Rethinking Serverless for Machine Learning Inference

Anish Reddy Ellore

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
Daniel J. Williams
Liting Hu

June 26, 2023
Blacksburg, Virginia

Keywords: Serverles, FaaS, Machine Learning Inference, Model Serving, Container

Rethinking Serverless for Machine Learning Inference

Anish Reddy Ellore

ABSTRACT

In the era of artificial intelligence and machine learning, AI/ML inference tasks have become exceedingly popular. However, executing these workloads on dedicated hardware may not be feasible for many users due to high maintenance costs, varying load patterns, and time to production. Furthermore, ML inference workloads are stateless, and most of them are not extremely latency sensitive. For example, tasks such as fake review removal, abusive language detection, tweet classification, image tagging, and free-tier-chat-bots do not require real-time inference. All these characteristics make serverless platforms a good fit for deployment, and in this work, we identify the bottlenecks involved in hosting these inference jobs on serverless and optimize serverless for better performance and resource utilization. Specifically, we identify model loading and model memory duplication as major bottlenecks in Serverless Inference, and to address these problems, we propose a new approach that rethinks the way we serve FaaS requests. To support this design, we employ a hybrid scaling approach to implement the autoscale feature of serverless.

Rethinking Serverless for Machine Learning Inference

Anish Reddy Ellore

GENERAL AUDIENCE ABSTRACT

Most modern software applications leverage the power of machine learning to incorporate intelligent features. For instance, platforms like Yelp employ machine learning algorithms to detect fake reviews, while intelligent chatbots such as ChatGPT provide interactive conversations. Even Netflix relies on machine learning to recommend personalized content to its users. The process of creating these machine learning services involves several stages, including data collection, model training using the collected data, and serving the trained model to deploy the service. This final stage, known as inference, is crucial for delivering real-time predictions or responses to user queries. In our research, we focus on selecting serverless computing as the preferred infrastructure for deploying these popular inference workloads. Serverless, also referred to as Function as a Service (FaaS), is an execution paradigm in cloud computing that allows users to efficiently run their code by providing scalability, elasticity and fine-grained billing. In this work we identified, model loading and model memory duplication as major bottlenecks in Serverless Inference. To solve these problems we propose a new approach which rethinks the way we serve FaaS requests. To support this design we use a hybrid scaling approach to implement the autoscale feature of serverless.

Dedicated to family and friends.

Acknowledgments

I would like to express my sincere gratitude to my committee members, Dr. Butt, Dr. Dan, and Dr. Liting, for their support and guidance throughout my MS journey.

I am extremely thankful to my advisor, Dr. Butt, and my lab mates for their unwavering support during my MS journey and job search. Their guidance and collaboration have been invaluable to me.

A special thanks goes to Dr. Dan for his collaboration and valuable feedback, which greatly contributed to the success of my work. I would also like to thank my lab mates, Abhijit and Tanuj, for their assistance and collaboration during crucial experiments.

I would like to extend my heartfelt thanks to my family, roommates and friends for their love, encouragement, and unwavering support throughout this incredible journey. Your presence has meant the world to me.

Contents

List of Figures

List of Tables

List of Abbreviations

FaaS Function as a service

ML Machine Learning

NLP Natural Language Processing

Chapter 1

Introduction

1.1 Motivation

Deep learning has transformed how we approach complex tasks such as image recognition [1, 2, 3, 4], natural language processing [5, 6, 7, 8, 9, 10], recommendation systems [11, 12], and many others. These learning techniques, empowered by vast amounts of data and advances in computational power, have achieved remarkable accuracy and performance across a wide range of applications. However, once these models are trained, they must be deployed to make predictions on new data, a process commonly referred to as *inference*. Inference is a critical step in leveraging the power of machine learning and deep learning models, as it allows us to apply the acquired knowledge to real-world scenarios.

The demand for performing inference cost-effectively and efficiently has grown significantly as machine learning and deep learning models find broader adoption. ChatGPT [13] currently has 1.16 billion users and has set the record for the fastest-growing user base [14]. As a result, organizations and individuals seek practical solutions that deliver accurate predictions, optimize resource utilization, and minimize operational costs. Traditionally, inference workloads are run on high-performance servers with dedicated hardware infrastructures like multiple CPU/GPU/TPU units. This approach, although effective, can be costly, requiring substantial upfront investments and operational costs. Moreover, there are significant challenges in scaling up such systems during peak load and minimizing resource provision-

ing during low-activity hours. To overcome such problems we look at a cloud computing execution model called Serverless or FaaS.

Serverless computing is an emerging paradigm in cloud computing that enables developers to focus solely on writing and deploying code without the need to manage or provision servers. In this model, the cloud provider manages all the infrastructure and resource allocation, dynamically scaling the resources based on the application's demands, thus eliminating the need for traditional server provisioning. It allows developers to focus on their core application logic, promotes faster development cycles, allows fine-grained billing for resource usage, and provides the flexibility to scale applications seamlessly based on real-time demands. Serverless computing is gaining popularity in academia [15, 16, 17, 18, 19] for supporting Machine Learning Inference workflows. Cloud Providers like AWS [20] have pursued this idea and built a fully-managed Serverless Platform [21] to host ML Inference jobs.

Generally, serverless workloads [22] consist of short-running functions with a low memory footprint. However, Inference workloads usually require loading large models into memory and using specialized support, like GPU or FPGA-based accelerators, to vectorize computations. Most public FaaS offerings don't have GPU support and usually have a limit on the function's available memory and execution time. Further, FaaS platforms have a well-documented problem of cold start [23, 24]. Cold start occurs when we try to service a function request when there are no corresponding functions in the memory. This is a huge challenge for FaaS providers because customers don't pay for the time function in the cold start phase as it is a FaaS provider bottleneck. There are also other problems of huge tail latency, which can lead to violating customer SLAs (Service Level Agreements), making it difficult for latency-critical applications to use FaaS.

In my thesis, we delve into the analysis of Inference workloads on FaaS platforms and investigate their performance characteristics and resource utilization. We performed some of

our experiments on OpenFaaS [25], a widely used open-source FaaS platform that uses containers for function sandboxing, and Kubernetes [26], a container orchestration platform to analyze the bottlenecks and system limitations involved in scaling ML inference jobs to a data center scale. Experiments we performed show that the cold start problem is substantially more in serverless inference. This is mainly because during inference, in addition to VM boot-time, functions also need to load the machine learning model to memory. The time taken to load the model can overshadow VM boot time if the model is large enough. Furthermore, current machine learning trends show us that models keep getting larger with time, so the problem of model loading will only increase. VM boot might not even be a bottleneck in the future because of the large model loading times. Additionally, in serverless inference, FaaS providers increase the function replicas to handle more requests, aka horizontal scaling. During function scaling, if the replicas are present on the same node then the function replicas copy the same machine-learning model in memory resulting in a waste of memory due to memory duplication.

To solve the problems of model loading and duplication, in my thesis, we propose a new design that rethinks the way serverless functions are executed and deployed. This design complements the research done in the latest state-of-the-art papers [27, 28].

1.2 Contributions

In this thesis we identify the bottlenecks involved in running inference services on serverless platforms and propose an intuitive container model which avoids ML model loading thereby addressing the cold start problem, reduces memory footprint and the time to scale. To enable auto-scale for this container model we use a hybrid scaling approach and demonstrate it's benefits over existing state of the art serverless inference works and serverless platforms.

Summary of the contributions:

- Identified that ML Inference falls under a class of workloads that scales with CPU and has a predictable upper bound for memory.
- Proposed a new hybrid autoscaling design for modern workloads to scale with CPU and pods.
  - This design still has the concept of pod to benefit from all the Kubernetes features
  - Reduces memory footprint significantly and avoids colds starts
- Non intrusive design benefits from all the recent innovations in serverless, containers and microVMs.

1.3 Thesis Organization

Chapter 2 introduces concepts and of serverless computing and machine learning inference. Here we also talk about containers, microVMs, different multi tenant security models and design choices.

Chapter 3 describes the relevant work done on serverless computing, machine learning inference and serverless inference. Since this is a new and evolving area lot of the state of the art work is being done in industry so we also talk about different inference platforms.

Chapter 4 documents all the experiments performed to analyze inference on serverless platforms. Here we also talk about our discoveries and recommend different platform choices for ML inference.

