university of groningen

faculty of science and engineering

Comparitive Study Between GPU Utilization and Inflight Requests Based Autoscaling systems

Master's Thesis

Siddharth Mitra
S4138430

Master of Science in Computing Science
University of Groningen

Internal Supervisor:
Prof. Dr. Alexander Lazovik
University of Groningen

Internal Supervisor:
Michel Medema
University of Groningen

External Supervisor:
Arend Jan Kramer
Intel Corp

Contents

Page

Acknowledgments 4
Abstract 5

1 Introduction **6**
- 1.1 Related Work 7
- 1.2 Research Questions 8
- 1.3 Milestones 8
- 1.4 Scope/Limitation 8
- 1.5 Target Group 9
- 1.6 Thesis Outline 9

2 Theoretical Background **10**
- 2.1 Cloud Computing 10
- 2.2 Kubernetes 10
- 2.3 Autoscaling 10
  - 2.3.1 Horizontal Pod Autoscaler 11
  - 2.3.2 Inflight Request Based Autoscalers 12
- 2.4 GPU Memory Utilization Metric 13
- 2.5 Metric collection using Prometheus 13

3 System Architecture **14**
- 3.1 Base Architecture 14
  - 3.1.1 Request Flow 15
- 3.2 Inflight Request Based Autoscaling System 17
- 3.3 GPU Based Autoscaling System 18

4 Approach **20**
- 4.1 Loading the systems with a constant traffic 20
- 4.2 Load system with variable traffic 20
- 4.3 Reducing the user-defined threshold of GPU Memory utilisation to observe GPU scaling 21

5 Experimental Setup **22**
5.1 Tools and Technologies 23
5.2 Evaluation Metrics 24
5.2.1 Autoscaling performance metrics proposed by SPEC Cloud Group 24
5.2.2 Inference response time 25
5.2.3 GPU Activity 25
5.2.4 Cost 25

6 Results **26**
6.1 Experiment 1 - Loading the system with constant traffic 26
6.1.1 Inference latency statistics 27
6.1.2 Scaling pattern 27
6.1.3 GPU memory utilization 28
6.1.4 Monthly Cost 31
6.2 Experiment 2 - Loading the system with variable traffic 32
6.2.1 Inference latency statistics 32
6.2.2 Scaling pattern 34
6.2.3 GPU Memory utilization 36
6.2.4 Monthly Cost 38
6.3 Experiment 3: Reducing the user-defined threshold for average GPU memory utilization 39
6.3.1 Load the system with constant traffic 39
6.3.2 Loading the system with variable traffic 43

7 Result Discussion **48**

8 Conclusion **50**
8.1 Future Work 51

Bibliography **53**

Acknowledgments

I would like to express my sincere gratitude to everyone who supported me during my graduation project. First of all, I would like to thank Dr. Alexander Lazovik for his supervision and his sharp as well as insightful comments during our progress meetings. Next, I would like to thank both my second supervisor, Michel Medama, and external supervisor, Arend Jan Kramer for their continuous guidance and support during the project. I would also like to thank Dr. Yannis Katramados for offering me the internship position at COSMONiO (now Intel). I could not have asked for a better place to work. I also want to thank all my colleagues who helped me throughout the internship and shared their knowledge.

Abstract

With the advances in Machine Learning, the deployment of Deep Learning models requiring GPUs at inference time is becoming increasingly common. GPUs are expensive resources that are often present in limited numbers as project resources. In a Kubernetes environment, where the inference services run in a serverless platform, autoscaling GPUs during inference time is a challenge. Companies often need to make informed decisions on the autoscaling approach to use while designing and implementing an inference serving system in such platforms.
When selecting an appropriate autoscaling approach, there is often a requirement to trade-off between optimizing GPU utilization and having lower inference latency. One approach to improve GPU utilization is to scale based on GPU-based utilization metrics. However, these are not easily available on all Kubernetes platforms, and configuring them with the inbuilt Kubernetes resource, Horizontal Pod Autoscalers is non-trivial. GPU utilization-based autoscaling is not served out of the box in Kubernetes platforms, unlike its CPU autoscaling counterpart which boasts of a more mature system ecosystem of off-the-shelf autoscaling controls. One approach for autoscaling GPUs, provided as an off-the-shelf solution, uses the number of inflight requests to make a scaling decision (requests which are being served and not yet responded to).
In this thesis, we design and implement a simple autoscaling system that scales the GPUs based on the average GPU memory utilization. We compare both these approaches by studying their behavior in response to different environmental conditions that incrementally simulate real-world characteristics. These simulations model systems that are pounded by inference requests at a constant rate and another in which the systems are loaded with variable traffic. Through experiments, we show that the request-based autoscaling approach is better suited for use cases where the focus is on providing lower inference latency rather than better GPU utilization. In contrast, the GPU utilization-based autoscaling approach provides a more conservative way to utilize GPUs, generally leaving GPUs available for other use but at the cost of providing slow inference response times.

CHAPTER 1

Introduction

Machine learning models which use GPUs to accelerate performance are becoming increasingly common.[Kaiser(2019b)] For many use cases, GPU-based inference is the only way to achieve acceptable latency [Kaiser(2019a)]. GPUs are expensive resources and as a result, are often present in limited capacities as project resources.
Knowledge of GPU resource pool utilization and their efficient management at runtime is of vital importance for the following reasons:

- For some business cases, making a wrong decision on the number of GPUs required can severely impact the business in financial costs.
- Given the limited availability of GPU resources, better management of GPU resources provides flexibility in how to utilize and share them between computing tasks and projects.
- Monitoring the health of deployed systems and placing automatic alerts or controls on scaling operations in case of unexpected behavior.

