per container. As a consequence, the allocation of base resources is fixed during runtime, and the performance of each stage of the pipeline $h(m, R_m)$ will be a function of used model variant m, and its throughput $h(m)$, and the number of replicas n_s.

For profiling, we follow [58] and record latency and throughput on the power of two increments of 1 to 64 batch sizes. Profiling per all batch sizes is costly; therefore, following [56] for each model variant, we fit the observed results on the profiled batch sizes under the base resource allocation to a quadratic polynomial function $l_m(b_s) = \alpha b_s^2 + \beta b_s + \gamma$ that can infer the latency for unmeasured batch sizes with a lower Mean Squared Error (MSE) than a linear function, $\alpha b_s + \beta$. Multiple model variants are available per-stage of the inference pipelines, and the mentioned approach can decrease the profiling cost by an order of magnitude.

P3.4.3 *Optimization Formulation*

We used Integer Programming (IP) as a robust optimization framework to address the intricate challenges associated with configuring inference pipelines. As we have discussed later in Section P3.4.4, due to the optimality of the IP results compared to heuristic solutions, we have chosen IP modeling alongside the Gurobi solver to guarantee the optimality of the results. We have discussed the scalability and limitations of the IP formulation and Gurobi solver in Section P3.5.3. The goal of IPA is to maximize accuracy and minimize cost while guaranteeing SLAs.

$$\text{Objectives} = \begin{cases} \text{Maximizing the Accuracy} \\ \text{Minimizing the Cost} \end{cases} \tag{P3.2}$$

There are $|M_s|$ model variants for each inference stage $s \in P$ in the pipeline P. Each $m \in M_s$ model variant performs the same inference tasks (e.g., image classification) that exhibit different resource requirements, latency, throughput, and accuracy. The resource requirements of the models are estimated offline in the profiling step (Section P3.4.2). The profiler provides the resource requirements of the model variants per task and their latencies under different batch sizes. The two main goals of IPA are maximizing the precision of the pipelines and minimizing the resource cost using lighter models. We define $I_{s,m}$ as an indicator of whether a selected model variant is currently active for task s or not:

$$I_{s,m} = \begin{cases} 1 & \text{if } m \text{ is active in stage } s \\ 0 & \text{Otherwise} \end{cases} \tag{P3.3}$$

At each point in time, only one model variant m can be active for each inference stage; therefore, the resource requirement of each stage model replicas R_s is equal to that stage active variant resource requirement R_m.

$$R_s = \sum_{m \in M_s} R_m \cdot I_{s,m} \tag{P3.4}$$

Similarly, the latency and throughput of each pipeline stage (l_s and h_s) are calculated based on the latency and throughput of the active model variant for that stage.

$$l_s = \sum_{m \in M_s} l_{s,m}(b_s) \cdot I_{s,m} \tag{P3.5}$$

$$h_s = \sum_{m \in M_s} h_{s,m}(b_s) \cdot I_{s,m} \tag{P3.6}$$

Another contributing factor to the pipeline's end-to-end latency is the time spent on the queue of each inference stage. For queue modeling, we have used the theoretical upper bound formulation introduced in [56]:

$$q_s(b_s) = \frac{b_s - 1}{\lambda} \tag{P3.7}$$

Equation P3.7 illustrates the worst-case queuing delay based on the arrival rate and the batch size. The first request that arrives in a batch should wait for $b_s - 1$ additional request before being sent to the models.

The formal definition of the metric *PAS* (explained in Section P3.4.1) for the accuracy of the end-to-end pipeline is defined in Equation P3.8. *PAS* is computed by multiplying the accuracies of the active models in each stage of the inference pipeline.

$$PAS = \prod_{s \in P} \left(\sum_{m \in M_s} a_{s,m} \cdot I_{s,m} \right) \tag{P3.8}$$

The multi-objective goal (P3.9) is to maximize the end-to-end accuracy of the pipeline and minimize the cost. The cost objective is achieved in two ways, using smaller models (models with fewer resource requirements) and using the least number of replicas for them. The batch size for each model should be chosen carefully, as larger batch sizes will increase the utilization and throughput of the entire load and the per batch latency. It is worth mentioning that the IPA multi-objective formulation is agnostic to inference pipeline accuracy definition (as also shown later in Appendix P3.12). The existing end-to-end inference graph accuracy definition can be substituted with potentially better future inference pipeline accuracy definitions.

$$\begin{aligned} f(n,s,I) = \alpha \cdot PAS \\ - \beta \sum_{s \in P} n_s \cdot R_s \\ - \delta \sum_{s \in P} b_s \end{aligned} \tag{P3.9}$$

The two variables α and β adjust the preference level given to each objective. We should find a batch that increases utilization on a reasonable scale. Following [56], we have added a small penalty term for batch size in the objective function δ that tries to minimize batch sizes until lowering batch sizes causes instability in the system (the system throughput becomes lower than the incoming workload). We now can describe the auto-configuration of the three online configuration knobs explained in Section P3.2.2 as an IP problem:

$$
\begin{aligned}
\max \quad & f(n,s,I) && \text{(P3.10a)}\\
\text{subject to} \quad & \sum_{s\in P} l_s(b_s) + q_s(b_s) \leq SLA_P, && \text{(P3.10b)}\\
& \text{if } I_{s,m} = 1, \text{ then} && \\
& \quad n_s \cdot h_s(b_s) \geq \lambda_p, \quad \forall s \in P && \text{(P3.10c)}\\
& \sum_{m\in M_s} I_{s,m} = 1, \quad \forall s \in P && \text{(P3.10d)}\\
& n_s, b_s \in \mathbb{Z}^+, \quad I_{s,m} \in \{0,1\}, \quad \forall s \in S, \forall m \in M_s && \text{(P3.10e)}
\end{aligned}
$$

The chosen combination of models should be able to meet the latency (P3.10b) constraint of the pipeline. The pipeline SLA_P is the aggregate of per-stage SLA_s as described in Section P3.4.2. The end-to-end latency of the pipeline is obtained by summing the inference latency of the chosen model variant $l_s(b_s)$ and the queueing time of each model server $q_s(b_s)$ in the inference path. Also, the sum of the throughput of all replicas of an active model should be higher than the arrival rate P3.10c into the pipeline.

In summary, the objective function tries to find the most accurate combination of models in the inference pipeline while allocating the least number of physical resources based on the pipeline designer's preference. This trade-off between the two objectives is configurable by modifying the α and β weights for accuracy and resource allocation. Therefore, the inputs of the optimization formulation are resource requirements R, latency l, precision of all models a, and throughputs h of all model variants m in all stages s under all possible batch sizes b alongside the queueing latency model q under all batch sizes, the incoming load λ coming to the pipeline and pipeline latency SLA_p. The output returns the optimal number of replicas n, batch size b, and the chosen model variant I for each pipeline stage.

P3.4.4 *Gurobi Solver*

IPA has been designed to meet production-level requirements, (1) guaranteeing an optimal solution, (2) no need for additional pretraining or retraining costs, and (3) negligible overhead. Other approaches like heuristics are ad hoc solutions that are hard to generalize to different

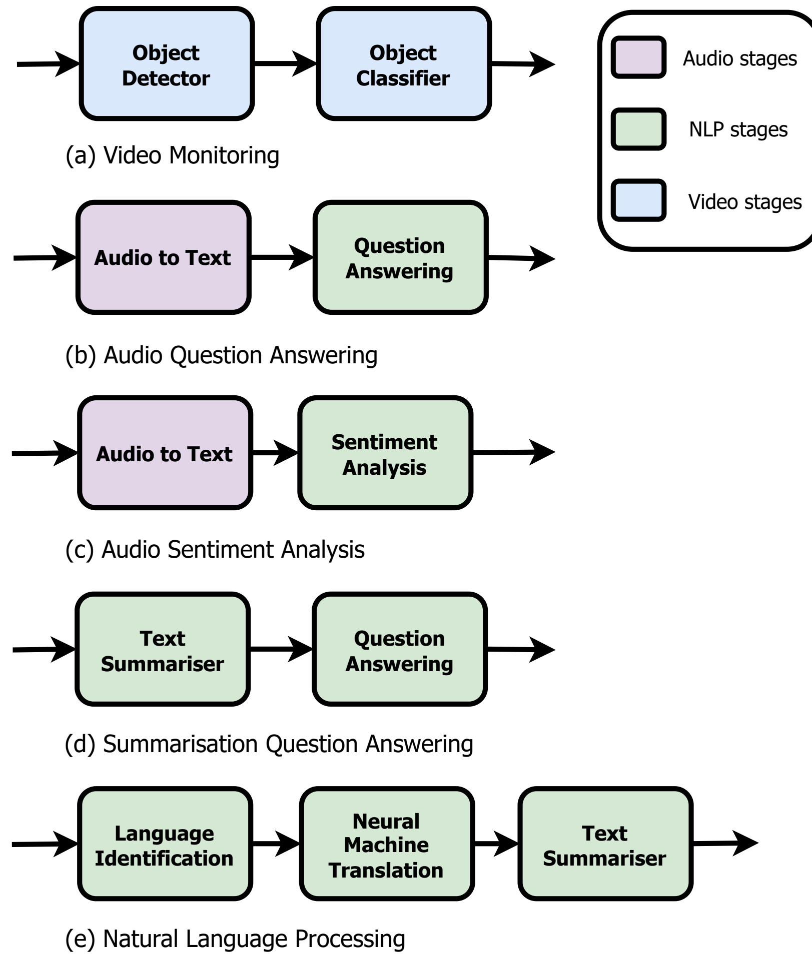


Figure P3.6: Representative pipelines used in this work.

systems, and ML approaches do not guarantee the optimal solution (failing to find the optimal solution results in over-provisioning or under-provisioning) and typically incur long training times (for example, reinforcement learning). On the contrary, although the IP formulation is NP-hard (as shown in [58]), it meets the above requirements for a production-ready system. In our case, we chose the Gurobi solver [10] that guarantees the optimal solution; the downside of using the solvers is that in the case of very large search spaces or under very tight SLA requirements, it might take a long time to find the desirable configuration. In our case, Gurobi solved the problem formulation in Formula P3.10 in less than a second.

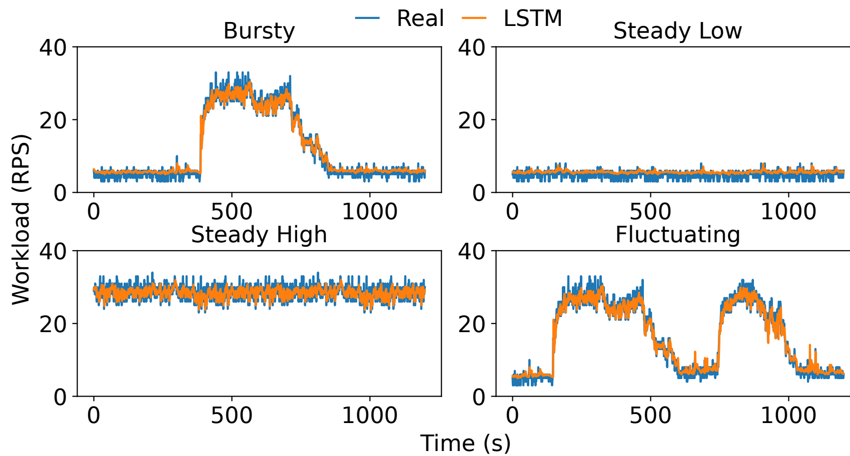


Figure P3.7: Representative tested load patterns from the Twitter trace[5], showing LSTM predictions.

P3.4.5 *Dropping*

High workloads may cause high back pressure in the upstream queues. If a request has already passed its *SLA* at any stage in the inference pipeline, then there might be no point in continuing to serve it until the last stage and placing high pressure on the system. A mechanism we have used is to drop a request at any stage of the pipeline if it has already passed *SLA* in the previous steps. We also consider that a request is dropped if its current latency exceeds $2\times$ the SLA to avoid constant back pressure on the queues.

P3.5 EVALUATION

In this section, we conduct extensive experiments to demonstrate the practical efficacy of IPA in real-world scenarios using a diverse set of workloads. The code and data for replicating the results are available at `Link-obscured-for-double-blind`.

P3.5.1 *Experimental Setup*

Most previous works on inference pipelines [9, 16, 55, 59] have implemented the entire pipeline on their self-made infrastructures. We have implemented our frameworks on top of Kubernetes, the de facto

Table P3.6: per-stage and end-to-end SLA of inference pipelines (in seconds).

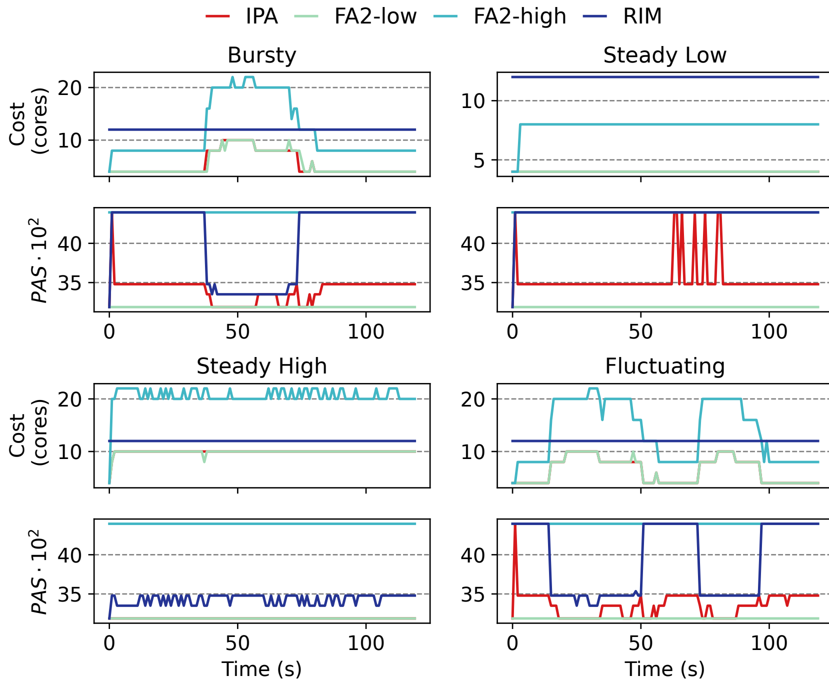
Pipelines	Stage 1	Stage 2	Stage 3	E2E
Video Monitoring	4.62	2.27	×	6.89
Audio QA	8.34	0.89	×	9.23
Audio Sentiment	8.34	1.08	×	9.42
Sum QA	2.52	1.32	×	3.84
NLP	0.97	12.76	3.87	17.61

standard in the containerized world and widely used in industry. This will enable easier access to the framework for future use by developers. IPA is implemented in Python with over 8K lines of code, including the adapter, simulator, queuing, load balancer, and model container implementations.

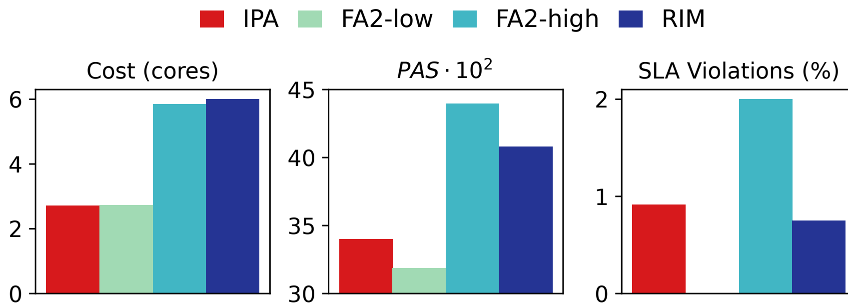
Hardware We used six physical machines from Chameleon Cloud [41] to evaluate IPA. Each server is equipped with 96 Intel(R) Xeon(R) Gold 6240R CPU @ 2.40GHz cores and 188 Gb of RAM.

Pipelines We used five descriptive pipelines with a wide variety of models for each stage, as shown in Figure P3.6. The pipelines are adapted from previous work and also from industrial examples. The video monitoring pipeline (pipeline a) is a commonly used pipeline in previous works [16, 74] and industry [14] which an object detector sends the cropped images to a later model to perform classification tasks like detecting a license plate or human recognition. The audio and question answering / sentence analysis pipelines (pipelines b and c) are adapted from use cases that comprise multiple types of ML model [19]. NLP pipelines (pipelines d and e) are representative examples of emerging use cases of language models [71, 72]. For a full specification of the models used in each stage of the pipelines, refer to Appendix P3.10. Also, for the α, β and δ values of Equation P3.9 used for the experiments IPA, see Appendix P3.11.

Baselines We compare IPA with variations of two similar systems, namely FA2 [56] and RIM [34]. FA2 is a recent system that achieves cost efficiency using scaling and batching; however, compared to IPA, it does not have model switching as an optimization angle. RIM, on the other hand, does not have scaling as a configuration knob but uses model switching to adapt to dynamic workloads. The original RIM does not include batching; we also add batching to RIM for a fair comparison. As RIM does not support scaling, we statically set the



(a) Temporal analysis under different workloads.



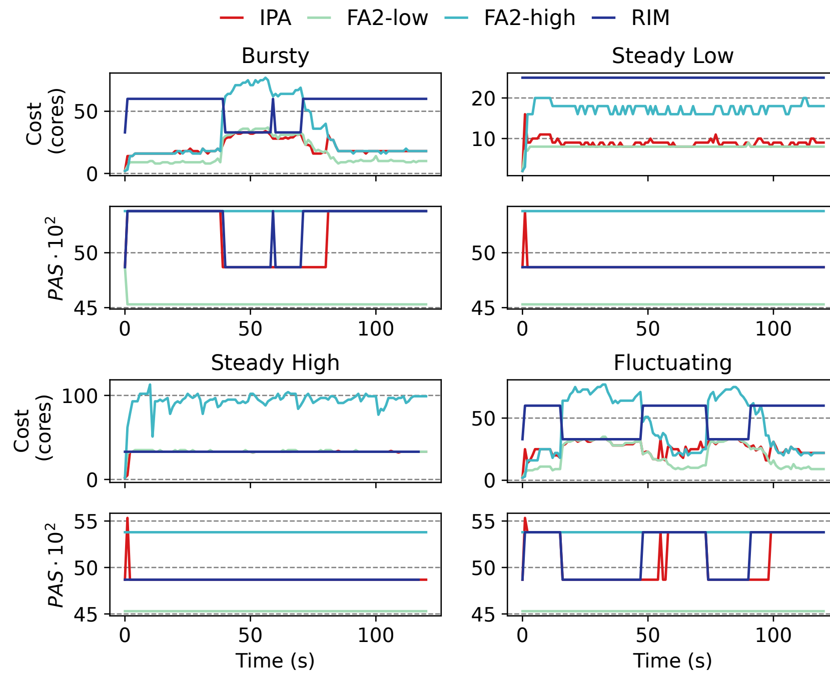
(b) Average analysis on bursty workload

Figure P3.8: Performance analysis of the Video pipeline.

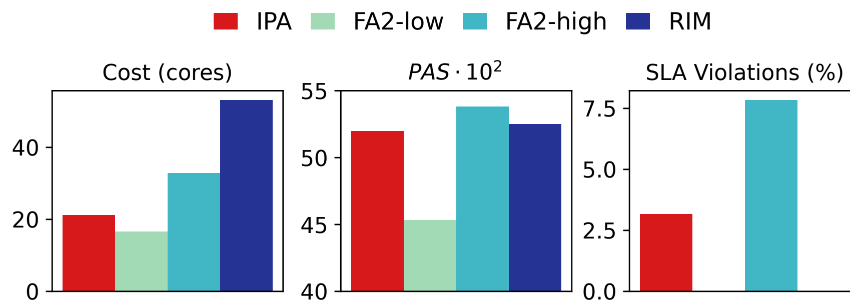
scaling of each stage of the inference pipeline to a high value. Similarly, FA2 does not support model switching; therefore, we use two versions of it, one FA2-low, which sets the model variants to the lightest models, and FA2-high, which sets the model variants to a heavy combination of models on each stage [1]. The three systems compared benefit from the LSTM predictor explained in Section P3.3.

Workload Figure P3.7 shows excerpts from Twitter trace [5] that have been used to evaluate the performance of IPA against four parts of the data set. It includes bursty, fluctuating, steady low, and steady

1 Ideally we should have set the FA2-high to the heaviest models but due to resource limitations, we set it to models that on average give better accuracy compared to IPA



(a) Temporal analysis under different workloads.

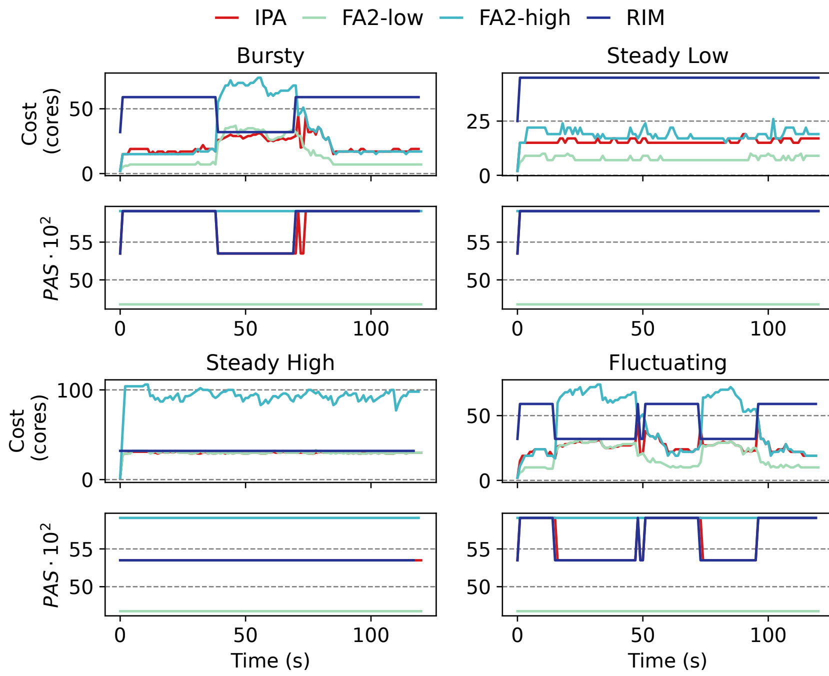


(b) Average analysis on bursty workload.

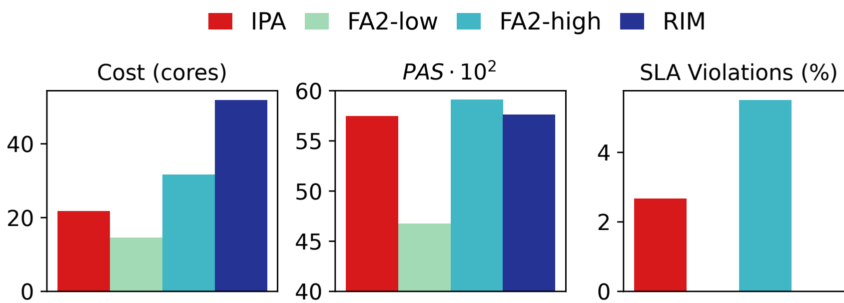
Figure P3.9: Performance analysis of the Audio-qa pipeline.

high workloads. For the train and test split during the training of the LSTM module, we trained the LSTM module on 14 days of the Twitter trace and chose the four mentioned workload excerpts from the other 7 unseen parts of the dataset. The LSTM predictor can predict the workload in less than 50*ms* with a Symmetric Mean Absolute Percentage Error (SMAPE) [31] of 6.6% that is comparable to the predictors used in systems with similar context [78]. Furthermore, an asynchronous load tester was implemented to emulate the behavior of users in real-world data centers.

SLA Table P3.6 shows the pipeline SLA of each inference pipeline that is calculated by summing the heuristic per-stage SLAs explained in Section P3.4.2.



(a) Temporal analysis under different workloads.

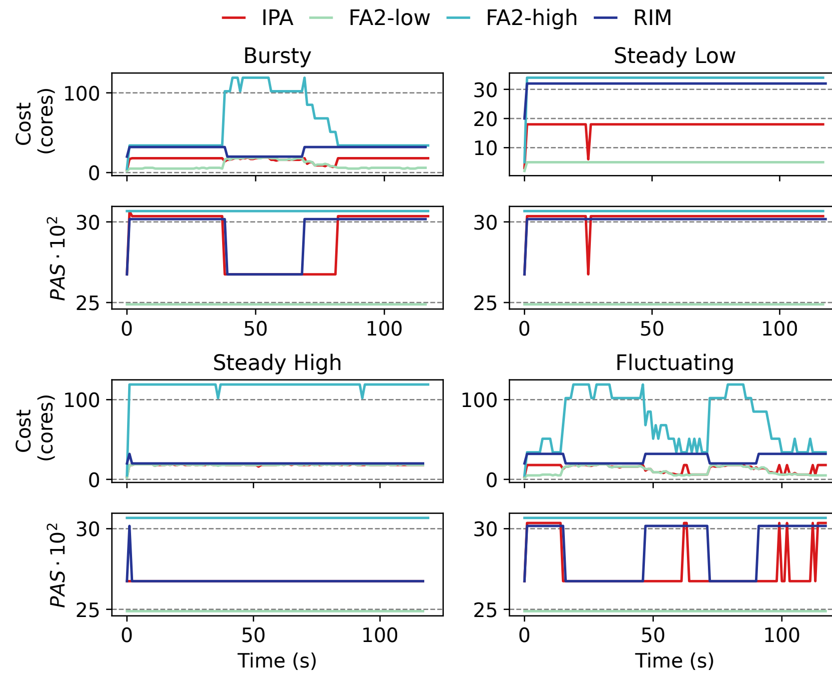


(b) Average analysis on bursty workload.

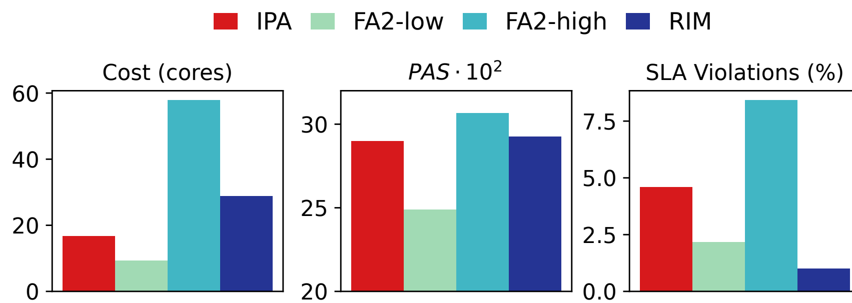
Figure P3.10: Performance analysis of the Audio-sent pipeline.

P3.5.2 *End-to-End Evaluation*

Figure P3.8 shows the evaluation results for the video pipeline in the four bursty, steady high, steady low, and fluctuating workloads. Since FA2-high and FA2-low are always set to the lightest and heaviest variants, they will always provide the lowest and highest possible *PAS* despite load fluctuations. In the three bursty, steady low, and fluctuating workloads IPA can always achieve a trade-off between the cost objective and, in steady high workload IPA, diverge to a configuration that uses the lowest-cost model variants to adapt to the high resource demands of steady high workload. The only available adaptation mechanism for RIM is to change the models; therefore,



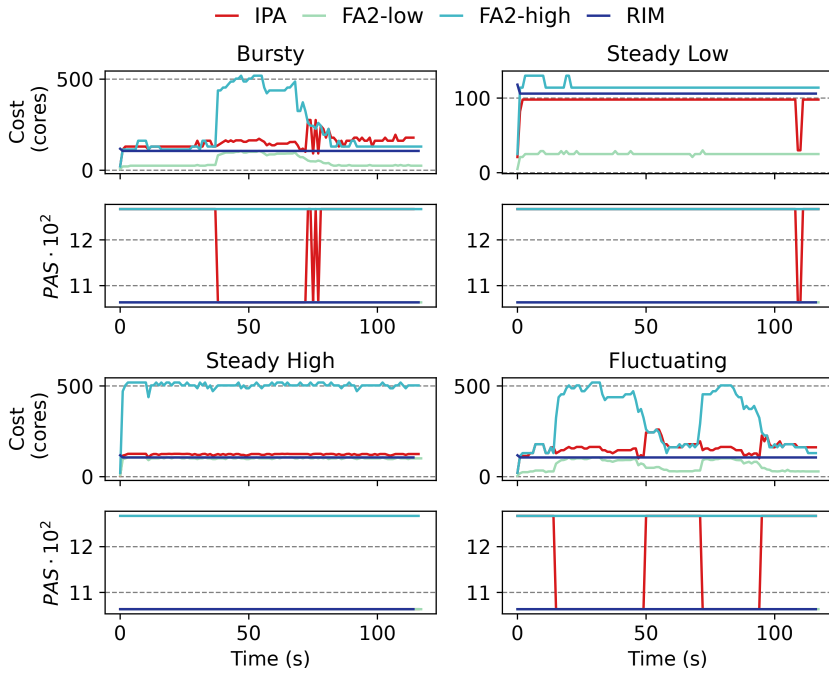
(a) Temporal analysis under different workloads.



