Towards SLO-aware ML Inference on Serverless Platforms

Abhijit Tripathy

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science and Application

Ali R. Butt, Chair

Dimitrios S Nikolopoulos

M. Mustafa Rafique

June 27, 2023

Blacksburg, Virginia

Keywords: Machine Learning, Deep Learning, Serverless Inference, Autoscaling, Load Balancing, Response Time, Tail Latency

Copyright 2023, Abhijit Tripathy

Towards SLO-aware ML Inference on Serverless Platforms

Abhijit Tripathy

(ABSTRACT)

The rapid advancement of Machine Learning (ML) and Deep Learning (DL) has revolutionized various domains, necessitating efficient and cost-effective ML inference capabilities. Function-as-a-Service (FaaS) has emerged as a promising approach for hosting ML inference services, providing a serverless computing environment that streamlines development cycles and offers scalability and simplified infrastructure management. However, existing autoscaling strategies employed by popular FaaS platforms often overlook critical factors such as response time and tail latency. Additionally, Python's Global Interpreter Lock (GIL) poses challenges for parallel computing in high-request traffic scenarios. This thesis addresses the need for efficient and cost-effective Machine Learning (ML) inference capabilities by exploring batching and autoscaling strategies for Serverless Inference instances. The study proposes a prototype FaaS framework that provides adaptive request batching, reactive autoscaling policies, and SLO monitoring, thus allowing Serverless Inference workloads to meet their SLO targets even during peak traffic. The proposed approach aims to optimize resource utilization, mitigate tail latency, and improve overall system performance.

Towards SLO-aware ML Inference on Serverless Platforms

Abhijit Tripathy

(GENERAL AUDIENCE ABSTRACT)

Machine Learning (ML) and Deep Learning (DL) are sophisticated methodologies that enable computers to acquire knowledge from data and make predictions or decisions autonomously. This has led to significant advancements in various fields. Inference refers to the process of applying a trained ML model to new data to make predictions or extract insights. In the context of ML, there is a growing need for efficient and cost-effective inference capabilities. A new approach called Function-as-a-Service (FaaS) has emerged that can address this need. FaaS is a way of abstracting the server infrastructure away from the developers. This means developers can focus on writing the ML code without worrying about managing the underlying infrastructure. FaaS offers benefits such as scalability, simplified infrastructure management, and faster development cycles. However, existing FaaS platforms face challenges in ensuring fast response times and handling high levels of incoming requests. This thesis aims to address these challenges by proposing a prototype FaaS framework. The framework incorporates adaptive request batching, reactive autoscaling policies, and Service-Level Objectives (SLOs) monitoring. Request batching allows the framework to process multiple requests together, improving efficiency. Autoscaling policies ensure the system dynamically adjusts its resources based on the incoming workload. Monitoring SLOs helps track and meet performance targets, even during peak traffic. By optimizing resource utilization, reducing delays in processing requests, and improving overall system performance, the proposed approach seeks to provide efficient and cost-effective ML inference capabilities in a serverless environment.

Acknowledgments

The completion of this Master's thesis would not have been possible without the contributions and support of several individuals, to whom I would like to express my deepest gratitude. First and foremost, I am immensely grateful to Dr. Ali R. Butt, my advisor, for his expertise in computer science and dedication to research. Dr. Butt's guidance and mentorship have been instrumental in shaping me into a better researcher. The countless hours he dedicated to reviewing my work, offering valuable insights, and pushing me to reach my full potential have been invaluable. I am honored to have had the opportunity to work under Dr. Butt's supervision and greatly appreciate his unwavering support throughout my graduate studies. I would also like to extend my heartfelt thanks to the members of my thesis committee, Dr. Dimitrios Nikolopoulos and Dr. M. Mustafa Rafique. Their expertise in computer science and scholarly guidance has played a significant role in refining my research and expanding my knowledge. The time and effort they providing constructive feedback and challenging me to think critically have been greatly appreciated. In addition, I would like to express my gratitude to Dr. Li and Dr. Williams for their valuable contributions to my research in Serverless Systems and File Storage. Their insightful feedback and expertise in these areas have greatly enhanced the quality and direction of my work. Furthermore, I would like to acknowledge the labmates at the Distributed Systems and Storage Laboratory at VT. Their camaraderie and shared passion for research have made my experience in grad school both enjoyable and intellectually stimulating. The collaborative environment we fostered within the lab has allowed me to grow both professionally and personally. I am grateful for the insightful discussions, exchange of ideas, and collective pursuit of knowledge that have significantly enriched my graduate journey. The support and friendship of each labmate

have been invaluable, and I appreciate their contributions to making my academic pursuit an exciting and fulfilling one. Finally, I would like to express my heartfelt appreciation to my family and friends for their unwavering support throughout my academic endeavors. Their encouragement, understanding, and belief in my abilities have been the pillars of my success. Their constant presence and words of encouragement have provided me with the strength and resilience to overcome challenges and pursue my aspirations. I am truly grateful for their love and support. In conclusion, I extend my sincere gratitude to all the individuals mentioned above for their contributions, guidance, and support. Their belief in my abilities and their commitment to my academic and personal growth has played an instrumental role in shaping me as a researcher. I am forever grateful for their invaluable contributions, and I will carry the knowledge and experiences gained from this journey into my future endeavors.

Contents

List of Figures ix

List of Tables x

1 Introduction 1

1.1 Motivation 1

1.2 Approach 4

1.3 Contributions 5

1.4 Document Overview 6

2 Background 7

2.1 Serverless Computing 7

2.2 Machine Learning Inference 9

2.3 Challenges 10

2.3.1 Cold Start Latency 10

2.3.2 Resource Limitations 12

2.3.3 Throughput 12

2.3.4 Latency 14

3 Design & Implementation **16**

3.1 Overview 16

3.2 Adaptive Batching 18

3.3 Feedback control-based Autoscaling 21

3.3.1 Real-time Request-Arrival-Rate Monitoring 21

3.3.2 SLO Monitor 23

4 Evaluation **24**

4.1 Experimental Setup 24

4.2 Prototype vs. TorchServe 25

4.3 Replica Count vs. Latency 26

4.4 Autoscaling vs. Latency 27

4.5 Stress Testing 32

4.6 CPU and Memory Utilization 33

5 Related Work **34**

5.1 Related Work 34

6 Conclusion **37**

6.1 Limitations and Future Work 37

6.2 Summary 39

List of Figures

2.1 *CDF* for concurrent requests to BERT service on OpenFaaS with 8 replicas . 15

3.1 Prototype Architecture 17

4.1 CDF for Load Testing on BERT Service running on our Prototype vs TorchServe 25

4.2 P80 Latency of Prototype vs TorchServe 26

4.3 *CDF* for concurrent load testing on the BERT service with varying replica counts. All tests are done with 500 concurrent users, with randomized spawn rates to simulate sporadic request traffic. $(S_B = 128, T_B = 100ms)$ 28

4.4 *CDF* for concurrent load testing on the BERT service with varying replica counts. All tests are done with 1000 concurrent users, with randomized spawn rates to simulate sporadic request traffic. $(S_B = 128, T_B = 100ms)$ 29

4.5 Distributed Load Testing for the BERT Service with autoscaling engine enabled. The demarcation line denotes the timestamp when the autoscaler successfully scales up the number of pods from $1-3$. 30

4.6 Distributed Load Testing for the BERT Service with autoscaling engine enabled. The autoscaler scales the number of replicas from $1-5$ by 12:30:24 PM. The highest throughput reaches close to 3000 rps. 31

4.7 *CDF* for concurrent requests to BERT service on ServeML(1 - 8 replica count) 32

List of Tables

1.1 Table of various autoscaling strategies provided for Functions associated with HTTP events 3

2.1 Total Execution time for running Inference using Serverless Functions on OpenFaaS 11

2.2 Inference Time (ms) for various batch sizes for *bert-base-multilingual-uncased-sentiment* model 13

4.1 Throughput(RPS), and tail latency(P80-P100) for distributed load testing on BERT Service running on our prototype 27

4.2 Request Arrival Rate distribution and Request Latency for BERT service with 5 function instances 30

Chapter 1

Introduction

1.1 Motivation

The advent of Machine Learning (ML) and Deep Learning (DL) has sparked a transformative revolution across numerous domains, leaving an indelible mark on fields like image recognition [1, 2, 3], natural language processing [4, 5, 6, 7], recommendation systems [8, 9], and more. Within the realm of ML, workflows can be broadly classified into two key stages: *training* and *inference*. Extensive research [10, 11, 12, 13, 14] has been conducted on ML training in modern cloud environments, yielding invaluable insights and optimizations. However, equal emphasis must be placed on the subsequent stage of the ML pipeline: inference. Once these models have been trained, their true potential is realized when deployed to make predictions and decisions on new, previously unseen data (e.g., discerning textual data content or identifying subjects within an image). The growing prevalence of ML and DL has fueled a corresponding surge in demand for efficient and cost-effective inference capabilities [15, 16, 17]. Consequently, exploring novel strategies to streamline and elevate the inference process has emerged as a critical endeavor, propelling the boundaries of machine intelligence's achievable goals [18, 19].

Organizations are exploring innovative solutions to host and deploy their ML models as the demand for efficient and cost-effective ML inference capabilities continues to rise. One emerging approach gaining traction is the adoption of Function-as-a-Service (FaaS) for host-

ing ML inference services. FaaS provides a serverless computing environment where developers can focus solely on writing application logic without the need to manage the underlying infrastructure. FaaS promotes faster development cycles and promises effortless scalability, fine-grained resource billing, and simplified infrastructure management. As a result, FaaS has gained significant attention from academia for supporting ML Inference workflows [20, 21, 22, 23, 24]. Commercial Cloud Providers like AWS [25] have built a fully-managed Serverless Platform [26] to run Inference workloads.

While ML *training* predominantly takes place offline, *inference* is a real-time and synchronous operation requiring a timely response to user requests. Typically, users send HTTP requests to the system, expecting quick and efficient processing of their queries. In the context of ML inference, these requests come with stringent latency requirements, often a few hundred milliseconds. These latency targets, called response-time Service Level Objectives (SLOs), are crucial metrics for measuring and ensuring optimal user experience.

Meeting response-time SLOs is paramount, as failing to comply can lead to deteriorated user satisfaction, loss of business opportunities, and financial repercussions. When considering the adoption of Serverless Platforms for ML inference, it becomes imperative for these platforms to offer robust response-time guarantees for their function instances. To achieve strong response-time guarantees, Serverless Platforms for ML Inference must employ optimizations like downscale stabilization, minimizing cold start delays, and efficiently handling workload fluctuations. Techniques like proactive resource allocation [27], predictive scaling [28], and smart load balancing [29] become critical in maintaining low-latency response times and meeting the defined SLOs.

The tail latency of HTTP requests to Serverless Inference Functions is influenced by several factors, particularly in high-request traffic scenarios. When the request load increases, Serverless Function instances must be scaled up to distribute the incoming traffic effectively.