Chapter 5 talks about our container model which solves the aforementioned problems mentioned in chapter 4. Further, we also talk about the scaling problems of our design and how we can overcome them using a hybrid scaling approach.

Chapter 6 and Chapter 7 compares our work with existing industry standard approaches and we summarize our work.

Chapter 2

Background

We provide a background of Serverless Computing platforms and Deep Learning Inference workloads and their resource usage behavior.

2.1 Machine Learning

Machine learning is a field of artificial intelligence that focuses on developing algorithms and models that enable computers to learn and make predictions or take actions without being explicitly programmed. In a more mathematical terms every task or a service is a mathematical function $f(x)$ but some tasks are very complex for us to formulate so we generate data points and use machine learning to approximate an ideal function $f(x)$ with a model $g(x)$. Machine learning is widely used in industry these days and any machine learning pipeline has 4 stages to it:

- **Data Collection:** Gathering relevant data that is representative of the problem or task at hand. This data serves as the input for the learning process.
- **Data Preprocessing:** Cleaning and transforming the data to ensure its quality and suitability for the learning algorithms. This step may involve handling missing values, normalizing or scaling features, and encoding categorical variables.
- **Model Training:** Choosing an appropriate machine learning model or algorithm and

training it on the prepared data. The model learns from the data and adjusts its internal parameters to make accurate predictions or decisions.

- **Inference:** In stage of inference we deploy this model to serve unseen real world requests. Inference happens anytime the service gets a request and an inference operation generally takes less than few seconds for most models.

2.1.1 Popular ML Models:

In this section we introduce a few popular ML models used in real world applications.

- **BERT for NLP:** BERT large model has approximately 340 million parameters and it used in NLP applications like fake review detection, chatbots, etc. BERT model when loaded takes 500MB to 1GB space in memory.

- **GPT for NLP:** GPT models are typically large due to their transformer-based architecture and extensive pre-training. The original GPT model (GPT-1) had approximately 117 million parameters. More recent versions, such as GPT-2 and GPT-3, are significantly larger, with GPT-2 having 1.5 billion parameters in its largest variant, and GPT-3 having up to 175 billion parameters in its largest variant. GPT can be seen in more modern chatbots like chatGPT[13] and Bard[29].

- **CNN for image processing:** CNN models are used for feature extraction and generally seen in applications like face detection, image captioning, etc. These models are smaller in size compared to NLP models like BERT and GPT.

- **GAN for content generation:** GANs are used for image generation, data anonymization, and deep fakes, etc.

2.2 Machine Learning Inference

Inference workloads usually involve applying a pre-trained ML/deep neural network model to generate content, make predictions, or extract meaningful information from input data. In contrast to the training phase, where the model learns to recognize patterns and relationships in the data, inference focuses on utilizing the trained model to make accurate predictions on new, unseen data. Inference plays a crucial role in various applications, including image and speech recognition, natural language processing, recommendation systems, and autonomous driving.

Deep learning models excel at capturing intricate patterns and representations from high-dimensional data, enabling them to make accurate predictions. However, their computational complexity and resource requirements pose challenges for deploying and scaling them in production environments. Traditional approaches to deep learning inference involve running the model on dedicated hardware resources, such as graphical processing units (GPUs) or FPGA-based accelerators like TPUs [30]. While these setups offer high performance, they often require significant upfront investment and may not be flexible enough to handle varying workloads efficiently. This has led to the emergence of serverless computing platforms as an alternative solution for deploying deep learning inference workloads.

2.3 Serverless Computing

Serverless computing, also known as Functions as a Service (FaaS), is a cloud computing model that enables developers to focus on writing and deploying code without the need to manage the underlying infrastructure. In this model, the cloud provider dynamically allocates and manages resources to execute code in response to specific events or requests.

Serverless computing exhibits several key characteristics that distinguish it from traditional cloud computing models:

2.3.1 Event-Driven Architecture

- **Event-Driven Architecture:** Serverless applications are built around an event-driven architecture. Events, such as HTTP requests, database updates, or scheduled events, trigger the execution of associated functions. When an event occurs, the corresponding function is invoked, performs its task, and exits. This event-driven approach enables developers to create applications that respond quickly and efficiently to various events.

- **Unit of Execution:** Different serverless platforms have different sandboxes to run untrusted user functions. Two major function sandboxes or isolated units of execution are containers [31] and microVMs [24, 32]. Containers are preferred as function sandboxes when the platform is a single-tenant platform or a trusted multi-tenant platform, whereas microVMs are the preferred function sandboxes for an untrusted multi-tenant setup [22, 24, 32]. For example, cloud service providers like AWS Lambda and Azure Functions use VMs for function code isolation, while single-tenant platforms like OpenFaaS [25] and OpenWhisk [33] use containers for isolation.

2.3.2 Features

- **No Server Management:** One of the primary advantages of serverless computing is the elimination of server management tasks. Developers are relieved from provisioning, scaling, and maintaining servers. The cloud provider takes care of these tasks automatically, allowing developers to concentrate on writing code and delivering value

to end-users.

- **Automatic Scaling:** Automatic scaling is a crucial feature of serverless computing, wherein the cloud provider dynamically adjusts the number of function instances based on the incoming workload. This automatic scaling ensures that the application can effectively handle fluctuations in demand. As a result, serverless architectures are highly scalable, offering the ability to handle large traffic surges without requiring manual intervention.

    1. **Vertical Scaling:** Vertical scaling involves adding more resources to a container or virtual machine to handle increased load. For instance, in Kubernetes, vertical scaling is employed to determine the precise resource requirements of a pod at runtime and minimize resource wastage. Vertical scaling is typically used for stateful monolithic applications where horizontal scaling would involve significant startup time and synchronization issues.
    2. **Horizontal Scaling:** Horizontal scaling is also called as scaling out where we add multiple containers or virtual machines to distributed the load and to maintain SLAs. For example, in Kubernetes horizontal scaling is done by adding more container pods to the cluster. This type of scaling is generally used for state less applications, web applications, etc.

- **Billing Based on Usage:** Serverless computing follows a pay-per-use pricing model. Instead of being charged for provisioned capacity or idle resources, users are billed for the actual execution time of their functions. This pricing model allows for cost efficiency, especially for applications with unpredictable or sporadic workloads. Users only pay for the resources consumed during function execution, reducing operational costs.

2.3.3 Limitations

FaaS has a lot of features but it also has some drawbacks and limitations. One of the main limitations of serverless platforms is their inherent cold start problem. When a function is triggered for the first time or after a period of inactivity, the serverless platform needs to provision the necessary resources and initialize the environment, resulting in a noticeable delay. This cold start latency can negatively impact real-time applications or those requiring strict response time. Although various optimization techniques have been developed to mitigate this issue, it remains a significant limitation for certain use cases. Another drawback of serverless platforms is the potential lack of control over the underlying infrastructure. With serverless platforms, developers are abstracted from the underlying infrastructure, which means they have limited control and visibility into the serverless environment. This lack of control can be problematic for applications with specific resource requirements or those that rely on specialized hardware. Additionally, serverless providers impose various limitations on resources, execution time, and storage, which can restrict the scalability and flexibility of certain applications. Moreover, most public cloud providers force functions to be stateless and require additional storage services to communicate.

2.3.4 Serverless Inference

In recent years, the intersection of deep learning inference and serverless platforms has gained significant attention from research communities and the industry. Researchers have explored various techniques to optimize the performance and efficiency of deep learning inference on serverless platforms, including tensor sharing [34], heterogeneous resource abstraction [15], and serverless function composition [19]. These efforts aim to improve inference latency, reduce resource consumption, and enhance cost-effectiveness for deploying deep learning

models on serverless platforms.

Chapter 3

Review of Literature

3.1 Serverless Computing

Serverless Commputing is an emerging cloud computing paradigm which allows users to run their workloads and get billed only for the time the application runs. To provide such guarantees, serverless computing starts the application only when a request comes. This results in cold start problems which increases tail latency, SLA violations and application cost.

Firecracker [24] Firecracker proposes a novel microVM design which uses a minimal VM to run serverless workloads securely and also reducing the cold start time along the way.

RunD [27] RunD focusses on high densisity deployment and concurrent and high concurrency container startup. They identified bottlenecks in concurrent container startup, rootfs and microVM memory footprint. They solve these problems by pre-creating a pool of cgroups, and using a micorVM template.