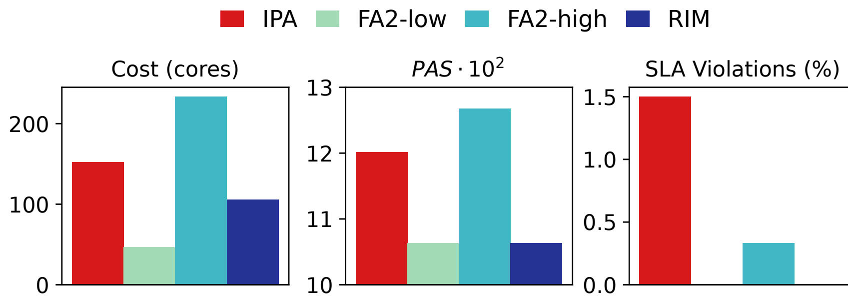
(b) Average analysis on bursty workload.

Figure P3.11: Performance analysis of the Sum-qa pipeline.

under load variations in bursty and fluctuating workload, it trades off accuracy for meeting the incoming load burst throughput. Failing to meet the incoming workload will result in request drops and SLA violations. As expected, FA2-low and FA2-high have the highest and lowest SLA attainment. In the video pipeline, IPA provides the same resource efficiency as FA2-low, as the base allocation (explained in Section P3.4) of variants used for the first stage in the video pipelines is similar in most cases, and changing the model in favor of latency reduction does not result in higher computational costs (this is the reason for the overlap between FA2-low and IPA in the cost temporal plots shown in Figure P3.8a.). In total, IPA can show a better balance between the two cost and accuracy objectives. Although FA2-high and RIM provide the highest accuracies, their cost efficiency is compro-



(a) Temporal analysis under different workloads.

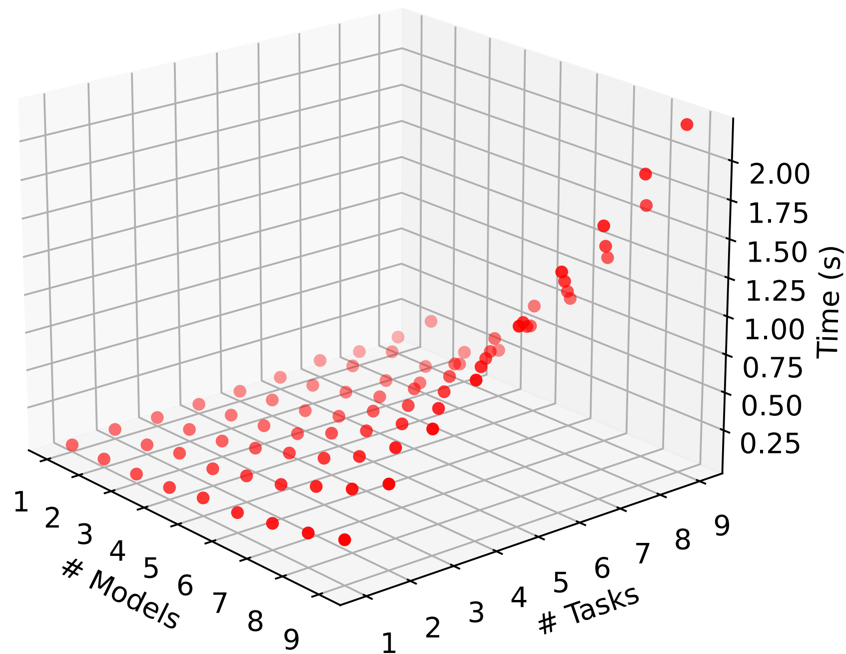


(b) Average analysis on bursty workload.

Figure P3.12: Performance analysis of the NLP pipeline.

mised due to employing more accurate variants per stage and a high scaling factor. FA2-low can meet the requirements of SLA and achieve the same cost efficiency as IPA but cannot improve accuracy because it is fixed in the lightest variants. The higher violation rate in FA2-high, RIM, and IPA compared to FA2-low is because SLAs are defined using the processing latency of average models, and SLAs become tighter for more accurate models, resulting in a higher tail latency violation. Figures P3.9a and P3.10a show the same temporal and average results on the audio-qa and audio-sent inference pipelines. Due to the lower number of variants used in these two pipelines ($5 \times 5 = 25$ for video and 5×2 and 5×3 in the audio-qa and audio-sent pipelines), we observe fewer fluctuations in RIM in all workloads. However, like the

Figure P3.13: Decision time of Gurobi optimizer for IPA formulation with respect to the number of models and tasks on the inference graph.



video pipeline, IPA achieved a trade-off between the accuracy and cost objectives.

Compared to the stages in the three mentioned pipelines, the base allocations for the summarization stage used in the sum-qa and NLP pipelines provide a larger span of changes in the required CPU cores (see Appendix P3.10). For example, the resource difference between the heaviest and lightest model in the Object Detection stage of the video pipeline is $8 - 1 = 7$, while it is more than doubled in the summarization stage ($16 - 1 = 15$). Consequently, we observe a longer span of differences between the FA2-low and FA2-high approaches in these two pipelines. In both approaches, IPA can adapt to the load using the second least heavy models, resulting in a cost reduction of 3x and 2x with only 2 and 1 loss value in the inference *PAS*. It should be noted that the initial spikes in *PAS* are due to the initial setting. The optimizer then immediately adjusts the models in response to the workload.

Figure P3.14: Comparison of IPA results for different trade-offs between accuracy and cost objectives, IPA can navigate effectively between the two cost and accuracy objectives.

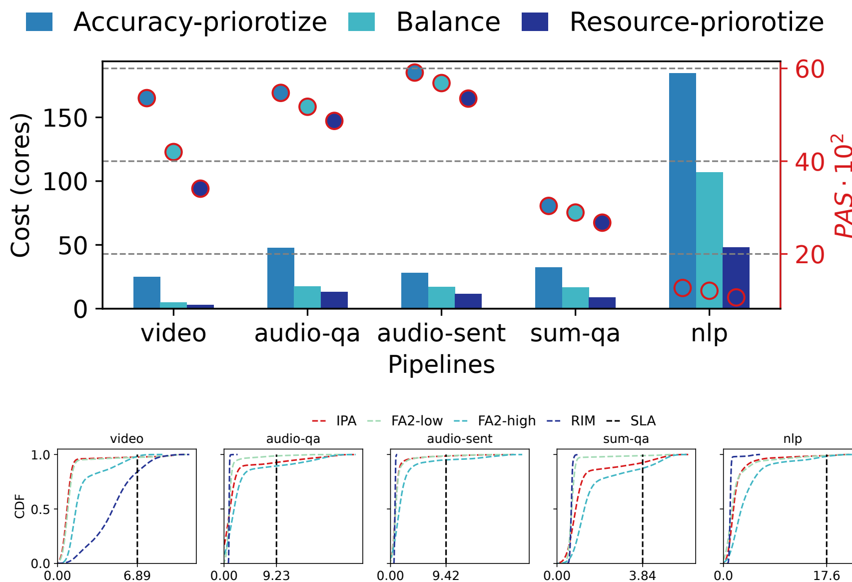


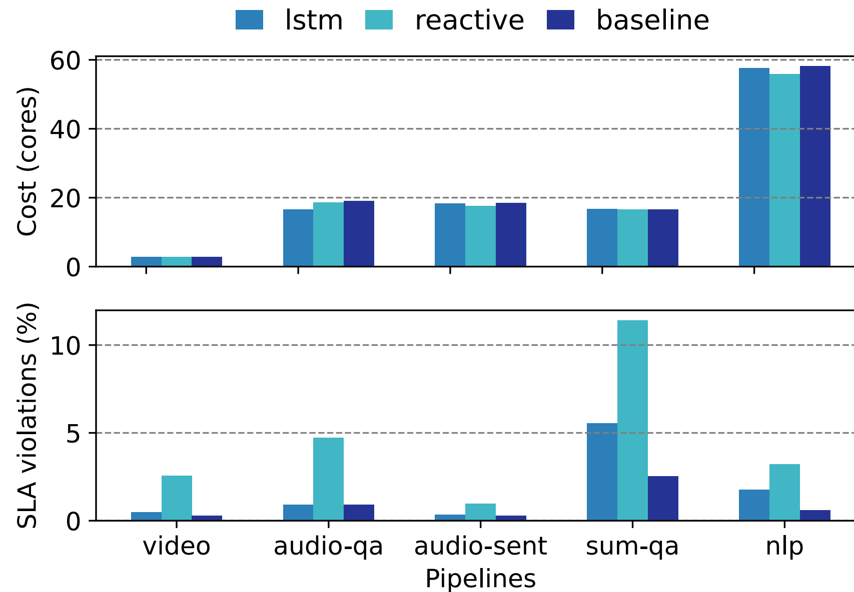
Figure P3.15: End-to-end latency distribution for the five tested inference pipelines under different approaches. IPA can achieve latency close to the FA2-low with light model variants, and only RIM is achieving better latency at the expense of high resource overprovisioning.

P3.5.3 *IPA Scalability*

System Scalability To examine the effectiveness of IPA in real-world systems, we included the NLP pipeline in our evaluations, which during bursts scales to more than 500 cores (Figure P3.12a). Due to the use of best production-grade ML deployment practices, such as lightweight containers and Kubernetes as the back-end with the benefit of distributed scheduling and cluster management, we believe IPA has the potential to scale to large clusters.

Optimizer Scalability As mentioned in Section P3.4, we have used the Gurobi solver to solve the IP optimization problem. One critique of using Gurobi is its limitations in the solvable problem space in the time constraints of a real-world autoscaler. To guarantee fast autoscaler adaptation to workload fluctuations, the autoscaler should be able to find the next configuration in less than two seconds to leave enough room for the adaptation process itself, which in our experiments was

Figure P3.16: Effect of using predictor on reducing SLA violations on bursty workload, IPA LSTM can reduce SLA violations up to 10x with the same resource usage. Also, using baseline predictors with perfect knowledge about the future reveals potential room for reduction of SLA with better predictors.



around the same number of eight seconds that sums up $8 + 2 = 10$ which we used as our adaptation monitoring interval. We conducted a set of simulated experiments shown in Figure P3.13 to examine the decision-making time of the IPA growth with respect to the change in the number of available model variants and the number of tasks in the inference pipelines (length of the inference graph). IPA can find the optimal configuration for inference pipelines with 10 stages, each with 10 models, in less than 2 seconds. Having an effective decision time for inference pipelines beyond these sizes requires faster optimization solutions. In addition, IPA will adapt to workloads with more SLA violations for pipelines that have SLA requirements lower than the decision-making time of IPA (although it will eventually adapt). However, most existing inference pipelines [53] rarely go beyond 10 stages; therefore, IPA will be effective for real-world use cases.

P3.5.4 *IPA Adaptability*

The main premise of IPA is to provide an adaptable framework to achieve a trade-off between the cost and accuracy objectives utilizing the three configuration knobs of model switching, scaling, and batching. Instead of using a fixed value for α and β in previous experiments, we examined the effect of changing the weights given to each objective by modifying the values of α and β for each inference pipeline. Figure P3.14 shows a set of experiments carried out on all five pipelines, where in one scenario, cost optimization (resource prioritize) is set as a priority by setting β to a higher value, and in another scenario, accuracy is set as a system priority by using a higher value for α. It is evident that IPA provides an adaptable approach to optimize different cost and accuracy preferences by the inference pipeline designer. For example, a highly accurate adaptation scenario can be chosen for the audio-sent pipeline with 28 CPU cores and an average *PAS* of 59 or a lower accurate result of 53 with 11 CPU cores.

Figure P3.15 shows the end-to-end latency CDF of the five pipelines tested to further show the flexibility of IPA in dynamic workloads. IPA leverages its fast adaptation by using heavy models only when the load is low and achieves nearly the same latency efficiency as FA2-low (with a higher accuracy than FA2). Only RIM can provide better latency compared to IPA, but as shown in the previous examples (e.g., Figure P3.9b for the Audio-qa pipeline) comes at the expense of high resource allocations (3x compared to IPA in the same pipeline).

P3.5.5 *IPA Predictor*

Predictors are effective in reducing SLA violations. Most previous work on inference pipeline serving [16, 33, 34, 56] has done reactive auto-configuration. In reactive approaches, configuration changes occur with live load monitoring and in response to load changes. [73, 78] for predicting load before load changes. IPA uses a proactive approach using an LSTM predictor that takes advantage of historical data. The ablation analysis provided in Figures P3.16 shows that using the LSTM predictor is beneficial in reducing SLA violations in all pipelines with a negligible difference in resource consumption. The LSTM module is trained in less than ten minutes for the 14 days of Twitter traces;

therefore, using it is practical in real-world scenarios. Furthermore, comparing with the baseline, which is a predictor that has complete knowledge of the load in the future interval (ground truth of the LSTM predictor), shows that in the case of video, audio-qa and audio-sent pipelines, the IPA predictor performs well with the baseline predictor. Moreover, the baseline predictor results in sum-nlp and the nlp results show that better load predictions in the prediction module can also potentially result in further reduction in SLA.

P3.6 RELATED WORKS

Single stage inference serving: Several approaches have been proposed in previous research to improve performance metrics without considering multiple stage inference models [3, 17, 26, 68, 73]. They intend to enhance a set of performance metrics, e.g., latency and throughput, and reduce resource utilization through adaptive batching, horizontal and vertical resource scaling, model switching, queue reordering, and efficient scheduling of models in heterogeneous clusters. A few works have considered joint optimization of qualitative metrics like accuracy and performance for single-stage inference-serving systems. Model Switching [75] proposes a quality adaptive framework for image processing applications. It switches between models trained for the same task configuration knob and switches from heavier models to lighter models in response to load spikes. However, the proposed prototype does not consider the interaction of model switching between other resource configuration knobs, such as autoscaling and batching. INFaaS [58] abstracts the selection of model variants in the single-stage setting from the user and automatically selects the best-performing model within the user-defined SLOs. It also actively loads and unloads models based on their usage frequency. InfAdapter [61] and Cocktail [28] propose joint optimization formulations to maximize accuracy and minimize cost with predictive autoscaling in single-stage inference scenarios.

Multi-stage inference serving: Several approaches have been proposed in previous research to improve inference performance metrics in multistage inference serving systems [16, 29, 33, 34, 37, 40, 45, 54–56, 59, 64, 69] since changing one model's configuration affects subsequent steps. InferLine [16] reduces the end-to-end latency of

ML service delivery by heuristically optimizing configurations such as batch sizes and horizontal scaling of each stage. Llama [59] is a pipeline configuration system specific to the use case designed exclusively for video inference systems. It attempts to reduce end-to-end latency by interactively decreasing the latency of each stage in the pipeline. Stages of the pipeline can be either ML inference or non-ML video tasks like decoding. GrandSLAm [40] is a system designed to minimize latency and ensure compliance with SLA requirements in the context of a chain of microservices mainly dedicated to ML tasks. The system achieves this by dynamically reordering incoming requests, prioritizing those with minimal computational overhead, and batching them to maximize each stage's throughput. FA2 [56] provides a graph transformation and dynamic programming solution in inference pipelines with shared models. The graph transformation part breaks the execution graph to make it solvable in real-time, and the dynamic programming solution returns the optimal batch size and scaling factor per each DNN stage of the pipeline. Multimodel and Multitask inference VR/AR/Metaverse pipelines [8, 44] are other emerging use cases of inference pipelines. However, unlike IPA, none of the above approaches considers the three pillars of accuracy, cost, and end-to-end latency/throughput jointly for multistage inference serving systems.

P3.7 CONCLUSION AND FUTURE WORKS

In this work, we introduced IPA, an online auto-configuration system for jointly improving the resource cost and accuracy over inference pipelines. IPA uses a combination of offline profiling with online optimization to find the suitable model variant, replication factor, and batch sizes for each step of the inference pipeline. Real-world implementation of IPA and experiments using real-world traces showed that it could preserve the same cost efficiency and SLA agreement while also having an improvement of up to 21% in the proposed end-to-end pipeline accuracy metric (*PAS*) compared to the two baseline approaches. The following are some directions for future work.

Scalability. IPA leverages Gurobi solver for finding the suitable configurations. This worked fine in our problem setting, as a limited number of model variants were used in each step of the pipeline. However,

an interesting future direction for IPA is to examine its performance where more model variants are available for each step of the pipeline and also in cases where we have more complicated larger graphs [54]. The adapter needs to be able to respond to bursts in less than one second, which demands either designing new heuristic methods that can find a good enough but not necessarily optimal solution or data-driven solutions like some of the methods that have been used before in similar auto-configuration context like Bayesian Optimization [4], Reinforcement [68] Learning or Causal Methods [35].

Emerging ML deployment paradigms. Multi-model serving [2] enables more efficient usage of GPUs. This feature enables the simultaneous loading of several models on the same server instead of running one microservice per model, as in IPA. Although the focus of IPA was on the CPU service, using it on GPUs and containerized platforms is not straightforward. To our knowledge, there is no built-in mechanism for sharing GPUs on mainstream container orchestration frameworks like Kubernetes. Making the IPA formulation consistent with the sharing of GPUs and also considering interference between multiple models on the scheduler [49] as part of the IPA is a potential future extension.

P3.8 ACKNOWLEDGMENTS

This work has been supported, in part, by the National Science Foundation (Awards 2007202, 2107463, 2038080, and 2233873), Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - Project-ID 210487104 - SFB 1053, and Roblox Corporation. The authors also thank Chameleon Cloud for providing cloud resources for the experiments.

P3.9 APPENDIX

P3.10 PIPELINES STAGES SPECIFICATIONS

List of models used per stage of the pipeline with their specifications. BA in the following tables refers to the Base Allocation explained in Section P3.4.2 in terms of CPU cores.

Object Detection

Performance Measure: Mean Average Precision (mAP)

Number of Variants: Five

Source: Ultralytics YOLOV5 [67]

Threshold: 4 RPS

Table P3.7: Object Detection Task Models

Model	Params (M)	BA	mAP
YOLOv5n	1.9	1	45.7
YOLOv5s	7.2	1	56.8
YOLOv5m	21.2	2	64.1
YOLOv5l	46.5	4	67.3
YOLOv5x	86.7	8	68.9

Object Classification

Performance Measure: Accuracy

Number of Variants: 5

Source: Torchvision [48]

Threshold: 4 RPS

Audio

Performance Measure: Word Error Rate (WER) [2]

2 WER is the primary metric for evaluating audio-to-text models, however, to preserve the higher means better property mentioned in Section P3.4.1 we used the 1-WER which is similar to the accuracy measure for other types of model.

Table P3.8: Object Classification Task Models

Model	Params (M)	BA	Accuracy
ResNet18	11.7	1	69.75
ResNet34	21.8	1	73.31
ResNet50	25.5	1	76.13
ResNet101	44.54	1	77.37
ResNet152	60.2	2	78.31

Number of Variants: Five

Source: HuggingFace

HuggingFace Source: facebook

Threshold: 1 RPS

Table P3.9: Audio Task Models

Model	Params (M)	BA	1 - WER
s2t-small-librispeech	29.5	1	58.72
s2t-medium-librispeech	71.2	2	64.88
wav2vec2-base	94.4	2	66.15
s2t-large-librispeech	267.8	4	66.74
wav2vec2-large	315.5	8	72.35

Question Answering

Performance Measure: F1 Score

Number of Variants: Two

Source: HuggingFace

HuggingFace Source: depeest

Threshold: 1 RPS

Table P3.10: Question Answering Task Models

Model	Params (M)	BA	F1 Score
roberta-base	277.45	1	77.14
roberta-large	558.8	1	83.79

Summarisation

Performance Measure: Recall-Oriented Understudy for Gisting Evaluation (ROUGE-L)

Number of Variants: Six

Source: HuggingFace

HuggingFace Source: sshleifer

Threshold: 5 RPS

Table P3.11: Summarisation Task Models

Model	Params (M)	BA	ROUGE-L
distilbart-1-1	82.9	1	32.26
distilbart-12-1	221.5	2	33.37
distilbart-6-6	229.9	4	35.73
distilbart-12-3	255.1	8	36.39
distilbart-9-6	267.7	8	36.61
distilbart-12-6	305.5	16	36.99

Sentiment Analysis

Performance Measure: Accuracy

Number of Variants: Three

Source: HuggingFace

HuggingFace Source: Souvikcmsa

Threshold: 1 RPS

Table P3.12: Sentiment Analysis Task Models

Model	Params (M)	BA	Accuracy
DistillBerT	66.9	1	79.6
Bert	109.4	1	79.9
Roberta	355.3	1	83

Language Identification Task Models

Performance Measure: Accuracy

Number of Variants: One

Source: HuggingFace

HuggingFace Source: dinalzein

Threshold: 4 RPS

Table P3.13: Language Identification Task Models

Model	Params (M)	BA	Accuracy
roberta-base-finetuned	278	1	79.62

Neural Machine Translation

Performance Measure: Bilingual Evaluation Understudy (BELU)

Number of Variants: Two

Source: HuggingFace

HuggingFace Source: Helsinki-NLP

Threshold: 4 RPS

Table P3.14: Neural Machine Translation Task Models

Model	Params (M)	BA	Accuracy
opus-mt-fr-en	74.6	4	33.1
opus-mt-tc-big-fr-en	230.6	8	34.4

P3.11 CONSTANT MULTIPLIERS VALUES

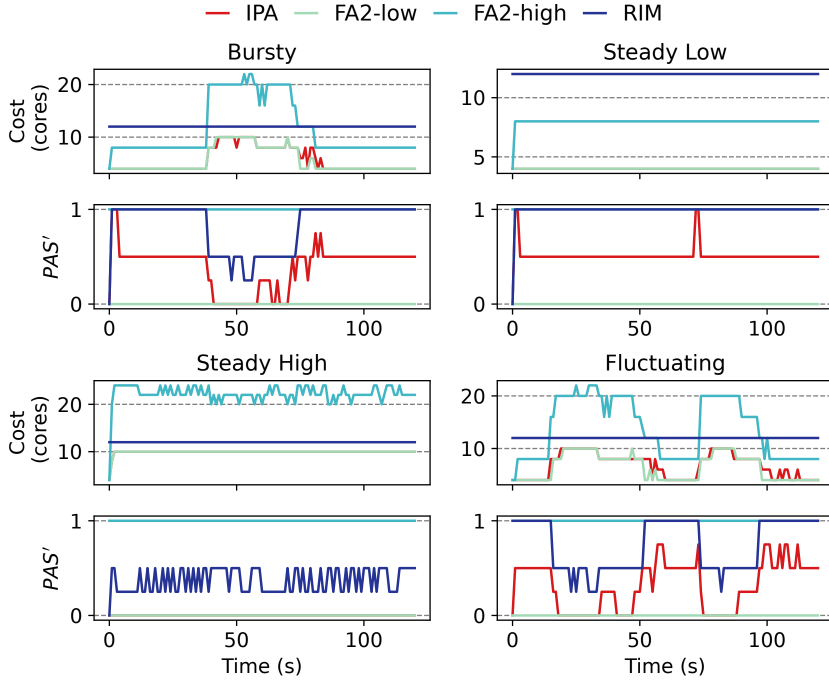
In this part, we have presented the values used for the multiplier α, β, and δ in Equation P3.9. These values were applied in the experiments discussed in Section P3.5.2. It is important to mention that because various objectives (like accuracy and cost) have different scales, the multiplier's scale is adjusted accordingly. We empirically found the best values for each objective multiplier.

Table P3.15: Pipelines objectives multiplier values

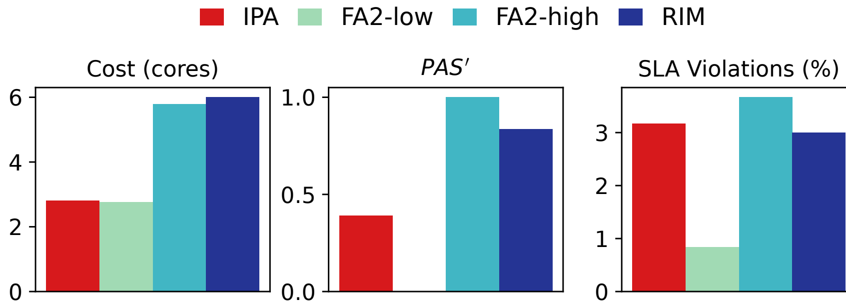
Pipeline	α	β	δ
Video	2	1	0.000001
Audio-qa	10	0.5	0.000001
Audio-sent	30	0.5	0.000001
Sum-qa	10	0.5	0.000001
NLP	40	0.5	0.000001

P3.12 ALTERNATE INFERENCE PIPELINE ACCURACY DEFINITION

In this section, we have presented results for an alternative accuracy metric in inference pipelines that we will refer to as PAS'. Our goal was to empirically show that two different accuracy metrics were able to provide similar results. To define a metric over the accuracy in a pipeline, we first sort the accuracy of each stage's model variants from lowest to highest. Then, we assign a zero scale to the least accurate model and one to the most accurate model (normalizing the accuracy). Intermediate model variants are assigned scaled accuracy values between zero and one, proportionally aligned with their rankings in the ordered list. For example, if three model variants exist, the model scaled accuracy is assigned 0, 0.5, and 1. The overall accuracy over the pipeline is considered as the sum of each stage's model variants' scaled accuracy value. For example, a two-stage pipeline with three model variants per stage and the second most accurate model chosen in each pipeline will have an end-to-end accuracy rank of 0.5 + 0.5 = 1. As a consequence, this will also change Equation P3.8 for the accuracy objective to sum up the normalized accuracy at each step instead of multiplication. Also, the accuracy values of each model and stage $a_{s,m}$ are the normalized accuracy explained rather than the actual



(a) Temporal analysis under different workloads.