Table 1.1: Table of various autoscaling strategies provided for Functions associated with HTTP events

FaaS Platforms	Autoscaling Strategy	Metrics
AWS Lambda	event-based scaling	rate(http_requests), Concurrency Utilization
Azure Functions	event-based scaling	rate(http_requests)
Google Cloud Functions	event-based scaling	rate(http_requests)
OpenFaaS	target-based scaling	CPU, RPS, Concurrency

However, factors not adequately captured by current autoscaling strategies employed by popular FaaS platforms can still affect tail latency.

We surveyed the autoscaling strategies utilized by the four most-popular FaaS platforms, including AWS Lambda [30], Azure Functions [31], Google Cloud Functions [32], and OpenFaaS [33], and we present our findings in Table 1.1. AWS Lambda offers autoscaling based on the number of incoming HTTP requests or the utilization rate of provisioned instances. Azure Functions and Google Cloud Functions primarily provide event-based scaling for HTTP-based triggers, similar to AWS Lambda. OpenFaaS, on the other hand, offers more flexibility by allowing the customization of autoscaling policies based on three custom metrics: CPU utilization, requests-per-second (RPS) for each function, and concurrency. However, none of these metrics directly capture the response time of the function instance or the tail latency experienced by users. Moreover, AWS Lambda, Azure Functions, and GCP Functions follow a one-to-one mapping for incoming requests and function instances, i.e., each function only processes one request at a time. Therefore, when multiple concurrent requests arrive at a function's API gateway, many concurrent instances of the functions are spawned with a one-to-one mapping with the number of outstanding requests. This will be extremely wasteful regarding resource utilization during high volumes of incoming traffic and has been a limiting factor for adopting Serverless in low-latency workloads.

Another crucial aspect of serverless inference workflows is the execution of custom inference

logic on pre-loaded models by inference functions. Python is a popular choice for writing deep learning code in this context. However, Python threads suffer from a limitation known as the Global Interpreter Lock (GIL) [34], which hinders parallel computing advantages. Consequently, when a Python inference function instance receives multiple concurrent requests, they experience serialization due to the GIL, resulting in significant tail latency. Thus, even with highly scaled function instances, high request traffic can lead to notable tail latency for users.

These insights highlight the need for more refined autoscaling strategies considering metrics like request arrival rate, response time, and tail latency rather than relying solely on request count or CPU/memory utilization rates. Additionally, exploring adaptive batching techniques to bypass the limitations imposed by the GIL can help mitigate the tail latency challenges experienced during high request traffic periods.

1.2 Approach

This paper examines the performance of Python-based Inference services and their ability to meet request-latency SLO targets when run as Serverless functions. We use OpenFaaS [33], an open-source FaaS framework that can be easily integrated with Kubernetes for a distributed deployment. Additionally, we explore a range of metrics to ensure autoscaling decisions align with the SLO target, such as request arrival rate, response latency, and batching hyperparameters. By comparing the performance of these metrics and evaluating their impact on autoscaling decisions, we aim to identify effective autoscaling strategies that can optimize resource allocation and enhance user experience in Serverless Inference environments.

Based on our findings, we build a prototype by modifying OpenFaaS components that use

adaptive batching and a combination of predictive and reactive autoscaling strategies to optimize Serverless Inference workflows. During peak hours, efficient load balancing becomes crucial for routing traffic and ensuring the smooth operation of Serverless Inference functions. The Load Balancer in our prototype uses an Exponentially Weighted Moving Average (EWMA) based algorithm to distribute incoming requests to the fastest endpoints. By evaluating factors such as request distribution, latency, and scalability, we serve to inform load-balancing strategies that can maximize resource utilization, minimize response times, and maintain robust performance during periods of peak traffic.

1.3 Contributions

We propose a prototype built on the OpenFaaS framework incorporating adaptive batching for Serverless Inference functions. Our prototype integrates feedback control-based autoscaling mechanisms, leveraging latency and request arrival rate metrics to drive autoscaling decisions. By dynamically adjusting batch sizes based on workload patterns and utilizing latency metrics as feedback signals, our approach aims to optimize resource utilization and mitigate tail latency issues, ultimately improving overall system performance.

Furthermore, we outline potential avenues for extending our work, such as incorporating predictive autoscaling techniques, enabling GPU support for accelerated inference, and developing a unified in-memory model store for efficient model sharing and retrieval. These future enhancements can further enhance the capabilities and flexibility of the proposed FaaS prototype, opening up opportunities for advanced optimization and customization in Serverless Inference environments.

1.4 Document Overview

Chapter 2 provides an in-depth overview of the key concepts and technologies underpinning Serverless Inference. It covers the fundamentals of Machine Learning inference, the benefits and challenges of deploying ML models in a serverless environment, adaptive request batching, and the importance of meeting response-time Service Level Objectives (SLOs) for optimal user experience.

Chapter 3 delves into the proposed approach for optimizing Serverless Inference. It includes an analysis of different autoscaling strategies, focusing on evaluating various metrics to ensure SLO adherence. Furthermore, this section explores adaptive batching for Serverless Inference Functions, and feedback control-based autoscaling using network-level metrics to drive scaling decisions.

Chapter 4 focuses on the empirical evaluation of the proposed approach. It presents the experimental setup, datasets, and performance metrics used to assess the effectiveness of the autoscaling strategies, load balancing methods, and the developed FaaS prototype.

Chapter 5 comprehensively reviews the existing literature and research efforts on optimizing Serverless Inference. It highlights notable studies on autoscaling strategies, load-balancing methods, and other relevant areas, providing a comprehensive context for the current research.

Chapter 6 discusses the potential for future work and summarizes this thesis's key findings and contributions.

Chapter 2

Background

2.1 Serverless Computing

In recent years, serverless computing has emerged as a transformative technology reshaping the landscape of cloud computing. Traditional server-based infrastructures require businesses to provision and manage servers, leading to capacity planning, infrastructure maintenance, and scalability challenges. Serverless computing addresses these limitations by abstracting away the server infrastructure and providing a pay-as-you-go model based on resource usage. By leveraging serverless platforms, developers can focus on writing and deploying code without worrying about infrastructure management. This paradigm shift offers compelling advantages, including automatic scaling, rapid development cycles, reduced operational overhead, and cost optimization. As a result, serverless computing has garnered significant attention and adoption across industries [30, 31, 32, 33] and academia [35, 36, 37], transforming how applications are developed and deployed.

The core concept of serverless computing lies in its ability to execute code responding to events or triggers, known as Function-as-a-Service (FaaS). When an event occurs, such as an HTTP request or a database update, the serverless platform dynamically provisions the necessary resources and executes the corresponding function. This event-driven approach enables highly scalable and elastic architectures, where resources are allocated on demand to meet workload fluctuations. Additionally, the serverless model promotes a stateless execution

environment, allowing functions to be executed independently without relying on a shared state or context. This statelessness contributes to improved scalability, fault tolerance, and isolation, as each function invocation is independent and can be executed in parallel.

The benefits of serverless computing extend beyond traditional cloud computing challenges and have found particular relevance in modern application development, including microservices architectures, linear algebra [38], and real-time video encoding [39]. Furthermore, serverless computing is gaining traction in emerging technologies such as machine learning, where it offers a scalable and cost-effective platform for deploying and executing machine learning models for inference tasks [20, 21, 22, 24, 26, 40, 41]. With its inherent advantages of automatic scaling, cost optimization, and simplified deployment, serverless computing provides an attractive environment for running resource-intensive machine learning workloads.

While serverless computing offers significant advantages, it is essential to acknowledge its limitations. One notable limitation is the potential for cold start latency. When a function is invoked for the first time or after a period of inactivity, there may be a delay as the serverless platform provisions and initializes the necessary resources to execute the function. This latency can impact real-time and interactive applications where low latency is crucial. Furthermore, serverless environments impose resource limitations such as CPU, memory, and storage restrictions. These limitations are implemented to ensure fair resource allocation among multiple functions running on the platform. It is important for developers to carefully consider these limitations and optimize their functions accordingly to avoid performance issues. We elaborate on these challenges and some of the proposed solutions in 2.3.

2.2 Machine Learning Inference

Machine learning has revolutionized various domains by enabling computers to learn patterns and make intelligent decisions from data. While the training of machine learning models receives considerable attention, the deployment and execution of these models in real-world applications are equally crucial. Machine learning inference, the process of applying pre-trained models to new data for making predictions or extracting insights, plays a pivotal role in leveraging the power of machine learning in practical scenarios. Inference tasks are characterized by their need for low latency, real-time processing, and high-throughput capabilities. As organizations increasingly rely on machine learning models for decision-making and automation, efficient and scalable inference deployment becomes paramount. This section provides an overview of machine learning inference, its significance, and the challenges it presents in terms of computational requirements and scalability.

Machine learning inference involves using pre-trained models to process new input data and produce predictions or valuable outputs. During training, machine learning models learn from labeled data to generalize patterns and relationships. Once trained, these models can be deployed in production systems to perform real-time inference tasks. *Inference* is commonly employed in a wide range of applications, including image and speech recognition [1, 3], natural language processing [4, 5, 6], fraud detection, recommendation systems [9], and autonomous vehicles, to name a few. ChatGPT [4] has 1.16 billion monthly users and has set the record for the fastest-growing user base [42].

Efficient and scalable execution of machine learning inference is crucial for ensuring timely and accurate predictions. Inference tasks often require processing large volumes of data and performing complex computations. Traditional approaches, such as running inference on dedicated servers or high-performance computing clusters, can be costly and difficult to scale.

The demand for cost-effective and scalable solutions has led researchers and practitioners to explore alternative deployment strategies, such as serverless computing [16, 17, 21, 22, 24, 40], which offers the potential for efficient and flexible execution of machine learning inference tasks.

The integration of serverless computing with machine learning inference holds great promise. Serverless architectures enable automatic scaling and resource allocation based on workload fluctuations, ensuring optimal performance and cost efficiency for inference tasks. By leveraging the event-driven nature of serverless platforms, machine learning models can be seamlessly integrated into serverless functions, enabling real-time and on-demand processing of new data. Additionally, the pay-per-use pricing model of serverless computing aligns well with the sporadic nature of inference workloads, allowing organizations to optimize costs based on actual resource consumption.

2.3 Challenges

2.3.1 Cold Start Latency

One notable challenge in serverless computing that can significantly impact ML inference is the issue of cold start latency. Cold start refers to the delay experienced when a function is invoked for the first time or after a period of inactivity. This delay occurs because the serverless platform needs to provision and initialize the necessary resources to execute the function. For OpenFaaS, this includes launching a new pod with the container image of the running function and letting the function init process boot up. This can lead to a cold start latency of $4 - 5$ seconds. This latency will increase further if the function instances need to download the ML models during the function init process. We ran inference on two

different models using Python functions on OpenFaaS to demonstrate the effect of cold starts. We used a lightweight Convolutional Neural Network and Google's BERT [43], a language transformer similar to modern-day GPT in architecture. Table 2.1 shows that cold start latency is significant compared to actual inference time. Moreover, as models grow larger in size, loading them in memory to serve inference requests also becomes significantly larger than the actual inference computation time. For Google's T5 [44] text-to-text transformer model (*t5-large*), model loading time is almost 20× the inference cost.