Faasm [35] This paper introduces Faaslets, an isolation abstraction for high-performance serverless computing, addressing limitations in existing platforms by enabling direct memory sharing between functions. Faaslets leverage software-fault isolation (SFI) with WebAssembly and restore from pre-initialized snapshots for faster initialization. Compared to container-based platforms, Faasm achieves significant speed-ups and resource savings in

machine learning model training and inference tasks.

3.2 Machine Learning Inference

Machine learning inference and serverless inference are relatively new research areas and there are no industry standards or guidelines in this area. In FaaS Inference research focus has mostly been on improving the latency to avoid SLA violations [15, 36, 37]. These works apply optimizations like batching, model profiling, providing heterogeneous resource abstractions for CPU and accelerators, etc.

INFaaS [17] offers an economic model-less distributed inference system where users need to specify their application's performance and accuracy requirements. The system navigates the vast trade-off space of model variants on behalf of the user to meet the application requirements. INFaaS selects the model-variant, model optimizations, and hardware and combines VM-level horizontal auto-scaling characteristics similar to FaaS systems.

Clockwork [38] create their own managed runtime for distributed model serving and they demonstrate that DNN inference jobs have a deterministic performance for GPU since it does not contain any conditional branches. The goal of the paper is to avoid tail latencies by having a central unified view on every resource to better predict latencies.They create their own distributed ML Inference system Clockwork that can serve thousands of models while providing 100ms latency targets for 99.9999% of their requests.

AMPS-Inf [18] proposes an automated model partitioning framework that aims to find the most cost-efficient serverless inference deployment strategy by taking into consideration of memory allocation, layer size, intermediate state transition cost among Lambdas, shape of the partitioned model, execution time and pricing model. The partitioning strategy is

mainly based on the serverless computing characteristic that more memory for a single instance means more computing cycles from the assigned vCPU. There is still work to be investigated on how much parallelized model itself could bring to the performance.

INFLess [15] proposes a dedicated ML serverless platform to provide resource abstraction between CPU and accelerator resources and autotuning of resource provisioning. It first identifies the bottlenecks of existing serverless platforms on ML workloads: High latency for large inference jobs; High latency for small models in a batched setting; Resource over-provisioning and low utilization. Then it proposes its area-specific serverless platforms that proactively predicts the performance latency of requested model by analyzing its DAG; uses an auto-scaling engine to find the best batch size at resource change; tackling cold-start with a LSTH strategy.

Tetris [34] conducts a systematic investigation on characteristics and made the following observations: Inference jobs induce massive startup memory footprint due to the loading of massive model parameters; said model parameters also induce massive computing memory footprint; runtimes are duplicated for the same function's multiple instances; tensors of model parameters and constants are duplicated for multiple instances. Tetris comes up with a tensor sharing mechanism by storing tensors into memory and not loading existing tensors. One thing that Tetris fails to explicitly tackle is the massive memory footprint model parameters except only keeping a single copy. There is no specific near-data scheduling design to tackle this problem. We continue to read more papers to improve on the design for our system.

Chapter 4

Analysis

In our analysis section we demonstrate all the problems with existing serverless inference. Especially, we focus on model loading, memory duplication, and different techniques to load the model. Our first iteration of experiments target container-based Serverless ML Inference jobs . To that end, we use OpenFaaS [25], an open-source container-based Serverless framework that uses Kubernetes to manage the cluster, and exposes APIs to register and deploy functions.

OpenFaaS uses containers to run functions in a secure and isolated way. For serverless inference, we train our ML models and package them as docker images to run on OpenFaaS. With our experiments, we wanted to identify the bottleneck of serverless inference, so we measured the time taken by various phases involved in completing an inference request. Serverless inference can be divided into the following tasks, 1.) Function boot up, 2.) Model loading and 3.) Inference. Function boot depends on the size of the docker image, libraries used, etc.; Model loading depends on model storage location and model size; inference depends on GPU, CPU speed, input size, and complexity of the model. Further, to understand the differences between using different ML models, we use two models 1.) Name Classifier (CNN) and 2.) Bert for NLP[6]. Name Classifier takes text as input to predict the country the person with that name belongs to and with our Bert model we classify text into categories like sports, entertainment, politics, gaming, etc. Both these tasks use machine learning models on paper but Bert is extremely complex than the name classifier which uses simple CNNs. This

difference can be noticed in the trainable parameters, model size, and the docker image size mentioned in Table 4.1. Figure 4.3 and 4.4 show the layer-wise architecture and summary of the machine learning models we used for testing. Figure 4.1 and 4.2 show that cold start is considerably slower than warm start confirming our initial hypothesis. Figure 4.2 shows that as the model size increases model loading time increases, VM boot up stays relatively constant and inference time increases. It is also important to note that warm start doesn't have any model loading phase as the model is already in memory. In figure 4.5 we can see a time series graph of function memory consumption from boot to inference. As expected, during model loading there is a spike in memory consumption. Also, note that memory consumption is very high for a small period during model loading and it reduces once the load operation is complete. This can be a problem in production systems and this behavior is generally called as noisy neighbors. In table 4.3, we can see that function boot up in cold start takes around 3.5s for Name Classifier and 7s for BERT. Model loading is taking 2ms for Name-Classifier and 2500ms for Bert. And the inference time has been relatively low at 4ms for Name Classifier's typical workload of size 10B and for BERT it is taking about 2500ms for 1-1000 paragraphs. For typical workloads, we found that model inference time stays relatively consistent. However, when the input size increase inference time quickly grows and the reason behind this behavior is dependent on model design, input preprocessing, etc. For instance, when the input has 10K paragraphs, BERT's inference time shoots up to 27s. But, having an input of 10K paragraphs is very high for a normal text classification request. To improve model inference time we need to use GPUs instead of CPUs but that is not the focus of our work.

Next, we perform a series of experiments on the Ray platform [39]. Ray is a state of the art distributed computing framework designed for building scalable and high performance applications. Further, Ray model serving is also popular in the machine learning community.

In fact chatGPT is run on the Ray platform. Now we show different optimizations in model loading and the performance benefits of these optimizations on the Ray platform. From table 4.2 and figure 4.6 we can see that single copy and zero copy significantly reduces the model the loading time. This optimization shows us that with zero copy we can spin up the function only when we get an inference request and respond without violating any customer SLAs. Other benefits of zero-copy include the ability to reduce peak memory usage of the function and there won't be any spikes as seen in figure 4.5. This is very useful in production systems as we won't page out other function pages, aka reducing noisy neighbors.

From these experiments and current ML trends, we can see that model loading is the bottleneck in serverless inference, and model sharing will help us by avoiding cold start, reducing the overall memory footprint, and it also increases the number of functions the FaaS provider can host in a single node. Based on all our analysis we propose a design which solves all the mentioned problems in a very intuitive way in chapter 5.

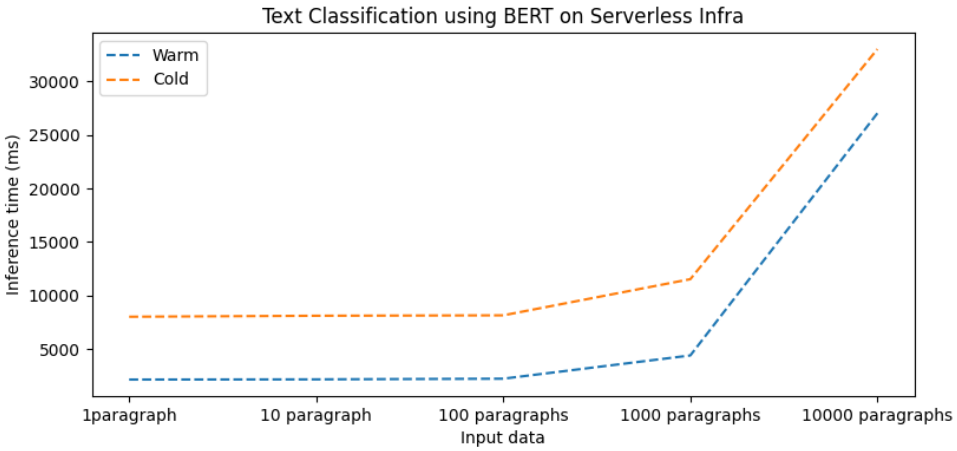


Figure 4.1: Comparing the total inference time of BERT model for cold and warm start.