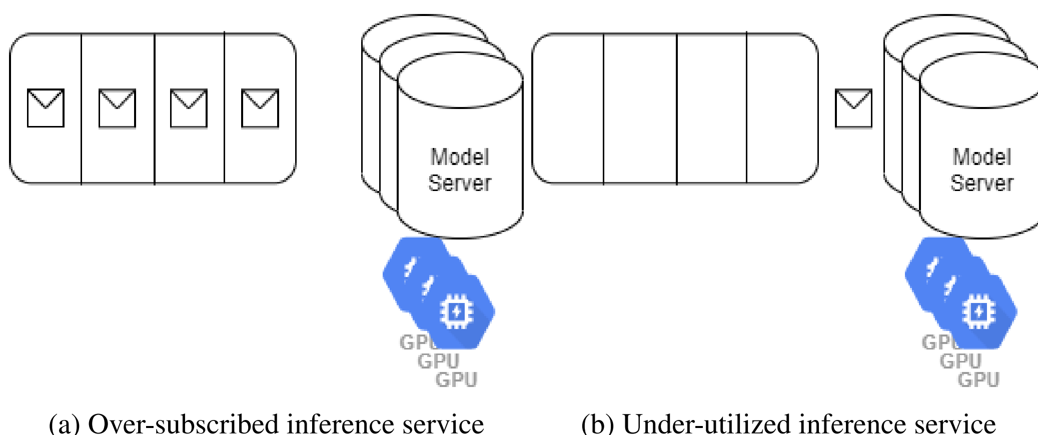
In real-world settings, there is often no guarantee on how 'GPU hungry' the incoming requests are or the rate at which incoming traffic will flow in. It is not uncommon to have inference requests which use up 100% of the GPU instance.[Kaiser(2019b)] On the other hand, it would be wasteful to spawn more GPU instances while underutilizing those already present to meet demand quicker, without proper considerations in place. This hints at a requirement to dynamically scale the number of GPU resources to meet the incoming demand. Circumventing the need for manual intervention, automatic scaling solutions help businesses:

- to provide the minimal operational GPU resources required during deployment
- to scale out the number of GPU resources in the event of an increase in the GPU usage demands
- to scale down the number of GPU resources to maintain the minimal operational limit when no usage demand is required
- to automate this scaling process 1-3 without human operator intervention

Most off-the-shelf solutions, such as Seldon Core[Core(2021)], Cortex[Cortex(2021)] as well as KF-Serving [KFServing(2021)] provide autoscaling solutions that focus on scaling the inference workload based on the number of inflight requests (requests which are being served and not yet responded to). There are no ready-made solutions that provide autoscaling based on the GPU utilization metrics. This highlights the need for autoscaling based on GPU utilization metrics and simultaneously builds a case for it to become a more readily available solution.
This thesis investigates the need by providing a comparative study on both autoscaling approaches. To do this, the project proposes the design as well as an implementation of a simple autoscaler that scales on the basis of the rolling average of the GPU memory utilization metric. We discover that choosing an appropriate autoscaling approach often involves a trade-off between efficient GPU utilization and lower inference latency.

1.1 Related Work

Autoscaling for CPU-based clusters has been studied in depth such as in [Al-Haidari et al.(2013)] and [Casalicchio(2019)]. Similarly, while [bol(2021)] presents the learnings and significant business impact from a real-world autoscaling solution deployed for CPUs, it still provides relevant pointers that can inspire the GPU-autoscaling design paradigm.
In the report [Cox et al.(2020)], the authors present three possible approaches to autoscale GPU-based inference systems in Kubernetes. The approaches discussed are based on GPU utilization, inference latency, and the number of inflight requests. According to the report, autoscaling based on GPU utilization is difficult as not all Kubernetes platforms provide GPU duty cycle metrics. In some use cases, it may be required to combine CPU utilization with GPU utilization to produce a complex autoscaling metric to make a scaling decision. Reconciling such a metric is not a trivial task.
According to the article, [Kaiser(2019b)], the inference latency-based autoscaling approach makes use of a user-defined latency target as its trigger. However, the approach is not a favorable one, since one would need to know the target latency before launching the inference service. This is risky since if an updated model having a longer inference latency compared to the previous version is deployed, it could trigger infinite scaling. Furthermore, inference latency is a poor indicator for deciding to scale down. The reason behind this is that the inference latency of a request only changes when requests start waiting in queues. Figure 1.1a depicts a situation wherein the inference service replicas are oversubscribed since the requests start queuing up. As a consequence, the latency per request would increase. This would result in the system deciding to scale up.



(a) Over-subscribed inference service (b) Under-utilized inference service

Figure 1.1: Scaling behavior for system A

When the inference service replicas receive requests at a rate slower than the system can handle, there will be no change in the latency as there are no queued requests. This can be seen in figure 1.1b. If the inference latency does not reduce as the intensity of the demand reduces, it cannot be trusted as a good signal for scaling down.
The final approach discussed deals with scaling the number of GPU resources based on the number of inflight requests in the system. As mentioned previously, most off-the-shelf GPU autoscaling approaches [Core(2021)] [Cortex(2021)] [KFServing(2021)] embrace this approach. [KFServing(2021)] provides a scale to zero GPU autoscaling solution based on this approach.

1.2 Research Questions

The main research goal is to investigate the GPU utilization-based autoscaling approach by drawing a comparison to the inflight requests-based autoscaling approach. To support this objective, the research question this thesis targets to answer is:

- How to best perform GPU resource autoscaling in ML inference services and recommend between the autoscaling approaches in the context of efficient GPU utilization and reduced inference latency?

The question is broken down into several smaller research problems:

Q1. How to build an autoscaling component that leverages GPU utilization metrics to make an autoscaling decision?

Q2. How does the inflight requests-based autoscaling system compare with the GPU utilization-based autoscaling system when the systems are loaded with constant traffic?

Q3. How do the two approaches compare when the systems are loaded with variable traffic?

1.3 Milestones

M1	Investigating GPU based autoscaling approaches in Kubernetes
M2	Design and implement a system that scales GPUs on the basis of average GPU utilization.
M3	Setup the environment with a deployed ML inference service and run experiments
M4	Compare and evaluate the performance of both the systems for each experiment.

Table 1.1: Project Milestones

1.4 Scope/Limitation

To build towards its objective of comparing the different GPU autoscaling approaches, this study makes a few working assumptions, which can be relaxed in subsequent investigations. To conduct the experiments, a complex system of several interacting components and services had to be implemented (as described in Chapter 3). In absence of off-the-shelf solutions, a simple GPU-utilisation-based autoscaling system was implemented which is restricted to the use of Nvidia GPUs (More information on the same is available in the appendix). Second, the maximum number of GPUs available to either

system was provisioned at six, and this number was always made available to the system. This is equivalent to reserving a cluster of six online GPUs and could result in overestimating the cost of the cluster setup. However, in the real world, one can dynamically request cloud compute instances and provision or release instances in real-time which would further reduce costs.