Cold start latency can considerably impact real-time applications requiring low-latency responses in serverless inference. The time taken to initialize the infrastructure and load the machine learning model can lead to increased response times, negatively affecting the overall user experience. Addressing the cold start latency challenge in serverless inference is crucial for ensuring the fast and efficient execution of machine learning models, particularly in time-sensitive applications such as real-time recommendations, fraud detection, or autonomous systems. Researchers and practitioners are actively exploring techniques to mitigate cold start latency, such as pre-warming functions [45], optimizing container reuse [46], and employing advanced resource allocation strategies [47] to enable seamless function responses.

Table 2.1: Total Execution time for running Inference using Serverless Functions on OpenFaaS

Model	**Size**	**Boot (s)**	**Model Load Time (s)**	**Inference (s)**
bert-base-uncased	419MB	4.1	1.78412	0.06519
bert-large-uncased	1.3GB	4.1	4.9135	0.07318
t5-small	232MB	4.5	1.08435	0.10406
t5-base	852MB	4.5	3.5046	0.19616
t5-large	2.8GB	4.7	10.69622	0.46764
roberta-base	478MB	3.5	2.21808	0.0392
roberta-large	1.4GB	4.2	5.67003	0.0673

2.3.2 Resource Limitations

Cloud providers typically enforce constraints on CPU, memory, and storage usage for individual serverless functions to ensure fair resource allocation and prevent resource abuse. While these limitations are necessary for maintaining platform stability and preventing resource contention, they can pose challenges for resource-intensive machine learning inference workloads. Machine learning models often require significant computational resources and memory to process datasets and perform complex computations. In scenarios where the resource requirements of a machine learning model exceed the allocated limits, it can lead to performance degradation, out-of-memory errors, or even function failures. Moreover, resource limitations can restrict the size and complexity of the models that can be deployed, potentially limiting the accuracy and capabilities of machine learning inference. Careful optimization and efficient utilization of computational resources are crucial to overcome these challenges. Techniques such as model compression, quantization, and optimizing memory usage can help alleviate the impact of resource limitations on machine learning inference in serverless computing environments. Additionally, monitoring and proactive resource management strategies can help ensure that the allocated resources align with the specific requirements of the machine learning inference tasks, striking a balance between performance and cost efficiency.

2.3.3 Throughput

High throughput is critical in machine learning inference tasks, particularly in scenarios where large requests need to be processed efficiently. In this context, throughput refers to the rate at which inference requests can be handled and is typically measured in terms of the number of inference requests processed per unit of time. High throughput is crucial in

time-sensitive applications, such as real-time recommendations or processing data streams, where low-latency and near-real-time responses are required. However, serverless computing may face challenges providing the desired throughput for machine learning inference.

While enabling scalability, the event-driven nature of serverless platforms can introduce overhead in function initialization and resource provisioning, which may lead to delays in processing subsequent requests. Moreover, individual requests must be served parallel to maintain a decent throughput during a high volume of inference requests. However, Python-based inference code cannot be run in parallel, as Python threads are linearized by the global interpreter lock, thus taking away any parallel computing benefits.

Inference Request Batching

As Python code cannot perform parallel computing, one approach towards scaling Python-based inference backends is to perform request batching. The computational nature of ML inference tasks involves linear algebra and matrix/vector arithmetic. Most computations incur very little overhead when applied to a batch of data instead of a single input. Table 2.2 shows the model inference time for the BERT model variation, *bert-base-multilingual-uncased-sentiment*, for various batch sizes, with fixed `max_length` size of the input. We can observe that with a 16× increase in batch size, the model inference time only increases 4×.

Table 2.2: Inference Time (ms) for various batch sizes for *bert-base-multilingual-uncased-sentiment* model

Batch Size	**Inference Time(ms)**
8	56
16	68
32	95
64	141
128	235

2.3.4 Latency

We conducted a series of experiments to quantify the request latency associated with plain Serverless Functions deployed on the OpenFaaS platform, with and without batching. We use Apache Server Benchmarking tool [48] to generate HTTP traffic toward our inference function. The inference function runs the Sentiment Analysis on text input, using Hugginface's *bert-base-multilingual-uncased-sentiment* transformer model. The Apache Server Benchmarking tool is configured to send N concurrent requests and repeat that process 20 times to record the request latencies, where $N \in \{100, 200, 500, 700, 1000\}$. Figure 2.1 represent the Cumulative Distribution Function for the request latency incurred by our recommendations. Our findings reveal that the request latency exceeds acceptable thresholds as concurrent users increase. This latency issue can be attributed to the inherent limitations of Python-based backends in performing parallel computations effectively. Consequently, incoming requests to a function instance become staggered due to ongoing computations, further contributing to increased latency. To address this challenge, we introduced request batching to minimize the frequency of invocations made to the Python backend. During periods of high request arrival rates, the batcher efficiently utilizes its buffer, resulting in a reduced number of HTTP calls made to the Python backend. This optimization significantly mitigates the tail latency experienced by requests, even when faced with high concurrent user counts. We can observe in Figure 2.1 that with batching, the BERT serverless functions perform better for 1000 concurrent requests than it does for 200 requests with batching disabled.

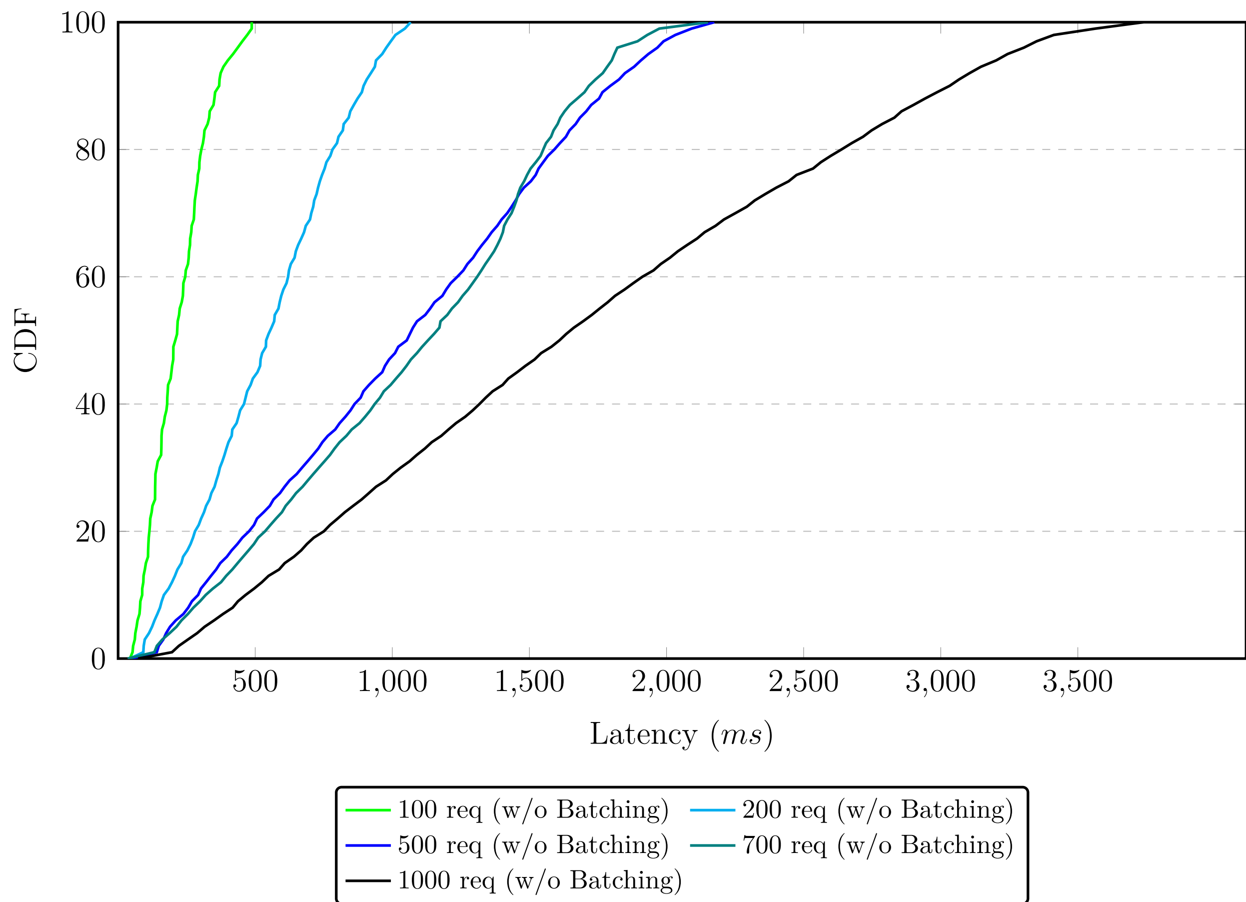

Figure 2.1: *CDF* for concurrent requests to BERT service on OpenFaaS with 8 replicas

Chapter 3

Design & Implementation

3.1 Overview

Our prototype is built upon OpenFaaS, which we have customized and deployed in a Kubernetes cluster. Figure 3.1 shows the overall architecture of our prototype. The Kubernetes cluster consists of the following components:

- **API Gateway**: This component exposes a common endpoint on the internet for users to send their inference queries. It is implemented using an `IngressRoute` in Kubernetes that points to the load balancer.
- **Load Balancer**: The Load Balancer component performs two important functions for ensuring the efficient distribution of incoming requests to function instances.
  - The Load Balancer uses an Exponentially Weighted Moving Average (EWMA) based algorithm for detecting the fastest function instances and dispatching incoming requests to those instances. The Load Balancer also detects newly launched function instances and starts dispatching traffic to new instances when they clear the health check on the inference endpoint.
  - The Load Balancer also acts as a Metrics server, exposing network and pod-level metrics to the autoscaler. The metrics exposed by the load balancer include

Request Arrival Rate, Request Processing Rate, Throughput, and Latency percentiles ($P_{50} - P_{99}$). These metrics are used by the autoscaler in conjunction with other inputs while deciding the number of function instances to launch.

- **Kube Metrics Server**: We have integrated the Kubernetes Metrics Server to gather metrics on pod-level resource usage, such as CPU usage, memory utilization, network bandwidth, etc.

- **Autoscaler**: The autoscaler is the core controller component of our proposed system. The autoscaler fetches network and pod-level metrics from the load balancer and the Kube metrics server. The autoscaler also includes an SLO monitoring module that tracks SLO violations and directs the autoscaler to scale up the number of replicas.