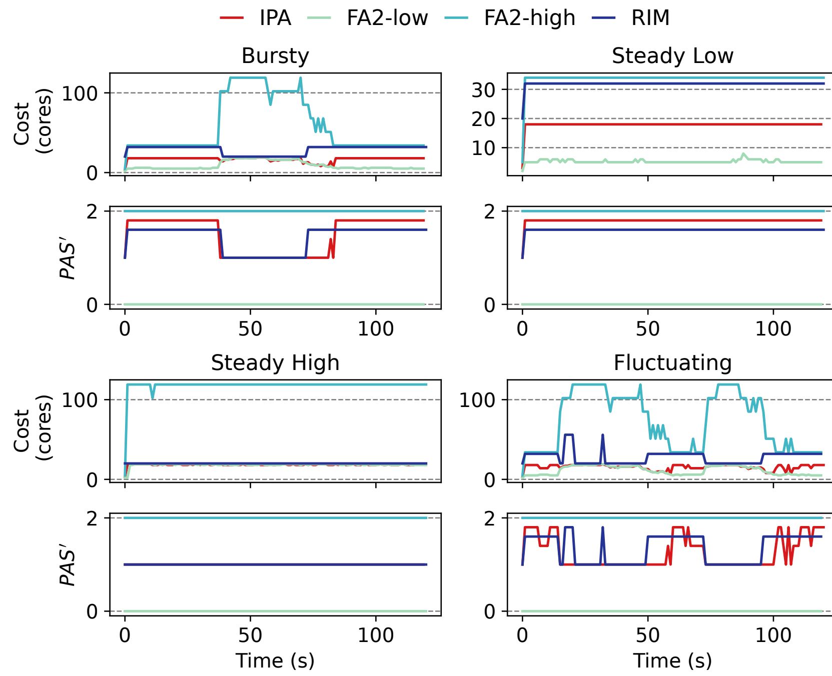
(b) Average analysis on bursty workload.

Figure P3.17: Performance analysis of the Video pipeline.

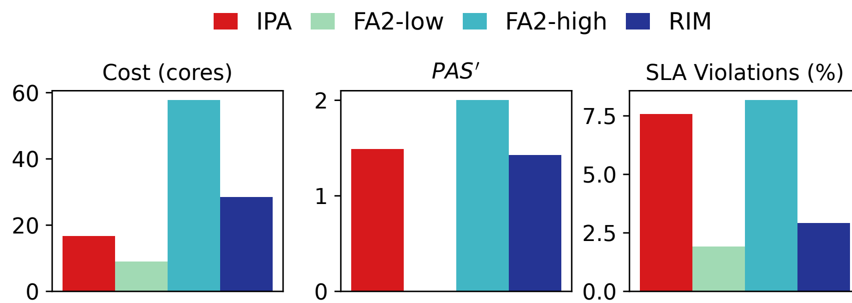
accuracy values. This transforms Equation P3.8 into Equation P3.11. This PAS' replaces the PAS metric in Equation P3.9. The rest of the IP formulation presented in Section P3.4 will remain the same.

$$PAS' = \sum_{s \in P} (\sum_{m \in M_s} a_{s,m} \cdot I_{s,m}) \tag{P3.11}$$

We replicated the end-to-end experiments outlined in Section P3.5.1 by substituting the accuracy metric with a new metric across all pipelines. In all cases, this alternative metric exhibited the same trend of the multiplication heuristic results across different methods. Figures P3.17 and P3.18 present the results obtained using this novel accuracy metric.



(a) Temporal analysis under different workloads.



(b) Average analysis on bursty workload.

Figure P3.18: Performance analysis of the Sum-qa pipeline.

A comparison with the results in Section P3.5.2, specifically Figures P3.8 and P3.11, reveals that both sets of results align in terms of resource allocation and accuracy optimization.

REFERENCES

[1] *AI Techniques in Medical Imaging May Lead to False Positives and False Negatives*. https://tinyurl.com/628z9tn4. Published on May 12, 2020. News-Medical.net.

[2] Sherif Akoush, Andrei Paleyes, Arnaud Van Looveren, and Clive Cox. "Desiderata for next generation of ML model serving." In: *arXiv preprint arXiv:2210.14665* (2022).

[3] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. "Batch: Machine learning inference serving on serverless platforms with adaptive batching." In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2020, pp. 1–15.

[4] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics." In: *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 2017, pp. 469–482.

[5] archiveteam. *Archiveteam-twitter-stream-2021-08*. https://archive.org/details/archiveteam-twitter-stream-2021-08. 2021.

[6] The Prometheus Authors. *Prometheus monitoring and alerting toolkit*. https://prometheus.io/. Accessed on 30.01.2024. 2024.

[7] Jeff Bar. *Amazon EC2 ML inference*. https://tinyurl.com/5n8yb5ub. 2019.

[8] Giovanni Bartolomeo, Simon Bäurle, Nitinder Mohan, and Jörg Ott. "Oakestra: An orchestration framework for edge computing." In: *Proceedings of the SIGCOMM'22 Poster and Demo Sessions*. 2022, pp. 34–36.

[9] Vivek M Bhasi, Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita Das. "Kraken: Adaptive container provisioning for deploying dynamic DAGs in serverless platforms." In: *Proceedings of the ACM Symposium on Cloud Computing*. 2021, pp. 153–167.

[10] Stephen P Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[11] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. "Language models are few-shot learners." In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

[12] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. "Once for All: Train One Network and Specialize it for Efficient Deployment." In: *International Conference on Learning Representations*. 2020.

[13] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. "Serving heterogeneous machine learning models on multi-GPU servers with spatio-temporal sharing." In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 2022, pp. 199–216.

[14] clarifai. *Clarifai*.

[15] CNBC. *Tesla crash that killed two men*. https://tinyurl.com/mr25a5mv. Published on April 18, 2021.

[16] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. "InferLine: Latency-aware provisioning and scaling for prediction serving pipelines." In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. 2020, pp. 477–491.

[17] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion 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.

[18] Tathagata Das, Yuan Zhong, Ion Stoica, and Scott Shenker. "Adaptive stream processing using dynamic batch sizing." In: *ACM Symposium on Cloud Computing (SoCC)*. 2014, pp. 1–13.

[19] Ruofei Du, Na Li, Jing Jin, Michelle Carney, Scott Miles, Maria Kleiner, Xiuxiu Yuan, Yinda Zhang, Anuva Kulkarni, Xingyu Liu, et al. "Rapsai: Accelerating Machine Learning Prototyping of Multimedia Applications through Visual Programming." In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 2023, pp. 1–23.

[20] Andre Esteva, Alexandre Robicquet, Bharath Ramsundar, Volodymyr Kuleshov, Mark DePristo, Katherine Chou, Claire Cui, Greg Corrado, Sebastian Thrun, and Jeff Dean. "A guide to deep learning in healthcare." In: *Nature medicine* 25.1 (2019), pp. 24–29.

[21] Maria Fazio, Antonio Celesti, Rajiv Ranjan, Chang Liu, Lydia Chen, and Massimo Villari. "Open issues in scheduling microservices in the cloud." In: *IEEE Cloud Computing* 3.5 (2016), pp. 81–88.

[22] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. "Dhalion: self-regulating stream processing in heron." In: *Very Large Data Bases (PVLDB)* 10.12 (2017), pp. 1825–1836.

[23] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. "A survey of quantization methods for efficient neural network inference." In: *arXiv preprint arXiv:2103.13630* (2021).

[24] Alim Ul Gias, Giuliano Casale, and Murray Woodside. "ATOM: Model-driven autoscaling for microservices." In: *IEEE International Conference on Distributed Computing Systems (ICDCS)*. 2019, pp. 1994–2004.

[25] *gRPC*. https://grpc.io. Accessed on 29.10.2021.

[26] Arpan Gujarati, Sameh Elnikety, Yuxiong He, Kathryn S McKinley, and Björn B Brandenburg. "Swayam: distributed autoscaling to meet slas of machine learning inference services with resource efficiency." In: *ACM/IFIP/USENIX Middleware Conference*. 2017, pp. 109–120.

[27] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. "Serving DNNs like Clockwork: Performance Predictability from the Bottom Up." In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 443–462. ISBN: 978-1-939133-19-9.

[28] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R Das. "Cocktail: A multidimensional optimization for model serving in cloud." In: *USENIX NSDI*. 2022, pp. 1041–1057.

[29] Udit Gupta, Samuel Hsia, Jeff Zhang, Mark Wilkening, Javin Pombra, Hsien-Hsin Sean Lee, Gu-Yeon Wei, Carole-Jean Wu, and David Brooks. "RecPipe: Co-designing models and hardware to jointly optimize recommendation quality and performance." In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 2021, pp. 870–884.

[30] Harvard Business Review. *When Machine Learning Goes Off the Rails*. https://hbr.org/2021/01/when-machine-learning-goes-off-the-rails. 2021.

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition." In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[32] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory." In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[33] Yitao Hu, Rajrup Ghosh, and Ramesh Govindan. "Scrooge: A cost-effective deep learning inference system." In: *Proceedings of the ACM Symposium on Cloud Computing*. 2021, pp. 624–638.

[34] Yitao Hu, Weiwu Pang, Xiaochen Liu, Rajrup Ghosh, Bongjun Ko, Wei-Han Lee, and Ramesh Govindan. "Rim: Offloading Inference to the Edge." In: *Proceedings of the International Conference on Internet-of-Things Design and Implementation*. 2021, pp. 80–92.

[35] Md Shahriar Iqbal, Rahul Krishna, Mohammad Ali Javidian, Baishakhi Ray, and Pooyan Jamshidi. "Unicorn: Reasoning about configurable system performance through the lens of causality." In: *Proceedings of the Seventeenth European Conference on Computer Systems*. 2022, pp. 199–217.

[36] *Istio*.

[37] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. "Chameleon: Scalable adaptation of video analytics." In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 2018, pp. 253–266.

[38] Maree Johnson, Samuel Lapkin, Vanessa Long, Paula Sanchez, Hanna Suominen, Jim Basilakis, and Linda Dawson. "A systematic review of speech recognition technology in health care." In: *BMC medical informatics and decision making* 14.1 (2014), pp. 1–14.

[39] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. "Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows." In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2018, pp. 783–798.

[40] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. "Grandslam: Guaranteeing SLAs for jobs in microservices execution frameworks." In: *Proceedings of the Fourteenth EuroSys Conference 2019*. 2019, pp. 1–16.

[41] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. "Lessons learned from the Chameleon testbed." In: *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, 2020.

[42] *Kubernetes Python client*. https://github.com/kubernetes-client/python.

[43] Darpan Kulshreshtha. *10 Instances Where AI Went Wrong*. https://tinyurl.com/2p8ywtpd. Published on LinkedIn.

[44] Hyoukjun Kwon, Krishnakumar Nair, Jamin Seo, Jason Yik, Debabrata Mohapatra, Dongyuan Zhan, Jinook Song, Peter Capak, Peizhao Zhang, Peter Vajda, et al. "XRBench: An extended reality (XR) machine learning benchmark suite for the metaverse." In: *arXiv preprint arXiv:2211.08675* (2022).

[45] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. "PRETZEL: Opening the black box of machine learning prediction serving systems." In: *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 2018, pp. 611–626.

[46] Nima Mahmoudi and Hamzeh Khazaei. "Performance modeling of serverless computing platforms." In: *IEEE Transactions on Cloud Computing* (2020), pp. 1–15.

[47] Ankur Mallick, Kevin Hsieh, Behnaz Arzani, and Gauri Joshi. "Matchmaker: Data Drift Mitigation in Machine Learning for Large-Scale Systems." In: *Proceedings of Machine Learning and*

Systems. Ed. by D. Marculescu, Y. Chi, and C. Wu. Vol. 4. 2022, pp. 77–94.

[48] Sébastien Marcel and Yann Rodriguez. "Torchvision the machine-vision package of torch." In: *Proceedings of the 18th ACM international conference on Multimedia*. 2010, pp. 1485–1488.

[49] Daniel Mendoza, Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. "Interference-aware scheduling for inference serving." In: *Proceedings of the 1st Workshop on Machine Learning and Systems*. 2021, pp. 80–88.

[50] "MinIO." In.

[51] "MLServer." In.

[52] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. "Deep learning recommendation model for personalization and recommendation systems." In: *arXiv preprint arXiv:1906.00091* (2019).

[53] Nvidia. `https://docs.nvidia.com/metropolis/deepstream/dev-guide/text/DS_Zero_Coding_Sample_Graphs.html`. Deepstream reference graphs.

[54] "Nvidia DeepStream." In.

[55] Alex Poms, Will Crichton, Pat Hanrahan, and Kayvon Fatahalian. "Scanner: Efficient Video Analysis at Scale." In: *ACM Trans. Graph.* 37.4 (July 2018), 138:1–138:13. ISSN: 0730-0301. DOI: `10.1145/3197517.3201394`.

[56] Kamran Razavi, Manisha Luthra, Boris Koldehofe, Max Mühlhäuser, and Lin Wang. "FA2: Fast, Accurate Autoscaling for Serving Deep Learning Inference with SLA Guarantees." In: *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2022, pp. 146–159. DOI: `10.1109/RTAS54340.2022.00020`.

[57] *Reduce False Positives with Machine Learning*. `https://complyadvantage.com/insights/reduce-false-positives-with-machine-learning/`. Accessed on July 27, 2023. ComplyAdvantage.

[58] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. "INFaaS: Automated model-less inference serving." In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 2021, pp. 397–411.

[59] Francisco Romero, Mark Zhao, Neeraja J Yadwadkar, and Christos Kozyrakis. "Llama: A Heterogeneous & Serverless Framework for Auto-Tuning Video Analytics Pipelines." In: *ACM Symposium on Cloud Computing (SoCC)*. 2021, pp. 1–17.

[60] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, et al. "Autopilot: Workload autoscaling at Google." In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020, pp. 1–16.

[61] Mehran Salmani, Saeid Ghafouri, Alireza Sanaee, Kamran Razavi, Max Mühlhäuser, Joseph Doyle, Pooyan Jamshidi, and Mohsen Sharifi. "Reconciling High Accuracy, Cost-Efficiency, and Low Latency of Inference Serving Systems." In: *Proceedings of the 3rd Workshop on Machine Learning and Systems*. 2023, pp. 78–86.

[62] *Seldon*. https://github.com/SeldonIO/seldon-core. 2023.

[63] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. "Nexus: A GPU cluster engine for accelerating DNN-based video analysis." In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 2019, pp. 322–337.

[64] Vikram Sreekanti, Harikaran Subbaraj, Chenggang Wu, Joseph E Gonzalez, and Joseph M Hellerstein. "Optimizing prediction serving on low-latency serverless dataflow." In: *arXiv preprint arXiv:2007.05832* (2020).

[65] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. "Cloudburst: Stateful Functions-as-a-Service." In: *Very Large Data Bases (PVLDB)* 13.12 (July 2020), pp. 2438–2452. ISSN: 2150-8097. DOI: 10.14778/3407790.3407836.

[66] NVIDIA TensorRT. *Programmable Inference Accelerator, 2018*.

[67] ultralytics. *YOLOv5*. https://github.com/ultralytics/yolov5. Accessed on 30.01.2024. 2024.

[68] Luping Wang, Lingyun Yang, Yinghao Yu, Wei Wang, Bo Li, Xianchao Sun, Jian He, and Liping Zhang. "Morphling: Fast, Near-Optimal Auto-Configuration for Cloud-Native Model Serving." In: *Proceedings of the ACM Symposium on Cloud Computing*. 2021, pp. 639–653.

[69] Wei Wang, Sheng Wang, Jinyang Gao, Meihui Zhang, Gang Chen, Teck Khim Ng, and Beng Chin Ooi. "Rafiki: Machine learning as an analytics service system." In: *arXiv preprint arXiv:1804.06087* (2018).

[70] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. "MLaaS in the wild: Workload analysis and scheduling in large-scale heterogeneous GPU clusters." In: *19th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 22)*. 2022.

[71] Tongshuang Wu, Ellen Jiang, Aaron Donsbach, Jeff Gray, Alejandra Molina, Michael Terry, and Carrie J Cai. "Promptchainer: Chaining large language model prompts through visual programming." In: *CHI Conference on Human Factors in Computing Systems Extended Abstracts*. 2022, pp. 1–10.

[72] Tongshuang Wu, Michael Terry, and Carrie Jun Cai. "Ai chains: Transparent and controllable human-ai interaction by chaining large language model prompts." In: *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 2022, pp. 1–22.

[73] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. "Mark: Exploiting cloud services for cost-effective, SLO-aware machine learning inference serving." In: *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 2019, pp. 1049–1062.

[74] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J Freedman. "Live video analytics at scale with approximation and delay-tolerance." In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 2017, pp. 377–392.

[75] Jeff Zhang, Sameh Elnikety, Shuayb Zarar, Atul Gupta, and Siddharth Garg. "Model-switching: Dealing with fluctuating workloads in machine-learning-as-a-service systems." In: *12th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. 2020.

[76] Tianlei Zheng, Xi Zheng, Yuqun Zhang, Yao Deng, ErXi Dong, Rui Zhang, and Xiao Liu. "SmartVM: a SLA-aware microservice deployment framework." In: *World Wide Web* 22.1 (2019), pp. 275–293.

[77] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. "Overload control for scaling wechat microservices." In: *ACM Symposium on Cloud Computing (SoCC)*. 2018, pp. 149–161.

[78] Zhuangzhuang Zhou, Yanqi Zhang, and Christina Delimitrou. "AQUATOPE: QoS-and-Uncertainty-Aware Resource Management for Multi-stage Serverless Workflows." In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 2022, pp. 1–14.

P4

BISCALE: INTEGRATING HORIZONTAL AND VERTICAL SCALING FOR INFERENCE SERVING SYSTEMS

ABSTRACT

Inference serving is of great importance in deploying machine learning models in real-world applications, ensuring efficient processing and quick responses to inference requests. However, managing resources in these systems poses significant challenges, particularly in maintaining performance under varying and unpredictable workloads. Two primary scaling strategies, horizontal and vertical scaling, offer different advantages and limitations. Horizontal scaling adds more instances to handle increased loads but can suffer from cold start issues and increased management complexity. Vertical scaling boosts the capacity of existing instances, allowing for quicker responses but is limited by hardware and model parallelization capabilities.

This paper introduces Biscale, a system designed to leverage the benefits of both horizontal and vertical scaling in inference serving systems. Biscale employs a two-stage autoscaling strategy: initially using in-place vertical scaling to handle workload surges and then switching to horizontal scaling to optimize resource efficiency once the workload stabilizes. The system profiles the processing latency of deep learning models, calculates queuing delays, and employs different dynamic programming algorithms to solve the joint horizontal and vertical scaling problem optimally based on the workload situation. Extensive evaluations with real-world workload traces demonstrate over 10× SLO violation reduction compared to the state-of-the-art

horizontal or vertical autoscaling approaches while maintaining resource efficiency when the workload is stable.

P4.1 INTRODUCTION

Cloud-based deep learning (DL) inference is a common component in modern intelligent applications and services, often involving multiple DL models interconnected in a dataflow multi-model (pipeline) system [42]. An example is a real-time video analytics application for traffic management, which may include models for video frame extraction, object detection, object classification, and tracking [18]. The performance of such systems is mostly evaluated based on two indicators: user satisfaction, quantified through Service Level Objectives (SLOs) mainly on the end-to-end latency [26], and resource efficiency [57], referring to the optimal utilization of computational resources. These performance indicators ensure the delivery of high-quality results while efficiently using resources, which is crucial for the scalability and sustainability of cloud-based DL applications [2, 28, 35, 36, 42, 57].

Efficient resource management in DL inference serving systems is crucial to maintaining system performance, especially under variable and unpredictable workloads [41, 47]. Two primary scaling strategies are often used to manage these dynamic workloads: horizontal and vertical scaling. Horizontal scaling involves the addition of more instances of the same DL model to handle increased loads, facilitating workload distribution without altering the resource configuration of existing instances. By distributing the workload across several instances, horizontal scaling can handle large workloads and maintain performance as demands grow. However, bringing up new instances in horizontal scaling involves cold start issues [33, 35, 39, 43] due to booting up additional instances, configuring them, and joining them into the system, which can take several seconds to minutes. The cold start issue reduces system responsiveness when the workload changes unpredictably, leading to SLO violations and reduced user satisfaction. Additionally, horizontal scaling increases system management's complexity, requiring more sophisticated load balancing.

Vertical scaling, conversely, focuses on adding resources such as CPU cores to existing DL model instances, thereby boosting the capacity of existing instances to handle more tasks simultaneously. A recently announced feature named in-place vertical scaling [5] by Kubernetes [22]—a dominant open source system to orchestrate and manage the containers in the system—allows additional resources to be added to existing instances without the need for rebooting or downtime, offering a significant advantage concerning the cold start issue. Additional computing resources, such as extra CPU cores, allow inference requests to be batched and processed concurrently, improving the system throughput. This capability enables a much quicker response to increased load compared to setting up new instances as required in horizontal scaling. While vertical scaling can provide rapid speed-up by instantly increasing the resources of a single instance, it is often limited by the physical capabilities of the hardware and the parallelizability of the DL model. Once those limits are reached, no further speed-up is possible, thus reaching the maximum throughput. In contrast, horizontal scaling can continue to expand by adding more instances. Furthermore, vertical scaling can lead to higher costs due to the need for high-end hardware, and it presents a single point of failure risk, where if the single instance fails, it can significantly impact system stability.

Existing inference serving systems mainly focus on horizontal scaling mainly due to the benefits of the scalability of horizontal scaling and the limitations of vertical scaling. Studies like INFaaS [43] focus on individual DL models where horizontal scaling is used in response to workload changes by bringing up new virtual machines. Others have considered DL pipelines and use horizontal scaling to optimize resource costs by switching between hardware types or model variants when the workload changes [12, 25, 42]. All of the mentioned approaches suffer from cold start issues when facing dramatic workload increases. Sponge [41] uses in-place vertical scaling and changes the CPU allocation of the running instance to absorb the sudden changes in the workload. However, they only consider a single DL model and ignore the challenges behind the data dependency of pipeline inference serving systems. Moreover, their approach suffers from SLO violations under high request rates that are beyond the capabilities of one powerful instance due to parallelization and physical hardware limitations, as discussed above. In short, existing autoscaling solutions suffer from non-responsiveness under high workload variability,

and to compensate for that, they require a significant level of over-provisioning. Ideal autoscaling should be responsive, meaning it can immediately react to sudden workload changes, and resource-efficient, avoiding over-provisioning, while limiting SLO violations by ensuring end-to-end inference latency.

In this work, we explore how to reduce SLO violations and resource costs by studying an autoscaling mechanism that combines both vertical and horizontal scaling mechanisms and leverages the benefits of both mechanisms in the pipeline inference serving system. Based on the insights from such an exploration, we design an autoscaler, named Biscale, that uses a two-stage autoscaling strategy, where it first leverages the responsiveness characteristics of in-place vertical scaling to absorb the unpredictable surge of requests in the workload. Second, Biscale performs horizontal scaling to optimize resource efficiency when the workload tends to stabilize by transitioning to a set of less powerful instances.

To be precise for the vertical scaling strategy, after getting the application pipeline with its SLO, Biscale addresses the challenges regarding the in-place vertical scaling decision-making in pipelines by first profiling the processing latency of all the DL models with respect to different CPU and batch size allocations, and calculating the queuing delay based on the given batch size and the arrival rate. Next, Biscale finds the right amount of resources to serve the incoming workload (avoiding under- or over-provisioning) for all the models in the system, in a single shot, by encapsulating the autoscaling problem in an Integer Program (IP) and solving it using dynamic programming. In contrast to existing solutions where heuristics are used for scaling decision-making that results in sub-optimal solutions, we use dynamic programming to solve the IPs which generates optimal solutions by breaking the problem into simpler subproblems and solving each subproblem only once, storing their solutions to avoid redundant calculations. For the transitions between vertical and horizontal scaling strategies, Biscale analyzes why an autoscaler needs to switch between scaling strategies, how to transit between them, and when to do it. For that, Biscale designs a state transition policy to decide when to use which autoscaling strategy and uses it during runtime to enable responsiveness and resource efficiency while guaranteeing end-to-end inference latency.

In short, this paper makes the following contributions. After presenting the motivation and identifying the challenges (§P4.2), we

- present the design of Biscale, including its overall architecture and system components (§P4.3);
- introduces an integer program to capture the vertical and horizontal resource scaling problem in DL inference serving systems and present our solutions based on dynamic programming (§P4.4);
- build a system prototype for Biscale and evaluate it with real-world workload traces (§P4.6). Overall, Biscale reduces the SLO violation by over 10× compared to the baselines.

§P4.7 summarizes related work. §P4.8 draws final conclusions.

P4.2 BACKGROUND AND MOTIVATION

Inference services, a critical component of online platforms and mainly composed of multiple models that are chained together, have unique characteristics that make them both latency-sensitive and resource-intensive. The latency sensitivity of these services arises from their interaction with online users. Users expect quick responses when interacting with these services, making delay or latency undesirable. On the other hand, inference services are also resource-intensive due to the heavy computations they need to perform [12, 27]. The dual characteristics of latency sensitivity and resource-intensiveness make designing and managing inference services challenging. It is essential to find a balance between providing quick responses to user interactions and managing the heavy computational resources required by these services.

Figure P4.1 depicts the vertical and horizontal scaling response time of a ResNet car detection model [29] to the workload changes under the SLO of 1000 ms. The number of requests starts from 20 requests per second (RPS) and then rises 6× (to 120 RPS) for 5 seconds and then drops back to 20 RPS. If the system captures the workload changes immediately and uses horizontal scaling, the new instances will become available after 5-6 seconds as evaluated in [42]. In this scenario, all the requests (100) in that short period will violate their SLO, and the scal-

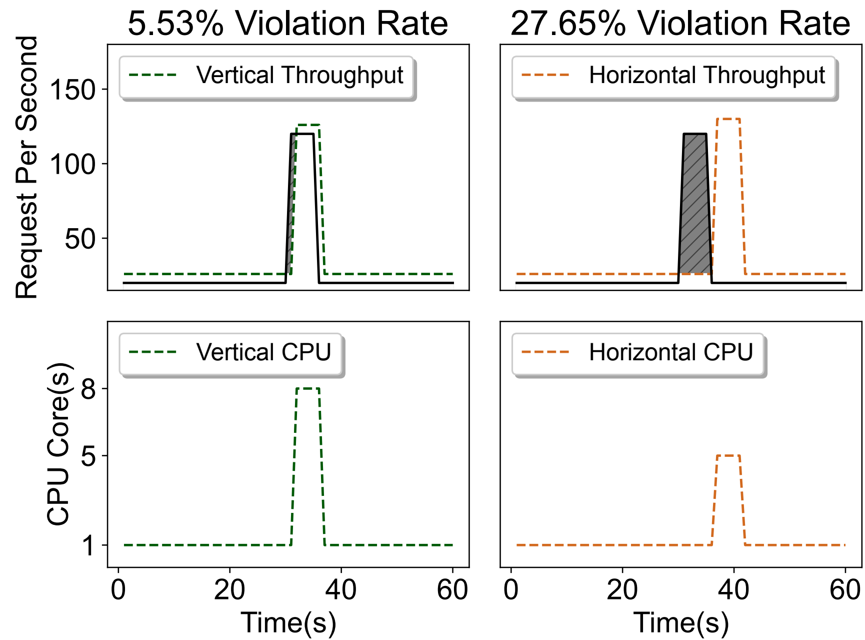


Figure P4.1: Vertical scaling vs. horizontal scaling reaction time in case of workload bursts. Horizontal scaling is not responsive. The gray area indicates SLO violation.