Third, it is non-trivial to assess and compare the efficiency of the two systems in real-time environments, particularly as this would require access to real-time production environments and their incoming requests. Hence, the project attempts to simulate these environments. Further, since there are a large number of variations to real-time environments to be considered, only a limited number of environments are simulated.

Fourth, in terms of concurrent processing, running multiple models on the same GPU resource can help boost GPU utilization. However, since most ML model servers do not readily provide this functionality, it is out of the scope of this thesis. In the same vein, parallel processing of multiple requests for a single ML-model system on the same GPU resource has been left to future work owing to the intertwined complexities of multiprocessing and Deep Learning (DL) algorithm implementations. For instance, DL libraries such as TensorFlow will process subsequent requests sequentially since TensorFlow uses a global single compute stream for each physical GPU device [1]. The current study assumes that the threshold concurrency limit for each individual model server is 1. Note, the study however does implement a certain degree of multiprocessing by sending requests to multiple GPU instances.

1.5 Target Group

This thesis targets data scientists, software, and DevOps engineers working with ML inference serving systems to fulfill and sustain service level agreements (SLA) for quality attributes such as availability and performance. The thesis could also be used in the project planning phase to make an informed decision on the type of autoscaling approach to use while designing the inference serving system.

1.6 Thesis Outline

The thesis is organized into sections as follows. Chapter 2 introduces the reader to the technical background on critical concepts used, while Chapter 3 provides a detailed description of the system architecture set up for the study. The experimental approach, setup, as well as evaluation metrics, are described in Chapters 4 and 5. Chapter 6 documents the results and comparisons between the different setups, while Chapter 7 presents main result inferences and Chapter 8 concludes.

[1] https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/common$_runtime/gpu/gpu_device.ccL$284

CHAPTER 2

Theoretical Background

The study involves an elaborate technical ecosystem. The background behind certain concepts and algorithms is described here.

2.1 Cloud Computing

This thesis uses Google Cloud Platform [1] to deploy the inference serving systems and compare their performance in different environments. Since the systems rely on the cloud it is important to understand what cloud computing is. NIST defines Cloud Computing as a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. [Mell et al.(2011)]

2.2 Kubernetes

Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications [Kubernetes(2021)]. It was inspired by the Borg cluster management system from Google[Google(2015)] and later donated to the Cloud Native Foundation. In the past decade, Kubernetes has seen a growth in the number of users. Several enterprises now use Kubernetes to deploy their applications in private data centers, public clouds as well as hybrid cloud environments.

2.3 Autoscaling

Autoscaling is the ability of a system to dynamically scale the number of instances. This means that if the system experiences a surge in the incoming traffic, to maintain the desired level of availability of the service provided by the system, the number of instances required to serve the traffic should be increased as well. Likewise, the system should reduce the number of instances in case the load decreases. Thus by automating the scaling decisions according to pre-determined objectives in quick

[1] https://cloud.google.com/

response to changing conditions, the system can avoid wasting resources. In the context of this thesis, instances refer to GPU resources, and the process of scaling means to increase or decrease the number of GPU resources.
Horizontal scaling refers to the process of adding or removing instances. The incoming traffic is distributed uniformly among the instances currently available, this is usually handled by a load balancer. Vertical scaling means increasing the capacity of the currently running instance. Vertical scaling is generally associated with a certain degree of downtime while upgrading the resource, therefore horizontal scaling is preferred when dealing with environments that consist of fluctuating load.
Both the autoscaling approaches discussed in this thesis are threshold-based autoscaling approaches. Threshold autoscaling is a reactive approach where the system monitors the values of metrics and makes a decision to scale out in case the values exceed the use of user-defined thresholds.

2.3.1 Horizontal Pod Autoscaler

According to the documentation mentioned in [Kubernetes(2021)], A Horizontal Pod Autoscaler or HPA is a Kubernetes resource which dynamically scales the number of pod replicas in a deployment based on the observed value of a single or multiple metrics specified in the HPA definition. The following is an example of a HPA definition.

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: hpa-demo
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: target-deployment
  minReplicas: 1
  maxReplicas: 6
  targetCPUUtilizationPercentage: 80
```

Since the GPU utilization-based autoscaling approach makes use of the HPA to scale the number of GPU resources it is important to understand the underlying algorithm of a HPA.
The following describes the HPA Autoscaling algorithm as mentioned in [Kubernetes(2021)]:

1. The HPA is implemented as a **Control Loop** with the default duration of every period set as 30 seconds.

2. The HPA periodically queries the targeted set of pods to collect the value of the metric specified in the HPA definition.

3. It compares the collected value with the target value specified in the definition.

4. Scaling of the number of replicas take place with the following conditions:

    - MinReplicas $\leq$ Replicas $\leq$ MaxReplicas
    - desiredReplicas = ceil[currentReplicas * (currentMetricValue / desiredMetricValue)]

5. The HPA takes five minutes before deciding to scale down the number of replicas (in case a scaling operation just took place), this is known as the cool-down period. The purpose of the cool-down period is to introduce a degree of robustness to the system against frequent conflicting decisions.

6. There is only a single policy for scaling down which allows 100% of the currently running replicas to be removed which means the scaling target can be scaled down to the minimum allowed replicas.

7. For scaling up there is no cool-down period. When the metrics indicate that the target should be scaled up the target is scaled up immediately.

8. Any scaling will be skipped if the ratio, CurrentMetricValue/DesiredMetricValue is sufficiently close to 1 (10% tolerance).

[Kubernetes(2021)] states that approach taken by the HPA has two benefits:

- The autoscaling algorithm for the HPA works conservatively. It rapidly scales up the number of pods when user load is detected. However, scaling down the number of pods is not treated as urgent.

- The HPA algorithm avoids thrashing, i.e. it prevents rapid execution of conflicting decisions if the load is not stable.

2.3.2 Inflight Request Based Autoscalers

The autoscaling approach used by this autoscaler is to scale the number of instances by counting the number of inflight requests. Inflight requests refer to the requests which are currently in the system and have not been processed yet. The autoscaler can scale the number of inference service replicas and hence GPU resources to closely match the incoming demand. It is even able to scale to zero in case there is no demand.
For this thesis, the Knative Pod Autoscaler which uses the inflight requests-based autoscaling approach is used. The following is an example of the definition of a service using the inflight requests-based autoscaler.

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: demo-service
  namespace: default
spec:
  template:
    metadata:
      annotations:
        # Target 10 in-flight-requests per pod.
        autoscaling.knative.dev/target: "10"
        autoscaling.knative.dev/minScale: "2"
        autoscaling.knative.dev/maxScale: "6"
    spec:
      containers:
      - image: gcr.io/knative-samples/autoscale-go:0.1
```