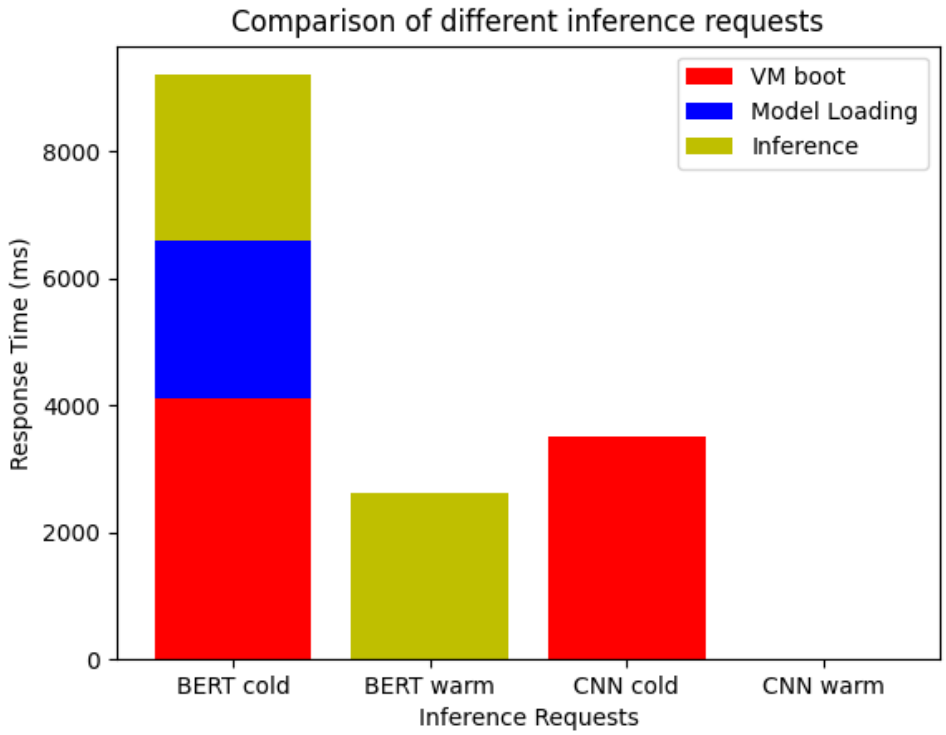


Figure 4.2: Comparing the time taken by different phases of cold and warm start of BERT and CNN.

Workload	Model	Size on disk	Parameters	Function Memory	Docker Image Size
Name Classification	CNN	80 KB	21,840	145 M	2.91 G
Text Classification	BERT	527.79 MB	138,357,544	1103 M	4.3 G

Table 4.1: Comparing FaaS characteristics of CNN and BERT

Model	No optimization	Single Copy	Zero copy
BERT	1.83	0.134	0..06835
BigBird	1.8	0.176	0.006531
GPT2	1.54	0.244	0.008338

Table 4.2: Comparing model loading times of different ML models with no optimization single copy, zero copy

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 224, 224]           1,792
              ReLU-2         [-1, 64, 224, 224]               0
            Conv2d-3         [-1, 64, 224, 224]          36,928
              ReLU-4         [-1, 64, 224, 224]               0
         MaxPool2d-5         [-1, 64, 112, 112]               0
            Conv2d-6        [-1, 128, 112, 112]          73,856
              ReLU-7        [-1, 128, 112, 112]               0
            Conv2d-8        [-1, 128, 112, 112]         147,584
              ReLU-9        [-1, 128, 112, 112]               0
        MaxPool2d-10          [-1, 128, 56, 56]               0
           Conv2d-11          [-1, 256, 56, 56]         295,168
             ReLU-12          [-1, 256, 56, 56]               0
           Conv2d-13          [-1, 256, 56, 56]         590,080
             ReLU-14          [-1, 256, 56, 56]               0
           Conv2d-15          [-1, 256, 56, 56]         590,080
             ReLU-16          [-1, 256, 56, 56]               0
        MaxPool2d-17          [-1, 256, 28, 28]               0
           Conv2d-18          [-1, 512, 28, 28]       1,180,160
             ReLU-19          [-1, 512, 28, 28]               0
           Conv2d-20          [-1, 512, 28, 28]       2,359,808
             ReLU-21          [-1, 512, 28, 28]               0
           Conv2d-22          [-1, 512, 28, 28]       2,359,808
             ReLU-23          [-1, 512, 28, 28]               0
        MaxPool2d-24          [-1, 512, 14, 14]               0
           Conv2d-25          [-1, 512, 14, 14]       2,359,808
             ReLU-26          [-1, 512, 14, 14]               0
           Conv2d-27          [-1, 512, 14, 14]       2,359,808
             ReLU-28          [-1, 512, 14, 14]               0
           Conv2d-29          [-1, 512, 14, 14]       2,359,808
             ReLU-30          [-1, 512, 14, 14]               0
        MaxPool2d-31            [-1, 512, 7, 7]               0
AdaptiveAvgPool2d-32            [-1, 512, 7, 7]               0
           Linear-33                 [-1, 4096]     102,764,544
             ReLU-34                 [-1, 4096]               0
          Dropout-35                 [-1, 4096]               0
           Linear-36                 [-1, 4096]      16,781,312
             ReLU-37                 [-1, 4096]               0
          Dropout-38                 [-1, 4096]               0
           Linear-39                 [-1, 1000]       4,097,000
================================================================
Total params: 138,357,544
Trainable params: 138,357,544
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 218.78
Params size (MB): 527.79
Estimated Total Size (MB): 747.15
----------------------------------------------------------------
```

Figure 4.3: BERT model summary

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 10, 24, 24]             260
            Conv2d-2            [-1, 20, 8, 8]           5,020
         Dropout2d-3            [-1, 20, 8, 8]               0
            Linear-4                  [-1, 50]          16,050
            Linear-5                  [-1, 10]             510
================================================================
Total params: 21,840
Trainable params: 21,840
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.06
Params size (MB): 0.08
Estimated Total Size (MB): 0.15
----------------------------------------------------------------
```