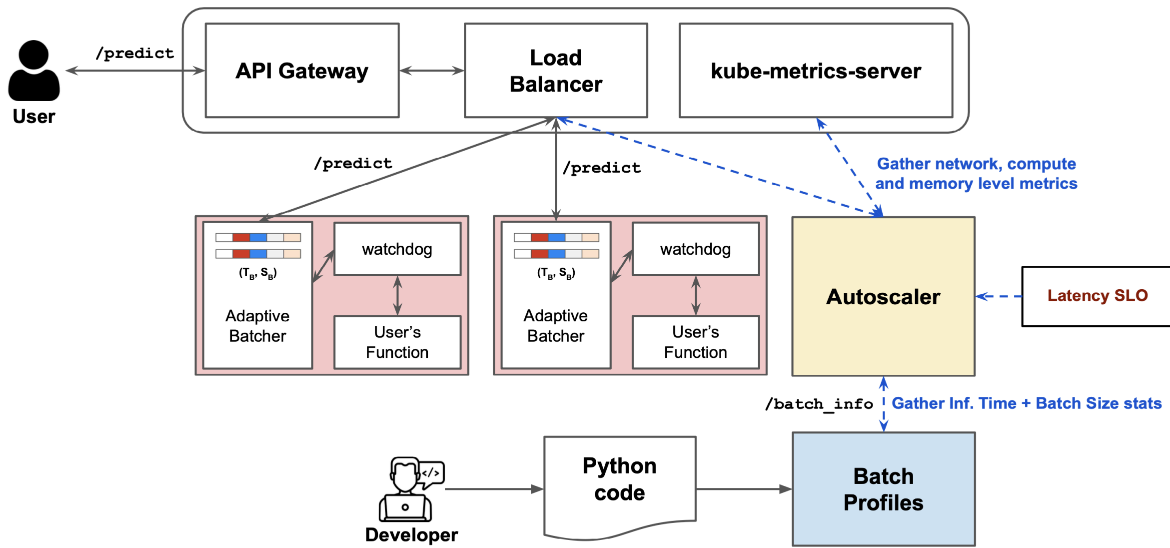

Figure 3.1: Prototype Architecture

The cluster has the core OpenFaaS components running that are required to dispatch new functions. Developers can conveniently write their inference services using Python and submit them to our OpenFaaS prototype. OpenFaaS incorporates a server component called

watchdog written in Go, responsible for receiving incoming requests and forwarding them to the respective function handler.

In our prototype, we have injected a sidecar container that contains the Adaptive Batching Module. The module intercepts incoming requests and, after applying appropriate filters, either forwards the requests directly to the watchdog component or in batches. Before deployment, the user's code undergoes offline profiling to capture inference time statistics across varying batch sizes. This profiling helps in tuning the hyperparameters of the Adaptive Batching module. Developers also provide information regarding their services' desired Service Level Objective (SLO) target. The autoscaling engine collects crucial metrics, including CPU and memory utilization, from the kube-metrics-server and network-level metrics, such as Request arrival rate, throughput, and latency, from the load balancer. Furthermore, it incorporates statistics derived from the offline profiling of the submitted functions, acquiring insights into inference time across different batch sizes. With this wealth of data, the autoscaling engine makes informed decisions regarding pod scaling, ensuring optimal resource allocation while maintaining the SLO target.

SLO Format. We follow the standard format to represent SLOs as (T_{slo}, P_{slo}). Here, T_{slo} represents the maximum acceptable response-time latency, whereas P_{slo} denotes the percentile of requests that have a response time, $T_{resp} \leq T_{slo}$.

3.2 Adaptive Batching

The Adaptive Batching Module is a lightweight sidecar container that runs an HTTP server written in Go. It is an integral component within our Serverless Framework prototype, enhancing the efficiency of Python-based inference functions. The module applies filters to the request path, determining whether requests should be directly forwarded to the OpenFaaS

watchdog or undergo batching before delivering. The module utilizes a lightweight, single-host reverse proxy server to facilitate request forwarding. This server efficiently manages the routing of requests based on the batching logic implemented by the Adaptive Batching Module. The module incorporates two hyperparameters: BatchSize and BatchTimeout. These parameters provide flexibility and control over the batching behavior.

BatchSize (S_B) dictates the maximum number of requests accumulating in a batch before being forwarded to the OpenFaaS watchdog. Once the batch size reaches the specified threshold, the module initiates the forwarding process, ensuring optimal utilization of resources.

BatchTimeout (T_B) establishes the maximum duration a batch can accumulate before being forwarded to the watchdog. Upon starting a new batch, the Adaptive Batching Module initiates a timer. If the BatchTimeout period elapses before reaching the BatchSize, the module forwards the current batch to the watchdog, ensuring timely processing and reducing latency.

The module takes immediate action in scenarios where the batch size equals the BatchSize parameter, indicating that the maximum limit has been reached. It stops the timer, forwards the requests to the OpenFaaS watchdog, and resets the batch, allowing the module to handle subsequent requests efficiently. The module uses lightweight Go channels to wait for the response of any given request, allowing them to yield the CPU to other Go threads.

Algorithm 1 provides the pseudocode run by each Go routine serving an HTTP request. The subroutine `resetAndForwardBatch()` resets the batch and then forwards the batch asynchronously, thus allowing available Go threads to server any pending requests. The asynchronous forward routine uses a single-host reverse proxy server to send the current batch to the OpenFaaS watchdog component. Upon receiving the batch response, the forwarding routine enqueues individual responses to each request's `responseChannel`.

Algorithm 1 Pseudocode for Adaptive Batching Module

```
queue.AcquireLock()
if queue ≠ nil then
    queue.append(request.Queries...)
    if len(queue) > BatchSize then
        stopTimer()
        resetAndForwardBatch()
    end if
else
    initQueue()
    queue.append(request.Queries...)
    startTimer(BatchTimeout, resetAndForwardBatch)
end if
queue.ReleaseLock()
response ← request.responseChannel
```

The hyperparameters for the adaptive batching module can be tuned in various ways, including an online learning-based algorithm. We approach the hyperparameter tuning problem in a heuristic way. We perform lightweight profiling on the deployed function to evaluate the mean inference time for a single request without batching. Let's call it $T_{\mu inf}$. For hyperparameters, Batchsize is denoted as S_B, and the BatchTimeout is denoted as T_B; we define another function $T_p(b)$ that provides the total processing time for a batch of size b. We then evaluate the optimal batch configuration using the following equation:

$$S_B^*, T_B^* = \arg\max_{S_B, T_B}(T_B + T_p(S_B) \leq min(T_{slo}, \frac{S_B}{T_{\mu inf}}))$$

The following equation can be structured as a dynamic programming problem by ranging batch size and batch timeout from 1 till the optimization reaches the global maxima. Overall, the Adaptive Batching Module is a critical component within our Serverless Framework prototype, enabling intelligent request management through dynamic batching. The autoscaling engine uses the hyperparameters through the *Batch Profile* to decide the optimal request arrival rate for each function instance. By leveraging configurable parameters and

employing an efficient reverse proxy server, the module optimizes resource utilization and improves the overall performance of serverless inference workflows.

3.3 Feedback control-based Autoscaling

The Adaptive Batching module can significantly improve the tail latency of user requests. Still, as the request arrival rate grows, a single function instance will not be enough to serve the requests while maintaining the tail latency below the SLO target. The function instance could be scaled vertically by assigning more vCPU resources to the pod. However, we have observed that the Python backend cannot perform parallel computing on a per-request level and would not benefit from increased CPU resources. However, scaling out the number of function instances, and delegating the load balancer to distribute incoming requests across all function instances evenly, would reduce the average request-arrival rate per function instance.

3.3.1 Real-time Request-Arrival-Rate Monitoring

The autoscaling engine monitors our inference service's request-arrival rate for each pod or functions replica. This information is exposed through an HTTP endpoint in the Load Balancer. The Load Balancer gathers this information using a sidecar proxy container injected into our function instances. It exposes metrics compatible with Prometheus, allowing the autoscaling engine to process the data easily in real time.

Let's assume the user sets a P80 latency target of T_{slo} for their function instance $(T_{slo}, 80)$. Based on T_{slo}, we can get the value of S_B^* and T_B^* fro, Section 3.2. Based on these values, we can evaluate the theoretical maximum processing rate of a function instance as $R_B^* =$

$\frac{S_B^*}{T_{resp}(S_B^*)}$, where $T_{resp}(b)$ denotes the total response time for a batch of size b. $T_{resp}(S_B^*)$ can be approximated as $T_B^* + T_p(S_B^*)$. R_B^* provides us with an upper bound on the request-arrival rate of a function instance that would ensure deterministic response time latencies. Let's denote the average request arrival rate for any of our function instances as R_μ. If $R_\mu > R_B^*$, we dispatch new replicas for our function based on the following formula:

$\frac{S_B^*}{T_B^* + T_p(S_B^*)}$

$$C_n = min(C_{max}, max(C_{min}, ceil(C_{n-1} \times \frac{R_\mu}{R^*})))$$

Here, C_n denotes the replica count for our function instance after the current iteration of decision-making by the autoscaler. C_{max} and C_{min} are the function instance's user-specified minimum and maximum concurrency limits. If $\frac{R_B^*}{2} < R_\mu < R_B^*$, we maintain the current replica count to prevent frequent oscillation of the function instance during sporadic request arrival rates. If $R_\mu < \frac{R_B^*}{2}$, we scale down the function instances.

The autoscaler can be further configured to enable downscale stabilization. Downscale stabilization is a strategy to prevent unnecessary oscillations or "flapping" of the pod replicas due to fluctuating metrics or transient traffic bursts. It introduces a delay or cooldown period before the autoscaler can scale down the number of pods after a scaling event. During this stabilization period, the autoscaler observes the metrics and evaluates whether further downscaling is necessary. Downscale stabilization aims to avoid rapidly scaling down pods in response to short-lived spikes in resource usage.

Predictive Autoscaling

Although the above approach can react to an increase in request arrival rate and increase the number of replicas, it is more effective to predict an increase in request arrival rate and

act accordingly. Various approaches can be used to predict the request arrival rate for our functions. We use a linear regression-based approach to profile our incoming traffic for a given amount of time and then start Prometheus's `predict_linear` feature to predict the incoming request arrival rate. This information is fed to the autoscaler and is used to replace the value of R_μ.

3.3.2 SLO Monitor

Besides the request-arrival rate for each pod, the autoscaler also tracks the various percentiles of request latency for each pod. The latency is averaged over a moving window of $30s$, using Prometheus's `histogram_quantile` function. This is done to ensure that SLO violations are caught during unpredictable traffic, and therefore, corrective action can be taken by the autoscaler by launching more functions. For example, if the target SLO for a given function is a P80 latency of T_{slo}, then the autoscaler will monitor the P80 latency for our Inference service and increase the function instance count if $T_{P80} \geq T_{slo}$. This is a reactive approach towards autoscaling as we increase the function instances after the SLO is violated. However, this policy can effectively minimize SLO violations when coupled with real-time monitoring of the request-arrival rate as discussed in 3.3.1. We implement the SLO Monitor as a Go routine that periodically polls the Load Balancer to get the appropriate latency percentile for the Inference requests. Based on the available metrics and the configured SLO target, the SLO monitor signals the autoscaler to scale up the number of function instances.

Chapter 4

Evaluation

4.1 Experimental Setup

We have built our prototype on a Kubernetes v1.25 cluster with five nodes. Three nodes are running the Kubernetes control plane components in High Availability mode, allowing these components to be replicated. All five nodes also act as worker nodes for running the Function workloads. The nodes are running Ubuntu 22.04.2 on Intel(R) Xeon(R) 6240R @2.40GHz CPU with 95 CPU cores spread across two NUMA nodes. The two NUMA nodes combined can contribute 187 GiB of memory to the pods.