In the definition provided above, a service called 'demo-service' is defined. The configurations for the autoscaler are present as annotations. The number of inflight requests the service can handle, beyond which the autoscaler would deploy a new instance of the service is 10. The minimum number of instances of the service is 2 and the maximum possible instances the autoscaler can scale up to is 6.
The default behavior of the autoscaler is to scale to zero if the minimum scale is not defined and in theory can scale out the number of instances to infinity if maximum scale is not defined.

2.4 GPU Memory Utilization Metric

The GPU utilization based autoscaling component makes it's scaling decison based on the rolling average of the GPU memory utilization metric. This metric represents the percentage of time over the last second that the GPU's memory controller was being utilized to either read or write from memory.

2.5 Metric collection using Prometheus

Prometheus is an open source monitoring system [2] which collects metrics and stores them in a time-series database. Prometheus relies on the *pull model* to collect the metrics. The process of collecting metrics is called *scraping*. The instrumented services expose their metrics in a format understandable by Prometheus.
A potential challenge with this model of metric collection is that Prometheus needs to know where to collect the metrics from. Statically configuring the endpoint is not an option since services in a micro-service architecture are generally ephemeral, hence there is a need to dynamically configure the location of the services. Prometheus servers have inbuilt solutions for service discovery (service monitors) which helps them overcome this challenge.

[2] https://prometheus.io/

CHAPTER 3

System Architecture

This chapter describes the architecture of the composite systems studied.

3.1 Base Architecture

Both inflight requests based as well as GPU utilization based autoscaling systems are built upon on a common base architecure. Figure 3.1 represents the base architecure for both the systems.

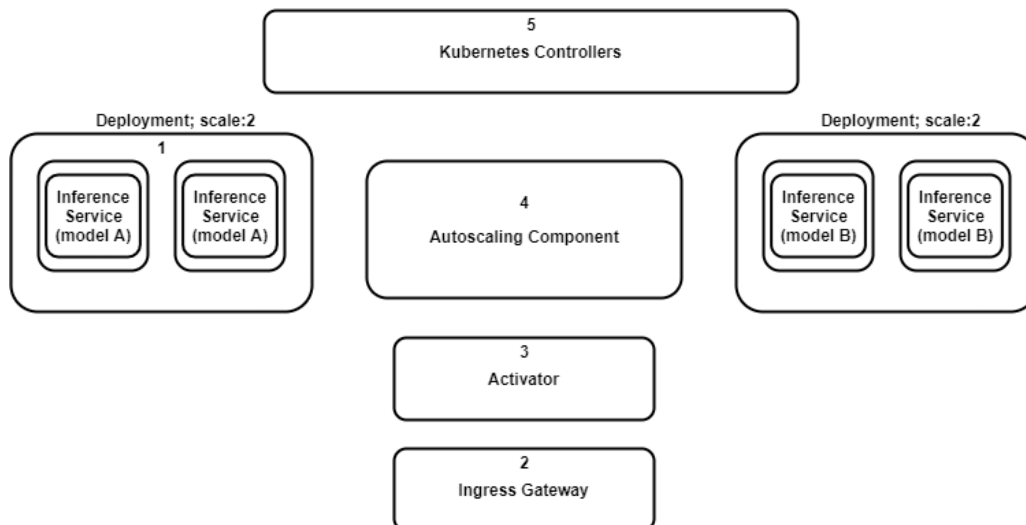


Figure 3.1: Base Architecture

The components of the base architecture are:

1. **Inference Service Deployment**
   This is a kubernetes *deployment* of pods which contain containers running inference services. Each inference service pod required exactly one GPU resource.

2. **Ingress Gateway**
   This is a gateway for routing requests to the appropriate inference service. It acts as a load balancer sitting at the edge of the system. Requests sent by external or internal applications to the ingress gateway are either routed to the activator (in case of insufficient inference service pods in the inference service deployment) or directly to the pod running the inference service.

3. **Activator**
   The Activator is a critical component of the architecture. The following are the purposes the activator serves:

   (a) It buffers the incoming requests, in case there are an insufficient number of inference service pods and it sends metrics to the autoscaler.

   (b) As inference service pods become available, it balances the load efficiently among the newly created inference service pods. This prevents overloading of any pod. Essentially protecting the system from bursts so that we do not see requests queuing in the pods.

4. **Autoscaler**
   This is the main component of study and is different for both the systems in terms of the approach it takes to scale the number of inference service pods, and hence GPU resources in both the systems.

5. **Kubernetes controllers**:
   This includes all the Kubernetes controllers responsible for deploying the inference service pods as well as other relevant resources.

3.1.1 Request Flow

In this section the general flow of an inference request in the inference serving system is discussed. Figure 3.2 depicts the various stages in the flow of the inference request.

Figure 3.2: Request flow

The various stages are described below:

- **Stage 1**
  The client sends an inference request to the ingress gateway.
- **Stage 2**
  On reaching the ingress gateway, if there are no inference services available to handle the incoming request, the request is forwarded to the activator, where it is stored in a buffer.
- **Stage 3**
  The activator then 'pokes' the autoscaler in order to trigger the deployment of the required number of inference service replicas.
- **Stage 4**
  The autoscaler communicates with the kubernetes controllers in order to scale the number of replicas of the inference service in order to meet the incoming demand.
- **Stage 5**
  The kubernetes controllers increase the scale of the inference service deployment, deploying new replicas.
- **Stage 6**
  The activator sends the stored request to the newly deployed inference service replica.
- **Stage 7**
  Once the inference service processes the request, it sends the response to the ingress gateway, which sends it back to the client.
- **Stage 8**
  The next request sent to the ingress gateway is directly sent to the inference service replica, provided it is available. The process repeats from stage 2 if the inference service is not available.