Figure 4.4: CNN model summary

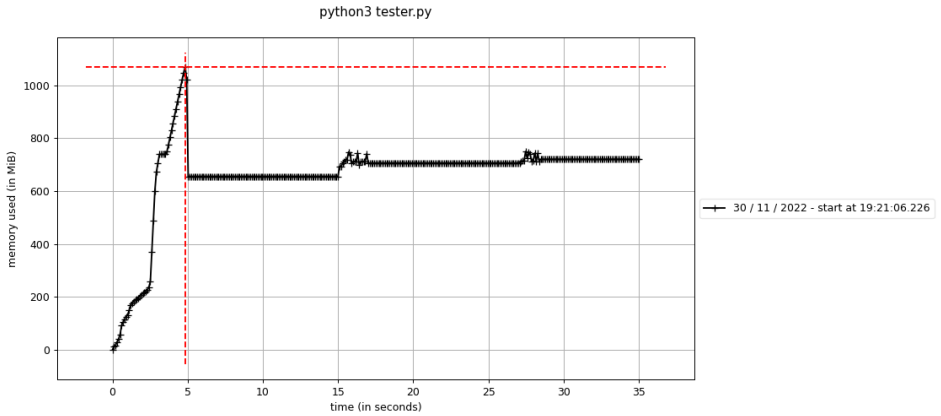


Figure 4.5: Memory consumption of BERT function from boot to inference

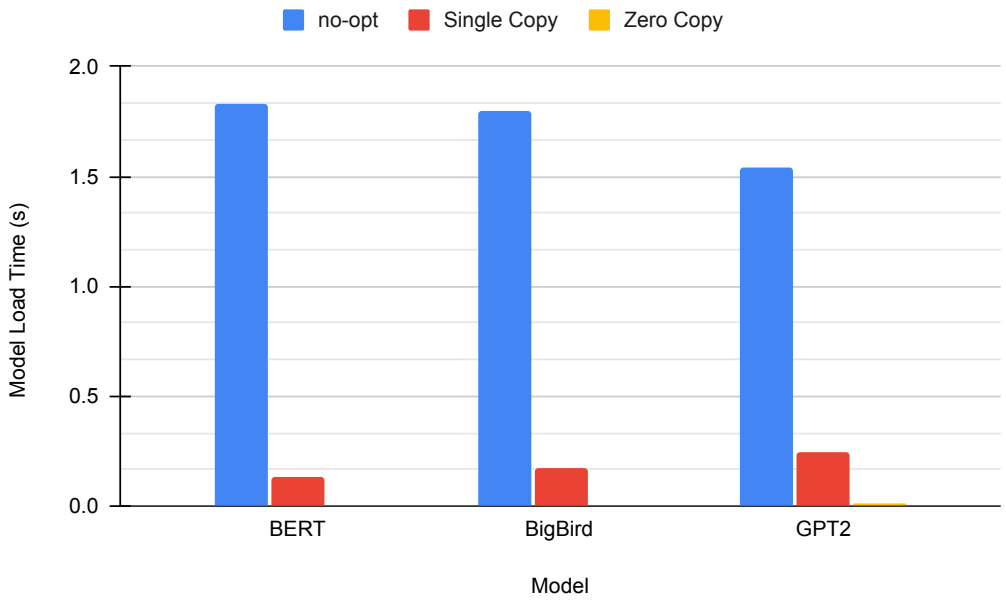


Figure 4.6: Model load time for different copy mechanisms

Model Name	Start Type	Input Data Size	Paragraphs	Total Time(ms)	Model Load Time (ms)	Inference Time(ms)
CNN	cold	10B	<1	3500	2	4
CNN	warm	10B	<1	14	2	4
CNN	warm	1KB	3	187	2	174
CNN	warm	10KB	30	1035	2	1020
CNN	warm	36KB	100	4742	3	4732
CNN	warm	36KB	100	4742	3	122262
Bert	cold	10B	<1	8140	2690	1547
Bert	warm	10B	<1	2636	2487	2622
Bert	warm	36KB	100	2219	2487	2188
Bert	warm	332KB	1000	4390	2487	4375
Bert	warm	3.2MB	10000	27022	2487	27022

Table 4.3: Breakdown of inference time for CNN and BERT's cold and warm start

Chapter 5

Design and Implementation

The design patterns of Function-as-a-Service (FaaS) or serverless architectures are predominantly influenced by the choices made by AWS Lambda, which often introduces challenges in the pursuit of user simplicity. Recent research on FaaS has focused on solving these issues in various security settings, such as single-tenant and multi-tenant environments. While single-tenant FaaS problems are relatively easier to solve due to lower security abstractions, higher control over machines, and better understanding of workloads, multi-tenant FaaS challenges are more complex and have a broader impact because of resource sharing, high utilization, and low cost. In this paper, we aim to address FaaS inference in a multi-tenant context, where a cloud service provider hosts machine learning inference workloads from different customers. We compare our design with current multi-tenant function hosting solutions such as AWS Lambda and OpenFaaS, providing insights into potential improvements for the industry. Also, please note that in our discussion we interchange containers and microVMs because our design works for both these sandboxing mechanisms.

5.1 Threads VS microVMs

Cloud FaaS providers utilize microVMs[24] to isolate function execution from untrusted tenants. To service requests, they follow these steps: 1) Check for an idle microVM hosting the function; 2) If not found, start a microVM and boot the function container; 3) Service

the request. Additionally, each microVM serves only one request at a time; to handle more requests, FaaS providers simply launch more microVMs. This approach is seemingly inefficient, but since these functions are short-lived and generally use less memory, it is acceptable for the time being. However, employing the same method for hosting ML inference workloads proves highly inefficient and does not scale well due to high model loading times and memory footprints.

To address these autoscaling issues we propose a design where to serve more requests we spawn more threads in a container instead of starting more microVM/container replicas. We hypothesize that: 1) Spawning threads is faster than starting microVMs/containers; and 2) Model is shared in thread-based scaling hence avoiding model duplication, model loading and VM boot. In line with these assumptions, we propose a design where a unit of function comprises: 1) a microVM; 2) a container; and 3) a worker thread pool to serve requests. The differences between AWS Lambda and our approach are illustrated in these two figures [5.1, 5.2]. As demonstrated by these figures, our design can accommodate more multi-tenant functions on a single node and service requests more rapidly.

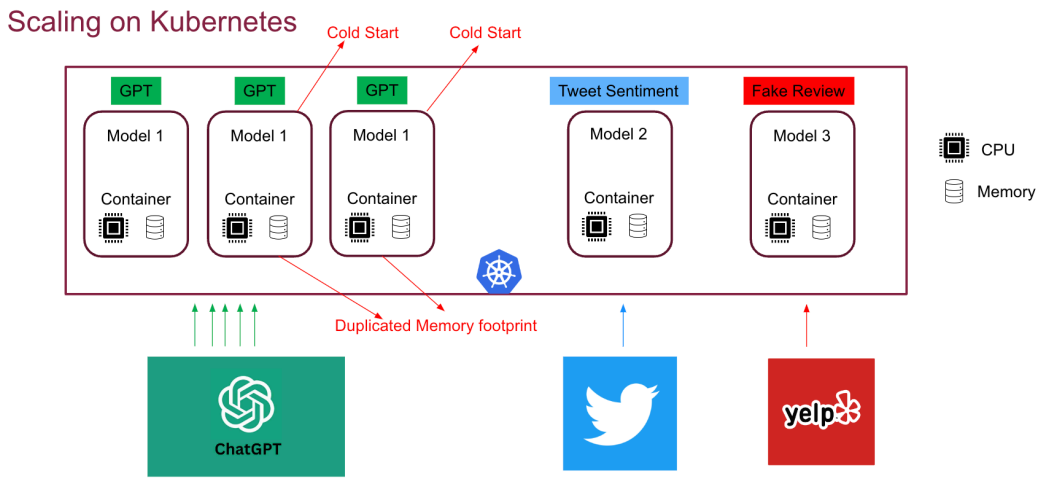


Figure 5.1: AWS Lambda's design, general kubernetes autoscaling design for serving ML models

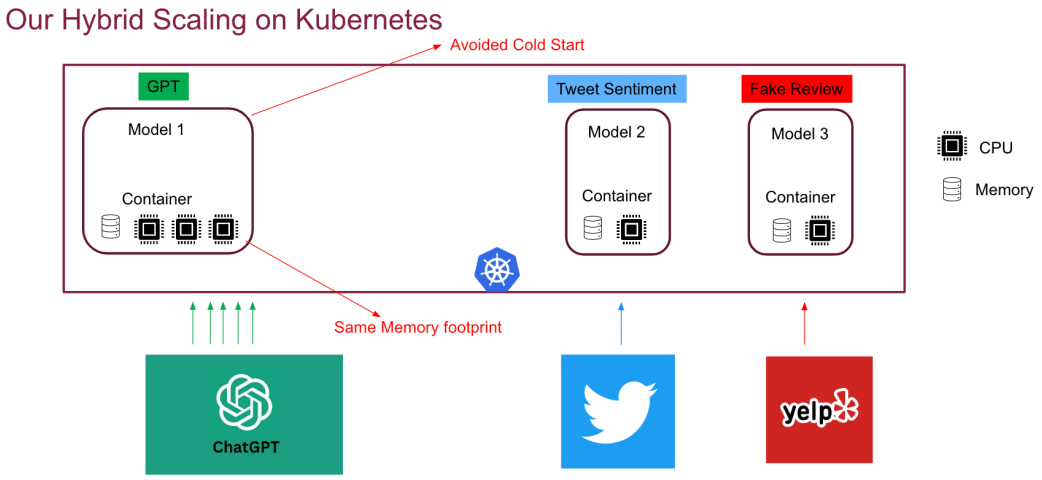


Figure 5.2: Our autoscaling design for serving ML models

5.1.1 Problems of thread based scaling

- **Memory Fragmentation:** References [27, 32] discuss the issues of memory fragmentation and reclamation in a multi-container per VM approach. This is not a problem with our approach because our memory requirements don't change much for handling more concurrent requests because model stays the same and we would need little more buffer to handle the requests which is negligible compared to the model size. This behavior is depicted in figure 5.3 and 5.4 when we used apache-bench to stress test our container with 100 concurrent requests. From figure 5.4 we can see that most of the data accesses are done for model and the data access pattern doesn't change with scale.