We use Locust [49], an open-source distributed load testing framework for load testing. Locust uses an event-based model instead of a thread-based model, thus reducing interference on the background running processes of Kubernetes, e.g., the inference function instances. We use a Python Inference service function that performs sentiment analysis on textual input using the *bert-base-multilingual-uncased-sentiment* model. We use the default handler code as we would get from PyTorch's torchserve examples. We configure each function instance to be deployed with 3 GiB of memory, and 4 vCPUs, to handle the inference computation efficiently. The batching module is configured with a *BatchSize* of 128 and a *BatchTimeout* of 100ms.

4.2 Prototype vs. TorchServe

We compare our prototype with TorchServe [50], a production-ready inference server running PyTorch models. We configure the TorchServe heap to utilize 32 GiB of memory and allot 16 parallel workers for our models. The batching configuration for our prototype as well as Torchserve, is kept the same, i.e., `BatchSize` = 128, `BatchTimeout` = 100 ms. Our function prototype is assigned 4 vCPUs and 2 GiB of memory. We run a load test on the two services using Locust, where we vary the number of users from 10 to 50.

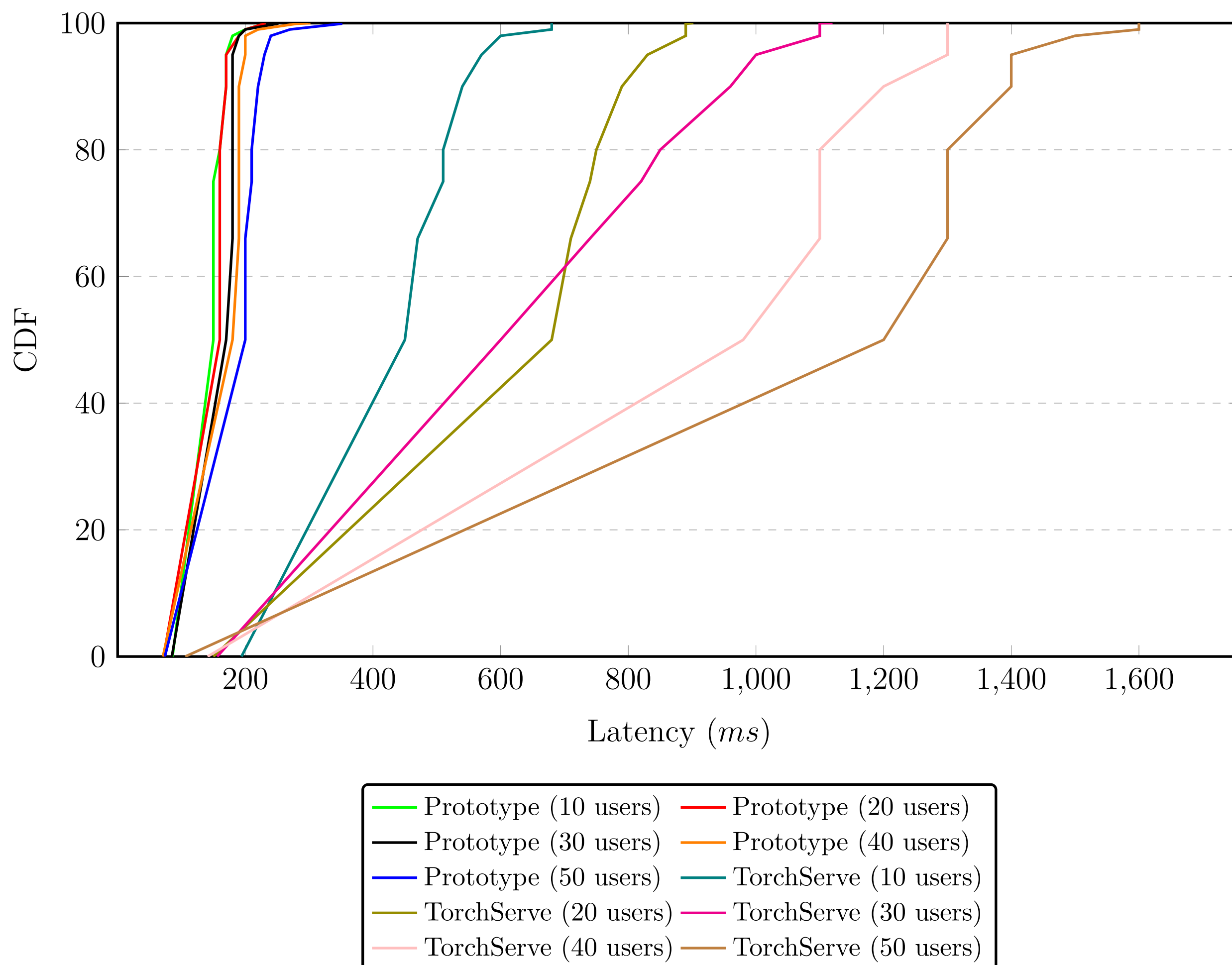

Figure 4.1: CDF for Load Testing on BERT Service running on our Prototype vs TorchServe

Figure 4.1 shows the CDF for the request latency to both services. Figure 4.2 shows the P90

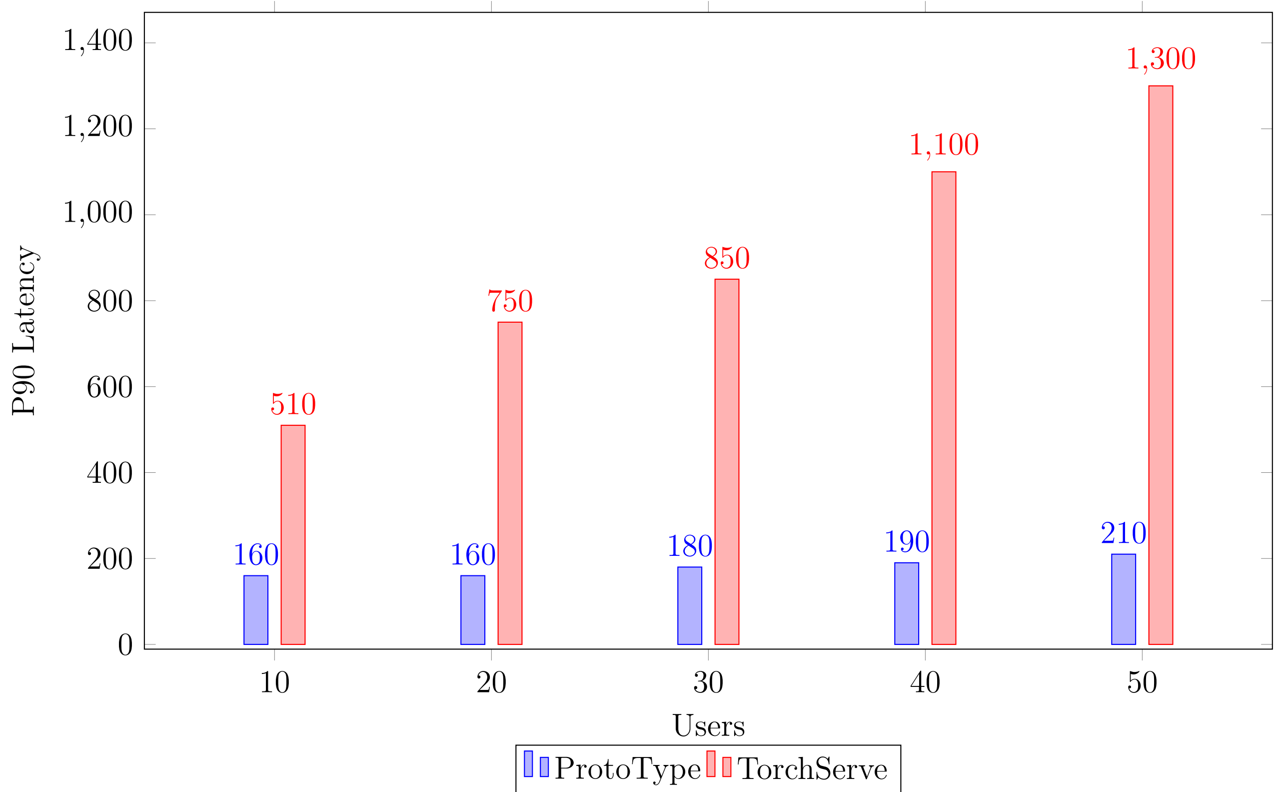

Figure 4.2: P80 Latency of Prototype vs TorchServe

latency for TorchServe and our prototype. From these experiments, we conclude that our prototype can maintain request latency during a spike in the number of users compared to the TorchServe instance. We can also demonstrate that our prototype performs better while consuming a fraction of the memory allocated to the TorchServe instance.

4.3 Replica Count vs. Latency

We conduct a series of load-testing experiments on the Inference Function as we vary the function replica count from $1-4$. We conduct these tests with Locust virtual users firing HTTP requests to the API Gateway in an infinite loop, starting with 0 users, and adding 50 new users every second, till we reach 500 virtual users. Figure 4.3 represents the Cumulative

Distribution Function for the request latency of our function instance as we vary the replica count. We observe that as we increase the replica count, the request tail latency becomes approximately equal to the optimal processing rate of each function instance based on data from Table 2.2. We also observe that the improvement in request tail latency saturates after reaching a replica count of four. Moreover, with four replicas, we can achieve an RPS of 1438, which is approximately 3× the number of concurrent users. We conducted the same experiment with 1000 virtual users and varied the replica count from $2-5$. We observed that with four replicas, the request tail latency becomes approximately equal to the optimal processing rate. Once again, we can observe that with $4-5$ replicas, we can achieve an RPS of 2833, almost 3× number of concurrent users. Figure 4.4 represents the CDF for load testing with 1000 concurrent users. Table 4.1 represents the data for the above experiments, with the throughput and tail latency recorded.