3.2 Inflight Request Based Autoscaling System

As mentioned before, the inflight requests based autoscaling system builds upon the base architecture (Figure 3.1). Figure 3.3 depicts the architecture of the system which uses an autoscaler that scales on the basis of the number of inflight requests.

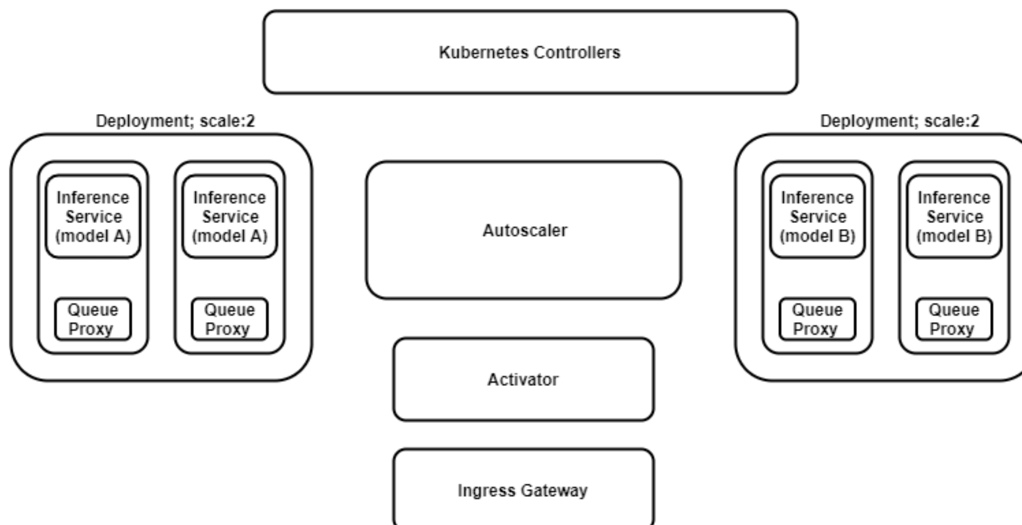


Figure 3.3: Inference serving system which scales on the basis of the number of inflight requests

The components which this system adds/ makes changes to the base architecture are:

1. **Inference service deployment**
   The inference service deployment now has a configurable concurrency threshold, which denotes the number of inflight requests the inference service can handle. As mentioned in section 1.4 the limit is set to 1.

2. **Queue Proxy**:
   Every inference service pod deployed has a corresponding queue proxy. The purpose of the queue proxy is:

   (a) To ensure the configured concurrency for the inference service.

   (b) Emit metrics regarding the number of inflight requests to the autoscaler.

3. **Inflight requests based autoscaling component**
   This component is responsible for autoscaling the number of replicas of the inference service in order to closely match the incoming demand. This includes scaling the number of replicas to zero in case there is zero incoming traffic. The autoscaler monitors the incoming requests to each replica and makes a decision to scale the number of replicas based on the configured concurrency for the revision.

3.3 GPU Based Autoscaling System

The architecture of the GPU based autoscaling system is represented by figure 3.4.

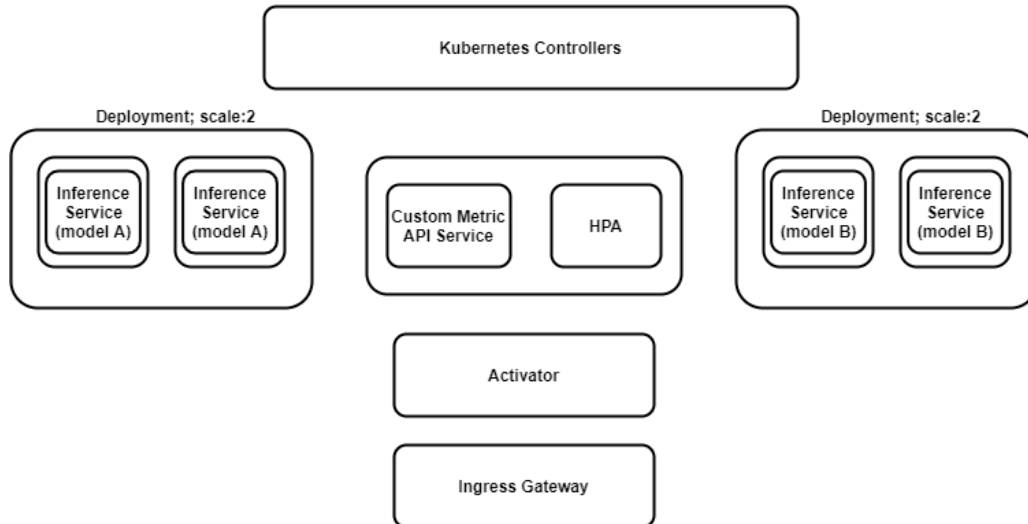


Figure 3.4: Inference server system using autoscaler based on GPU-utilzation

The architecture deviates from the previous system with respect to the working principal of the autoscaling component as well the absence of the queue proxy.
The inference services no longer have a configurable concurrency threshold but instead have a configurable average GPU memory utilization threshold. The system uses the native Kubernetes resource, called the Horizontal Pod Autoscaler(HPA) [1] as the autoscaling component. The HPA scales the number of replicas of the inference service based on the average GPU memory utilization metric. If the observed value crosses the user-defined threshold the system uses the HPA algorithm to scale the number of GPU resources.
Since GPU metrics are not easily available in every Kubernetes platforms, a general approach to extract GPU metrics and export them as custom Kubernetes API is required. Figure 3.5 depicts a simple approach to compute the GPU memory utilization of the GPU resources being used and export the rolling average of this GPU memory utilization value (over a window of 1 minute) as a custom metric to be used by the HPA. The solution relies on the Data Center GPU Manager(DCGM[2]) exporters to pull and make available GPU related metrics. These metrics are then scraped by the Prometheus server, where they are stored in a time series database. To compute the GPU memory utilization value of only the GPU resources being utilized at that instant, a component named 'custom exporter' is used. The component contains a python script running in a container. The script computes the GPU memory utilization of the GPU resources being utilized and makes them available for the prometheus server to scrape and store. Once the new metric is stored in the prometheus server, a prometheus adapter uses PromQL[3] to compute the rolling average of the GPU memory utilization and exposes it as part of the custom metrics API endpoint. The HPA then receives the average GPU memory utilization value by accessing this endpoint.