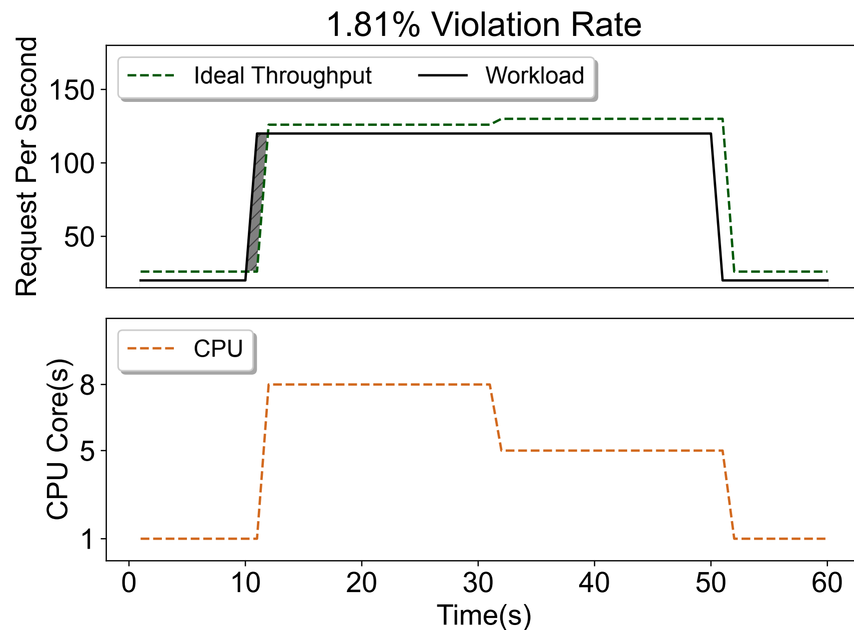


Figure P4.2: Using vertical and horizontal scaling jointly to absorb bursts and reduce operational costs. Vertical scaling provides responsiveness, while horizontal scaling provides cost efficiency. The gray area indicates SLO violation.

ing decisions cannot capture the burstiness and waste resources until a new scaling decision is made. On the other hand, by using in-place vertical scaling, we can change the computational resources of the

Table P4.1: Comparison of Biscale with previous works; Pipeline: A chain of models or just a single model? Resource Efficiency: Does this work use horizontal scaling for the highest resource efficiency? Responsiveness: Does this approach use in-place vertical scaling to respond quickly to the changes in the workload? (∗) uses model-variants as a way of vertical scaling.

System	Pipeline	Resource Efficiency	Responsiveness
INFaaS	✗	✓	✗∗
InferLine	✓	✓	✗
GrandSLAm	✓	✗	✗
FA2	✓	✓	✗
Scrooge	✓	✓	✗
Cocktail	✗	✗	✓
InfAdapter	✗	✓	✗∗
IPA	✓	✓	✗∗
Sponge	✗	✗	✓
Biscale	✓	✓	✓

available DL models and absorb the burstiness almost instantly (with an overhead of less than 100 ms). Next, as the workload decreases, we reduce the allocated resources, reducing operational costs. Using in-place vertical scaling for a short duration of 5 seconds, we could reduce the SLO violation rate by roughly 5×. It should be noted that the amount of resources in vertical scaling is higher than in horizontal scaling, as discussed in the previous section, meaning that horizontal scaling provides higher throughput. The next question arises: can we switch to horizontal scaling after we absorb the burstiness with vertical scaling? Figure P4.2 answers this question by suggesting that when the workload becomes stable again (the stabilization is discussed in Section P4.5.1.2), we can switch to horizontal scaling and reduce the operational cost by 40% in this simple car detection scenario. Therefore, a combination of vertical and horizontal scaling can react to workload changes fast enough, simultaneously reducing total resource consumption.

Table P4.1 presents an overview of related latency-sensitive inference serving systems. While most of the previous works use only horizontal scaling or model-variant changes for resource efficiency, they all suffer significantly from rapid changes in the workload if they do not over-provision their resources, which directly impacts operational costs negatively. On the other hand, the only work that considers re-

sponsiveness using vertical scaling in terms of changing the resources (not the model variant or model ensemble) becomes unpractical when the workload surpasses the capacity of one DL model with the highest possible resource allocation (hardware limitation). Furthermore, it does not consider pipeline, hence the challenges of data dependency using in-place vertical scaling are not addressed.

P4.3 BISCALE

In this section, we first discuss the challenges of designing a new autoscaler that benefits from both vertical and horizontal autoscaling mechanisms. Next, we give an overview of Biscale, its architecture, and the main components. Finally, we discuss the assumptions and the limitations of Biscale.

P4.3.1 *Challenges*

We have identified the following challenges in designing a new strategy that symbiotically combines both horizontal and vertical scaling:

Data dependency [DD]. The resource allocation decision becomes even more challenging when DL models have data dependencies (pipelines) in the serving system since the resource scaling decision for one DL model may affect downstream DL models. Therefore, we need a fast enough approach to find solutions for all the DL models in the pipeline in a single shot.

Adaptation period [AP]. After using vertical scaling, we need to identify when is a good time to switch to horizontal scaling to save resources. To be precise, we need to find an answer to this question: "How to correctly predict that the workload is stable enough that we do not need to allocate extra resources for vertical scaling?"

Hardware limitation for vertical scaling [HL]. One of the drawbacks of vertical scaling compared to horizontal scaling is that the DL model's maximum capacity is bound to the hardware capabilities. Every hardware has a maximum limit for processing power; once it is reached, the DL model cannot scale up any further. Thus, we must find a place-

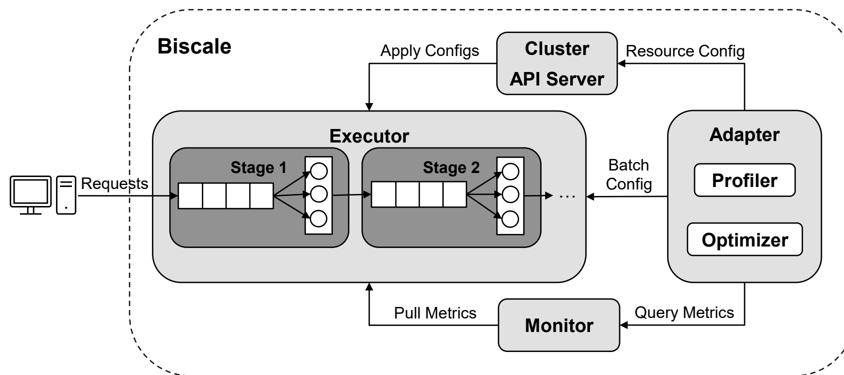


Figure P4.3: An overview of the Biscale architecture. The executor receives the requests and processes them. The monitor service collects the metric data from the stages. The optimizer makes both vertical and horizontal scaling decisions for the DL models. The Adapter enforces the scaling decisions by configuring the queues and the executor.

ment algorithm for assigning DL models in a pipeline in a distributed cluster.

Batch size [BS]. Apart from resource allocation and resource placement in inference serving systems, there is a dominant factor that directly affects the latency and throughput of DL models, named batch size. The batch size can be changed online to increase the DL model's throughput based on the demand at the cost of higher inference latency. As a result, we need to find the optimal batch size. Large batches can significantly impact the response time of requests within a batch. On the other hand, small batches might miss out on opportunities for improved throughput and cost efficiency.

P4.3.2 *System Design*

Figure P4.3 provides an overview of Biscale. The system comprises five main parts (profiler, executor, monitor, optimizer, and adapter). The profiler creates a new performance model for any registered DL model in the system. The stage component gets the requests and executes the DL model. The monitor component keeps track of the pipeline's request rate (workload distribution). The optimizer uses data from the monitor and the profiler components to find an optimal resource allocation solution based on in-place vertical or horizontal scaling.

Finally, the adapter enforces the decisions made by the optimizer to the system.

The **profiler** component sends profiling requests to the DL model with different configurations (different CPU and batch size allocations) (answering [BS]). It then records the latencies under each configuration and uses them to create the performance model of each DL model in the system. Furthermore, these latency profiles are used for in-place vertical scaling in the optimizer. Note that this component runs offline, and the performance profiles generated by this component will be used online.

The **executor** consists of multiple stages, each of which is composed of the queuing and the model processing parts. Each stage has one centralized queue and one or more processing model instances. The queuing component fetches the requests from the user or other stages, sends the arrival rate statistic to the optimizer, batches the requests based on the optimizer's decision, and sends the batches to their associated models for inference. It uses a round-robin approach to distribute batches among the processing instances (decided by the optimizer) in its stage. The processing instance has the DL model and the computing power (determined by the optimizer based on the incoming workload) to process the batches sent by the stage queuing component. After processing the batches, the instance sends the results to the next stage or the user. Both components keep track of the queuing and processing latencies. In the last stage, a query with the gathered latencies is generated and sent to the monitor component for further analysis.

The **monitor** gets the arrival rate statistics from the first stage queue and the total request latency from the last model instances. The optimizer will fetch the arrival rate statistics to find optimal resources and batch sizes for the whole system while guaranteeing end-to-end request latencies, and the total request latency will be used to report the system and the profiler's performance. To reduce the monitor overhead, we append the latencies to the same request as they pass through different pipeline stages.

The **optimizer** gets the arrival rate from the first queuing component and feeds it to the horizontal autoscaler. Next, it uses an LSTM predictor to estimate the highest workload in the following 10-second time window and feeds the predicted workload to the horizontal autoscaler

(answering [AP]). The horizontal scaling results are sent to the adapter if no additional resources are required. If not, the optimizer checks if there are enough physical hardware resources for using in-place vertical scaling to support the current workload (answering [HL]). If there are enough resources, the system uses in-place vertical scaling to respond to changes in the workload by finding the optimal resource allocation for all the DL models in the system in a single shot (answering [DD]). If not, the optimizer calculates the maximum throughput using vertical scaling and switches to horizontal scaling for the remaining requests.

The **adapter** is responsible for enforcing the configurations made by the optimizer to the cluster. The configurations for each DL model in the system are batch size, resource allocation of model instances, and the number of instances. After the new configuration is received from the optimizer, the adapter first compares the new set of configurations with the current state of configurations. If there are changes, the adapter makes two calls for each model's new configuration, one to the queue for adjusting the batch size and one to the executor to adjust the computing resources of the models (vertical scaling) or change the instance number of the model (horizontal scaling). The batch size will be immediately updated, and changes in the computing resources will be almost instant (less than 100 ms). The scale in/out of the instances will take seconds (roughly 5-6 seconds).

P4.3.3 *Assumptions and Limitations*

The in-place vertical scaling part of Biscale uses the vertical scaling feature of Kubernetes, which currently is on the alpha branch and only supports CPU and memory real-time modification. However, inference serving systems can benefit greatly from other accelerators, such as GPUs and TPUs, which, unfortunately, are not supported at this moment. Also, Biscale uses simple queuing management, whereas a more complex queuing simple may provide a more accurate queuing latency. Moreover, multiple variants for the same DL models with different properties, such as accuracy/latency trade-offs, can be leveraged to reduce costs further. Finally, we consider uniform SLOs, where all DL inference requests in the system have the same end-to-end latency SLO requirement.

P4.4 AUTOSCALING PROBLEM

In this section, we provide a mathematical representation of the joint horizontal-vertical autoscaling problem in an inference serving system to react to the changes in the workload. We encapsulate the joint autoscaling problem into an Integer Program (IP) that decides the batch size, computing resource, and the number of instances of each DL model in the system in a single shot. The goal of the IP is to minimize the cost while guaranteeing the requests SLO. It achieves it by using vertical scaling when there are sudden changes in the workload to capture the workload burstiness. Furthermore, when the workload is stable (see Section P4.5.1.2), the IP switches to horizontal scaling to save operational costs. We use different dynamic programming approaches to solve the IP based on the workload status.

P4.4.1 *Performance Profiling*

A DL model processes requests with different latencies based on the allocated computational resources and batch sizes. Previous works have shown that processing latency has a linear/quadratic relationship with batch size [31, 42]. Sponge [41] has shown that the processing latency relationship is inverse to the core allocation in inference serving systems. We follow the same guidelines and use the following formula to build a performance profile for each DL model in the system:

$$l(b,c) = \frac{\gamma \times b}{c} + \frac{\epsilon}{c} + \delta \times b + \eta. \tag{P4.1}$$

Formula P4.1 predicts the processing latency with the given batch size (b) and computing resources (c) of the DL model. We execute the DL model with different batch sizes and resource allocation configurations to find the static variables ($\gamma, \epsilon, \delta, \eta$) to get the processing latencies. We then use the latency data and fit the above formulation to get the DL performance profile. We perform the same approach offline for all the DL models in the system and create a unique performance profile for each.

All the used notations are available in Table P4.2.

Table P4.2: Notations

Symbol	Description
S	Set of all models for an application
SLO	Latency SLO of the application
n_s	number of instances of model $s \in S$
b_s	batch size of model $s \in S$
c_s	CPU core allocation of model $s \in S$
λ	Request arrival rate
$l_s(b,c)$	Processing time of model $s \in S$ with allocation core c and batch size b
$q_s(b)$	Queuing time of model $s \in S$ with batch size b
$h_s(b,c)$	Throughput of model $s \in S$ with allocation core c and batch size b

P4.4.2 *Queue*

Due to the differences in the number of DL instances, computing resources, and batch size, the DL model instances process requests with different latencies. Also, to form the batches to increase the system performance, the first request in the batch must wait for the arrival of the last request in the batch, causing an extra queuing delay. Because of the mentioned reasons, the requests in batches leave the queue at different speeds (i.e., the last request in the batch leaves the queue immediately, and the first request in the batch must wait the longest, hence it has the slowest speed in terms of leaving the queue), causing a dynamic queuing latency. FA2 [42] uses Equation (P4.2) for the worst-case scenarios. The worst-case scenario can happen when there are no available instances (out of the n instances for the same DL model) to process the current batch (meaning that all n instances are busy and the next batch needs to wait for one of the instances to be free ($l(b) - \frac{nb+1}{\lambda}$)) or the first request waits for the arrival of the last request in the same batch ($\frac{b-1}{\lambda}$). Therefore, the worst case is the maximum of these two equations, as shown in the following equation.

$$q(b,n) = \max\left(\frac{b-1}{\lambda}, l(b) - \frac{nb+1}{\lambda}\right). \tag{P4.2}$$

However, they do not consider the speed-up/down caused by changing the computing resources of the DL model instances, which directly

affects the queuing drainage. Therefore, we propose the following equation to incorporate the computing resources in the DL model as well:

$$q(b,c,n) = \max\left(\frac{b-1}{\lambda}, l(b,c) - \frac{nb+1}{\lambda}\right). \quad \text{(P4.3)}$$

Similar to FA2, we argue that the worst-case scenario happens for the latter scenario, where the first request needs to wait for the last request of the same batch due to the fact that by increasing the computing resources and reducing the processing latency, the DL models catch up with the workload. This means that the aggregate throughput of all the instances will equal or exceed the workload, i.e., $n \times h(b,c) \geq \lambda$, i.e., the second part of the above equation is always less than zero. Therefore, we use a simplified queuing latency prediction as:

$$q(b) = \frac{b-1}{\lambda}. \quad \text{(P4.4)}$$

P4.4.3 *Problem Formulation*

The goal of the optimizer is to decide the least amount of allocated resources of all instances ($\sum_{s \in S} n_s \times c_s$), subject to guaranteeing the end-to-end latency of requests and supporting the workload.

The end-to-end latency of a request is the aggregation of queuing and processing latencies of all the stages in the pipeline, which should be lower than the given SLO:

$$\text{End-to-End Latency} = \sum_{s \in S} l_s(b_s, c_s) + q_s(b_s) \leq SLO. \quad \text{(P4.5)}$$

To maintain system stability, the combined throughput of all instances of a stage must meet or exceed the request rate. Formally, for any stage, the product of its instance's throughput and the number of instances should be greater than or equal to the sum of request rates across all instances, i.e., $s \in S$, $h_s(b_s, c_s) \times n_s \geq \lambda$. This constraint ensures adequate provisioning for all stages, effectively managing the queuing of inference requests at each stage.

The problem can be formulated with the following IP:

$$\begin{aligned} \min \quad & \sum_{s \in S} n_s \times c_s \\ \text{subject to} \quad & \sum_{s \in S} l_s(b_s, c_s) + q_s(b_s) \leq SLO \\ & \lambda \leq h_s(b_s, c_s) \times n_s, \forall s \in S \\ & b_s, c_s, n_s \in \mathbb{Z}^+, \forall s \in S \end{aligned} \tag{P4.6}$$

The last constraint in Equation P4.6 states that the batch sizes and number of instances should be positive. Next, we solve the above IP with different approaches based on the current workload situation.

P4.4.4 *Optimizer*

Two dynamic variables per stage affect the total cost significantly and can be adjusted in the IP from the previous section, namely core allocation, c, and instance number, n. The former is used in vertical scaling (changing the processing latency of a model), and the latter is used in horizontal scaling (distributing the workload to multiple instances). As we explain in Section P4.5, vertical scaling is not the most cost-efficient mechanism for serving requests compared to horizontal scaling, but the in-place vertical scaling feature is more responsive when there are changes in the workload. Therefore, we solve the IP with two different mindsets, using vertical scaling to respond quickly or saving more resources with horizontal scaling.

However, a similar IP cannot be solved efficiently in real-time as the problem grows exponentially as the number of stages grows [25, 42, 43]. Consequently, we need to use heuristics or limit the solver to some extent (reducing the feasible space to explore). We leverage the limited SLO (which is usually a few thousand milliseconds), limited batch sizes (1-16), and limited core allocation (limited to the hardware the model is placed on (1-16 CPU cores)) in inference serving systems and provide dynamic programming algorithms to find the optimal solution for either vertical or horizontal scaling. Both algorithms run in $O(SLO \times b_{max} \times c_{max} \times |S|)$ time complexity, which reduces the exponential execution time (dependent on the number of stages) to the dominant SLO time by incorporating the space complexity using dynamic programming. Section P4.5.1.3 discusses which algorithm to be executed at a given time.

P4.4.4.1 *Vertical Scaling*

As discussed above, we use dynamic programming to find the optimal solution for the vertical scaling to absorb the sudden changes in the workload.

Algorithm 3 aims to find the optimal CPU allocations for the given pipeline considering their performance profile and the specified SLO. The algorithm takes the pipeline SLO, a set of models denoted by S with their corresponding performance profile, and the workload λ as input and returns the values c_s and b_s for all the DL models $s \in S$. Dynamic programming solves the IP by first solving the problem with just having one stage and then increasing it to the pipeline one by one. For any added model, we consider all possible SLOs (1 to SLO) and divide the picked number into two parts, one for the current DL model and one for the previous DL models. We check whether it is possible with the current division to serve the workload in all the DL models. We continue the procedure until all the models are visited. In the end, if there is no possible solution, we use a binary search on the workload to find the maximum possible workload supported by vertical scaling and bring up new instances (horizontal scaling) to support the remaining workloads.

To be precise, we iterate first over the stages (line 1) and then iterate on all possible SLO values with different batch sizes and core allocations (lines 2-5). Then, we estimate the total latency by calculating the processing latency and then aggregating it with the queuing latency (lines 6-8). Next, if the current throughput supports the incoming workload and the aggregated latency is lower than the SLO, we check if the current model is the first in the pipeline since there is no need to divide the current SLO. If so, we store the current configuration (current CPU and batch size allocations) (lines 11-14). If not, we check if the current configuration is the least total resource allocations with the current SLO division (lines 15-20). After considering all the configurations, we check if there is a possible candidate (line 21) by checking if the algorithm has reached the last DL model in the pipeline. If not, we use binary search on the workload and feed it to the same algorithm to find how much of the workload vertical scaling can support (lines 22-29). We then calculate the needed instances for serving the remaining requests using horizontal scaling with the same CPU core allocations

Algorithm 3: Vertical Scaling

```
   input  : SLO, Set of models S and their latency model, λ
   output : c_s, b_s ∀s ∈ S
1  for s in [0,|S|,1] do
2      for i in [SLO,−1,−1] do
3          if (s == 0 and dp[s][i]) or s (!= 0 and dp[s - 1][i]) then
4              for c in [1,c_max] do
5                  for b in [1,b_max] do
6                      l = calculate l_s(b,c), q = (b−1)/λ
7                      h = calculate h_s(b,c)
8                      l += q
9                      if h ≤ λ and l ≤ SLO then
10                         continue
11                     if s == 0 then
12                         if dp[s][i + l] is False then
13                             dp[s][i + l] = True
14                             best[s][i + l] = (c,c,b)
15                     else if dp[s − 1][i] and i > 0 then
16                         if dp[s][i + l] is False then
17                             dp[s][i + l] = True
18                             best[s][i + l] = (best[s − 1][i][0] + c,c,b)
19                         else if best[s − 1][i][0] + c < best[s][i + l][0]
                            then
20                             best[s][i + l] = (best[s − 1][i][0] + c,c,b)
21 Check if there is a solution in dp[ |S| - 1].
22 if No solution then
23     left = 1, right = λ
24     while right - left > 1 do
25         mid = (right - left) // 2
26         if Vertical scaling with λ = mid then
27             right = mid
28         else
29             left = mid
30     return Vertical Scaling with λ = left and Horizontal Scaling with
        λ = λ − left with the same CPU as Vertical Scaling.
31 else
32     result = calculate recursively the optimal configurations for all the
        models using best.
33     return result
```

(line 30). If there is a valid configuration, we find the configurations for all the models recursively (lines 31-33).

Algorithm 4: Horizontal Scaling

```
   input  : SLO, Set of models S and their latency model, λ
   output : n_s, b_s ∀s ∈ S
1  for s in [0, |S|, 1] do
2      for i in [SLO, −1, −1] do
3          if (s == 0 and dp[s][i]) or s (!= 0 and dp[s - 1][i]) then
4              for b in [1, b_max] do
5                  l = calculate l_s(b, 1), q = (b−1)/λ
6                  h = calculate h_s(b, 1)
7                  l += q
8                  if h ≤ λ and l ≤ SLO then
9                      continue
10                 ins = workload // throughput
11                 if s == 0 then
12                     if dp[s][i + l] is False then
13                         dp[s][i + l] = True
14                         best[s][i + l] = (ins, ins, b)
15                 else if dp[s − 1][i] and i > 0 then
16                     if dp[s][i + l] is False then
17                         dp[s][i + l] = True
18                         best[s][i + l] = (best[s − 1][i][0] + ins, ins, b)
19                     else if best[s − 1][i][0] + ins < best[s][i + l][0] then
20                         best[s][i + l] = (best[s − 1][i][0] + ins, ins, b)
```

P4.4.4.2 *Horizontal Scaling*

Similar to vertical scaling, we use dynamic programming to find the optimal instance number (horizontal scaling) for all the DL models in the system in a single shot. The main difference between the vertical and horizontal scaling algorithms is that we can support any workload using horizontal scaling. Hence, we calculate the number of needed 1-core instances (line 5), and we calculate the least total resource-consuming configuration (lines 11-20) in Algorithm 4. Finally, we recursively find the optimal number of instances for all the models and send the results to the adapter for the system adaptation.

P4.5 TRANSITION

In this section, we explain the necessity of transitioning to horizontal scaling after initially using in-place vertical scaling to respond to changes in workload. We detail the circumstances under which

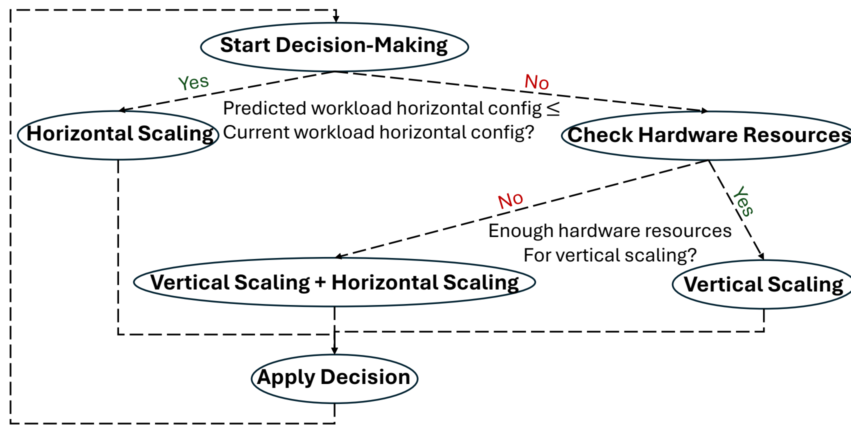


Figure P4.4: Transition states between vertical and horizontal scaling strategies.

vertical and horizontal scaling transitions should occur and provide a comprehensive method for switching between these mechanisms.

First, we present a mathematical proof demonstrating the superior performance of horizontal scaling in terms of throughput, addressing the *why* of this transition, contrasting it with the responsiveness offered by vertical scaling as discussed in Section P4.2. We outline the horizontal and vertical scaling transition steps based on this proof (answering *how*). Finally, to address the *when* of these transitions, we develop an LSTM model to predict the maximum requests per second (RPS) for the next few seconds, guiding the decision of when to switch from vertical scaling to horizontal scaling. All the transition states are available in Figure P4.4.

P4.5.1 *Vertical Scaling to Horizontal Scaling*

This part describes why we need to switch to horizontal scaling from vertical scaling and discusses how and when it must be done.

P4.5.1.1 *Why*

Amdahl's law describes the theoretical speed-up in the execution time of a parallelizable task [3]. In Formula P4.7, $L(r)$ calculates the speed-up of a task with the given computational resources (r) and the parallelization share (p) of the task.

$$L(r) = \frac{1}{(1-p) + \frac{p}{r}}. \tag{P4.7}$$

We use Amdahl's law to show that 1-core instances always provide the same or a higher throughput compared to the multiple-core instances in DL models under a fixed workload using the same amount of resources. We use induction to prove this statement by setting total resources $r = 2$ and then $r = n \times (n+1)$.

For $r = 2$, we have $2\times$ 1-core instances which implies $2 \times L(1)$ and $1\times$ 2-core instance which gives the task speed-up of $1 \times L(2)$. The execution time of $2 \times L(1)$ instances is $2 \times (\frac{1}{(1-p)+\frac{p}{1}}) = 2 \times 1 = 2$ while the latency of the execution time of $1 \times L(2)$ instances is $\frac{1}{(1-p)+\frac{p}{2}} = \frac{2}{2-p}$ which can be at most 2 as the parallelization share can be between 0 and 1. Therefore, $2 \times L(1) \geq 1 \times L(2)$, meaning that $2 \times 1 - core$ has a total higher speed-up $1\times$ 2-core, resulting in the same or higher throughput.

For $r = n \times (n+1)$, we calculate $(n+1)$ n-core instances ($(n+1) \times L(n)$) and n $n+1$-core instances ($n \times L(n+1)$) as follows:

$$(n+1) \times L(n) = (n+1) \times \frac{1}{(1-p) + \frac{p}{n}} = \frac{(n+1) \times n}{n - np + p}. \tag{P4.8}$$

$$n \times L(n+1) = n \times \frac{1}{(1-p) + \frac{p}{n+1}} = \frac{(n+1) \times n}{n - np + 1}. \tag{P4.9}$$

As $0 \leq p \leq 1$, $(n+1) \times L(n)$ process more requests than $n \times L(n+1)$ based on Formula P4.8 and Formula P4.9, $(n+1) \times L(n)$ results in a higher throughput.

Finally, as for any number of cores, more instances with a lower core number achieve a higher throughput. We conclude that for any $r = n$, $n \times 1 - core$ instances achieve the highest possible throughput under a fixed workload.

P4.5.1.2 *How*

There are two scenarios in which we switch to horizontal scaling from vertical scaling: *(i)* when the workload is stable, as described in the

next section, and *(ii)* when there are not enough hardware resources for vertical scaling to support the workload.

For the former scenario, we use the argument in the previous part and bring up as many 1-core instances as needed to serve the requests minus the currently running instances, which we reduce their resources to 1-core as soon as the rest of the instances are up and running. For instance, if currently, two 3-core instances are serving the requests and four 1-core instances can serve the same number of requests, we bring up two 1-core instances and reduce the 3-core instances' resources to 1-core instances using in-place resource scaling.

For the latter scenario, when there is a hardware limitation, or there is no speed-up possible because of the limited parallelism of the DL model, we serve as many requests as possible with the currently running instances by increasing their resources to the maximum to reduce the SLO violations and simultaneously bring up new 1-core instances to serve the remaining requests.

P4.5.1.3 *When*

To decide when to switch from vertical scaling to horizontal scaling in case of workload stability, we design an LSTM to predict the next ten seconds' maximum RPS and feed the predicted value and the current RPS to the horizontal scaling algorithm 4 to check whether the current and future configurations are the same. If so, we switch to horizontal scaling to save resources. We use the horizontal scaling algorithm to check if the workload is stable since the currently running instances may be able to serve more requests than the current workload. Therefore, more RPS may not need more resources.