Table 4.1: Throughput(RPS), and tail latency(P80-P100) for distributed load testing on BERT Service running on our prototype

Users	**Replicas**	**RPS**	**80%**	**90%**	**95%**	**98%**	**99%**	**99.99%**	**100%**
500	1	594	710	730	760	800	840	890	890
500	2	1078	390	410	450	470	490	610	610
500	3	1243	280	290	310	330	370	410	420
500	4	1438	280	280	290	300	330	380	380
1000	2	1295	700	720	740	770	800	1100	1100
1000	3	1702	480	500	520	560	590	720	720
1000	4	2723	300	320	340	360	400	560	580
1000	5	2833	300	300	320	370	410	500	510

4.4 Autoscaling vs. Latency

Next, we enable the autoscaler and configure it with $C_{min} = 1$ and $C_{max} = 5$. Based on Table 2.2, the autoscaler evaluates the optimal request arrival rate for each instance,

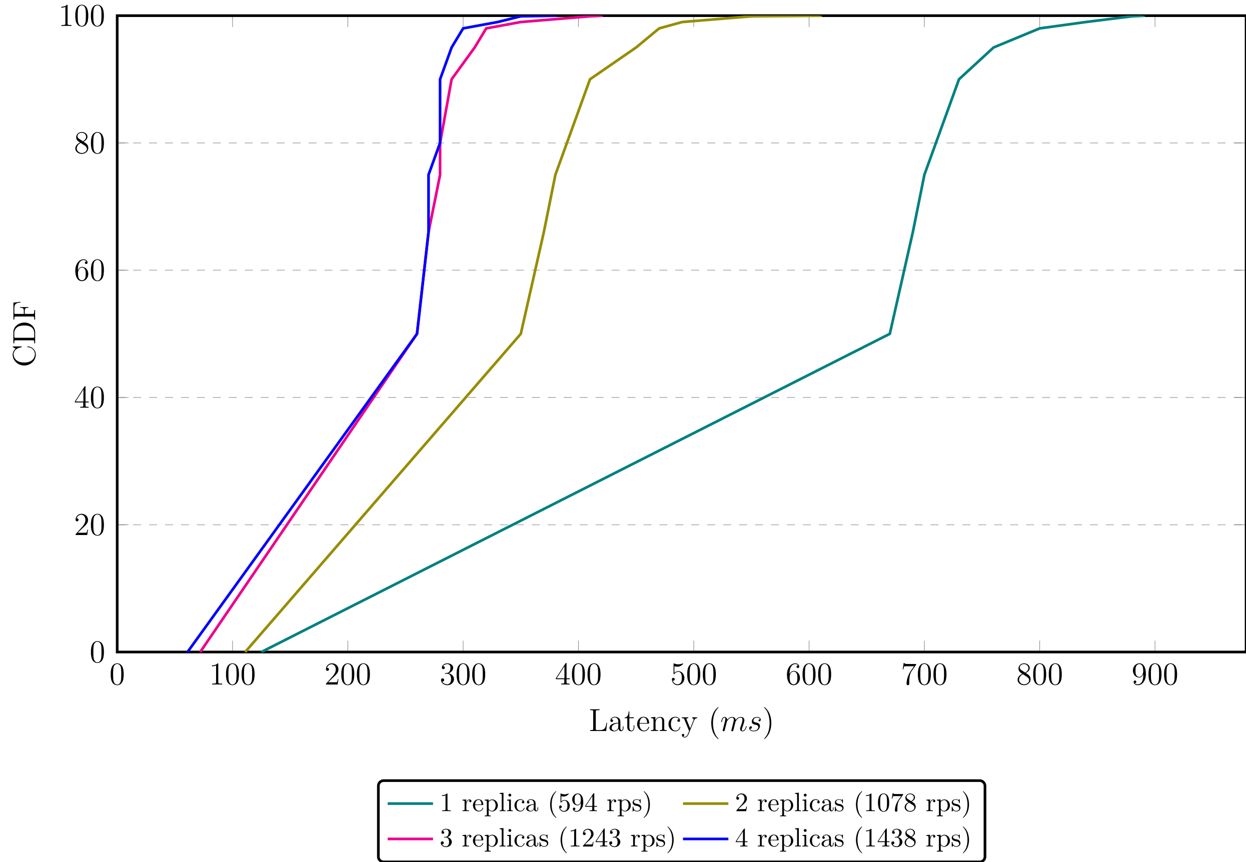

Figure 4.3: *CDF* for concurrent load testing on the BERT service with varying replica counts. All tests are done with 500 concurrent users, with randomized spawn rates to simulate sporadic request traffic. ($S_B = 128, T_B = 100ms$)

approximately 670 rps. Moreover, we provide an SLO target of $(310ms, 80)$ for the function instances. We also configure the batching module with $S_B = 128$ and $T_B = 100$ ms. For the distributed load test, we configure locust with three worker nodes, 500 virtual users, and a spawn rate of 50 users/second. Figure 4.5 shows the time-series graph generated by Locust and represents the response time and the number of active users with time. As the number of users increases, the autoscaler effectively spins up new replicas characterized by intermediate plateaus that follow a spike in the graph. Once the number of users becomes constant, thus denoting a constant request arrival rate, the autoscaler effectively scales up the function replica count to four. At the same time, the load balancer routes the traffic to

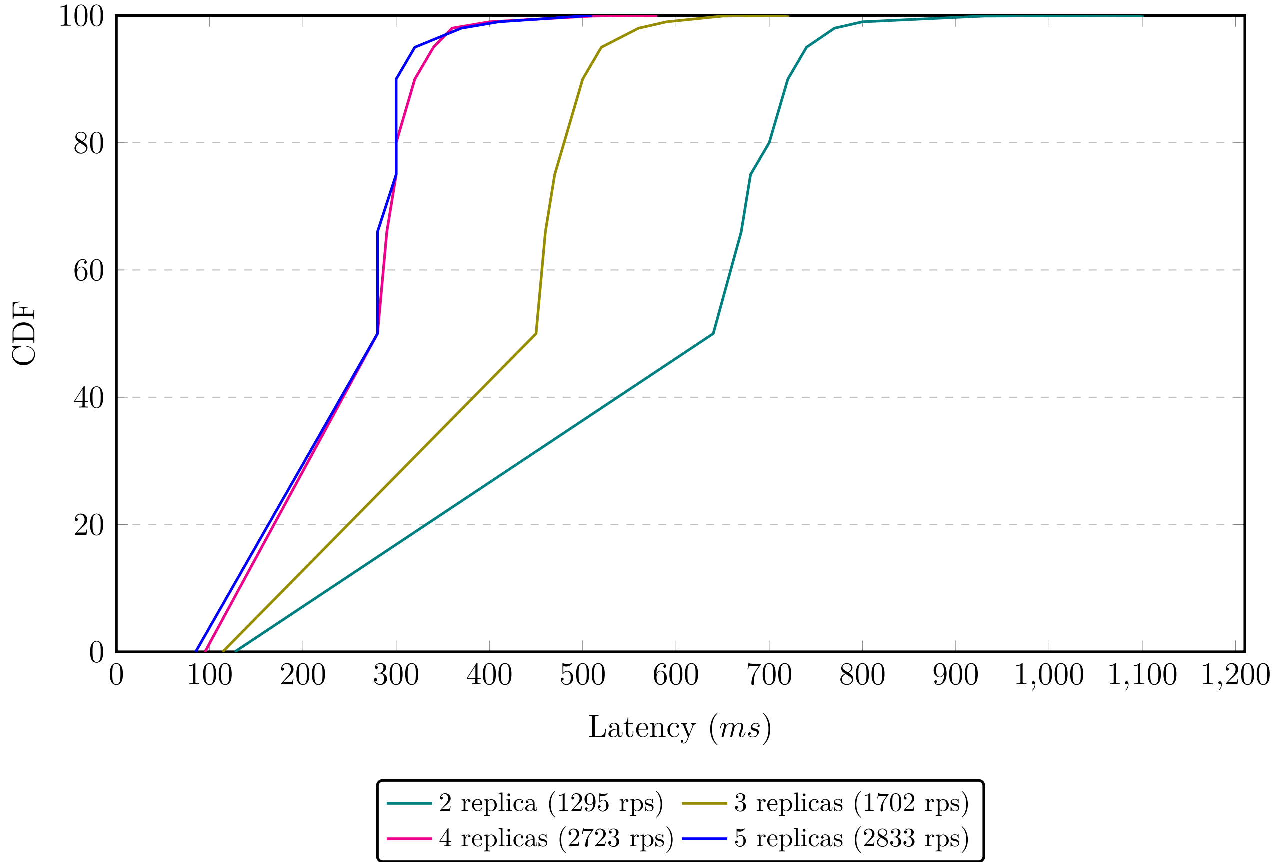

Figure 4.4: *CDF* for concurrent load testing on the BERT service with varying replica counts. All tests are done with 1000 concurrent users, with randomized spawn rates to simulate sporadic request traffic. ($S_B = 128, T_B = 100ms$)

the newly registered instances. As the function replicas are scaled up to four, the traffic is evenly distributed across the functions, and we achieve a P95 latency less than the configured SLO target of 310 ms. Next, we repeat this experiment for 1000 concurrent virtual users with the same configuration for the autoscaler, batcher, and SLO target. Figure 4.6 shows the time-series graph generated by Locust and represents the Throughput(RPS), response time, and the number of active users for the BERT Inference function. Table 4.2 shows the request arrival rate of each function instance while running the distributed load test for 1000 users. We can see that the load balancer effectively distributes the requests at the optimal processing rate of the function instance, determined from the *Batch Profile*, i.e., 670 ms.

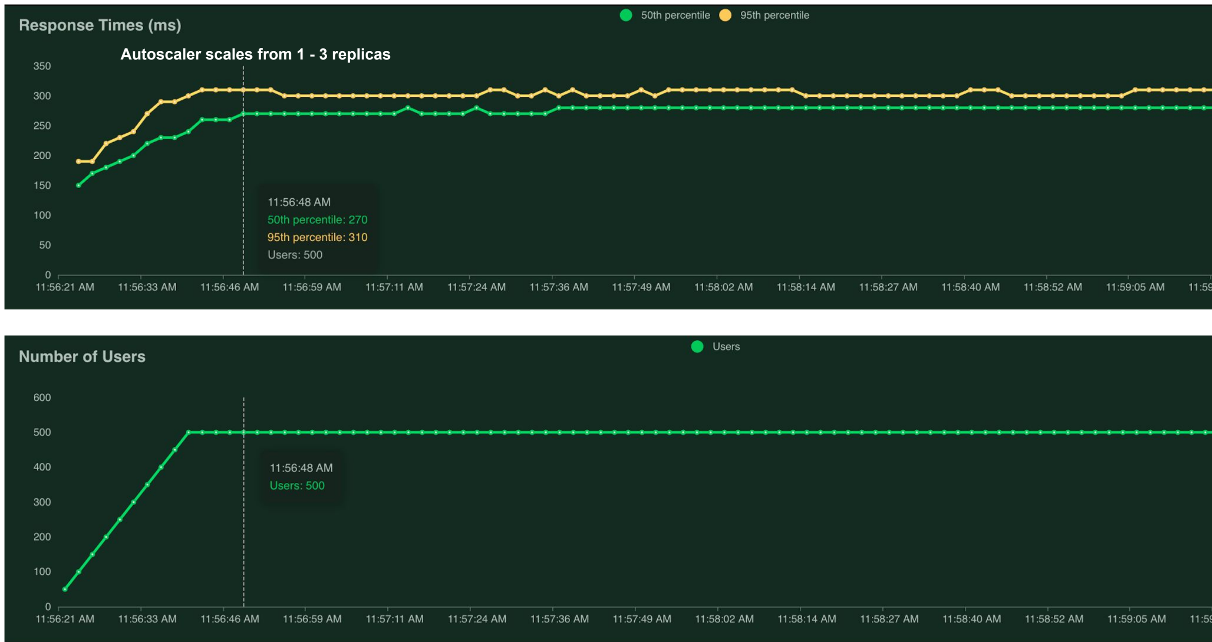

Figure 4.5: Distributed Load Testing for the BERT Service with autoscaling engine enabled. The demarcation line denotes the timestamp when the autoscaler successfully scales up the number of pods from $1-3$.