[1] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
[2] https://docs.nvidia.com/datacenter/dcgm/dcgm-user-guide/index.html
[3] https://prometheus.io/docs/prometheus/latest/querying/basics/

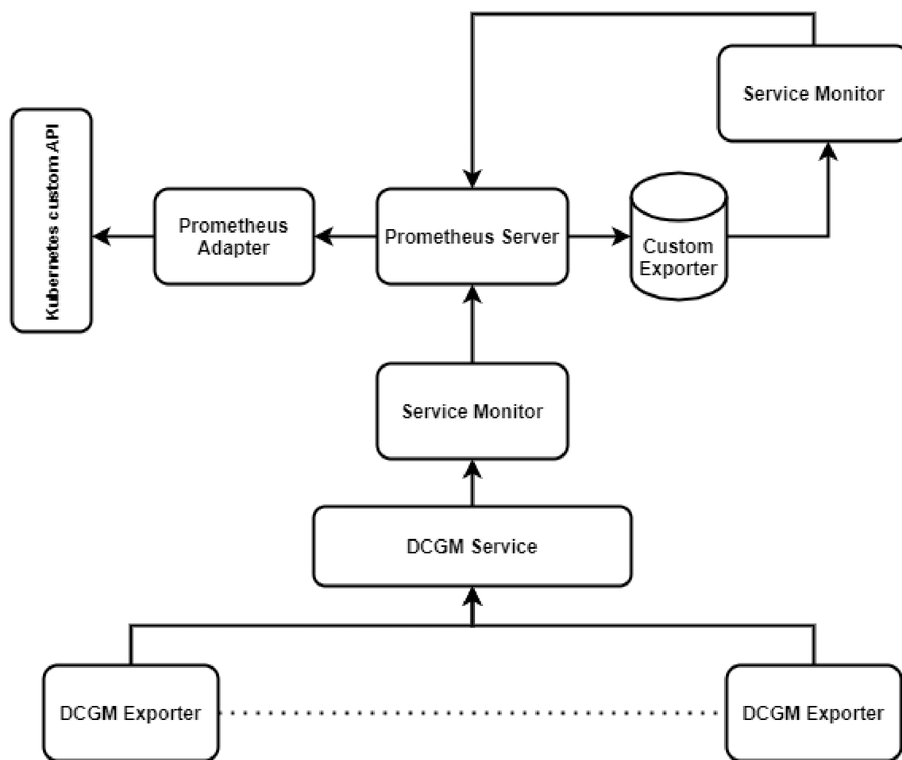


Figure 3.5: Exposing custom GPU utilization metrics

CHAPTER 4

Approach

Both the systems, discussed in the previous section, will be subjected to a set of experiments to evaluate their performance in different environments.

The following is the purpose for every experiment conducted:

1. Monitoring and deriving the relation between the ML inference service system's ability to serve a certain demand load, and the quantity cum duration of each GPU resource's utilisation.
2. Evaluate the system's decision to scale in order to meet the demand.
3. Analyze the latency statistics for both the systems.

The following are the experiments performed on both the systems:

4.1 Loading the systems with a constant traffic

The objective of running this experiment is to test the ability of the system to serve a fixed demand. A business use case which would generate such a load is offline batch processing - images are sent to the inference system in batches of fixed sizes. The rate at which the requests are sent to the system is more or less constant.
This experiment has been designed to mimic such a scenario, where in a large volume of images (15000 images) are sent to the inference serving system at a constant rate in batches of 6. For the GPU utilization based autoscaling system, the threshold for the average memory utilization has been set to 30 %.

4.2 Load system with variable traffic

The objective of running this experiment is to evaluate the ability of the system to make a scaling decision in order to meet the periodically fluctuating demand. The experiment attempts to mimic an environment, where the intensity of the incoming inference requests to the service varies with

each interval. In doing so, we can test the response of the system to upscaling and downscaling, and whether either system is more resilient and resource efficient to dramatic fluctuations.
The experiment is divided into four phases. A total of 48000 requests are sent to each system in four phases with varying rates. In the first phase, 16200 requests are sent with a frequency of 90 requests per second. In the second phase, 9600 requests are sent with a frequency of 20 requests per second. The third phase sees a rise in the request frequency with 16200 requests sent at 90 requests per second. In the final phase, 6000 requests are sent with a frequency of 10 request per second.
For the GPU utilization based autoscaling system, the threshold for the average memory utilization has been set to 30 %.

4.3 Reducing the user-defined threshold of GPU Memory utilisation to observe GPU scaling

The objective of this experiment is to evaluate the performance of the GPU utilization based autoscaling approach in an environment where the average GPU memory utilized by the GPU resources serving the inference load, frequently crosses the user-defined threshold. However, there is some rigidity in GPU memory utilisation imparted by the ML algorithm's design, and thus simply increasing the batch size of incoming requests without re-designing the computational efficiency (introducing parallelization) of the algorithm will not raise memory usage. As redesigning or improving parallelisation of the ML model's inference ability to raise GPU memory utilisation of the system is out of scope of this study, we instead simulate the scenario of over-utilisation by lowering the trigger threshold of the GPU Memory utilization below the observed memory utilization
To evaluate the performance, both Experiment 1 and 2 are performed on the system with the newly set threshold (6%).

CHAPTER 5

Experimental Setup

In this section we discuss the architecture of the setup required to conduct the experiments. Figure 5.1 depicts the underlying architecture for the setup.