- **Virtual Machine Resizing:** For our approach to work we need runtime resizing of containers and microVMs. Currently, containerd supports runtime resizing of containers and VMware also supports hot CPU and memory addition to microVMs. So we can leverage this existing technology to implement our design [40]. Further, as depicted in figure 5.3 we are not concerned about changing memory requirements as they stay

consistent for ML inference. So we only need to manage changing cpu requirements and that is a much easier problem.

- **Autoscaling:** Currently for autoscaling you can either do vertical scaling or horizontal scaling but not both together. This is because you wouldn't want two autoscalers changing your deployment back and forth. We need to modify this behaviour to fully realize our design. So we use a hybrid autoscaler which we talk about in the next section.

5.2 Hybrid Auto Scaling

From our design the objective of our autoscaling strategy is to make sure that there are no replicas of a pod on the same node. By doing so we can add the replica's cpu resources to our original pod avoiding memory duplication, container boot, etc. For our hybrid scaling strategy we use a horizontal pod autoscaler and when we decide to start a pod on a node we just check if there is already a replica of this pod on that node. If we find a replica we just add more cpu to that original replica or else we start this pod.

5.3 Implementation

5.3.1 Python Model Serving Server

To demonstrate the effectiveness of our approach, we implemented a proof of concept for our FaaS inference system. We began with a simple multithreaded Python Flask server to perform inference. During our experiments, we discovered that multithreading in Python is largely sequential due to the well-documented issue of the Global Interpreter Lock (GIL),

which results in serving requests sequentially at best, as seen in Figure 5.5. To address the GIL problem, some suggest using multiprocessing instead of multithreading. However, we aimed to test the efficacy of our thread-based program and therefore implemented a lightweight server in C++ that supports worker thread pools. It is important to note that Using multiprocessing would also give similar latency and memory benefits because of copy-on-write mechanism.

Throughout this process, we observed that while Python is the industry standard for training ML models, various emerging technologies convert inefficient Python code to more efficient language runtimes for serving purposes. Examples include PyTorch's TorchScript, ONNX runtime, and TensorFlow's C++ API. In our experiments, we utilized PyTorch's TorchScript API to convert our model into a C++ model.

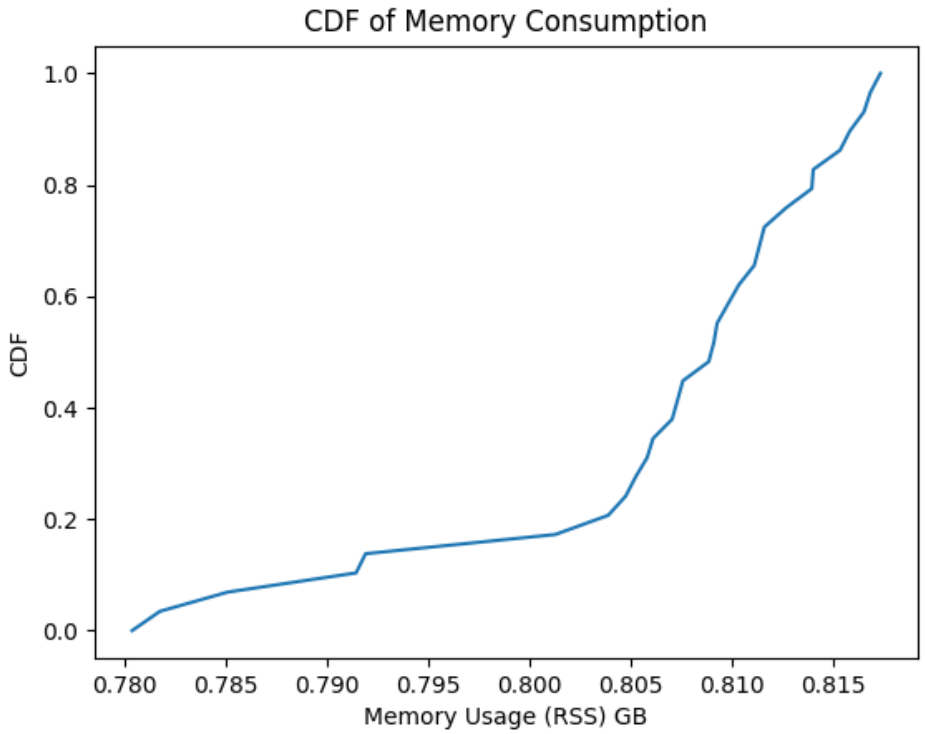


Figure 5.3: CDF distribution of memory consumption for 100 concurrent requests

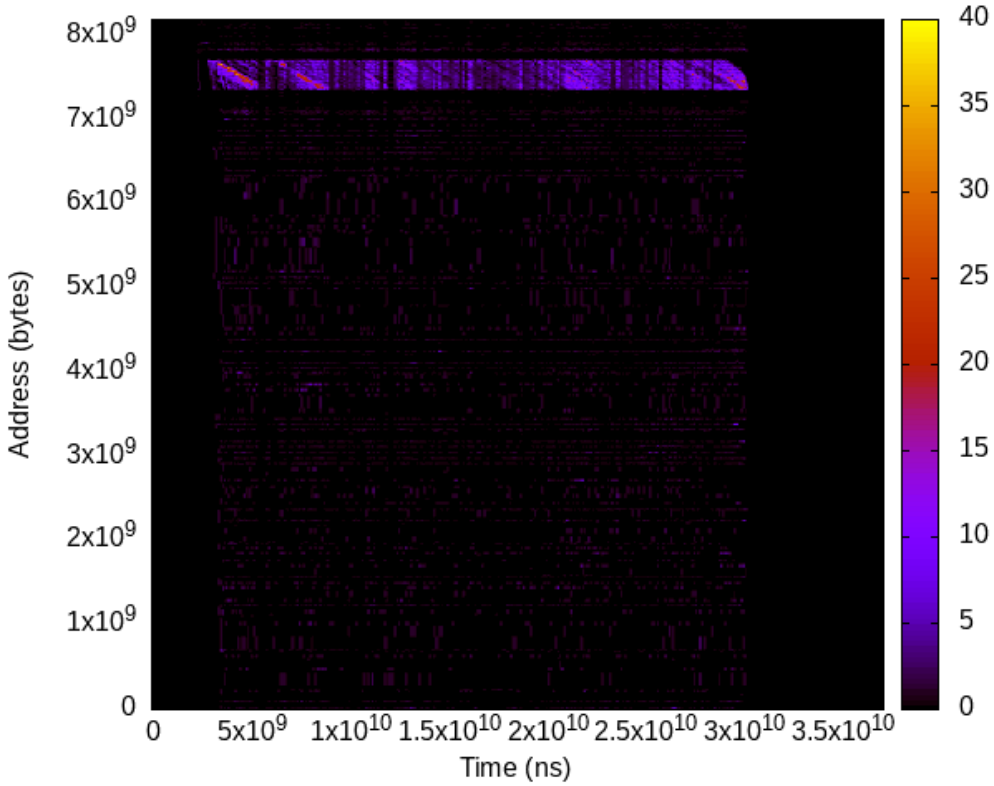


Figure 5.4: Heatmap for virtual memory data access patterns of BERT Inference

5.3.2 C++ Model Serving Server

To implement our solution, we utilized a C++ HTTP server and Torch modules to serve web requests and load the ML model, respectively. Next, we designed a CPU-intensive workload to assess whether the server scaled as expected. This CPU-intensive workload simply executed a large loop to keep the CPU busy. We conducted experiments with varying concurrency levels for both ML inference and CPU-intensive workloads, as depicted in Figure 5.6. One request for ML inference took approximately 200ms, while one request for the CPU-intensive workload required 500ms.

As seen in Figure 5.6, both the CPU-intensive workload and ML inference are scaling quite well. This shows the efficacy of our cpu server.