Table 4.2: Request Arrival Rate distribution and Request Latency for BERT service with 5 function instances

Instance	**Request Arrival Rate**	**P50**	**P95**	**P99**
Instance 1	688	250	295	299
Instance 2	571	244	284	297
Instance 3	620	257	337	363
Instance 4	729	263	315	395
Instance 5	669	302	308	401

Figure 4.6: Distributed Load Testing for the BERT Service with autoscaling engine enabled. The autoscaler scales the number of replicas from $1-5$ by 12:30:24 PM. The highest throughput reaches close to 3000 rps.

4.5 Stress Testing

We perform a stress test on our proposed system to see how much throughput can be generated with a maximum of 8 replicas, with autoscaling enabled. We configure Locust to run load testing with the user count in the range $[1500, 2000, 3000]$. The batching module is configured with a batch size of 128 and a batch timeout of 100 ms. We configure the autoscaler with $C_{min} = 1$ and $C_{max} = 8$, and we relax the SLO target to $(700ms, 80)$. Figure 4.7 shows the CDF for the request latency to the BERT service for varying user counts. We observe that the maximum P80 latency is capped at 600 ms, which is within the SLO target and can serve up to 3000 concurrent users at a maximum throughput of 4738 RPS.

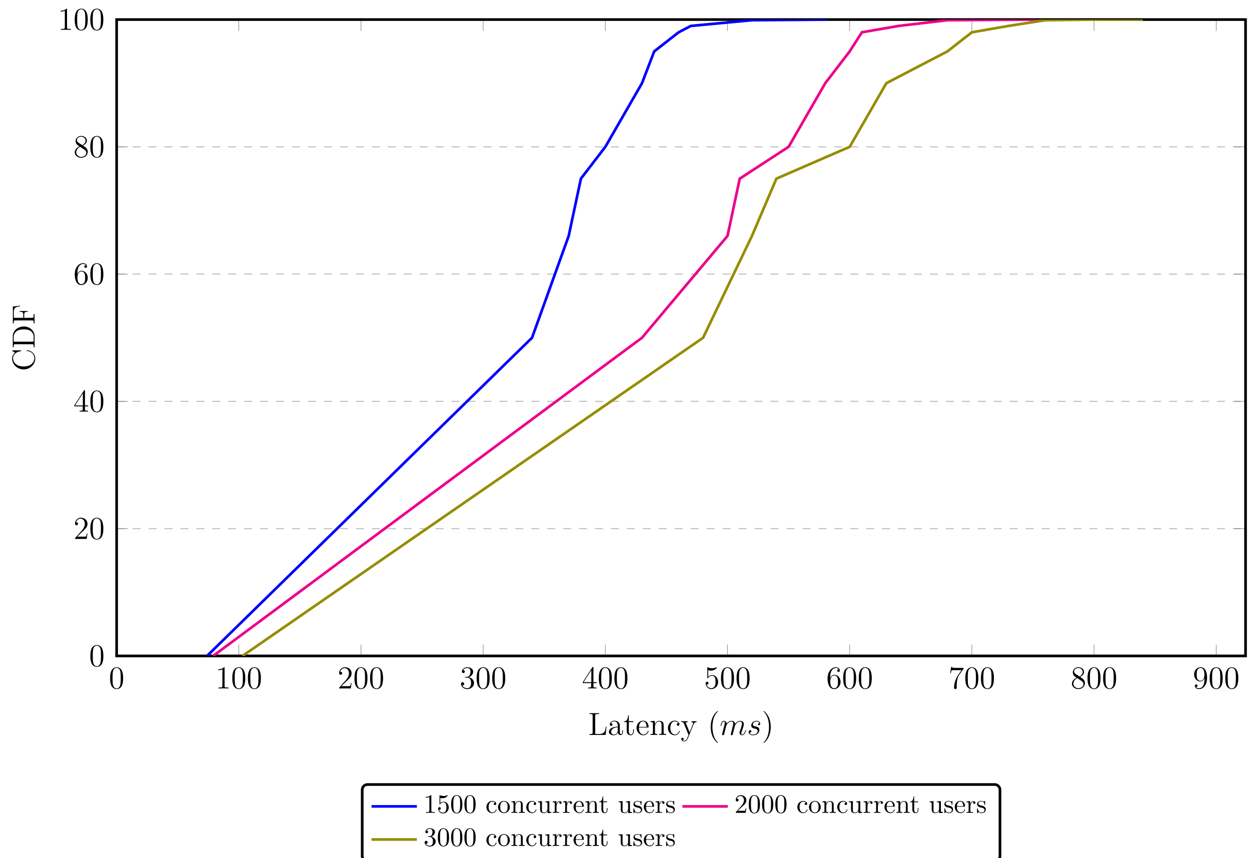

Figure 4.7: *CDF* for concurrent requests to BERT service on ServeML(1 - 8 replica count)

4.6 CPU and Memory Utilization

While running the experiments described in the above sections, we tracked the CPU and memory utilization of the pods. The model size for the *bert-base-multilingual-uncased-sentiment* is 673 MB on disk. We observe that when a function instance is booted up, or a pod is initialized, there is a sudden increase in memory consumption as the resident set size (RSS) increases to approximately 1216 MB. However, the memory utilization then drops to 800 MB and stays the same throughout the duration of the function instance. The sudden spike in memory consumption is caused due to unpacking the model data from the disk to the page cache, as the function instance keeps the model in memory to perform efficient inference. PyTorch unpacks the tensors that are stored in compressed form on disk. The constant memory utilization during the load-testing period shows the Python-based function only keeps the model in memory, and an increase in the number of requests would have no impact on the memory utilization of the function.

We assign four vCPUs to each function instance while pinning the batcher container to two cores and the Python-based backend to the remaining two cores to prevent interference between the two processes. We observe that the CPU utilization stays within 50% during the load testing period, implying that increasing the request arrival rate does not impact the CPU utilization of individual instances.

Chapter 5

Related Work

5.1 Related Work

In the field of distributed inference systems, several noteworthy contributions have been made by researchers and developers. One such system is INFaaS [21], which offers a model-less distributed inference system where users specify their application's performance and accuracy requirements. The system efficiently explores the trade-off space of model variants, model optimizations, and hardware selections to meet the application's specific needs. It incorporates horizontal auto-scaling characteristics similar to FaaS systems, providing scalability and adaptability.

Building upon the concepts of INFaaS, Arpan et al. [40] present their own managed runtime called Clockwork. Their research demonstrates that deep neural network (DNN) inference jobs exhibit deterministic performance on GPUs due to the absence of conditional branches. Clockwork is designed to serve thousands of models while meeting stringent latency targets of 100ms for 99.9999% of requests. The system achieves this by leveraging distributed model serving and optimizing resource allocation strategies.

AMPS-Inf, proposed by [20], introduces an automated model partitioning framework to find cost-efficient serverless inference deployment strategies. The framework considers factors such as memory allocation, layer size, intermediate state transition cost among Lambdas,

execution time, and pricing models. By exploiting the serverless computing characteristic that more memory means more computing cycles, AMPS-Inf focuses on optimizing inference jobs within individual instances. However, there is a scope for further investigation into the potential benefits of model parallelism and pipelining to improve overall inference performance.

INFLess, introduced by [22], addresses bottlenecks observed in existing serverless platforms for machine learning workloads. These bottlenecks include high latency for large inference jobs, high latency for small models in a batched setting, resource over-provisioning, and low utilization. INFLess tackles these challenges through resource abstraction, autotuning resource provisioning, and an LSTH strategy to handle cold-start issues. The platform proactively predicts performance latency by analyzing the requested models' directed acyclic graph (DAG) and adjusts the batch size dynamically based on resource changes.

Tetris, a research effort by [24], investigates the characteristics of inference jobs and proposes a tensor-sharing mechanism to reduce memory footprint. The study reveals that inference jobs induce significant startup memory footprint due to loading large model parameters, duplicated across runtimes and different instances. Tetris addresses this issue by storing tensors in memory and not reloading existing tensors. However, there is room for improvement in managing the massive memory footprint caused by model parameters and designing near-data scheduling strategies.

Upon further exploration of INFLess, it is discovered that profiling individual operators can be an effective approach to predicting performance. By associating each operator with five characteristics, such as execution time and memory usage, a static set of operators' versions can be profiled and combined to predict the overall performance of an inference function. The reported prediction accuracy is already impressive, with an error rate of less than 10%. Additionally, the auto-scaling mechanism in INFLess considers execution time but could

benefit from considering other factors, such as memory usage.

Zhang et al. [17] tackle the Service Level Objective (SLO) problem in machine learning model serving with their system called MArk (Model Ark). Built in Amazon Web Services (AWS), MArk is a general-purpose inference serving system designed to meet response-time SLOs and minimize serving costs. The system incorporates three key design choices tailored for inference workloads. It dynamically batches requests and utilizes hardware accelerators to optimize the performance-cost ratio. Predictive autoscaling is employed to handle dynamic workloads and minimize provisioning latency efficiently. Finally, serverless instances are leveraged for managing occasional load spikes that are challenging to predict. Comparative evaluations with the industrial ML serving platform SageMaker demonstrate superior latency performance and reduced serving costs by up to 7.8 times [17].

These notable contributions and advancements in distributed inference systems provide valuable insights and directions for improving the design of our system. By learning from their findings and incorporating novel approaches, we aim to enhance our own system's performance, scalability, and cost-effectiveness.

Chapter 6

Conclusion

6.1 Limitations and Future Work

Our proposed prototype can provide significant improvements over Python-based Serverless Inference functions. However, there are several areas where our approach is limited. Our experiments use a model size of approximately 700 MB, much smaller than modern AI-based models with sizes ranging in hundreds of gigabytes. Larger models would lead to huge cold start latencies, such that it would take a long time to redirect incoming traffic to the new function instances, thus increasing SLO violations. One area of future work to alleviate this problem could be implementing model partitioning [40], such that individual function instances are only responsible for a small partition of the model.

The autoscaling policies used by our prototype react to an increase in request arrival rate or SLO violations to decide when to scale up function instances. In the future, we can explore predictive strategies that try to predict resource demands and incoming traffic rates to make proactive autoscaling decisions. Several ML techniques can be used to perform predictive scaling, such as linear regression, Reinforcement Learning, etc. Reinforcement Learning is an effective tool to dynamically adjust the resources allocated to an application or system based on predicted future demands. Reinforcement learning models are trained using historical data, real-time metrics, and contextual information to forecast demand patterns and make intelligent decisions about scaling up or down. These models learn from past experiences,

rewards, and penalties to continuously refine their predictions and actions, adapting to changing workload conditions. By harnessing the predictive capabilities of reinforcement learning, our autoscaling algorithms can proactively respond to anticipated spikes or lulls in demand, thus helping optimize resource provisioning and meet service-level objectives while minimizing unnecessary overhead.

Another area of future work involves exploring other thread-based languages like C++ for writing Inference backends. All major ML/DL libraries are available as shared libraries in C++. This would allow our functions to perform parallel computation, thus improving inference time and the overall request latency. This can lead to more consistent and predictable behavior from the backend compared to Python.