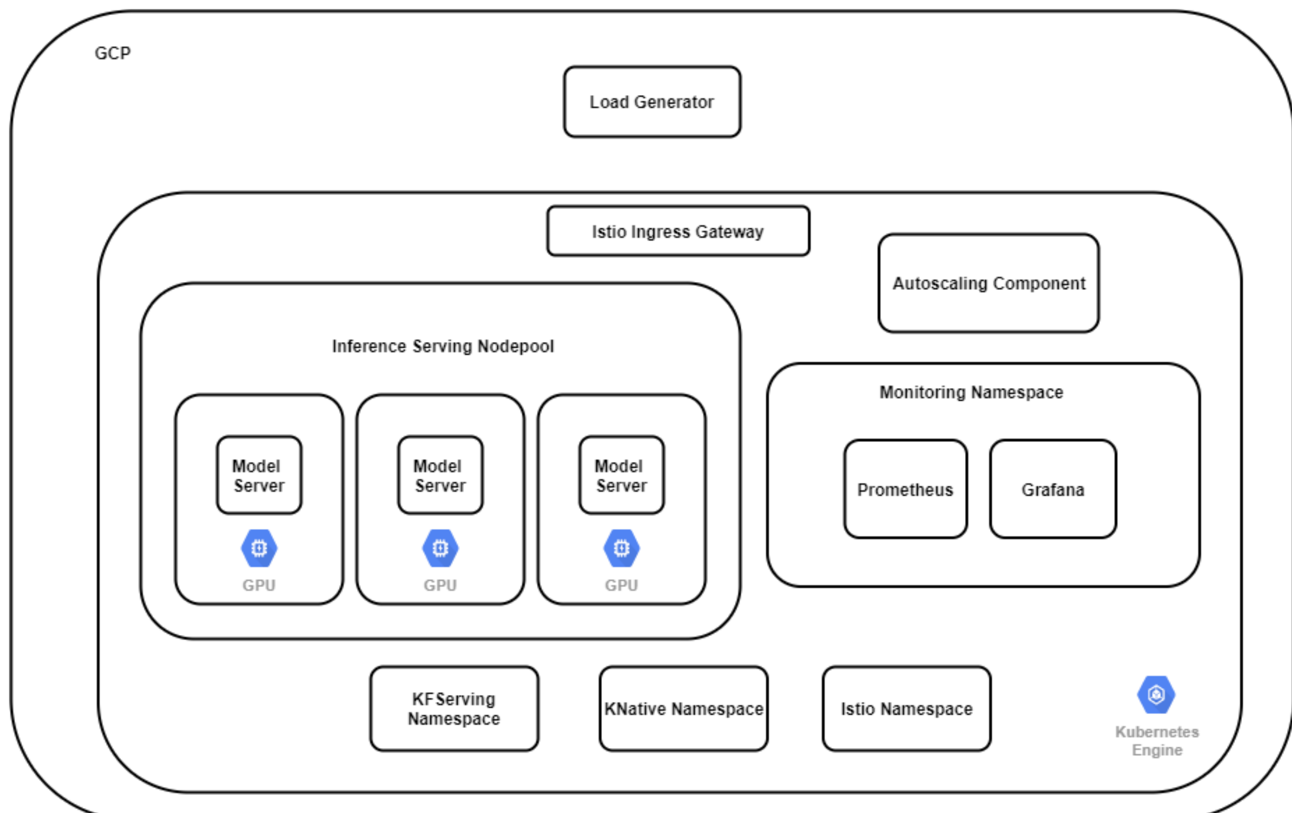


Figure 5.1: Experimental Setup

The following is a description of the main components of the experiment setup. 5.1.

1. **Load generator**: Generates and sends inference requests to the gateway.

2. **Kubernetes Cluster**: The cluster consists of six nodes. Each node has a single GPU (Nvidia Tesla P100) and is a N1-standard-8 machine(8 vCPUs, 30GB memory).

3. **Gateway**: This is the entry point of the serving system. The gateway is responsible for routing the requests to the appropriate service.

4. **Autoscaler**: This is the main component under study. The goal of the autoscaler is to make an autoscaling decision either based on GPU memory utilization or the number of inflight requests.

5. **Model server**: The model server hosts the trained inference model and exposes the predict logic via a GRPC / HTTP endpoint on the server. **A replica of the model server is always running to avoid the initial start-up delay.**

6. **GPU telemetry**: To monitor the utilization metrics of the GPUs in the cluster.

7. **Trained models**: The system uses pre-trained ML models. For the purpose of this study, we use a simple publicly available multi-class classification ML model [1], pretrained on a public dataset [2], with the objective to classify input images as different types of flowers.

5.1 Tools and Technologies

The following are the list of tools and technology used:

1. **Kubernetes:**
   Kubernetes provides the platform required to deploy and scale the serverless-inferencing systems. It is an open-source for automating deployment, scaling and management of containerized applications [Kubernetes(2021)].

2. **Google Kuberentes Engine:**
   Google Kuberentes Engine (GKE) provides a configurable Kubernetes cluster. The cluster is managed by Google using Google infrastructure. [3]

3. **Tensorflow:**
   An open source ML platform which provides the libraries for building and training ML models. It also provides libraries to serve the trained model as an inference service.

4. **KFServing:**
   KFServing provides a framework to serve inference services in Kubernetes providing features such as GPU Autoscaling, scale to zero, and canary rollouts to ML deployments. [KFServing(2021)]

5. **Istio:**
   KFserving utilizes the Istio [4] Ingress Gateway resource provided by Istio. The gateway helps route requests to the inference services.

6. **Knative Serving:**
   The study adopts the architecture set up by Knative serving. Knative Serving provides a platform to deploy serverless applications over kubernetes.

[1] https://www.tensorflow.org/hub/tutorials/image_feature_vector
[2] https://www.tensorflow.org/datasets/catalog/tf_flowers
[3] https://cloud.google.com/kubernetes-engine
[4] https://istio.io/

7. **Cert Manager:**
   Cert manager is used for provisioning the certificates for the KFServing webhook server.

8. **Hey Load Generator:**
   This is an open-source load testing tool [5] used to simulate an environment with multiple users, sending inference requests at set intevals.

9. **Data Center GPU Manager(DCGM):** For GPU telemetry, Nvidia's DCGM [6]) is used. It is a tool used to monitor the utilization of the GPUs in a cluster.

10. **Prometheus:** Prometheus is used to record the GPU metrics exported by a DCGM exporter and store it in a time-series database using a HTTP model.[7]

11. **Grafana:** Grafana [8] is used as the data visualization platform to analyse the GPU utilization as well as other collected metrics.