For building the LSTM model, we use an input layer, a 25-unit LSTM layer, followed by a one-unit dense output layer. We trained the network using the Adam optimizer with the MSE loss function. Figure P4.5 shows the LSTM result, where we train it with 14 days of the Twitter dataset [4] and seven days as validation (more details in Section P4.6). The workload prediction takes less than 30 ms, with a Mean Absolute Percentage Error of 5.8%, making it practical for getting live inference.

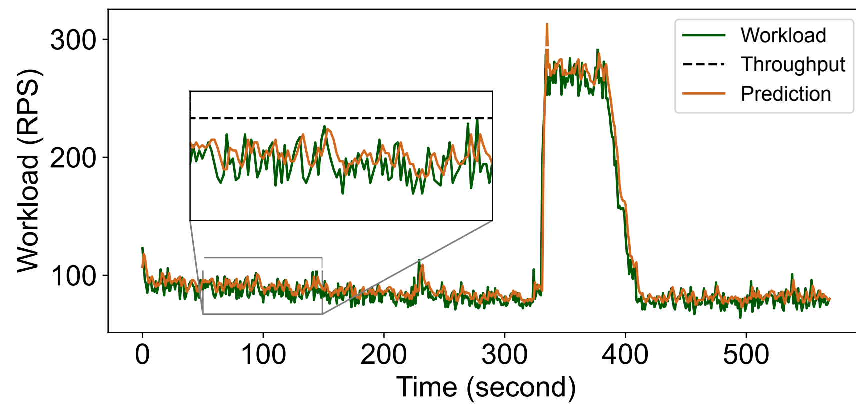


Figure P4.5: LSTM inference result.

P4.5.2 *Horizontal Scaling to Vertical Scaling*

This part describes why we need to switch to vertical scaling from horizontal scaling and discusses how and when it must be done.

P4.5.2.1 *Why and When*

When there is an increase in the workload, horizontal scaling needs to bring up new instances, load the required heavy libraries such as Tensorflow or PyTorch, and initialize the model. These actions take a few to tens of seconds, which drastically reduces the responsiveness of horizontal scaling. On the other hand, in-place vertical scaling can instantly increase the computing resources of the running instances, making it a better option in terms of responsiveness. Therefore, when there is a sudden surge of requests (when the current resource allocation cannot support the increased requests), we switch to vertical scaling to absorb as many extra requests as possible.

P4.5.2.2 *How*

Now, to answer how to transition from horizontal scaling to vertical scaling, let's assume that we have more than one 1-core instance in a stage since with just one 1-core instance, the previous part states how to scale. The following question arises when there is a change in the workload requiring more resources: Should we increase the resources of one instance, the resources of a subset of instances, or the resources of all the instances to react to the changes in the workload?

To answer this question, we use similar proof as in the previous section by first considering two 1-core instances, analyzing how to scale them vertically, and then generalizing them to any number of instances.

For $r = 2 \times n$, we calculate both configurations speed-up; once giving all the resources to one instance ($instance_1 = 2 \times n - 1$ and $instance_2 = 1$) and once distribute the resources evenly ($instance_1 = instance_2 = n$). The speed-up calculation for $r = 2 \times n - 1$ is as follows:

$$L(2 \times n - 1) = \frac{1}{(1-p) + \frac{p}{2 \times n - 1}} = 1 + \frac{2 \times p \times (n-1)}{2 \times (n - np + p) - 1}. \quad \text{(P4.10)}$$

By having the second instance with 1-core and no speed-up, we have the total speed-up of $2 + \frac{2 \times p \times (n-1)}{2 \times (n-np+p)-1}$. On the other hand, if we distribute the resources evenly to both instances, each with n CPU cores, we will have the total speed-up of:

$$2 \times L(n) = 2 \times \frac{1}{(1-p) + \frac{p}{n}} = 2 + \frac{2 \times p \times (n-1)}{n - np + p}. \quad \text{(P4.11)}$$

Now, if $n - np + p \leq 2 \times (n - np + p) - 1$, we have shown that distributing resources evenly has a higher total speed-up, resulting in a higher throughput. The equation can be proven as follows:

$$\begin{aligned} n - np + p &\leq 2 \times (n - np + p) - 1 \\ 1 &\leq n - np + p \\ 1 - p &\leq n \times (1 - p) \end{aligned} \quad \text{(P4.12)}$$

The last part is true since n (the number of allocated CPU cores) is always greater than 1.

The same proof applies to any number of instances, which is omitted to reduce repeatability. Therefore, evenly distributing resources generates a higher throughput compared to giving a subset of instances more resources. We follow the same practice in our system, and when vertical scaling is needed, we scale all the instances to the same amount of resources.

P4.6 EXPERIMENTAL EVALUATION

We implemented the inference services using PyTorch [38] and Hugging Face [19]. The queuing component is implemented using Python

AsyncIO [21]. We implemented Biscale in Python [1]. We evaluated Biscale on a testbed consisting of two servers, each equipped with Core i9-9940x CPUs at 3.3GHz and interconnected with a stable private Ethernet network (1Gbps). We set up a Kubernetes cluster using MicroK8s [50] on these two servers and configured the InPlaceVerticalScaling feature gate, an alpha-stage feature that enables in-place vertical scaling.

Workload: To evaluate Biscale, we developed a workload generator that uses the real-world Twitter traces [4] as the workload pattern (requests per second) and sends requests to the pipelines following a Poisson distribution to mimic the workloads on data centers. Furthermore, we use roughly 10-minute traces from the validation data discussed in Section P4.5.1.3 for evaluation purposes. The selected traces show a steady workload and heavy bursts, demonstrating the systems' behavior under a realistic and dynamic workload. We scale the traces for each pipeline to match the hardware capacity we run the experiments on.

DL Models: To compare Biscale with existing solutions in realistic environments, we use DL models across the computer vision, natural language processing, and audio recognition domains. These domains are also extensively used in other DL inference serving systems [12, 25, 31, 42–44]. Table P4.3 summarises the DL models we use to evaluate Biscale. We further assessed the equation in Formula P4.1 with 512 requests per batch size and CPU core configuration as demonstrated in Figure P4.6 to show the effectiveness of the equation.

Pipelines: We evaluate Biscale using three pipelines as specified in Table P4.3 similar to the pipelines used in recent works [11, 16, 55, 58]. For the SLO of each pipeline, we follow a similar methodology described in [26] and calculate the processing latency with the least batch size and core allocation ($b = c = 1$), aggregate the processing latencies of DL models, and multiply the result by a factor of three.

Baselines: We compare Biscale to a state-of-the-art horizontal scaling (FA2 [42]) and an extended version of the recently proposed in-place vertical scaling (Sponge [41]) in inference serving systems. FA2 solves the horizontal autoscaling problem by simultaneously considering the joint problem of batch size and number of instance selections. It increases the resource utilization of the DL models' instances, reducing

1 The source code will be available after the acceptance.

Table P4.3: DL Models and Applications

DL Models

Task	Architecture	Abbreviation
Object Detection [52]	YOLOv5n	OD
Object Classification [23]	ResNet18	OC
Audio to Text [54]	FAIRSEQ S2T	AT
Sentiment Analysis [14]	DistillBERT	SA
Language Identification [56]	XLM-RoBERTa	LI
Neural Machine Translation [17]	Elan-mt	NT
Text Summarization [34]	T5-small	TS

Applications

Name	Pipeline	SLO
Video Monitoring	OD→OC	780
Audio Sentiment Analysis	AT→SA	1350
Natural Language Processing	LI→NT→TS	2550

the needed instances. Sponge uses in-place vertical scaling to react to sudden changes in the workload. However, their approach only considers one model in the system. To be able to compare Biscale with Sponge, we use the approach in Algorithm 3 without the horizontal scaling part (one instance per DL model) as an extension of Sponge to solve the in-place vertical autoscaling problem in inference pipelines.

Metrics: We consider the following metrics in the evaluation, collected by Prometheus [8]:

- SLO violation rate: We collect each stage latency (consisting of queuing and processing latencies), aggregate them, and use them to check whether a request has violated the application's SLO.
- Cost: We use the number of instances × allocation CPU cores per instance of each DL model and report the aggregated results.
- Request dropping: We consider request dropping to avoid pending requests in the queues. We collect the dropped requests and use them to calculate the total SLO violation rate.

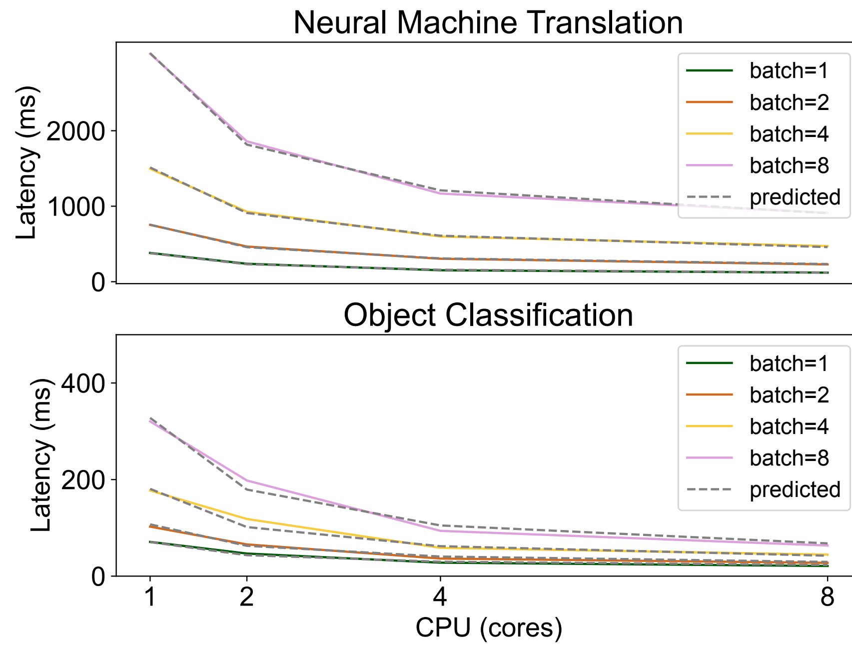


Figure P4.6: Performance profile of Translator and Classifier DL models with different CPU and batch configurations.

- P99 latency: Each second, multiple requests are being served. We pick the P99 of the requests to measure Biscale and compare it with the P99 latency of the baselines.

P4.6.1 *End-to-End Performance*

Video Monitoring. Figure P4.7 provides a detailed evaluation of Biscale, horizontal scaling, and vertical scaling used to handle the Twitter trace workloads in the video monitoring pipeline. The top subfigure shows the workload, starting low and stable, sharply increasing around the 50th second, peaking near the 180th second, and gradually returning to the initial RPS around the 450th second.

When the workload starts, horizontal scaling violates more than half of the requests since the initial state with one instance on each DL model is not capable of supporting the incoming workload. To support the workload, horizontal scaling brings new instances up, where the cold start-up issue kicks in and starts violating the requests until the instances are spawned and warmed up. On the other hand, both Biscale and vertical scaling immediately increase the number of CPUs for

their running instances to increase their throughput, hence supporting the starting workload.

After a few seconds, in Biscale, when the optimizer detects the workload is stable with the help of the LSTM, it switches to more 1-core instances to reduce the costs and, at the same time, to increase the DL models' throughput (second 10). The situation becomes complex when the workload increases to over 40RPS from 10RPS, where none of the approaches have enough up-and-running resources to capture the surge of requests, resulting in violating the SLOs. However, Biscale and vertical scaling approaches change the computing resources of the running instances to the maximum to reduce the SLO violations. Biscale has a slightly lower SLO violation rate since, in the previous seconds, it has brought up more 1-core instances to reduce the overall cost. The horizontal scaling approach initially has the highest violation rate since it needs to bring up new instances to support the current workload, but after the instances are up and running, it distributes the workload to multiple instances and stops violating the requests, which takes roughly 15 seconds. The vertical scaling approach adapts itself to the initial part of the workload. However, it starts suffering from SLO violation since one instance, even with the maximum resource allocation per DL model, is insufficient to support the incoming workload. After detecting the workload surge, Biscale changes the computing resources of the running instances to maximum and simultaneously commands the system to bring up new instances to support the increased workload. With this approach, Biscale violates the requests in less than a few seconds by compensating it with more cost at the 55th second. After capturing the surge, the optimizer reduces the allocated resources to half to save costs by switching to horizontal scaling. After that, there are some fluctuations in the workload where Biscale is capable of capturing immediately due to the responsiveness of in-place vertical scaling, but horizontal scaling still shows lots of SLO violations (over 10% in some seconds), where it needs to bring up new instances and violates SLOs meanwhile. After the workload reaches its initial RPS (450th second), all the approaches serve requests efficiently.

Audio Sentiment Analysis. Figure P4.8 depicts the end-to-end evaluation over the audio sentiment analysis pipeline. Both models in the audio sentiment analysis pipeline are heavier than the ones in video monitoring, and with the same hardware, the system can process a

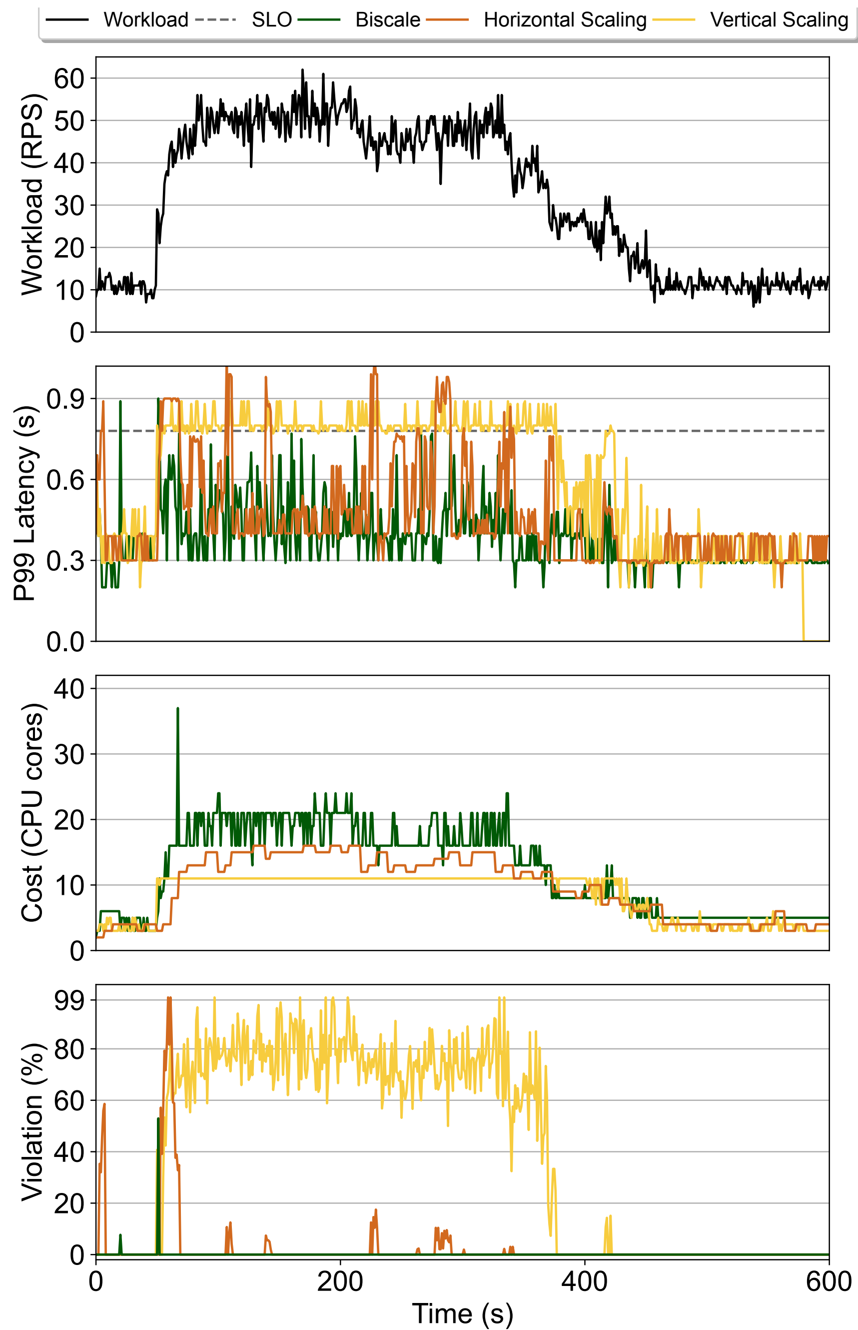


Figure P4.7: End-to-end comparison on the video monitoring pipeline. Biscale reduces the SLO violation to roughly 0.1% by using both scaling mechanisms jointly. Horizontal and vertical scaling mechanisms violate 2.4% and 39.3% of the requests' SLO, respectively.

lower RPS. Therefore, we use almost half of the previous workload in this experiment.

Unlike the video monitoring pipeline, the gap between the SLO and the processing latency with $b = c = 1$ is greater, and Biscale uses the opportunity and increases the DL models' batch size more aggressively, resulting in higher throughput and lower needed resources. Because of the bigger batch sizes, the system does not need to switch to lower CPU cores using in-place vertical scaling, unlike the video monitoring pipeline, which changes the CPU cores more frequently to keep up with the workload. On the other hand, the bigger batch sizes increase the pipeline SLO violation from 0.1% to 0.5%. The violation rate increases to 6.5% in horizontal scaling due to the fact that the models are heavier, and the initialization takes longer, thereby a slower reaction time and a higher SLO violation rate. Vertical scaling suffers from the queuing back pressure throughout the experiment, demonstrating the importance of distributing the requests to multiple instances. Also, at the moment, we depend on the LSTM and the performance profiles to switch to horizontal scaling to improve resource efficiency. Our LSTM might delay this process by predicting higher values for the maximum workload of the next period, as depicted in the same figure.

Natural Language Processing. Figure P4.9 demonstrates the end-to-end evaluation over the natural language processing pipeline. The pipeline contains three DL models (compared to two DL models in the other pipelines), which are the most resource-intensive DL models in our experiments. Therefore, we scale the RPS to the range of (2,12). Following the same pattern, Biscale scales the resources vertically at the beginning, and after capturing the initial workload, it reduces the resources to a minimum to match the workload. Horizontal scaling suffers the most in this scenario since the DL models are heavy and bring up new instances with the weight loading time taking over 10 seconds, resulting in violating roughly half of the requests SLO. Vertical scaling drops almost all the requests (over 96%) due to not having enough resources.

Overall, Biscale demonstrates adaptability to workload changes, effectively scaling resources up and down in response to demand. Despite workload spikes, the system manages to keep P99 latency mostly within the SLO threshold, indicating efficient latency management. The CPU usage graph reflects resource efficiency, increasing during high demand and conserving resources when demand is low. The

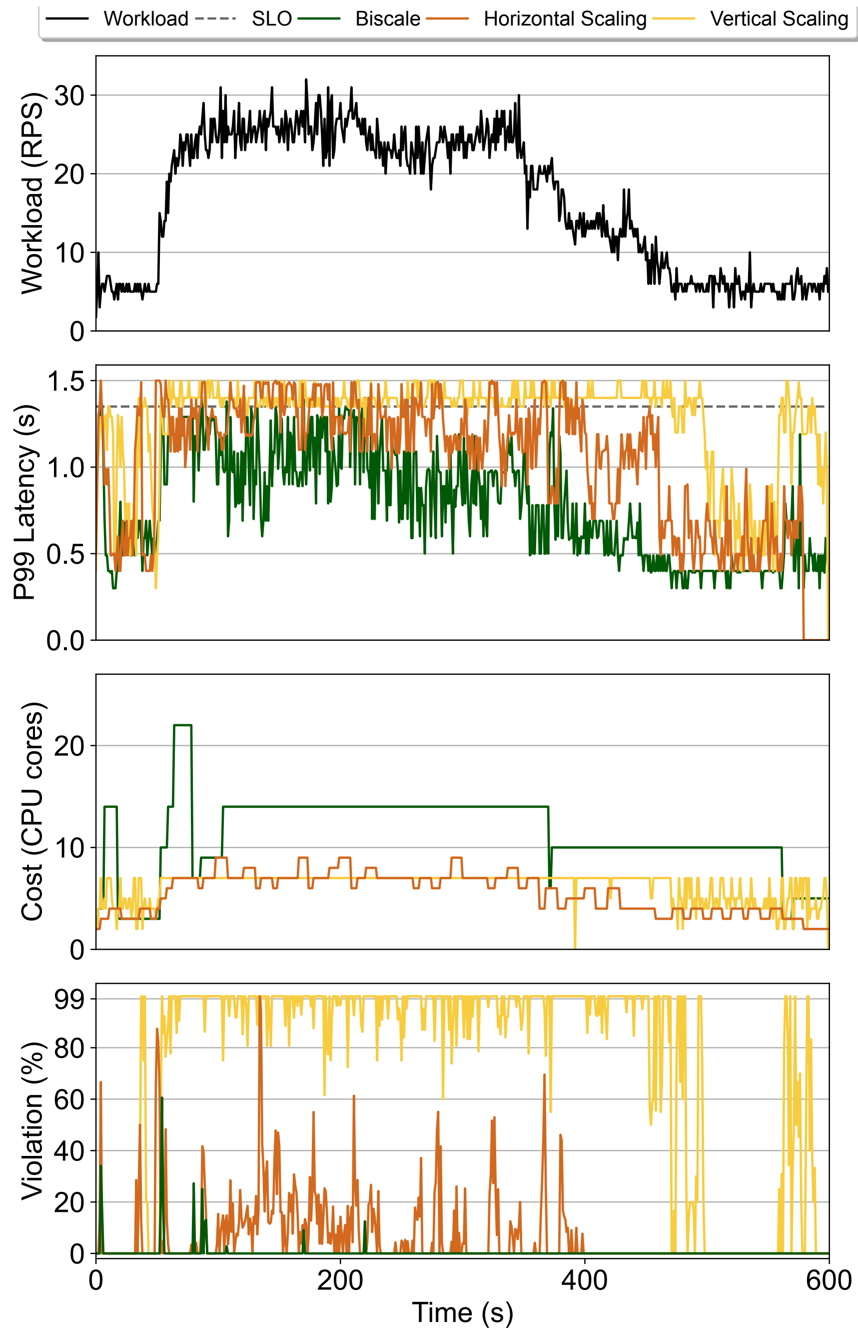


Figure P4.8: End-to-end comparison on the audio sentiment analysis pipeline. Biscale reduces the SLO violation to less than 0.5%. Horizontal scaling violates 6.5% of the requests, while vertical scaling violates most of the requests, reaching over 72% of the total SLO violation.

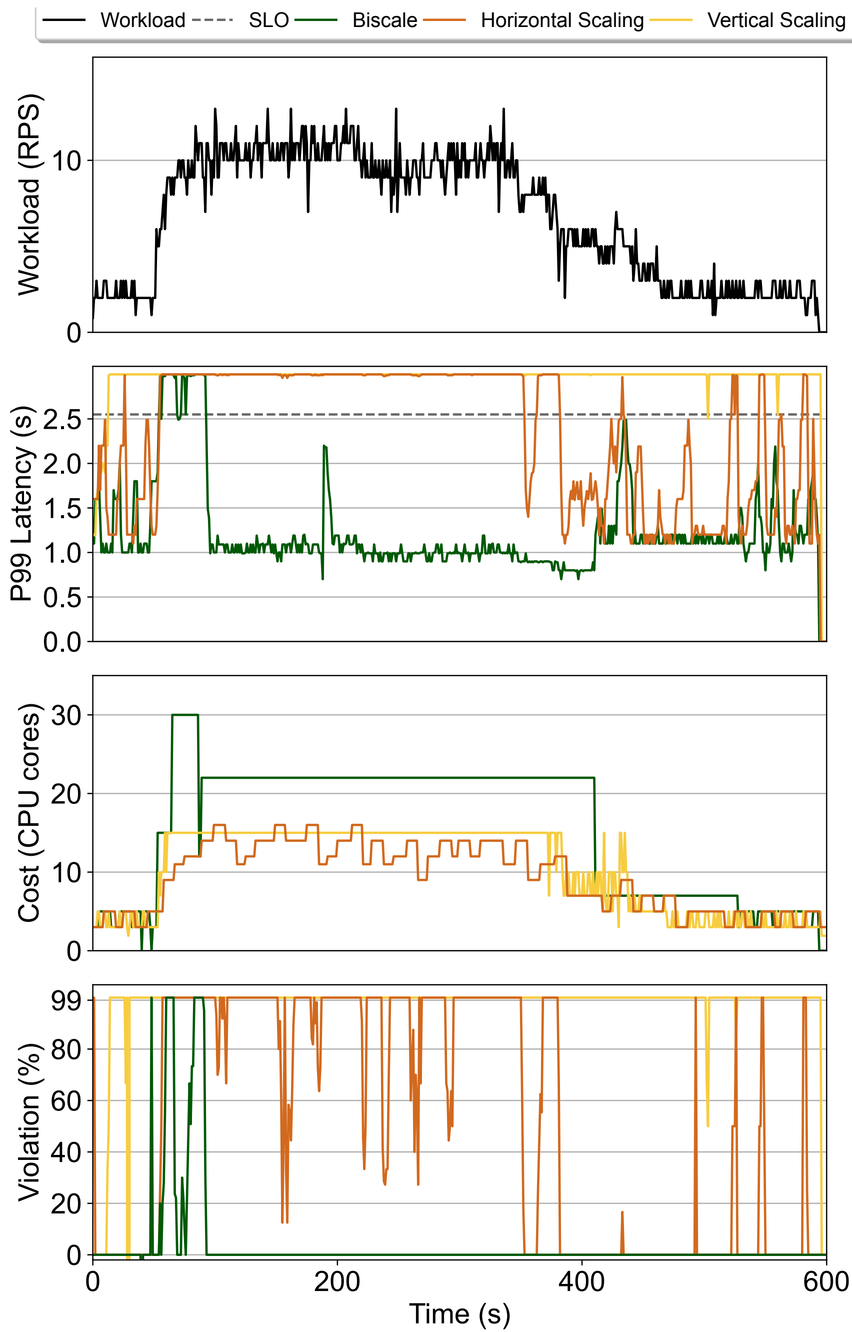


Figure P4.9: End-to-end comparison on the natural language processing pipeline. Biscale reduces the SLO violation to less than 3.8%. Horizontal scaling violates 50.7% of the requests SLO, and vertical scaling violates 96.7% of the requests SLO.

figure highlights the effectiveness of these strategies in managing varying workloads, maintaining low latency and low cost, and minimizing SLO violations.

P4.6.2 *Intra and Inter Parallelism*

In deep learning frameworks like TensorFlow [1] and PyTorch [38], two key parameters significantly impact request processing latency: *inter* and *intra* operation parallelism. These parameters have been extensively studied to determine their optimal configurations for various scenarios [59].

The *intra* operation parallelism parameter allows tasks within a single batch of requests to be parallelized, thereby enhancing processing efficiency within that batch. This parameter can be dynamically adjusted during runtime, offering flexibility and adaptability to changing workloads. In contrast, the *inter* operation parallelism parameter enables multiple tasks to be executed in parallel across different batches. However, unlike *intra* operation parallelism, the *inter* operation parameter is initialized once and remains fixed. This static nature can pose challenges, especially with in-place vertical scaling, where resource allocation needs may shift dynamically.

As shown in Figure P4.10, the *inter* operation parameter has minimal impact if new batches of requests are sent only after the previous batches have been fully processed. This suggests that the effectiveness of *inter* operation parallelism is context-dependent, proving more beneficial in scenarios with concurrent batch processing rather than sequential processing.