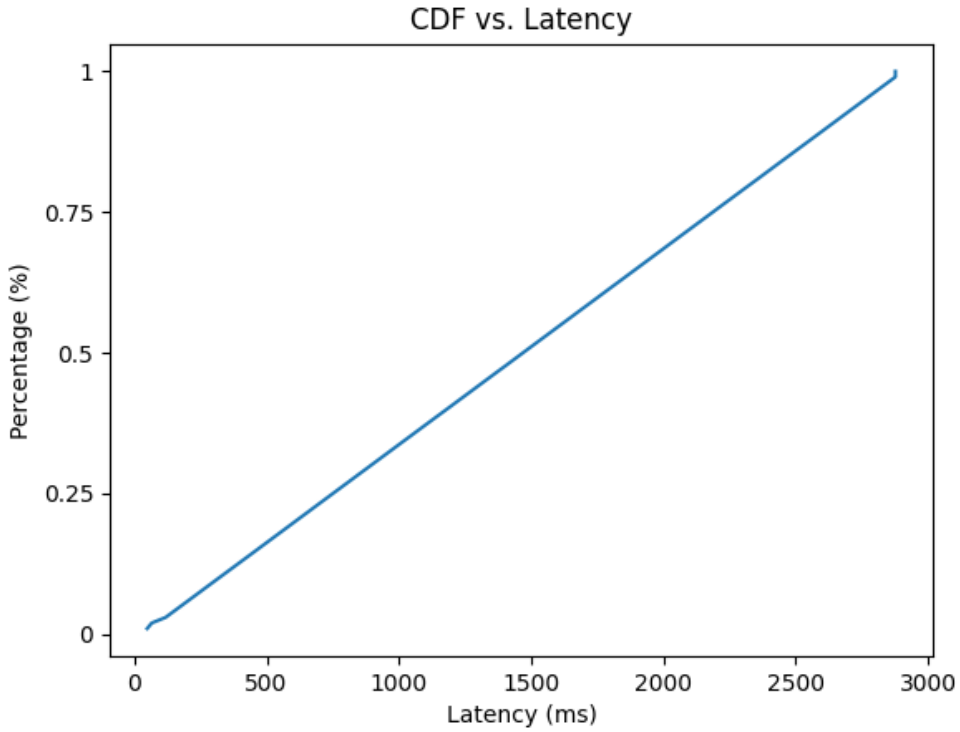
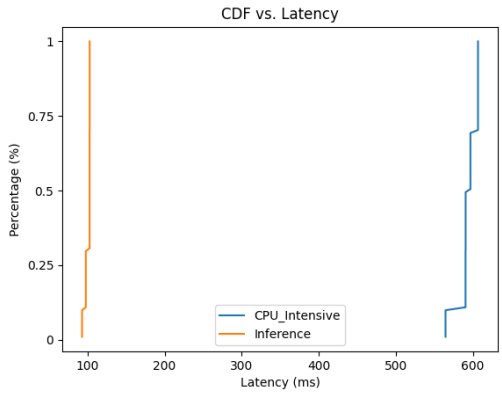


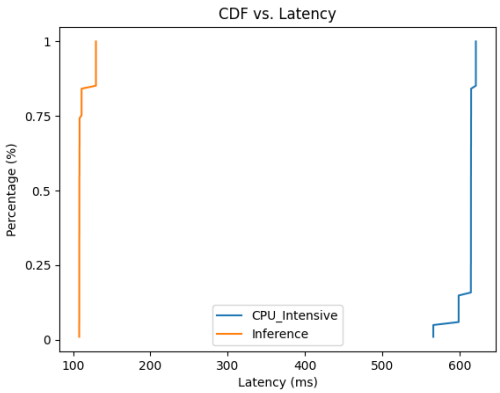
Figure 5.5: Model serving with Python's GIL

5.3.3 Kubernetes Autoscaler

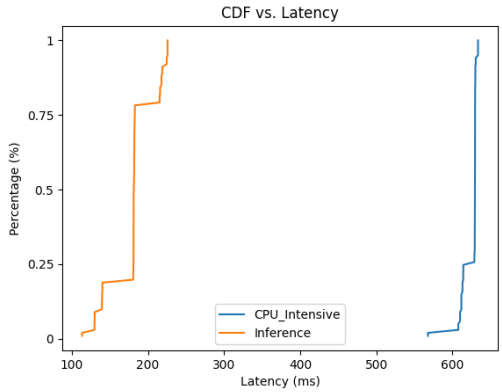
Kubernetes allows us to listen on custom metrics and set custom policies to auto scale. In our experiments we used P-99 latency to take scheduling decisions. We can also set different pod affinity rules to make sure that kubernetes Horizontal Pod Autoscaler prioritizes scheduling replicas on the same node. Coming to Vertical Pod Autoscaler, currently Kubernetes doesn't support restart free or live VPA but will roll out soon [41]. So to emulate our design we use a docker container and a custom proxy server to collect metrics required to scale up.



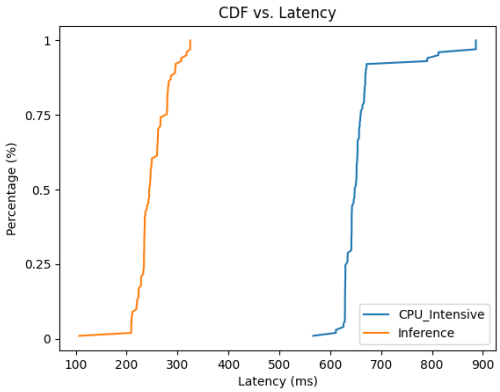
(a) 5 concurrent requests



(b) 10 concurrent requests



(c) 30 concurrent requests



(d) 50 concurrent requests

Figure 5.6: CDF over latency for ML inference and CPU intensive workload

Chapter 6

Results

In this chapter, we discuss our experimental setup and results, focusing on memory and latencies. Our final experiments were conducted on a single-node Kubernetes cluster with each node equipped with a 64-core CPU and 187 GiB of memory. We opted for a single-node cluster due to the lack of support for restart-free pod updates in Kubernetes and because the benefits of our design are only observed when pod replicas are scheduled to the same node.

Regarding the pod configuration, each pod was assigned a CPU limit of 1 vCPU, and the auto-scaling strategy involved doubling the number of pods every 5 seconds if the p99 latency exceeded 500ms. Consequently, during periods of high load, our hybrid approach continually doubles the vCPU count until it reaches 56 vCPUs, while the Horizontal Pod Autoscaler (HPA) doubles the number of pods until it reaches 56 pods. In case of a p99 latency violation, the vCPU and pod count change as follows: 1, 2, 4, 8, 16, 32, 56 for the hybrid approach and HPA.

For machine learning inference, we utilized the BERT model, and Python's Locust module was employed for load generation, setting the active user count to 100. Each user sent requests every 2-3 seconds. The load generation was executed for a few minutes, during which we collected latencies and memory consumption data for both the hybrid and HPA approaches. Figure 6.3 provides a visualization of this load generation process.

During our load tests, we observed that HPA initially started with 1 pod and scaled out to 56 pods, while the hybrid approach started with 1 vCPU and ended up with 56 vCPUs. The memory comparison, depicted in Figure 6.1, illustrates this behavior. The evaluation clearly demonstrates the amount of memory duplication required for ML inference services and emphasizes the advantages of employing a hybrid approach.

Moreover, Figure 6.2 illustrates that the hybrid approach outperforms HPA, particularly in terms of p-80 to p-100 latency. Tail latencies are influenced by multiple container boots, model loads, and the absence of cache locality. In contrast, the hybrid approach employs a thread pool, ensuring cache locality, efficient thread switching, and no model loading. Additionally, Figure 6.2 shows that the hybrid approach is slower than the bare-metal configuration with 56 vCPUs because we only scale up to 56 CPUs after 30 seconds. These experiments demonstrate that the hybrid approach provides near-native performance, while the HPA approach suffers from significant tail latencies.