We have explored Horizontal Autoscaling in our paper; however, when using function instances written in C++, we could enhance the throughput by allocating additional cores and memory to the function pod. Building upon this, we can develop intelligent scaling policies that dynamically determine when to scale up or out based on resource demands and workload characteristics. This combined approach of horizontal and vertical scaling opens up new avenues for maximizing the efficiency and scalability of serverless computing environments, offering improved throughput and enhanced performance for machine learning inference and other resource-intensive workloads.

6.2 Summary

We build a prototype for demonstrating Serverless Inference using adaptive request batching and autoscaling decisions based on request arrival rate, latency, and SLO violations. Our prototype demonstrates superior performance and throughput compared to FaaS-based Inference functions and ensures minimum SLO violations using predictive autoscaling policies. We demonstrate with our approach that CPU and memory utilization is not impacted by an increase in request arrival rate due to the normalizing nature of the adaptive request batching unit. Our prototype allows developers to write familiar ML/DL application logic in Python and uses the batching unit and autoscaler to forward requests efficiently to the function instances while maintaining the SLO target for the service. Our proposed prototype would allow developers to adopt Serverless functions for building inference logic for small to medium-sized ML models.

Bibliography

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," pp. 770–778, 06 2016.

[2] Z. Dai, H. Liu, Q. V. Le, and M. Tan, "Coatnet: Marrying convolution and attention for all data sizes," 2021.

[3] H. Touvron, A. Vedaldi, M. Douze, and H. Jegou, *Fixing the Train-Test Resolution Discrepancy*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[4] OpenAI, "Introducing chatgpt." https://openai.com/blog/chatgpt, 2022.

[5] OpenAI, "Gpt-4 technical report," 2023.

[6] R. Thoppilan, D. D. Freitas, J. Hall, N. Shazeer, A. Kulshreshtha, H.-T. Cheng, A. Jin, T. Bos, L. Baker, Y. Du, Y. Li, H. Lee, H. S. Zheng, A. Ghafouri, M. Menegali, Y. Huang, M. Krikun, D. Lepikhin, J. Qin, D. Chen, Y. Xu, Z. Chen, A. Roberts, M. Bosma, V. Zhao, Y. Zhou, C.-C. Chang, I. Krivokon, W. Rusch, M. Pickett, P. Srinivasan, L. Man, K. Meier-Hellstern, M. R. Morris, T. Doshi, R. D. Santos, T. Duke, J. Soraker, B. Zevenbergen, V. Prabhakaran, M. Diaz, B. Hutchinson, K. Olson, A. Molina, E. Hoffman-John, J. Lee, L. Aroyo, R. Rajakumar, A. Butryna, M. Lamm, V. Kuzmina, J. Fenton, A. Cohen, R. Bernstein, R. Kurzweil, B. Aguera-Arcas, C. Cui, M. Croak, E. Chi, and Q. Le, "Lamda: Language models for dialog applications," 2022.

[7] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," 2023.

[8] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T.-S. Chua, "Neural collaborative filtering," 2017.

[9] C. A. Gomez-Uribe and N. Hunt, "The netflix recommender system: Algorithms, business value, and innovation," *ACM Trans. Manage. Inf. Syst.*, vol. 6, dec 2016.

[10] M. S. Vallabhajosyula and R. Ramnath, "Towards practical, generalizable machine-learning training pipelines to build regression models for predicting application resource needs on hpc systems," in *Practice and Experience in Advanced Research Computing*, PEARC '22, (New York, NY, USA), Association for Computing Machinery, 2022.

[11] A. Harlap, A. Tumanov, A. Chung, G. R. Ganger, and P. B. Gibbons, "Proteus: Agile ml elasticity through tiered reliability in dynamic resource markets," in *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, (New York, NY, USA), p. 589–604, Association for Computing Machinery, 2017.

[12] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: An efficient dynamic resource scheduler for deep learning clusters," in *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, (New York, NY, USA), Association for Computing Machinery, 2018.

[13] H. Zhang, L. Stafman, A. Or, and M. J. Freedman, "Slaq: Quality-driven scheduling for distributed machine learning," in *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, (New York, NY, USA), p. 390–404, Association for Computing Machinery, 2017.

[14] X. Miao, X. Nie, Y. Shao, Z. Yang, J. Jiang, L. Ma, and B. Cui, "Heterogeneity-aware distributed machine learning training via partial reduce," in *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, (New York, NY, USA), p. 2262–2270, Association for Computing Machinery, 2021.

[15] C. Makaya, A. Iyer, J. Salfity, M. Athreya, and M. Lewis, “Cost-effective machine learning inference offload for edge computing,” 12 2020.

[16] Y. Hu, R. Ghosh, and R. Govindan, “Scrooge: A cost-effective deep learning inference system,” in *Proceedings of the ACM Symposium on Cloud Computing*, SoCC ’21, (New York, NY, USA), p. 624–638, Association for Computing Machinery, 2021.

[17] C. Zhang, M. Yu, W. Wang, and F. Yan, “MArk: Exploiting cloud services for Cost-Effective, SLO-Aware machine learning inference serving,” in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, (Renton, WA), pp. 1049–1062, USENIX Association, July 2019.

[18] Z. Zheng, X. Yang, P. Zhao, G. Long, K. Zhu, F. Zhu, W. Zhao, X. Liu, J. Yang, J. Zhai, S. L. Song, and W. Lin, “Astitch: Enabling a new multi-dimensional optimization space for memory-intensive ml training and inference on modern simt architectures,” in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’22, (New York, NY, USA), p. 359–373, Association for Computing Machinery, 2022.

[19] M. Hartmann, L. Weber, J. Wirth, L. Sommer, and A. Koch, “Optimizing a hardware network stack to realize an in-network ml inference application,” in *2021 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, pp. 21–32, 2021.

[20] J. Jarachanthan, L. Chen, F. Xu, and B. Li, “Amps-inf: Automatic model partitioning for serverless inference with cost efficiency,” in *50th International Conference on Parallel Processing*, ICPP 2021, (New York, NY, USA), Association for Computing Machinery, 2021.

[21] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis, "INFaaS: Automated model-less inference serving," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 397–411, USENIX Association, July 2021.

[22] Y. Yang, L. Zhao, Y. Li, H. Zhang, J. Li, M. Zhao, X. Chen, and K. Li, "Infless: A native serverless system for low-latency, high-throughput inference," ASPLOS '22, (New York, NY, USA), p. 768–781, Association for Computing Machinery, 2022.

[23] M. Wawrzoniak, I. Müller, R. Bruno, A. Klimovic, and G. Alonso, "Short-lived datacenter," 2022.

[24] J. Li, L. Zhao, Y. Yang, K. Zhan, and K. Li, "Tetris: Memory-efficient serverless inference through tensor sharing," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, (Carlsbad, CA), USENIX Association, July 2022.

[25] Amazon, "Cloud computing services - amazon web services (aws)." https://aws.amazon.com/.

[26] AWS, "Serverless inference: Amazon sagemaker." https://docs.aws.amazon.com/sagemaker/latest/dg/serverless-endpoints.html, 2021.

[27] J. Park, B. Choi, C. Lee, and D. Han, "Graf: A graph neural network based proactive resource allocation framework for slo-oriented microservices," in *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '21, (New York, NY, USA), p. 154–167, Association for Computing Machinery, 2021.

[28] S. Xue, C. Qu, X. Shi, C. Liao, S. Zhu, X. Tan, L. Ma, S. Wang, S. Wang, Y. Hu, L. Lei, Y. Zheng, J. Li, and J. Zhang, "A meta reinforcement learning approach for predictive autoscaling in the cloud," in *Proceedings of the 28th ACM SIGKDD Conference on*

Knowledge Discovery and Data Mining, KDD '22, (New York, NY, USA), p. 4290–4299, Association for Computing Machinery, 2022.

[29] B. Chang, A. Akella, L. D'Antoni, and K. Subramanian, "Learned load balancing," in *Proceedings of the 24th International Conference on Distributed Computing and Networking*, ICDCN '23, (New York, NY, USA), p. 177–187, Association for Computing Machinery, 2023.

[30] "Aws lambda : Run code without thinking about servers or clusters." https://aws.amazon.com/lambda, 2016.

[31] "Azure functions : Execute event-driven serverless code functions with an end-to-end development experience." https://azure.microsoft.com/products/functions, 2016.

[32] "Google cloud functions." https://cloud.google.com/functions, 2018.

[33] OpenFaaS, "Openfaas : Serverless functions, made simple." https://www.openfaas.com, 2019.

[34] S. Gross, "Pep 703 – making the global interpreter lock optional in cpython." https://peps.python.org/pep-0703/, 2023.

[35] L. Ao, G. Porter, and G. M. Voelker, "Faasnap: Faas made fast using snapshot-based vms," in *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, (New York, NY, USA), p. 730–746, Association for Computing Machinery, 2022.

[36] A. Wang, S. Chang, H. Tian, H. Wang, H. Yang, H. Li, R. Du, and Y. Cheng, "FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 443–457, USENIX Association, July 2021.

[37] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "SAND: Towards High-Performance serverless computing," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, (Boston, MA), pp. 923–935, USENIX Association, July 2018.

[38] V. Shankar, K. Krauth, Q. Pu, E. Jonas, S. Venkataraman, I. Stoica, B. Recht, and J. Ragan-Kelley, "numpywren: serverless linear algebra," 2018.

[39] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A serverless video processing framework," in *Proceedings of the ACM Symposium on Cloud Computing*, (New York, NY, USA), p. 263–274, Association for Computing Machinery, 2018.

[40] A. Gujarati, R. Karimi, S. Alzayat, W. Hao, A. Kaufmann, Y. Vigfusson, and J. Mace, "Serving dnns like clockwork: Performance predictability from the bottom up," in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, (USA), USENIX Association, 2020.

[41] A. Mahgoub, K. Shankar, S. Mitra, A. Klimovic, S. Chaterji, and S. Bagchi, "SONIC: Application-aware data passing for chained serverless applications," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pp. 285–301, USENIX Association, July 2021.

[42] K. Hu, "Chatgpt sets record for fastest-growing user base." https://www.reuters.com/technology/chatgpt-sets-record-fastest-growing-user-base-analyst-note-2023-02-01/, 2023.

[43] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019.

[44] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, jan 2020.

[45] D. Schall, A. Margaritov, D. Ustiugov, A. Sandberg, and B. Grot, "Lukewarm serverless functions: Characterization and optimization," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, (New York, NY, USA), p. 757–770, Association for Computing Machinery, 2022.

[46] S. Lee, D. Yoon, S. Yeo, and S. Oh, "Mitigating cold start problem in serverless computing with function fusion," *Sensors*, vol. 21, no. 24, 2021.

[47] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, "Agile cold starts for scalable serverless," in *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, (Renton, WA), USENIX Association, July 2019.

[48] A. S. Foundation, "ab - apache http server benchmarking tool." https://httpd.apache.org/docs/2.4/programs/ab.html.

[49] Locust, "An open source load testing tool." https://locust.io.

[50] PyTorch, "Serve, optimize, and scale pytorch models in production." https://github.com/pytorch/serve.