P4.6.3 *Request Dropping Effect*

When there is a surge of requests on the system, they may accumulate in the queues, leading to violations not only of their SLOs but also of new incoming requests, as older requests must be processed first. To prevent this, inference serving systems implement request dropping to alleviate queue back pressure [25, 42].

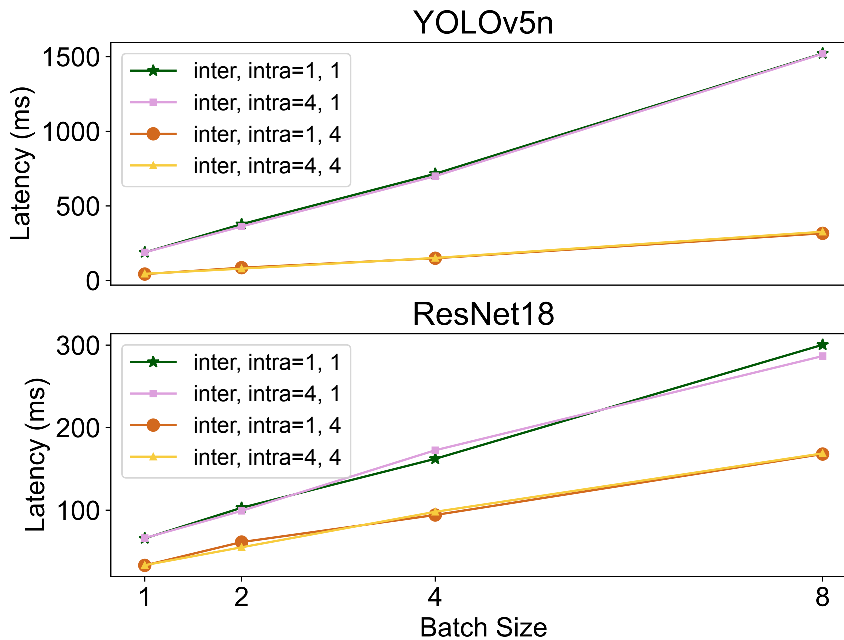


Figure P4.10: The effect of Intra and Inter parallelism parameters on the YOLOv5n and ResNet18 models with 4 CPU cores. Inter parallelism parameter does not affect the processing latency when there is one batch at a time in the DL model.

Different request-dropping strategies can be employed to minimize SLO violations. One strategy is to drop any request already reaching SLO at any processing or queuing stage. Another approach is to allow a buffer, such as three times the SLO, acknowledging that some delay might be acceptable. Finally, an approach is never to drop requests, opting to serve all requests regardless of the cost.

Figure P4.11 compares these three strategies using Biscale and baseline scaling approaches over the first 100 seconds of the workload depicted in Figure P4.7, which includes a sudden increase in demand. Biscale successfully serves almost all requests (over 99%) during this period with virtually no SLO violations. In contrast, when the dropping strategy is relaxed (3x SLO and No Dropping), both horizontal and vertical scaling mechanisms experience significant request dropping, with over 20% and 40% drop in horizontal and vertical scaling, respectively. Thus, the 1x SLO strategy is the most effective in minimizing SLO violations for these approaches.

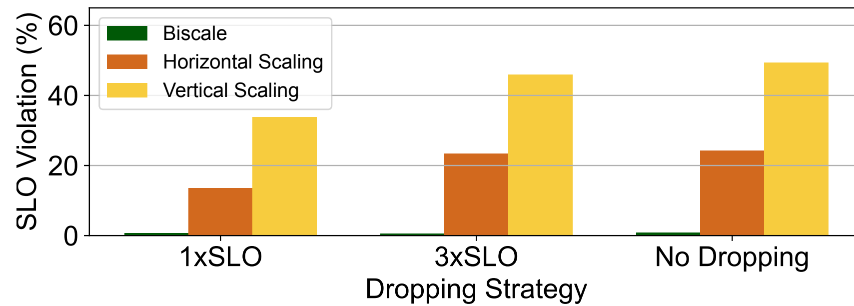


Figure P4.11: The effect of different dropping strategies. The 1xSLO dropping strategy helps reduce the total number of SLO violations.

P4.7 RELATED WORK

This section discusses autoscaling in inference serving systems and microservices since DL models are mainly encapsulated in microservices, and the inference pipelines are composed of several interconnected microservices as discussed in Section P4.2.

Autoscaling in Inference Serving Systems: Multiple approaches have been proposed to reduce total resource consumption either using vertical or horizontal scaling in inference serving systems while guaranteeing SLOs [12, 13, 15, 25, 27, 28, 30, 37, 43, 44, 47, 48]. FA2 [42] uses graph transformation and dynamic programming techniques to reduce the number of instances in horizontal scaling. The graph transformation component deconstructs the execution graph, simplifying it and enabling optimal solutions to be found in real time. Meanwhile, the dynamic programming solution determines the optimal batch size and scaling factor for each DL model within the pipeline, ensuring efficiency and performance optimization. Sponge [41] uses a greedy approach to guarantee end-to-end latency of one DL model under a dynamic network using in-place vertical scaling. Similar to this work, they encapsulate the autoscaling problem into an IP and solve it in real time.

Autoscaling in Microservices: Current practices in microservice autoscaling mechanisms are mainly rule-based heuristics [6, 7, 20, 32, 49, 51, 60], meta-heuristics [10, 24, 53], or recently, machine learning based [40, 45, 46]. Kubernetes Vertical Pod Autoscaler (VPA) [7] and Horizontal Pod Autoscaler (HPA) [6] manage the allocation of computing resources and the scaling of instances for microservices using threshold-based metrics. VPA adjusts the CPU and memory resources

allocated to individual pods based on their current usage to ensure optimal performance. Conversely, HPA scales the number of instances up or down in response to real-time demands by monitoring metrics like CPU and memory utilization. However, it is advised not to use VPA and HPA together, as they may interfere with each other's operations. When both are set, their concurrent adjustments can lead to conflicting actions, resulting in inefficient resource allocation and potential performance degradation. A close work, SHOWAR [9], uses the empirical variance in the historical resource usage for vertical scaling to have an improved resource allocation compared to the default commonly used metric-based Kubernetes Vertical Autoscaler [7] (VPA), and uses CPU scheduler's eBPF metrics to design a more accurate horizontal autoscaler than Kubernetes Horizontal Autoscaler [6] (HPA). In vertical scaling in SHOWAR and Kubernetes VPA, the microservices are bounded by a maximum of 1-core CPU allocation since microservices do not leverage multiple CPU cores to accelerate the microservices' performance in contrast to DL models. Furthermore, the vertical autoscaler in SHOWAR does not react to changes in the workload but to CPU and memory metrics, which results in SLO violations in terms of unpredictable changes in the workload.

P4.8 CONCLUSION

In this paper, we have presented Biscale, a novel system designed to enhance inference serving through an innovative two-stage autoscaling strategy. By combining in-place vertical scaling with horizontal scaling, BoS effectively addresses the challenges of managing resource allocation under variable and unpredictable workloads. Extensive evaluation with real-world workloads demonstrates Biscale reduces the SLO violation by over $10\times$ and minimizes the cost when the demand is low, showing Biscale's potential to provide responsive and cost-effective inference-serving solutions.

For future works, we consider enhancing Biscale's capability to utilize heterogeneous hardware resources effectively. Modern data centers often comprise a mix of CPUs, GPUs, TPUs, and specialized accelerators. Integrating Biscale with hardware-aware scaling policies can optimize the allocation of tasks to the most suitable hardware, further improving efficiency and performance. Moreover, different placement

strategies can be employed where new instances should reside. It requires answering the question of whether we should use a bin packing approach to utilize the physical machines to the maximum, a fair distribution of instances using a round-robin approach to not exhaust the machines, or if the most sparse instance distribution generates the ideal configuration in terms of guaranteeing SLOs.

P4.9 ACKNOWLEDGMENTS

This work has been supported by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project-ID 210487104 - SFB 1053.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. "Tensorflow: A system for large-scale machine learning." In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2016, pp. 265–283.

[2] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. "Batch: Machine learning inference serving on serverless platforms with adaptive batching." In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE. 2020, pp. 1–15.

[3] Gene M Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities." In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. 1967, pp. 483–485.

[4] archiveteam. *Archiveteam-twitter-stream-2021-08*. `https://archive.org/details/archiveteam-twitter-stream-2021-08`. 2021.

[5] The Kubernetes Authors. *In-place Resource Resize for Kubernetes Pods*. `https://kubernetes.io/blog/2023/05/12/in-place-pod-resize-alpha/`. Accessed on 30.01.2024. 2023.

[6] The Kubernetes Authors. *Kubernetes Horizontal Pod Autoscaling*. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/. Accessed on 30.01.2024. 2024.

[7] The Kubernetes Authors. *Kubernetes Vertical Pod Autoscaling*. https://cloud.google.com/kubernetes-engine/docs/concepts/verticalpodautoscaler/. Accessed on 30.01.2024. 2024.

[8] The Prometheus Authors. *Prometheus monitoring and alerting toolkit*. https://prometheus.io/. Accessed on 30.01.2024. 2024.

[9] Ataollah Fatahi Baarzi and George Kesidis. "Showar: Right-sizing and efficient scheduling of microservices." In: *Proceedings of the ACM Symposium on Cloud Computing*. 2021, pp. 427–441.

[10] Tao Chen and Rami Bahsoon. "Self-Adaptive Trade-off Decision Making for Autoscaling Cloud-Based Services." In: *IEEE Transactions on Services Computing* 10.4 (2017), pp. 618–632.

[11] clarifai. *Clarifai*.

[12] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. "InferLine: Latency-Aware Provisioning and Scaling for Prediction Serving Pipelines." In: *ACM Symposium on Cloud Computing (SoCC)*. 2020, pp. 477–491. DOI: 10.1145/3419111.3421285.

[13] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. "Clipper: A {Low-Latency} Online Prediction Serving System." In: *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2017, pp. 613–627.

[14] Souvick Das. *SentimentAnalysisDistillBERT*. https://huggingface.co/Souvikcmsa/SentimentAnalysisDistillBERT. Accessed on 30.01.2024. 2024.

[15] Aditya Dhakal, Sameer G. Kulkarni, and K. K. Ramakrishnan. "GSLICE: Controlled Spatial Sharing of GPUs for a Scalable Inference Platform." In: *ACM Symposium on Cloud Computing (SoCC)*. 2020, pp. 492–506. DOI: 10.1145/3419111.3421284.

[16] Ruofei Du, Na Li, Jing Jin, Michelle Carney, Scott Miles, Maria Kleiner, Xiuxiu Yuan, Yinda Zhang, Anuva Kulkarni, Xingyu Liu, et al. "Rapsai: Accelerating Machine Learning Prototyping of Multimedia Applications through Visual Programming." In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 2023, pp. 1–23.

[17] Mitsua Elan. *ElanMT-BT-ja-en*. https://huggingface.co/Mitsua/elan-mt-bt-ja-en. Accessed on 30.01.2024. 2024.

[18] Mohamed Elhoseny. "Multi-object detection and tracking (MODT) machine learning model for real-time video surveillance systems." In: *Circuits, Systems, and Signal Processing* 39.2 (2020), pp. 611–630.

[19] Hugging Face. *Hugging Face*. https://huggingface.co/. Accessed on 11.07.2024.

[20] Luca Florio and Elisabetta Di Nitto. "Gru: An Approach to Introduce Decentralized Autonomic Behavior in Microservices Architectures." In: *IEEE International Conference on Autonomic Computing (ICAC)*. 2016.

[21] python software foundation. *AsyncIO library*. https://docs.python.org/3.8/library/asyncio.html. 2022.

[22] The Linux Foundation. *Kubernetes*. https://kubernetes.io. Accessed on 29.10.2021. 2019.

[23] The PyTorch Foundation. *ResNet18*. https://pytorch.org/vision/main/models/generated/torchvision.models.resnet18.html. Accessed on 30.01.2024. 2024.

[24] Sören Frey, Florian Fittkau, and Wilhelm Hasselbring. "Search-based genetic optimization for deployment and reconfiguration of software in the cloud." In: *IEEE International Conference on Software Engineering (ICSE)*. 2013.

[25] Saeid Ghafouri, Kamran Razavi, Mehran Salmani, Alireza Sanaee, Tania Lorido-Botran, Lin Wang, Joseph Doyle, and Pooyan Jamshidi. *IPA: Inference Pipeline Adaptation to Achieve High Accuracy and Cost-Efficiency*. 2024. arXiv: 2308.12871 [cs.DC].

[26] Arpan Gujarati, Sameh Elnikety, Yuxiong He, Kathryn S McKinley, and Björn B Brandenburg. "Swayam: distributed autoscaling to meet slas of machine learning inference services with resource efficiency." In: *ACM/IFIP/USENIX Middleware Conference*. 2017, pp. 109–120.

[27] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. "Serving DNNs like Clockwork: Performance Predictability from the Bottom Up." In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2020, pp. 443–462.

[28] Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma, Mahmut Taylan Kandemir, and Chita R Das. "Cocktail: A multidimensional optimization for model serving in cloud." In: *USENIX NSDI*. 2022, pp. 1041–1057.

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition." In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[30] Yitao Hu, Weiwu Pang, Xiaochen Liu, Rajrup Ghosh, Bongjun Ko, Wei-Han Lee, and Ramesh Govindan. "Rim: Offloading Inference to the Edge." In: *Proceedings of the International Conference on Internet-of-Things Design and Implementation*. 2021, pp. 80–92.

[31] Ram Srivatsa Kannan, Lavanya Subramanian, Ashwin Raju, Jeongseob Ahn, Jason Mars, and Lingjia Tang. "GrandSLAm: Guaranteeing SLAs for Jobs in Microservices Execution Frameworks." In: *ACM European Conference on Computer Systems (EuroSys)*. 2019, pp. 1–16. ISBN: 978-1-4503-6281-8. DOI: 10.1145/3302424.3303958.

[32] Hamzeh Khazaei, Rajsimman Ravichandiran, Byungchul Park, Hadi Bannazadeh, Ali Tizghadam, and Alberto Leon-Garcia. "Elascale: autoscaling and monitoring as a service." In: *ACM Annual International Conference on Computer Science and Software Engineering (CASCON)*. 2017.

[33] Matthew LeMay, Shijian Li, and Tian Guo. "Perseus: Characterizing performance and cost of multi-tenant serving for cnn models." In: *2020 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE. 2020, pp. 66–72.

[34] Steven Liu. *T5-small*. https://huggingface.co/stevhliu/my_awesome_billsum_model. Accessed on 30.01.2024. 2024.

[35] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Guodong Yang, and Chengzhong Xu. "The power of prediction: microservice auto scaling via workload learning." In: *Proceedings of the 13th Symposium on Cloud Computing*. 2022, pp. 355–369.

[36] Motahare Mounesan, Mauro Lemus, Hemanth Yeddulapalli, Prasad Calyam, and Saptarshi Debroy. *Reinforcement Learning-driven Data-intensive Workflow Scheduling for Volunteer Edge-Cloud*. 2024. arXiv: 2407.01428 [cs.DC].

[37] Vinod Nigade, Pablo Bauszat, Henri Bal, and Lin Wang. "Jellyfish: Timely Inference Serving for Dynamic Edge Networks." In: *2022 IEEE Real-Time Systems Symposium (RTSS)*. 2022, pp. 277–290. DOI: 10.1109/RTSS55097.2022.00032.

[38] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. "Pytorch: An imperative style, high-performance deep learning library." In: *Advances in neural information processing systems* 32 (2019).

[39] Anna Pavlenko, Joyce Cahoon, Yiwen Zhu, Brian Kroth, Michael Nelson, Andrew Carter, David Liao, Travis Wright, Jesús Camacho-Rodríguez, and Karla Saur. "Vertically Autoscaling Monolithic Applications with CaaSPER: Scalable C ontainer-a s-a-S ervice P erformance E nhanced R esizing Algorithm for the Cloud." In: *Companion of the 2024 International Conference on Management of Data*. 2024, pp. 241–254.

[40] Haoran Qiu, Weichao Mao, Chen Wang, Hubertus Franke, Alaa Youssef, Zbigniew T Kalbarczyk, Tamer Başar, and Ravishankar K Iyer. "{AWARE}: Automate workload autoscaling with reinforcement learning in production cloud systems." In: *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 2023, pp. 387–402.

[41] Kamran Razavi, Saeid Ghafouri, Max Mühlhäuser, Pooyan Jamshidi, and Lin Wang. *Sponge: Inference Serving with Dynamic SLOs Using In-Place Vertical Scaling*. 2024.

[42] Kamran Razavi, Manisha Luthra, Boris Koldehofe, Max Mühlhäuser, and Lin Wang. "FA2: Fast, Accurate Autoscaling for Serving Deep Learning Inference with SLA Guarantees." In: *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2022, pp. 146–159. DOI: 10.1109/RTAS54340.2022.00020.

[43] Francisco Romero, Qian Li, Neeraja J Yadwadkar, and Christos Kozyrakis. "INFaaS: Automated Model-less Inference Serving." In: *USENIX Annual Technical Conference (ATC)*. 2021, pp. 397–411.

[44] Francisco Romero, Mark Zhao, Neeraja J Yadwadkar, and Christos Kozyrakis. "Llama: A Heterogeneous & Serverless Framework for Auto-Tuning Video Analytics Pipelines." In: *ACM Symposium on Cloud Computing (SoCC)*. 2021, pp. 1–17.

[45] Fabiana Rossi, Matteo Nardelli, and Valeria Cardellini. "Horizontal and vertical scaling of container-based applications using reinforcement learning." In: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE. 2019, pp. 329–338.

[46] Vighnesh Sachidananda and Anirudh Sivaraman. "Erlang: Application-Aware Autoscaling for Cloud Microservices." In: *Proceedings of the Nineteenth European Conference on Computer Systems*. 2024, pp. 888–923.

[47] Mehran Salmani, Saeid Ghafouri, Alireza Sanaee, Kamran Razavi, Max Mühlhäuser, Joseph Doyle, Pooyan Jamshidi, and Mohsen Sharifi. "Reconciling High Accuracy, Cost-Efficiency, and Low Latency of Inference Serving Systems." In: *Proceedings of the 3rd Workshop on Machine Learning and Systems*. 2023, pp. 78–86.

[48] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. "Nexus: a GPU cluster engine for accelerating DNN-based video analysis." In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*. Ed. by Tim Brecht and Carey Williamson. ACM, 2019, pp. 322–337. DOI: 10.1145/3341301.3359658.

[49] Akshitha Sriraman and Thomas F. Wenisch. "¯Tune: Auto-Tuned Threading for OLDI Microservices." In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2018.

[50] *The lightweight Kubernetes*. https://microk8s.io/.

[51] Giovanni Toffetti, Sandro Brunner, Martin Blöchlinger, Josef Spillner, and Thomas Michael Bohnert. "Self-managing cloud-native applications: Design, implementation, and experience." In: *Future Generation Computer Systems* 72 (2017), pp. 165–179.

[52] ultralytics. *YOLOv5*. https://github.com/ultralytics/yolov5. Accessed on 30.01.2024. 2024.

[53] Chad Verbowski, Ed Thayer, Paolo Costa, Hugh Leather, and Björn Franke. "Right-Sizing Server Capacity Headroom for Global Online Services." In: *IEEE International Conference on Distributed Computing Systems (ICDCS)*. 2018, pp. 645–659.

[54] Changhan Wang, Yun Tang, Xutai Ma, Anne Wu, Sravya Popuri, Dmytro Okhonko, and Juan Pino. "Fairseq S2T: Fast speech-to-text modeling with fairseq." In: *arXiv preprint arXiv:2010.05171* (2020).

[55] Tongshuang Wu, Michael Terry, and Carrie Jun Cai. "Ai chains: Transparent and controllable human-ai interaction by chaining large language model prompts." In: *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 2022, pp. 1–22.

[56] Dina El Zein. *xlm-roberta-base-finetuned-language-identification*. https://huggingface.co/dinalzein/xlm-roberta-base-finetuned-language-identification. Accessed on 30.01.2024. 2024.

[57] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. "Mark: Exploiting cloud services for cost-effective, SLO-aware machine learning inference serving." In: *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 2019, pp. 1049–1062.

[58] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J Freedman. "Live video analytics at scale with approximation and delay-tolerance." In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 2017, pp. 377–392.

[59] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P Xing, et al. "Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning." In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 2022, pp. 559–578.

[60] Tianlei Zheng, Xi Zheng, Yuqun Zhang, Yao Deng, ErXi Dong, Rui Zhang, and Xiao Liu. "SmartVM: a SLA-aware microservice deployment framework." In: *World Wide Web* 22.1 (2019), pp. 275–293.

P5

DISTRIBUTED DNN SERVING IN THE NETWORK DATA PLANE

ABSTRACT

Programmable networks have received tremendous attention recently. Apart from exciting network innovations, in-network computing has been explored as a means to accelerate a variety of distributed systems concerns, by leveraging programmable network devices. In this paper, we extend in-network computing to an important class of applications called deep neural network (DNN) serving. In particular, we propose to run DNN inferences in the network data plane in a distributed fashion and make our programmable network a powerful accelerator for DNN serving. We demonstrate the feasibility of this idea through a case study with a real-world DNN on a typical data center network architecture.

P5.1 INTRODUCTION

The emergence of programmable, high-performance network switches and SmartNICs, has not only enabled exciting innovations in networking but also inspired a new computing paradigm called in-network computing [6, 12]. With in-network computing, programmable network devices are instructed to accelerate application components by leveraging the high-throughput, low-latency processing capabilities, and convenient on-path placement of these devices [2]. Example applications that have been proven to benefit from in-network computing are caching [5], aggregation [10, 13], agreement [4], and database query processing [18].

Deep learning has become the de facto approach for various inference tasks such as object detection and speech recognition. With the widespread adoption of deep learning, it becomes critical that we can serve DNNs with high performance in terms of both throughput and latency. DNN serving is usually done preferably with high-end accelerators like GPUs/TPUs over CPUs due to higher efficiency and lower costs [14]. GPUs/TPUs typically employ batching to improve throughput, at the cost of increased inference latency [14].

Considering both the fast development of programmable network devices and the high demand for DNN serving, we ask a bold question: *Can we leverage a programmable network to perform DNN serving?* Given that a modern Tofino2 switch can process packets with nanosecond latency, and at the rate of billions of packets per second [1], DNN serving would achieve an unprecedented level of performance if the DNN can be executed entirely in the data plane of the programmable network. Note that with this design the inference can be performed "on the fly" while transferring DNN input data on the network, eliminating the need of accelerators.

Prior work has explored the intersection of programmable networks and machine learning. For example, in-network aggregation has been used to accelerate the gradient synchronization in data-parallel DNN training [10, 13]. Other work has explored data-plane packet classification by running per-packet inference tasks, like decision trees, SVMs and small (binary) neural networks, on programmable switches and SmartNICs [16, 19]. Confined to a single device, such approaches limit the size of the supported ML models, and work towards addressing this issue only involves new hardware architectures [17]. So far, and to the best of our knowledge, none of these efforts support the serving of large DNNs (models with millions of weights) across a network of programmable network devices targeting user applications.

In this paper, we propose an in-network system for fast, end-to-end DNN serving by distributing a DNN across a network of programmable switches, as depicted in Figure P5.1. Our inspiration stems from the observation that DNNs are dataflow computations similar to how packets flow through a network. Based on that, we ❶ map the neurons in the DNN to the physical network switches, ❷ craft and route packets carrying the input/intermediate data to go through the switches containing the corresponding neurons, and ❸ instruct each switch to perform the computations specified by the neurons assigned

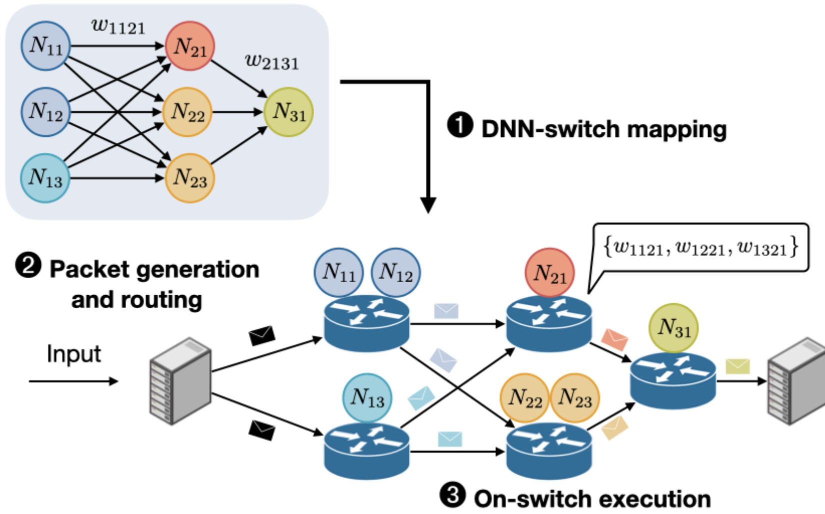


Figure P5.1: An overview of in-network DNN serving.

to the switch. In the rest of this paper, we demonstrate the feasibility of this idea through a case study with a real-world DNN and a typical data center network architecture. We also discuss further challenges.

P5.2 A CASE STUDY WITH MINI-ALEXNET

We use a simplified version of AlexNet [9] (mini-AlexNet) used in [11] and trained on CIFAR-10 [8] with three dimensions (height, width, and channel). We map the mini-AlexNet network to a set of programmable switches using the data center network architecture (spine/leaf) adapted from the Google infrastructure [15].

The mini-AlexNet neural network is described in Table P5.1. It consists of an input layer and three convolutional layers followed by three fully connected layers. The convolutional layers are composed of convolutional filters, ReLU activation functions, and 2D Max Pooling (2x2) layers. Due to the fact that the state-of-the-art programmable switches (Barefoot Tofino2 [3]) are not equipped with Floating Point Units (FPUs), we store the inputs and weights in 8-bit integers. To avoid multiplication and aggregation overflow, we can increase the input values to 32-bit integers as it does not affect the number of operations and the cost of storing results is either temporary, or minuscule compared to weights. Previous work has shown that fixed-point arithmetic is faster than floating point equivalent accuracy, and 8-bit precision is sufficient for DNN inference [17].

	mini-AlexNet (CIFAR-10)	Number of Parameters	Number of Operations	Memory (byte)	Mapped Switch
Layer 1	Input: $32 \times 32 \times 3$	0	0	3,072	S1
Layer 2	Conv1: $3 \times 3 \times 3 \times 64$	1,792	3,110,400	16,192	S2
Layer 3	Conv2: $3 \times 3 \times 64 \times 192$	110,784	37,347,648	117.696	S3
Layer 4	Conv3: $3 \times 3 \times 192 \times 384$	663,939	21,227,520	665,475	S4
Layer 5	FC1: 4096	6,295,552	12,582,911	6,299,648	S5
Layer 6	FC2: 2048	8,390,656	16,777,215	8,392,704	S6
Layer 7	FC3 (Output): 10	20,490	40,959	20,500	S7

Table P5.1: The mini-AlexNet network. It contains seven layers, with over 15 million parameters and 91 million operations requiring less than 16MB of memory to store.

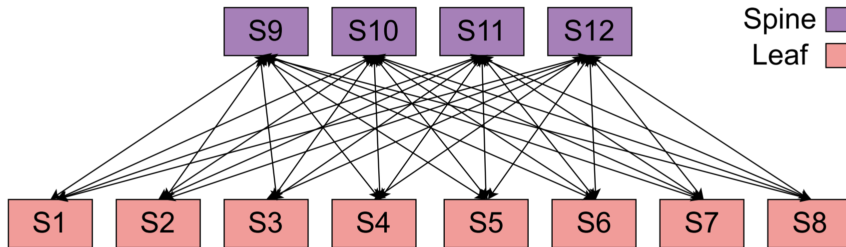


Figure P5.2: The spine/leaf network architecture.