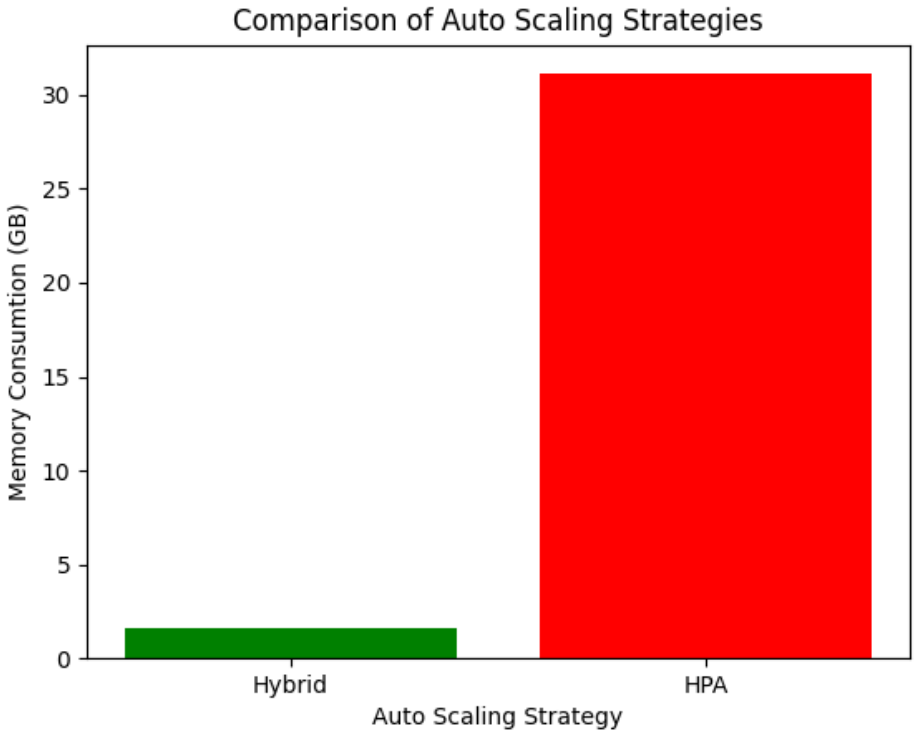


Figure 6.1: Memory consumption of ML inference service for Hybrid and HPA

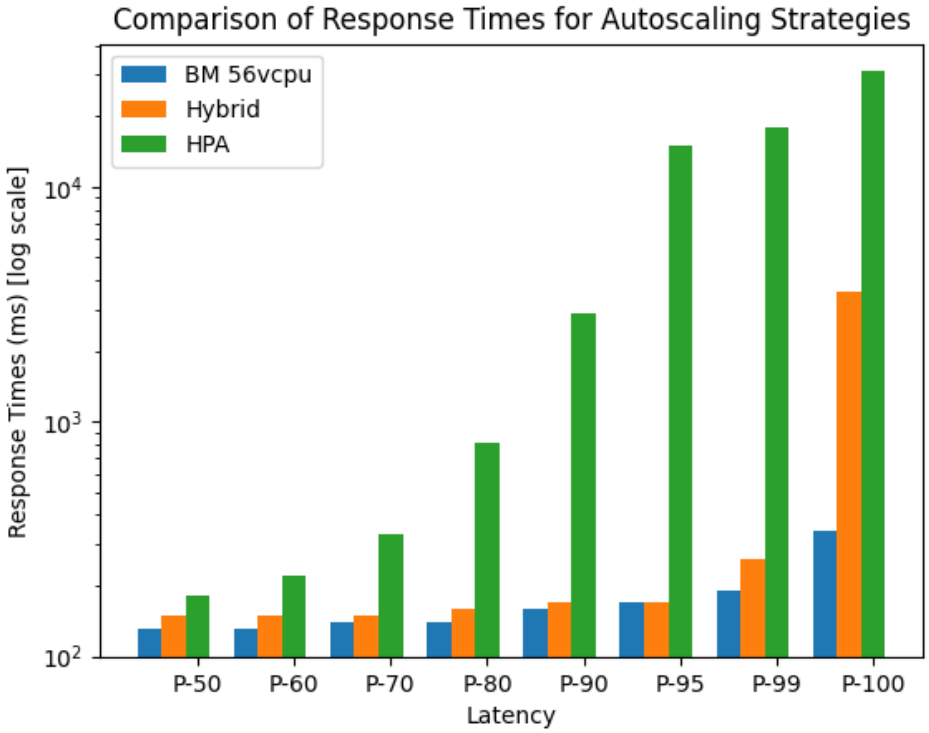


Figure 6.2: Latency comparison for Hybrid and HPA

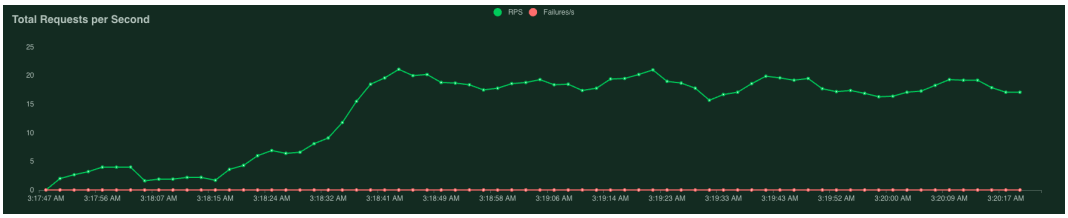


Figure 6.3: Total request per second of our load generator. Starts with 1 user and increases 2 users every second till we reach 100 concurrent users

Chapter 7

Conclusion

Various aspects of future work can be explored in this research. First, we aim to ensure that our proof of concept functions as intended with more experiments when there are multiple tenants focusing on fairness and performance. Second, we plan to deploy this proof of concept to a microVM and evaluate its performance. Third, we need to implement our autoscaling strategy on kubernetes when the live VPA feature is available. Finally, once our FaaS Inference platform is ready, we will compare it to existing FaaS platforms and model serving platforms, such as AWS Lambda, Kserve, and Ray Serve (state-of-the-art).

In summary, our experiments have identified model loading as a primary bottleneck in serverless inference jobs. Model sharing can help mitigate this issue by avoiding cold starts, reducing the overall memory footprint, and increasing the number of functions a FaaS provider can host on a single node. We have proposed an intuitive design to address these problems and show its efficacy with our experiments.

Bibliography

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

[2] Microsoft, "Resnet-152 v1.5." https://huggingface.co/microsoft/resnet-152, 2022.

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

[5] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.

[6] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[7] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[8] 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.

[9] 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.

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

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

[12] 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.

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

[14] 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.

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

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

[21] A. W. Services, "Build, train, and deploy machine learning (ml) models for any use case with fully managed infrastructure, tools, and workflows." https://aws.amazon.com/sagemaker/.

[22] M. Shahrad, R. Fonseca, I. n. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'20, (USA), USENIX Association, 2020.

[23] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, *et al.*, "Cloud programming simplified: A berkeley view on serverless computing," *arXiv preprint arXiv:1902.03383*, 2019.

[24] Firecracker, "Secure and fast micro-VMs for serverless computing." https://github.com/firecracker-microvm/firecracker.

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

[26] Kubernetes, "Production-grade container orchestration." https://kubernetes.io/.

[27] Z. Li, J. Cheng, Q. Chen, E. Guan, Z. Bian, Y. Tao, B. Zha, Q. Wang, W. Han, and M. Guo, "{RunD}: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pp. 53–68, 2022.

[28] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, analysis, and optimization of serverless function snapshots," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, (New York, NY, USA), p. 559–572, Association for Computing Machinery, 2021.

[29] "Bard." https://bard.google.com/. Accessed on June 3, 2023.

[30] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony,

K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, and J. Ross, "In-datacenter performance analysis of a tensor processing unit," 2017.

[31] "Open container initiative." https://opencontainers.org/. Accessed on June 3, 2023.

[32] Z. Li, L. Guo, J. Cheng, Q. Chen, B. He, and M. Guo, "The serverless computing survey: A technical primer for design architecture," *ACM Computing Surveys (CSUR)*, vol. 54, no. 10s, pp. 1–34, 2022.

[33] "Open source serverless cloud platform." https://openwhisk.apache.org/. Accessed on June 3, 2023.

[34] 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.

[35] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pp. 419–433, USENIX Association, July 2020.

[36] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "Batch: Machine learning inference serving on serverless platforms with adaptive batching," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, IEEE, 2020.

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

[38] 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.

[39] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica, "Ray: A distributed framework for emerging ai applications," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, (USA), p. 561–577, USENIX Association, 2018.

[40] "Vm cpu hot add." https://docs.vmware.com/en/VMware-vSphere/7.0/com.vmware.vsphere.vm_admin.doc/GUID-285BB774-CE69-4477-9011-598FEF1E9ACB.html. Accessed on May 3, 2023.

[41] "Kubernetes live pod vpa." https://github.com/kubernetes/enhancements/pull/1883. Accessed on June 3, 2023.