DNN-switch mapping. The data center network architecture we use is depicted in Figure P5.2. It consists of four programmable switches as spines and eight programmable switches as leaves. Each leaf switch is connected to all spine switches with 400GbE links. The spine/leaf architecture advantage is that any leaf (spine) switch is connected to any other leaf (spine) switch in exactly two hops. We use this property to map the layers of the DNN to just leaf switches (one layer is mapped to one switch) to avoid stragglers in the synchronization.

The Tofino2 switch we consider here has four processing pipelines, each equipped with eight 400GbE ports supporting up to 12.8Tb/s of aggregate throughput. To leverage the processing power of all four pipelines of each switch, we distribute the neurons of the same layer to all the pipelines on the same switch (as the neurons in the same layer do not need to communicate with each other) evenly based on the layer type: For the *convolutional layer* we divide the number of filters by the number of switch pipelines and allocate the convolutional filters to the respective pipelines. For the *dense layer*, we divide the number of dense neurons by the number of switch pipelines and store the dense neurons' weights in the respective pipelines.

In the mini-AlexNet scenario, after we get the input from the client (on S1), we allocate and store 16 convolutional filters to each pipeline, as the first convolutional layer (on S2) has 64 filters. After the on-switch execution (detailed later), the packets for the next layer are generated (detailed later) and are sent to the next convolutional layer (on S3). We follow the same procedure until we reach the first dense layer (on S5) with 4096 neurons. We divide the dense layer into four parts and allocate each part containing 1024 neurons to each of the pipelines and perform the neuron computations until we reach the output layer (on S7), where we obtain the prediction result and send it back to the client.

Packet generation and routing. Once the DNN has been partitioned and the neurons deployed on the switches, the next step is to generate packets and route them on the network following the DNN dataflow. For the input layer, the input data is encapsulated into as many packets as are required (based on the input and the packet header size) and is multicast to the switch's pipelines (as the neurons are distributed among all pipelines) hosting neurons that are directly connected to that layer. Upon completing its computations, a layer will emit packets encapsulating the input for the next layer to the spine switches, and these packets will be multicast to the next associated switch hosting the next layer. Each switch maintains a forwarding table applying the above logic to route the packets inside the network. Each packet carries a label through which its target layer can be identified. The switch multicasts the packet based on the label to all the pipelines of the next switch. Upon a packet's arrival, the switch knows the computations to apply to the data carried by that particular packet.

In the mini-AlexNet scenario, the input switch (S1) multicasts the input to all the spine switches (with recirculation), and the switches in the spine layer redirect the packets to the first convolutional layer (on S2). After the processing of the first convolutional layer is done, each pipeline has 1/4 of the input for the next layer. With the spine/leaf design, we transfer each input segment to the spine switches, and there we multicast the input segments to all four pipelines of the next switch (on S3). Each pipeline in S3 receives packets from a switch in the spine layer, shaping the current layer's input. We follow the same procedure until we reach the output layer, where we send back the final prediction.

On-switch execution. Switches will perform computations for the neurons they host upon packet arrivals. More specifically, as each pipeline of the switch gets the entire input, we perform the computations based on the layer type. If the layer is a convolutional layer, we apply the filter to a subset of the input and store the result of the dot product. If it is a dense layer, we multiply all the input values by all the weights and aggregate their results.

We decompose each multiplication into a number of shifts. The input is shifted left by i if the i-th bit of the 8-bit weight is set, otherwise by 0. All shifts are performed in parallel, in a single stage, and the intermediate results are stored in temporaries. This step takes one stage, assuming predicate instructions (to check if the i-th bit is set),

otherwise two. Then, a reduction step aggregates the intermediate results. Since we use 8-bit weights, this step takes three stages. Given N free ALUs in the first stage, we can perform $N/8$ multiplications in parallel, each of which has a maximum depth of five stages. This allows us to replicate the process a number of times without recirculation. The (intermediate) result of each multiplication is accumulated to a register, with a cost, in terms of stages, logarithmic to the number of multiplications performed. If the dot product requires more multiplications, the packet is recirculated.

Then we apply the ReLU activation function to the result of the dot product by checking whether the most significant bit (MSB) is 1 (sign bit) and replacing the value with 0. For the max pooling part, we check whether the current value is the last piece of the pooling window. A major challenge here is to find the pooling elements due to the fact that there is no mod operation in the switches available. To avoid this issue, we process the inputs in the order of the max pooling window. For a simple 2×2 pooling window, after we processed all four window's values, we get the maximum of them in four cycles (three comparisons in total; two cycles for the first two comparisons in parallel and storing the result and two cycles for the second stage comparison). To meet the promised 12.8Tb/s throughput, Tofino2 allows a limited number of operations per packet traversal (few 10's of multiplications like the one described above). Therefore, we need to recirculate to process all the inputs for all the filters in the same pipeline. For each filter/neuron on a switch, the switch accumulates the multiplication results and maintains a counter to ensure that packets from all weights have been processed before emitting the result as a packet to the downstream layers.

In Table P5.1 we calculate the number of operations and memory requirements for each layer of mini-AlexNet. Even the most memory hungry of the layers (Layer 6) requires less than 10% of the available memory on the Tofino2 (a couple of hundred of MBs). However, the number of operations greatly exceeds the 10's of operations we can perform in one traversal. To solve this issue, we recirculate the input in the switch with a new set of operations until all the required operations are done. In the most computation-intensive layer (Layer 3), each pipeline needs to compute roughly 10,000,000 operations, we need to recirculate the same input less than 1,000,000 times. The packet recirculating comes with a latency cost similar to parsing

another packet. Tofino2 can potentially process six billion packets (1.5 billion packets per pipeline), resulting in less than 1ms to process even the most computation-intensive layer in our scenario. In total, a set of available switches in the data centers require less than 2*ms* to perform inference in the mini-AlexNet scenario. Compared to the evaluations reported in [11], our in-network DNN serving system not only reduces the inference latency by over 2× and 2.5× compared to CPU and GPU, respectively but also eliminates the necessity of having inference servers.

P5.3 CHALLENGES AND DISCUSSION

Accuracy improvement. Floating point addition on programmable switches has been carefully explored in [20] while other more complex arithmetic operations are still not feasible with the current switch design. Currently, we use 8-bit fixed point weights as we are still not able to implement the floating point multiplication operations in the data plane respecting the memory access limitation of Tofino2 switches. We plan to explore how packet re-circulation can help work around this limitation.

Support for more complex layer types. So far, we have only discussed how to handle DNNs with convolutional and dense layers. However, popular DNNs typically involve a variety of layer types with more complex structures and activation functions, calling for a careful design of the data structure. There are also layers with nonlinear activation functions like `tanh` and `sigmoid`, which are currently hard to perform on programmable switches.

Fault tolerance. Switches and links can fail, and ensuring that the final prediction is generated without being affected by such failures is essential. Also, a packet loss between switches could render a DNN execution stagnation due to the use of the per-neuron counter. We acknowledge that achieving reliability for stateful in-network computing like DNN serving is a big challenge, which has not been extensively studied yet [7]. We leave this for future work.

P5.4 ACKNOWLEDGMENTS

This work has been partially funded by the German Research Foundation (DFG) within the Collaborative Research Center (CRC) 1053 MAKI, the Dutch Research Council (NWO) Open Competition Domain Science XS Grant 12611, and Google Research.

REFERENCES

[1] Anurag Agrawal and Changhoon Kim. "Intel tofino2–a 12.9 tbps p4-programmable ethernet switch." In: *2020 IEEE Hot Chips 32 Symposium (HCS)*. IEEE Computer Society. 2020, pp. 1–32.

[2] Marcel Blöcher, Lin Wang, Patrick Eugster, and Max Schmidt. "Switches for HIRE: resource scheduling for data center in-network computing." In: *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*. Ed. by Tim Sherwood, Emery D. Berger, and Christos Kozyrakis. ACM, 2021, pp. 268–285. DOI: 10.1145/3445814.3446760.

[3] Intel Corporation. *Intel Tofino 2*. Accessed on 21.9.2022.

[4] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. "NetChain: Scale-Free Sub-RTT Coordination." In: *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*. Ed. by Sujata Banerjee and Srinivasan Seshan. USENIX Association, 2018, pp. 35–49.

[5] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. "NetCache: Balancing Key-Value Stores with Fast In-Network Caching." In: *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 2017, pp. 121–136. DOI: 10.1145/3132747.3132764.

[6] George Karlos, Henri E. Bal, and Lin Wang. "Don't You Worry 'Bout a Packet: Unified Programming for In-Network Computing." In: *HotNets '21: The 20th ACM Workshop on Hot Topics in Networks, Virtual Event, United Kingdom, November 10-12, 2021*. ACM, 2021, pp. 99–107. DOI: 10.1145/3484266.3487395.

[7] Daehyeok Kim, Jacob Nelson, Dan R. K. Ports, Vyas Sekar, and Srinivasan Seshan. "RedPlane: enabling fault-tolerant stateful in-switch applications." In: *ACM SIGCOMM 2021 Conference, Virtual Event, USA, August 23-27, 2021*. Ed. by Fernando A. Kuipers and Matthew C. Caesar. ACM, 2021, pp. 223–244. DOI: 10.1145/3452296.3472905.

[8] Alex Krizhevsky. "Learning Multiple Layers of Features from Tiny Images." In: (2009), pp. 32–33.

[9] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks." In: *Communications of the ACM* 60.6 (2017), pp. 84–90.

[10] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael M. Swift. "ATP: In-network Aggregation for Multi-tenant Learning." In: *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*. Ed. by James Mickens and Renata Teixeira. USENIX Association, 2021, pp. 741–761.

[11] Seulki Lee and Shahriar Nirjon. "SubFlow: A Dynamic Induced-Subgraph Strategy Toward Real-Time DNN Inference and Training." In: *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2020, pp. 15–29. DOI: 10.1109/RTAS48715.2020.00-20.

[12] Dan R. K. Ports and Jacob Nelson. "When Should The Network Be The Computer?" In: *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS 2019, Bertinoro, Italy, May 13-15, 2019*. ACM, 2019, pp. 209–215. DOI: 10.1145/3317550.3321439.

[13] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter Richtárik. "Scaling Distributed Machine Learning with In-Network Aggregation." In: *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*. Ed. by James Mickens and Renata Teixeira. USENIX Association, 2021, pp. 785–808.

[14] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. "Nexus: a GPU cluster engine for accelerating DNN-based video analysis." In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*. Ed. by Tim Brecht and Carey

Williamson. ACM, 2019, pp. 322–337. DOI: 10.1145/3341301.3359658.

[15] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. "Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network." In: *ACM SIGCOMM computer communication review* 45.4 (2015), pp. 183–197.

[16] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Gianni Antichi, Paolo Costa, Hamed Haddadi, and Roberto Bifulco. "Re-architecting Traffic Analysis with Neural Network Interface Cards." In: *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, April 4-6, 2022*. USENIX Association, 2022, pp. 513–533.

[17] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, Ishan Gaur, and Kunle Olukotun. "Taurus: a data plane architecture for per-packet ML." In: *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*. Ed. by Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch. ACM, 2022, pp. 1099–1114. DOI: 10.1145/3503222.3507726.

[18] Muhammad Tirmazi, Ran Ben Basat, Jiaqi Gao, and Minlan Yu. "Cheetah: Accelerating Database Queries with Switch Pruning." In: *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. Ed. by David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo. ACM, 2020, pp. 2407–2422. DOI: 10.1145/3318464.3389698.

[19] Zhaoqi Xiong and Noa Zilberman. "Do Switches Dream of Machine Learning?: Toward In-Network Classification." In: *Proceedings of the 18th ACM Workshop on Hot Topics in Networks, HotNets 2019, Princeton, NJ, USA, November 13-15, 2019*. ACM, 2019, pp. 25–33. DOI: 10.1145/3365609.3365864.

[20] Yifan Yuan, Omar Alama, Amedeo Sapio, Jiawei Fei, Jacob Nelson, Dan R. K. Ports, Marco Canini, and Nam Sung Kim. "Unlocking the Power of Inline Floating-Point Operations on Programmable Switches." In: *19th USENIX Symposium on Networked*

Systems Design and Implementation, NSDI 2022, April 4-6, 2022. USENIX Association, 2022, pp. 683–700.

P6

NETNN: NEURAL NETWORK INTRUSION DETECTION SYSTEM IN PROGRAMMABLE SWITCHES

ABSTRACT

The rise of deep learning has led to various successful attempts to apply deep neural networks (DNNs) for important networking tasks such as intrusion detection. Yet, running DNNs in the network control plane, as typically done in existing proposals, suffers from high latency that impedes the practicality of such approaches.

This paper introduces NetNN, a novel DNN-based intrusion detection system that runs completely in the network data plane to achieve low latency. NetNN adopts raw packet information as input, avoiding complicated feature engineering. NetNN mimics the DNN dataflow execution by mapping DNN parts to a network of programmable switches, executing partial DNN computations on individual switches, and generating packets carrying intermediate execution results between these switches. We implement NetNN in P4 and demonstrate the feasibility of such an approach. Experimental results show that NetNN can improve the intrusion detection accuracy to 99% while meeting the real-time requirement.

P6.1 INTRODUCTION

In network security, Intrusion Detection Systems (IDSes) play an important role in identifying and responding to anomalous behaviors in

network traffic. Traditional IDSes commonly employ statistical methods which rely on a prior knowledge of known attack patterns [13, 26]. However, these methods are limited in detecting unknown attacks, necessitating a more advanced and adaptive approach. Recently, machine learning (ML) approaches, such as Decision Trees (DTs) and Random Forests (RFs), have made their way into networking tasks such as intrusion detection [1, 3, 27, 29]. While using DTs and RFs reduces the required expertise in the field and provides insights about the decisions similar to rule-based approaches, it still requires a critical step called feature engineering—selecting the best features among many to build the ML model. This step is known to suffer from missing or sensitive features, and is prone to overfitting.

Deep learning has gained tremendous attention due to its successes in various applications such as speech recognition, computer vision, and natural language processing [8, 17]. Deep neural networks (DNNs) can automatically extract features from large and complex data sets, and learn nonlinear relationships between input and output variables, making them a well-suited approach for complex decision-making applications such as detecting intrusions in network traffic. As a result, several successful attempts have been made to build IDSes based on DNNs, with significant detection accuracy improvement [6, 21, 23]. However, these attempts implement the DNN-based intrusion detection models in the network control plane, incurring high latency and thus failing to meet stringent timing requirements.

Over the last decade, programmable network devices (e.g., programmable switches and SmartNICs) have been increasingly adopted in modern networks, allowing to implement customized network functions in the network data plane. This has not only accelerated network innovations [16, 20, 21, 24], but also inspired new directions such as in-network computing where computations (e.g., aggregation, caching, and coordination) are offloaded to the network data plane for low-latency high-throughput processing [9, 10, 18].

Given the ever-increasing need for real-time network security and rapid advancements in programmable networks, it is imperative to explore the potential of leveraging programmable networks for efficient intrusion detection using DNNs. This raises an important question: Can we harness the capabilities of programmable networks, such as the packet processing speed and nanosecond-level latency offered by modern programmable switches, to effectively perform DNN-based

intrusion detection in a timely and efficient manner? DNN serving would achieve an unprecedented level of performance if the DNN is executed entirely in the network data plane. Since programmable switches are natively responsible for moving the DNN input data, with this design, the inference can be performed "on the fly," eliminating the need for external accelerators. However, to achieve inference at the line rate, it is essential to have pure data-plane DNN implementations which expose unique challenges due to complex DNN computations and limited hardware capabilities.

We present NetNN, the first intelligent data-plane system that enables the execution of DNNs using a set of programmable switches for intrusion detection. To this end, NetNN ❶ maps the DNN neurons and associated weights to a set of programmable switches, ❷ mimics the execution workflow of the DNN and routes the DNN execution results with network packets, and ❸ issues instructions to individual switches, wherein each switch is tasked with executing the computations dictated by the neurons specifically assigned to it. Overall, this paper makes the following contributions. After presenting the motivation for IDSes in network data planes and identifying the challenges (Section P6.2), we

- present the design of NetNN, including its overall architecture and components (Section P6.3);
- introduce a novel network design for DNN-based IDSes that not only advances state-of-the-art intrusion detection accuracy but also enables deep learning inference execution completely in the data plane (Section P6.3.2);
- implement a system prototype for NetNN[1]. using the P4 language, and evaluate it using a Covert Channel dataset (Section P6.4). Overall, NetNN increases the accuracy of intrusion detection systems on programmable switches to 83% without the need for feature engineering and 99% by adding the hardware timestamp when packets arrive (packets' inter-arrival time) and the majority voting of the same flow's packets classification.

1 NetNN open source code: https://github.com/shynfard/netnn

P6.2 MOTIVATION

P6.2.1 *Feature Extraction for Intrusion Detection*

To understand the importance of features engineering in traditional approaches for intrusion detection and the advantages of deep learning based approaches, we compare the accuracy achieved by the state-of-the-art decision-tree-based approach NetBeacon [29], an eXtreme Gradient Boosting (XGBoost) [5] decision trees model with randomly selected features, and NetNN, for both packet and flow classification on a convert channel detection dataset [3]. The covert channel detection dataset consists of 1000 flows encoded by two censorship resistance tools: Facet [12] and DeltaShaper [2].

Packet classification. For NetNN, we use the first 68 (maximum of UDP and IP header size) raw bytes of the packets (from the dataset [3]) and train a neural network with three layers (a 1D convolutional layer with 32 filters, a dense layer with 50 neurons, and another dense layer with 100 neurons) similar to [14]. For NetBeacon, we train an XGBoost model using the packet's length and inter-arrival time. For the random selection approach, we select a random subset of the same features of NetBeacon and train an XGBoost model. With only the packet features (length and inter-arrival time), NetBeacon reaches the accuracy of 58% using the XGBoost classifier. The accuracy drops to 51% when selecting a random set (10%) of lengths and inter-arrival times. On the other hand, NetNN achieves an accuracy of 83% by just using the first 68 bytes of raw representation of the packets.

Flow classification. For the flow classification of NetNN, we train a neural network (a 1D convolutional layer with 32 filters followed by another 1D convolutional layer with 64 filters, a dense layer with 50 neurons, another dense layer with 100 neurons, and a final output layer as depicted in Figure P6.1) with the first 68 bytes of the first power of two packets (1st, 2nd, 4th, ..., 1024th) with the minimum and maximum of inter-arrival times up to the 1024th packet. We injected a flow identifier as a feature to the input to help the neural network model generate similar outputs for the packets from the same flow. At the end, we aggregate the classification results of the ten packets and report the final category of the flow based on the majority votes. For NetBeacon, similar to FlowLens [3], we use buckets for length

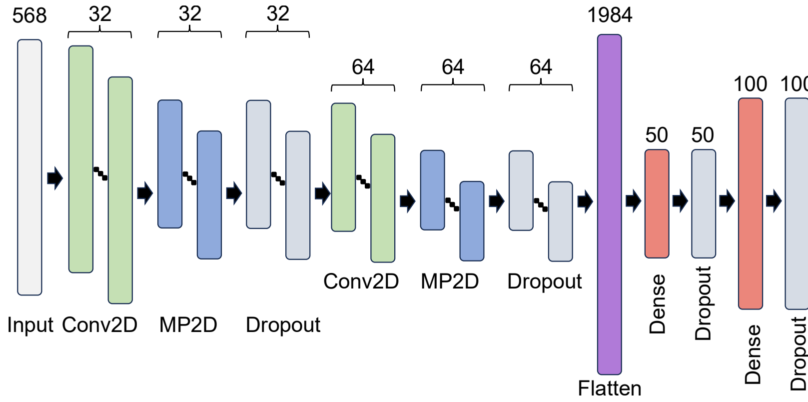


Figure P6.1: The DNN architecture used in NetNN with two Conv1D layers (Conv2D with the second dimension as one), followed by a flatten and three dense layers.

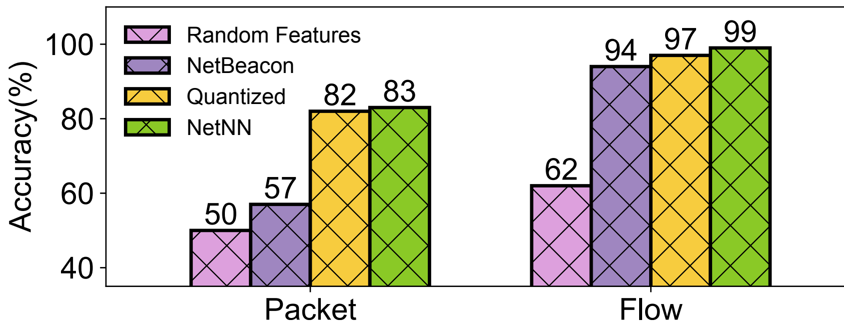


Figure P6.2: Comparison of NetBeacon, Random, and NetNN for both packet and flow classification.

and inter-arrival time distributions of all the packets in the dataset, roughly 14 million packets (7 million malicious and 7 million benign). Specifically, we use 1500 buckets for different lengths (1, 2, ..., 1500) and 3000 buckets for inter-arrival times (0-60 seconds distributed to 3000 buckets with lengths of 12 milliseconds). We randomly select 10% of lengths and inter-arrival times buckets for the random feature approach. With the flow level features, Netbeacon's accuracy reaches 94% while selecting a random subset of features reduces the accuracy considerably to 62%. On the contrary, NetNN achieves a validation accuracy of over 99% with less information as shown in Figure P6.2.

Flow-level feature extraction is quite expensive in the data plane as accurately calculating statistical features of packets such as inter-arrival rate or length distribution requires floating-point arithmetic, which is not available on the programmable switches data plane such as Intel Tofino. Moreover, storing and updating features require careful design and management of multiple hash tables [1, 3, 29]. On the other

hand, we observe the importance of careful feature selection by using a random set of 50 features from FlowLens [3], resulting in the lowest accuracy. The above combined raises the question of how to reduce the overhead of feature extraction.

P6.2.2 *Our Goal*

Our goal is to design a new intrusion detection system based on DNNs that run completely in the network data plane, leveraging the performance of modern programmable switches. With that, a network operator can scan network traffic at the line rate and analyze it accurately directly in the network data plane. The following objectives encapsulate the underlying motivations guiding our design principles.

No feature engineering. We aim to avoid the use of features for packet/flow classification since gathering features is both computing and storage expensive, as we discuss in Section P6.2.1. Therefore, our objective is mainly to use raw packet representation plus inter-arrival time information to feed a DNN model for intrusion detection.

Model scalability. We aim to deploy large DNNs with hundreds of thousands of operations per inference for high accuracy, which is not possible on only one switch as discussed in Section P6.2.3. Therefore, we propose to deploy a DNN by breaking it into multiple parts and distributing each part's weights to a different programmable switch similar to [16].

P6.2.3 *Intrusion Detection on Programmable Switches*

Intrusion detection on programmable switches (e.g., based on decision trees) has become popular due to the increasing demand for real-time network security [3, 13, 27–29]. By performing intrusion detection directly in the data plane, the processing overhead and the network traffic can be significantly reduced, resulting in faster and more efficient intrusion detection. Additionally, since the programmable switches are located at strategic points in the network topology, they can provide better visibility into the network traffic, making it easier to detect and prevent network attacks.

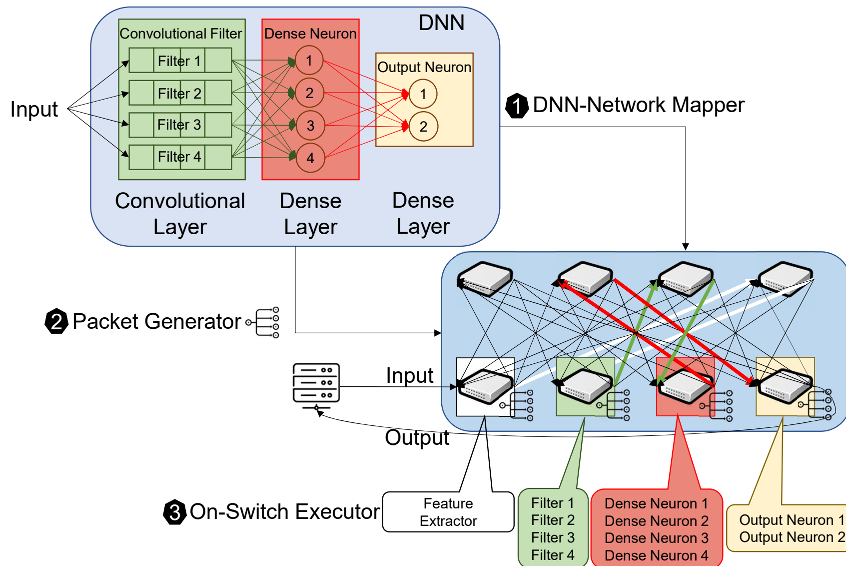


Figure P6.3: An overview of the NetNN architecture.

Processing constraint. To ensure line-rate processing in programmable switches, the data plane programs (e.g., written in P4) must employ simple packet processing instructions. Each pipeline stage in the switch imposes a fixed processing time on packets. This restriction limits the number and complexity of operations performed within each stage. Floating-point operations, loops, and even integer multiplication/division are usually not supported. Also, the actions associated with each table are constrained to restricted simpler operations, such as additions and bit shifts.

Memory constraint. The memory architecture of programmable switches imposes various constraints on the data structures used in P4 programs. Such switches incorporate two high-speed memory types: TCAM (Ternary Content Addressable Memory), which facilitates rapid table lookups, and SRAM (Static Random Access Memory), which allows P4 programs to maintain state across packets. However, the available stateful memory in programmable switches is limited, typically in 10s of MB. Also, accessing all available registers can be challenging since registers within one stage of the processing pipeline cannot be accessed from different stages.

P6.3 SYSTEM DESIGN

We present NetNN, a system that leverages the network data plane for fast and accurate intrusion detection. We start with a design overview and then dive into the major components.

P6.3.1 *System Overview*

A DNN is characterized by its layered architecture, consisting of multiple interconnected layers of neurons. This architecture typically includes an initial layer, one or more intermediate layers, and a final output layer. Within these layers, neurons are connected via weighted connections, allowing for complex information processing. This work focuses on three different layer types, Convolutional, MaxPooling, and Dense layers, where each of the layers has different properties such as convolutional kernel and maxpooling window.

Figure P6.3 (top left) shows a simple DNN and provides a high-level overview of NetNN consisting of three key components: *Mapper*, *Packet Generator*, and *Neural Network Executor*. The system takes a trained model as input and decides the number of needed switches by leveraging the layered architecture of the DNNs. Take the model in Figure P6.1 as an example. After the model has been provided to the system, the system decides the number of switches, in this case, layer by layer (Convolutional and Dense layers) per switch. Moreover, since switches have multiple pipelines to process packets in parallel, we divide the convolutional layers filter by filter and, divide the dense neurons by the number of neurons and map each partition to a switch pipeline (details in Section P6.3.2).

When a packet enters the *Feature Extractor* switch we first extract the packet identifier (5-tuple: `src.ip`, `src.port`, `dst.ip`, `dst.port`, and `proto`) and capture the packet's arrival time. Then, we check if the packet belongs to a flow recorded before. If it is recorded in the flow table, we calculate the simple inter-arrival time features of the flow up to the current packet. We further check whether the current packet should be treated as an inference point (see Section P6.3.3). If so, we extract the packet's first 68 bytes (maximum UDP/IP datagram size) and take each bit as input to our model, including the inter-

arrival time features. When the features are preprocessed, the feature extractor switch generates packets from the set of features based on the next layer type with respect to the network topology and the mappings as described in Section P6.3.4. The packets are generated using the convolution kernel if the next layer is a convolutional layer. Finally, we execute the DNN operations based on the layer type (see Section P6.3.3) using simple arithmetic operations (bit-shifting and addition). If the layer is convolutional, we dot product the neurons by the link weights and aggregate the results. After all the results are computed for a layer, a similar approach for the first layer packet generation is applied to the rest of the layers, e.g., the results are treated as new packets, and based on the next layer type, the packets are generated and routed.

P6.3.2 *Mapper*

In this part, we focus on the DNN partition problem to fit the model on a set of programmable switches and provide an efficient execution flow of the DNN. Most DNNs have a layered structure, and we decide to partition the DNNs across switches layer by layer, as done similarly in [16]. We follow the same practice and assign each layer to a switch. There are layers without weights, such as MaxPooling layers, typically employed after convolutional layers to reduce overfitting and increase computational efficiency. MaxPooling layers, unlike other layers in a neural network, do not involve learnable weight parameters. Instead, they perform computations based on a fixed operation to downsample and retain the most important information from the input data. Since these layers do not have weight parameters, we fit them alongside their previous convolutional layers. After mapping the layers/neurons to switches, we store the model weights on switches as follows.

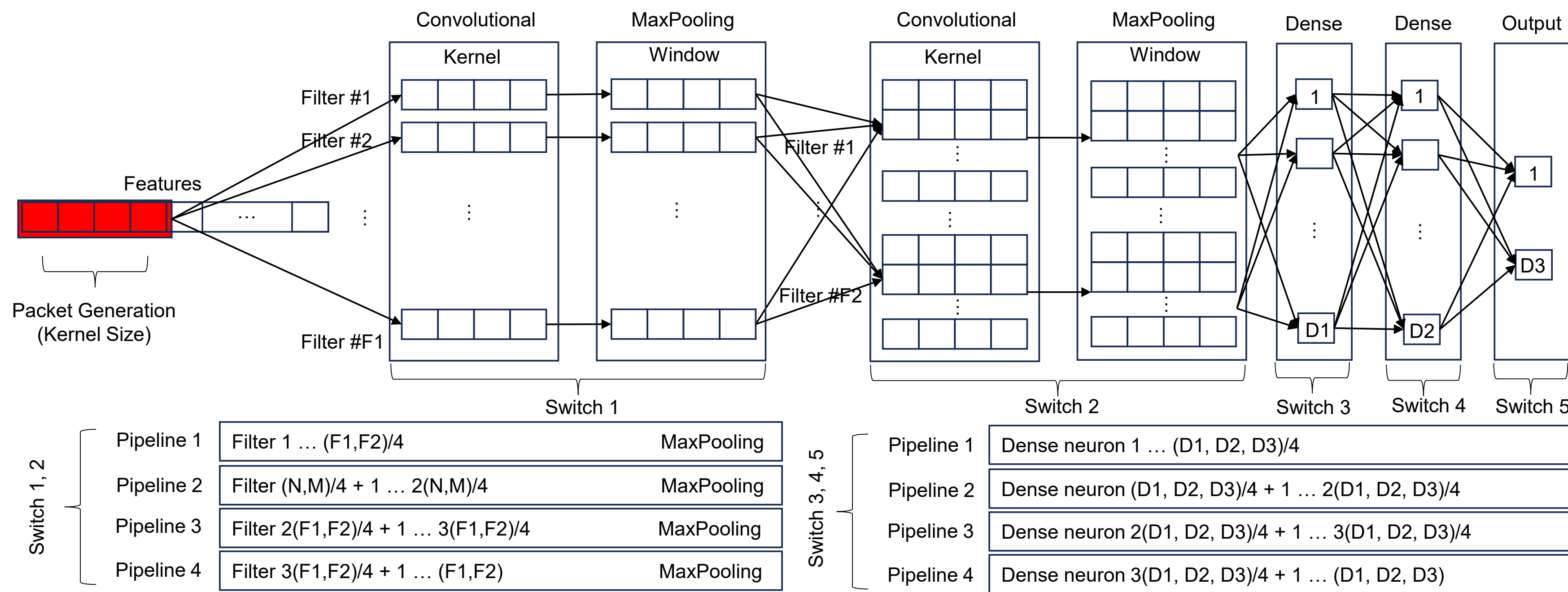


Figure P6.4: The workings of NetNN Mapper

Convolutional layers. We divide the filters by the pipelines on a switch. The filter-splitting approach suits the switch architecture well since there is no communication between any two filters in the same convolutional layer, and computations can be done in parallel. For the model in Figure P6.1, each pipeline in the first switch receives two filters, while the second switch receives four filters per pipeline. In the inference phase, after the convolutional layers, there are MaxPooling layers, which will fit in the same switches as shown in Figure P6.4.

There are two different approaches to storing the weights of convolution layers: register arrays in the data plane and table entries through the control plane. With register arrays, we can read and write them in real-time during packet processing since it does not involve software processing. However, register arrays often have limited capacity due to hardware constraints, and they are typically designed for transient data storage needed during packet processing and not for persistent data. With table entries, we benefit from simplified management of them since the modification is usually handled through APIs. However, accessing and processing entries in tables via the control plane can introduce a higher latency compared to data plane register arrays. This latency is due to software processing and lookup operations, making tables less suitable for real-time packet processing. We use table entries to access the persistent data (layers' weights) and register arrays for storing data(features)/intermediate data in real time.

To store the weights of the convolutional layers, we use the filter number and the position of the weight in that filter ($Filter, Weigth_i, Weight_j$) as the identifier. However, we need a more complex 3D table entry on the switch. Therefore, we need to map the 3D identifier to a 1D space, which is usually done by their multiplication, $Filter \times Height \times Width$, that maps a number from $\mathbb{R}^3 \longrightarrow \mathbb{R}$. However, calculating the multiplications on the data plane is computationally expensive as multiplications are not natively supported on the programmable switches. Therefore, we use an available hash function on the data plane to create a unique index using similar identifiers, i.e., $\text{Hash}(Filter \times Height \times Width)$.

Dense layers. We follow the same approach as in the convolutional layer, we divide the layer by the number of dense neurons. Similar arguments apply here as there is no communication between two dense neurons in the same layer, and the dense neuron computations can be executed in parallel. We use a similar data structure (table entry)

for storing the weights of dense layers. However, storing dense weights is simpler than convolutional weights as one can access the dense weights by a number (weight identifier) instead of three numbers.

So far, we have mapped a DNN with different layers to a set of switches. Next, we need to trigger the DNN computations through the events of packet arrivals. When a packet arrives at a switch, we need to mimic the DNN execution using network packets. We first identify the next layer where the input/intermediate data should be sent. Based on the layer type, we follow different strategies:

Convolutional layers. If the next layer type is a convolutional layer, we put the related data based on the kernel size of the convolutional layer to the packet header (including the necessary identifiers) and multicast the packet to all the next switch pipelines. For instance, from the feature extractor switch, we fetch a kernel size of features from the feature register, add the filter number and weight positions in the kernel ($kernel_i$,$kernel_j$) to the packet header, and create a packet. The result of this approach is that when a packet arrives at a convolutional switch, it can perform all the necessary computations related to that exact packet header data, except the MaxPooling part (discussed in Section P6.3.3) as it requires multiple packet headers. When the data in the packet headers is processed, we get a final value for the current packet header that will be used to generate the next packet header based on the following layer type.

Dense layers. If the next layer is a dense layer, we multicast each feature to all the next switch pipelines since each feature is used by all the neurons of a dense layer. After all the computations regarding the current data are completed, the result is stored in a register array on the switch. After all the computations regarding a dense neuron are finished, the finalized value is ready to be sent to the pipelines of the next switch based on the next layer type.

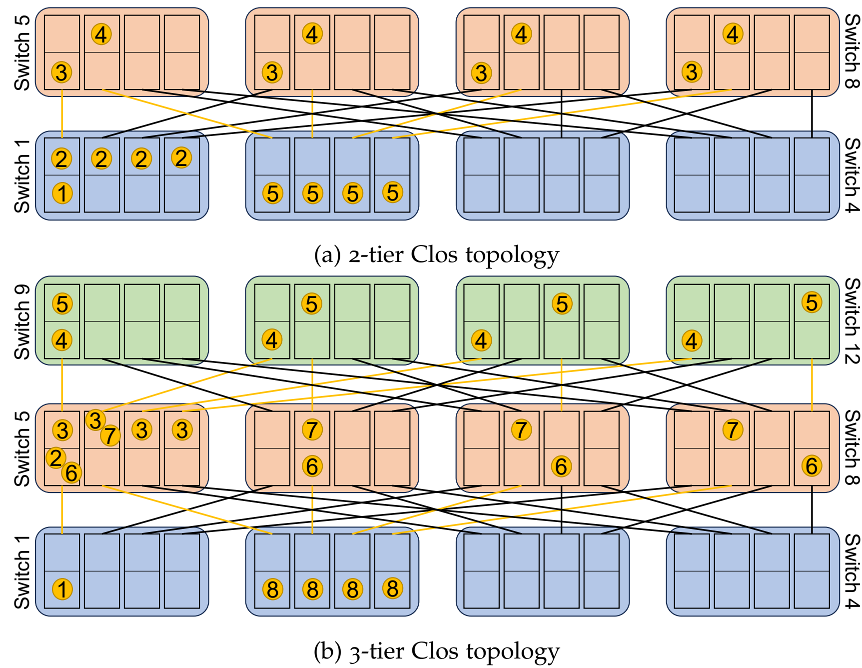
P6.3.3 *Neural Network Executor*

When a packet arrives in the initial switch, we capture the inter-arrival time and calculate the simple inter-arrival time features (min, max) up to the current packet. After that, we need to identify if the current packet is an inference point. Following NetBeacon [29], we check

whether a packet is in the order of the power of two, namely (1, 2, 4, ...), as it can be easily identified in a programmable switch by using a bitwise `AND` operation between the number and its decrement (if the result is zero, it means that the number is a power of two). Upon an inference point, we start generating packets as described before. We use the following procedures (based on the layer type) to process the generated packets.

Convolutional and MaxPooling layers. For convolutional layers, we first create the hash of the indices (filter number, height, and width) of the required weights. After fetching the weights from registers, we perform dot products for all the weights and the values. We break down each multiplication operation into a series of bit shifts for the dot production. Specifically, we shift the input left by an amount determined by the corresponding bit in the 8-bit weight. These shifts are executed concurrently in a single processing stage, and the interim results are stored temporarily. Then, a reduction step combines the interim results. For 8-bit weights, this step necessitates three stages. Performing integer multiplication in the data plane is viable for executing an inference, but it takes too many switch resources (stages). Therefore, we adopt an alternative approach where we store all the dot product results in a table to reduce the number of required stages. We store all possible dot products with 8-bit to 8-bit values and use a TCAM table to access the dot product with value | weight. This approach requires one stage instead of three stages for the dot product with the memory requirement of 2^{16} 8-bit values (65 KB of memory). Next, we employ the ReLU activation function to process the dot product result, which involves examining the most significant bit (MSB), the sign bit, and substituting the value with 0 when it is set to 1.

To perform MaxPooling operations, we need to be sure that all the values regarding a MaxPooling window (a MaxPooling window is a rectangular region of the values where the most important value is selected) have arrived at the switch. To ensure that, we use a bitmap (using a table) of each value and flag its indexes when they arrive. Anytime a packet arrives, we check the bitmap of the value. If all packets have arrived, we fetch the results of all the packets from the convolutional layer regarding the same MaxPooling window and find the maximum of the values within $log(size(MaxPoolingwindow))$. After calculating the maximum value in the MaxPooling window, we multicast it to the following switch pipelines.



(a) 2-tier Clos topology

(b) 3-tier Clos topology

Figure P6.5: The target network architecture.

Dense layers. For the dense layer, finding the index of the dense neuron is relatively straightforward. When a packet arrives at a switch running part of a dense layer, we use a counter to access the dense table entry (the data structure we use for storing the dense weights) to iterate over all the dense weights. After the calculation, we store the intermediate results in a register array and aggregate them as soon as the processing associated with that neuron is finished. We repeat the procedure until all variables from the previous layer are processed. After that, we perform a ReLU activation function (explained above) to see if we want to activate the neuron. Finally, we multicast the result to the next switch pipelines.

P6.3.4 *Packet Generator*

The data center network architecture we use is depicted in Figure P6.5, which represents a multi-tier Clos network topology [19]. Each tier has four programmable switches, each with four pipelines with ingress and egress parts. Each switch in a tier is connected to all the switches in the above and bellow tiers (if available) with 400 GbE links. For instance, there is exactly one link from *Switch 1* (pipeline 4) in the first tier to *Switch 8* (pipeline 1) in the second tier. The advantage of

Clos is that every switch from each tier is exactly two hops away from another switch in the same tier. Leveraging this property, we map the layers of the DNN exclusively to the lowest tier switches, mitigating synchronization issues and eliminating potential stragglers within the network. The downside of this architecture is that there is no direct link between all the pipelines of a switch in a tier to another switch in the above or below tier. Therefore, we may need another tier to send packets to the correct switch pipeline. In a similar example, there is no direct link between pipeline 1 of *Switch 1* to pipeline 1 of *Switch 8*. Consequently, if we want to have the packet emitting from pipeline 1 of *Switch 1* in ingress part of pipeline 1 of *Switch 8*, we need to send the packet to a middle switch and then get the packet from the middle switch that is directly connected to pipeline 1 of *Switch 8*, which in this scenario are pipeline 4 of *Switch 1* and pipeline 4 of *Switch 9*.

After a packet arrives in a switch (say *Switch 1* (1)), the processing part will start in the ingress part of the switch. If all the processing related to the packet is finished in the ingress (see Section P6.3.3), we use a 2-tier Clos network architecture as illustrated in Figure P6.5a. The result in the ingress part will be distributed to all egress parts of the switch using the traffic manager (2). Next, the result is encapsulated into packets in the egress part and the packets are emitted to the switches in a higher tier (3). Then, the switches in the higher tier direct the packets to the corresponding pipelines to be sent to a specific pipeline in the lower tier switch (4). Finally, all the pipelines of *Switch 2* receive the packets with the result from the initial switch (*Switch 1*)) (5). All the associated links are colored.

However, if the processing part requires both the ingress and ingress part of the pipeline, we may need another tier to multicast the result, as illustrated in Figure P6.5b. Taking a similar scenario, sending the result of a packet in pipeline 1 of *Switch 1* to all pipelines of *Switch 2*, instead of distributing the packet in *Switch 1*, we need to send the packet encapsulating the result to the second tier switch (2), *Switch 5* here, and distribute it to all pipelines using the traffic manager in the switch (3). After that, we follow the similar steps as a 2-tier Clos architecture until all the packets are in the correct pipelines of the second-tier switches (4, 5, 6, 7). One of the second-tier switches has the packet in the correct pipeline without sending it to a higher-tier switch (pipeline 2 of *Switch 5*). To avoid synchronization issues, we follow the same procedure for all the second-tier switches. Finally,

we send the packets to all pipelines of *Switch 2* (8). All the used links are colored in this scenario as well.

P6.4 EVALUATION

We implemented a preliminary version of NetNN using P4 language with the behavioral model bmv2, which serves as a P4 target. This prototype is used for testing within Mininet, a network emulator tailored for network experimentation. Within Mininet, we established a network topology illustrated in Figure P6.3 with four switches per layer (since there are four pipelines per switch in a Tofino switch) as depicted in Figure P6.4, and switches are connected to each other similar to the link connections in Figure P6.5b.

P6.4.1 *Resource Usage*

By executing the P4 implementation on hardware, we can achieve the switch's line-rate performance and naturally leverage the match-action pipeline's characteristics, including bounded latency guarantees and high-speed throughput. However, in our prototype, the software emulation on bmv2 lacks the necessary time accuracy for precise performance measurements at a granular level. Therefore, we report the memory usage, the number of operations, and the packet generation rates of the NetNN implementation. As presented in Table P6.1, the convolutional layers require much more operations per packet than dense layers, hence more switches are beneficial to distribute the computations. On the other hand, dense layers require more memories to store the weights, yet much lower than a programmable switch memory limitation.

P6.4.2 *Impact of Parameters*

Impact of inference points. The number of inference points used to determine classification is an important factor affecting performance. The more packets from the same flow that we observe and analyze, the more accurate the features at the flow level become, and hence,

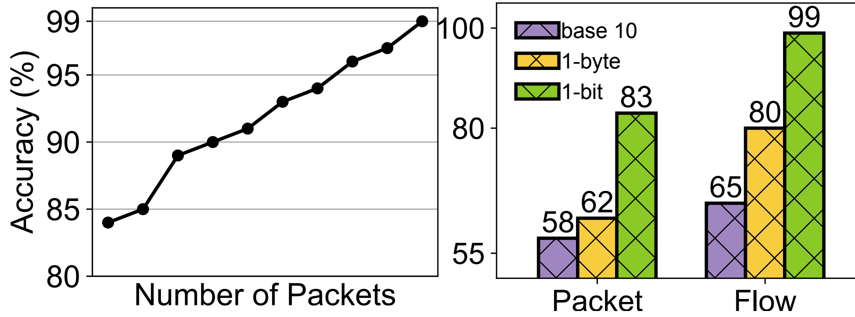


Figure P6.6: Inference points.

Figure P6.7: Input representation.

Table P6.1: DNN in-switch performance. Memory is in byte.

Layer	# of packets	Operations per packet	Memory
Conv1	3240	644	1296
Conv2	2592	38895	5680
Dense1	496	3159	13
Dense2	4	4950	25
Dense3	1	378	1

the more accurate the flow classification confidence. As depicted in Figure P6.6, a clear trend shows that the accuracy improves as we incorporate more inference points for the flow classification. If the goal of the classification is 90% accuracy, the system needs at least four packets of the same flow; if the goal is to reach over 95%, the system requires at least eight packets to be analyzed.

Impact of input representation. To fine-tune the DNN for intrusion detection, we used different input formats to improve the model's accuracy. We trained the same model with three different input formats: bit, byte, and base 10 to demonstrate the model accuracy with different data representations. Figure P6.7 illustrates the accuracy of the model trained with different input formats. Since the number of flows in the dataset is limited (1000 flows) and each flow provides few features, the model trained with base 10 format not only provides the least accuracy in both packet and flow classification but also requires feature engineering (calculating the features in base 10), while the model trained with 1-bit format provides the highest accuracy without the need for feature engineering since that is the natural format of network packets.

P6.5 RELATED WORK

Intrusion detection. Due to switches' strategic location and performance, there is an increased usage of switches for intrusion detection [7, 15, 23]. FlowLens [3] leverages programmable switches to support machine learning-based network security applications by collecting features of packet distributions and classifying flows on the switches. Jaqen [13] addresses the challenge of defending against volumetric DDoS attacks by leveraging universal sketch techniques, switch-native mitigation, and network-wide management. Poseidon [26] provides a modular policy abstraction for DDoS defense policies. NetBeacon [29] proposes a multi-phase sequential model architecture for dynamic analysis using a switch-based model representation.

ML-based applications. In-network ML has gained significant attentions [18, 24]. IIsy [27] is a framework for in-network classification using off-the-shelf network devices. It maps trained ML models to network devices without modifications. FPISA [25] proposes a floating point representation in programmable switches for efficient distributed training. Mousika [22] proposes a Binary Decision Tree (BDT) model that enables translation from complex models to BDT using knowledge distillation. N3IC [20] enables neural network inference on programmable NICs. Planter [28] maps ensemble models to programmable switches for efficient data classification. SwitchTree [11], pForest [4], and FlowRest [1] enhance in-switch inference with Random Forest models.

None of these works directly apply DNNs for intrusion detection completely in the network data plane.

P6.6 CONCLUSION

In this paper, we presented NetNN, a novel approach to enable the execution of DNNs in programmable switches. NetNN achieves this by splitting and distributing the neural network layers to multiple switches, generating packets similar to the flow execution of the neural network, and simplifying the needed mathematical operations for getting the inference based on the capabilities of the programmable switches. We prototype NetNN using P4 and evaluate it by designing a

no-feature-engineering-needed DNN using a Covert Channel dataset, showing an intrusion detection accuracy of 99%.

P6.7 ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments and suggestions. This work has been funded by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - Project-ID 210487104 - SFB 1053.

REFERENCES

[1] Aristide Tanyi-Jong Akem, Michele Gucciardo, and Marco Fiore. “Flowrest: Practical flow-level inference in programmable switches with random forests.” In: *IEEE INFOCOM 2023-IEEE Conference on Computer Communications*. IEEE. 2023, pp. 1–10.

[2] Diogo Barradas, Nuno Santos, and Luís Rodrigues. “DeltaShaper: Enabling unobservable censorship-resistant TCP tunneling over videoconferencing streams.” In: *Proceedings on Privacy Enhancing Technologies* (2017).

[3] Diogo Barradas, Nuno Santos, Luís Rodrigues, Salvatore Signorello, Fernando MV Ramos, and André Madeira. “FlowLens: Enabling Efficient Flow Classification for ML-based Network Security Applications.” In: *NDSS*. 2021.

[4] Coralie Busse-Grawitz, Roland Meier, Alexander Dietmüller, Tobias Bühler, and Laurent Vanbever. “pforest: In-network inference with random forests.” In: *arXiv preprint arXiv:1909.05680* (2019).

[5] Tianqi Chen and Carlos Guestrin. “Xgboost: A scalable tree boosting system.” In: *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 2016, pp. 785–794.

[6] Roberto Doriguzzi-Corin, Luis Augusto Dias Knob, Luca Mendozzi, Domenico Siracusa, and Marco Savi. “Introducing packet-level analysis in programmable data planes to advance Network Intrusion Detection.” In: *Computer Networks* 239 (2024), p. 110162.

[7] Chuanpu Fu, Qi Li, Meng Shen, and Ke Xu. "Realtime robust malicious traffic detection via frequency domain analysis." In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2021, pp. 3431–3446.

[8] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. "Serving DNNs like Clockwork: Performance Predictability from the Bottom Up." In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2020, pp. 443–462.

[9] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. "NetChain: Scale-Free Sub-RTT Coordination." In: *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*. Ed. by Sujata Banerjee and Srinivasan Seshan. USENIX Association, 2018, pp. 35–49.

[10] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. "NetCache: Balancing Key-Value Stores with Fast In-Network Caching." In: *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 2017, pp. 121–136. DOI: 10.1145/3132747.3132764.

[11] Jong-Hyouk Lee and Kamal Singh. "Switchtree: in-network computing and traffic analyses with random forests." In: *Neural Computing and Applications* (2020), pp. 1–12.

[12] Shuai Li, Mike Schliep, and Nick Hopper. "Facet: Streaming over videoconferencing for censorship circumvention." In: *Proceedings of the 13th Workshop on Privacy in the Electronic Society*. 2014, pp. 163–172.

[13] Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. "Jaqen: A High-Performance Switch-Native approach for detecting and mitigating volumetric DDoS attacks with programmable switches." In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 3829–3846.

[14] Gonzalo Marín, Pedro Casas, and Germán Capdehourat. "DeepSec meets RawPower-deep learning for detection of network attacks using raw representations." In: *ACM SIGMETRICS Performance Evaluation Review* 46.3 (2019), pp. 147–150.

[15] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. “Kitsune: an ensemble of autoencoders for online network intrusion detection.” In: *arXiv preprint arXiv:1802.09089* (2018).

[16] Kamran Razavi, George Karlos, Vinod Nigade, Max Mühlhäuser, and Lin Wang. “Distributed DNN serving in the network data plane.” In: *Proceedings of the 5th International Workshop on P4 in Europe*. 2022, pp. 67–70.

[17] Kamran Razavi, Manisha Luthra, Boris Koldehofe, Max Mühlhäuser, and Lin Wang. “FA2: Fast, Accurate Autoscaling for Serving Deep Learning Inference with SLA Guarantees.” In: *2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2022, pp. 146–159. DOI: 10.1109/RTAS54340.2022.00020.

[18] Amedeo Sapio, Marco Canini, Chen-Yu Ho, Jacob Nelson, Panos Kalnis, Changhoon Kim, Arvind Krishnamurthy, Masoud Moshref, Dan R. K. Ports, and Peter Richtárik. “Scaling Distributed Machine Learning with In-Network Aggregation.” In: *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021*. Ed. by James Mickens and Renata Teixeira. USENIX Association, 2021, pp. 785–808.

[19] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. “Jupiter rising: A decade of clos topologies and centralized control in google’s datacenter network.” In: *ACM SIGCOMM computer communication review* 45.4 (2015), pp. 183–197.

[20] Giuseppe Siracusano, Salvator Galea, Davide Sanvito, Mohammad Malekzadeh, Gianni Antichi, Paolo Costa, Hamed Haddadi, and Roberto Bifulco. “Re-architecting Traffic Analysis with Neural Network Interface Cards.” In: *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, April 4-6, 2022*. USENIX Association, 2022, pp. 513–533.

[21] Tushar Swamy, Alexander Rucker, Muhammad Shahbaz, Ishan Gaur, and Kunle Olukotun. “Taurus: a data plane architecture for per-packet ML.” In: *ASPLOS ’22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*. Ed. by Babak Falsafi, Michael Ferdman, Shan

Lu, and Thomas F. Wenisch. ACM, 2022, pp. 1099–1114. DOI: 10.1145/3503222.3507726.

[22] Guorui Xie, Qing Li, Yutao Dong, Guanglin Duan, Yong Jiang, and Jingpu Duan. "Mousika: Enable general in-network intelligence in programmable switches by knowledge distillation." In: *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*. IEEE. 2022, pp. 1938–1947.

[23] Jiarong Xing, Qiao Kang, and Ang Chen. "NetWarden: Mitigating Network Covert Channels while Preserving Performance." In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 2039–2056.

[24] Zhaoqi Xiong and Noa Zilberman. "Do Switches Dream of Machine Learning?: Toward In-Network Classification." In: *Proceedings of the 18th ACM Workshop on Hot Topics in Networks, HotNets 2019, Princeton, NJ, USA, November 13-15, 2019*. ACM, 2019, pp. 25–33. DOI: 10.1145/3365609.3365864.

[25] Yifan Yuan, Omar Alama, Jiawei Fei, Jacob Nelson, Dan RK Ports, Amedeo Sapio, Marco Canini, and Nam Sung Kim. "Unlocking the power of inline Floating-Point operations on programmable switches." In: *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 2022, pp. 683–700.

[26] Menghao Zhang, Guanyu Li, Shicheng Wang, Chang Liu, Ang Chen, Hongxin Hu, Guofei Gu, Qianqian Li, Mingwei Xu, and Jianping Wu. "Poseidon: Mitigating volumetric ddos attacks with programmable switches." In: *the 27th Network and Distributed System Security Symposium (NDSS 2020)*. 2020.

[27] Changgang Zheng, Zhaoqi Xiong, Thanh T Bui, Siim Kaupmees, Riyad Bensoussane, Antoine Bernabeu, Shay Vargaftik, Yaniv Ben-Itzhak, and Noa Zilberman. "IIsy: Practical in-network classification." In: *arXiv preprint arXiv:2205.08243* (2022).

[28] Changgang Zheng and Noa Zilberman. "Planter: seeding trees within switches." In: *Proceedings of the SIGCOMM'21 Poster and Demo Sessions*. 2021, pp. 12–14.

[29] Guangmeng Zhou, Zhuotao Liu, Chuanpu Fu, Qi Li, and Ke Xu. "An efficient design of intelligent network data plane." In: *32nd USENIX Security Symposium (USENIX Security 23)*. 2023, pp. 6203–6220.

ERKLÄRUNG

Hiermit erkläre ich, die vorgelegte Arbeit zur Erlangung des akademischen Grades Doktor rerum naturalium (Dr. rer. nat.) mit dem Titel

Resource Efficient Inference Serving With SLO Guarantee

selbständig und ausschließlich unter Verwendung der angegebenen Hilfsmittel erstellt zu haben. Ich habe bisher noch keinen Promotionsversuch unternommen.

Darmstadt, 12.08.2024

Kamran Razavi, M.Sc.