5.2 Evaluation Metrics

To evaluate the performance of both the autoscaling systems, the following autosacling performance metrics defined by SPEC [Herbst et al.(2016)] are used:

5.2.1 Autoscaling performance metrics proposed by SPEC Cloud Group

These metrics quantify the autoscaling capabilities of the two autoscaling approaches and help the developer community to select the appropriate strategy for their workload. The metrics include:

- **Provisioning accuracy metrics**
  Represented as θ_U and θ_O, describe the relative amount of under-provisioned or over-provisioned GPU resources, respectively, during the measurement interval.

$$\theta_U[\%] = \frac{100}{T} \sum_{t=1}^{T} \frac{max(d_t - s_t, 0)}{d_t} \Delta t$$

$$\theta_O[\%] = \frac{100}{T} \sum_{t=1}^{T} \frac{max(s_t - d_t, 0)}{d_t} \Delta t$$

- **Wrong-provisioning timeshare metrics**
  Represented as τ_U and τ_O, measure the time in which the autoscaler underprovisions or over-provisions, respectively, during the time of the experiment.

$$\tau_U[\%] = \frac{100}{T} \sum_{t=1}^{T} max(sgn(d_t - s_t, 0))\Delta t$$

$$\tau_O[\%] = \frac{100}{T} \sum_{t=1}^{T} max(sgn(s_t - d_t, 0))\Delta t$$

[5] https://github.com/rakyll/hey
[6] https://docs.nvidia.com/datacenter/dcgm/dcgm-user-guide/index.html
[7] https://prometheus.io/
[8] https://grafana.com/

- **Overall provisioning accuracy metric**

$$\theta = \frac{\theta_U + \theta_O}{2}$$

- **Overall wrong provisioning timeshare metric**

$$\tau = \frac{\tau_U + \tau_O}{2}$$

Here we define:

1. T as the duration of the experiment and the current time as $t \in T$
2. d_t represents the number of GPU resources demanded by the autoscaler and and s_t denotes the number of GPU resources supplied by the system at time t.
3. Δt denotes the time interval between the last and the current change in either demand (d) or supply (s).
4. *sgn* is the signum function.

Lower values for these metrics indicate a better autoscaling strategy.

5.2.2 Inference response time

The inference latency statistics for both the autoscaling systems are compared for both the experiments. Lower inference latency indicates a faster autoscaling approach.

5.2.3 GPU Activity

This metric evaluates the amount of time the GPU resource is active for over the duration of the experiment. A higher value for the same indicates that the GPU resource was highly active during the experiment while a lower value could mean that the GPU resource was sitting idle for longer periods of time during the experiment.

$$\text{GPU Activity \%} = \frac{\text{Amount of time GPU is active}}{\text{Duration of the Experiment}} * 100$$

5.2.4 Cost

The autoscaling systems are compared based on the monthly costs generated for running the clusters. The costs are calculated based on the Google Cloud Platform(GCP) billing policy for GPU resources [Google(2021)]. The breakdown includes a fixed provisioning cost per GPU instance, followed by a variable cost proportionate to the number of seconds a GPU is run for in the experiment. The Google Cloud Pricing Calculator[9] has been used to generate the monthly pricing estimates. To arrive at these estimates, it has been assumed that the experiment is run once a day for 5 days a week.

[9] https://cloud.google.com/products/calculator/

CHAPTER 6

Results

This chapter provides quantitative experiment results to evaluate the performance of both the systems when placed in different environments. As discussed in experiment outline in chapter 4, the systems are first loaded with constant request traffic, following which the systems are made to experience variable traffic. In the final experiment the memory utilization threshold of the GPU utilization based autoscaling system is reduced to a value below the observed average memory utilization and both the preceding experiments are performed on this system. We use the following terminology to demarcate the different autoscaling systems studied:(1) *System A*: The inflight requests based austoscaling system, and (2) *System B*: GPU utilization metric based autoscaling system

6.1 Experiment 1 - Loading the system with constant traffic

Both the systems are loaded with 15000 requests at a steady rate. The following is a graph depicting the total number of inference requests sent by the load generator measured every 15 seconds.

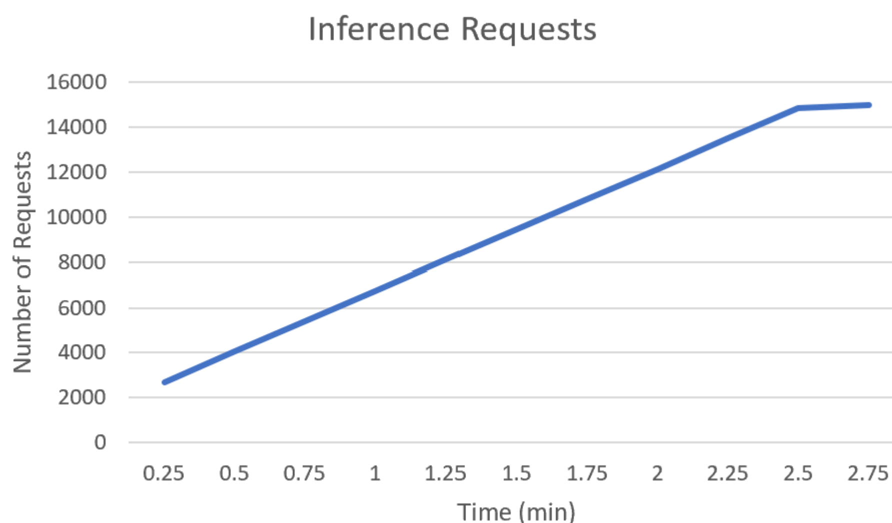


Figure 6.1: Inference Requests sent by the load generator

6.1.1 Inference latency statistics

The following tables represent the latency distribution (Table 6.1) as well as general statistics on the observed response times (Table 6.2).

System	10%	25%	50%	75%	90%	95%	99%
A	0.0528s	0.0756s	0.0961s	0.0987s	0.0997s	0.1003s	0.1063s
B	0.4007s	0.4915s	0.5003s	0.5638s	0.5997s	0.6008s	0.6945s

Table 6.1: Latency distribution

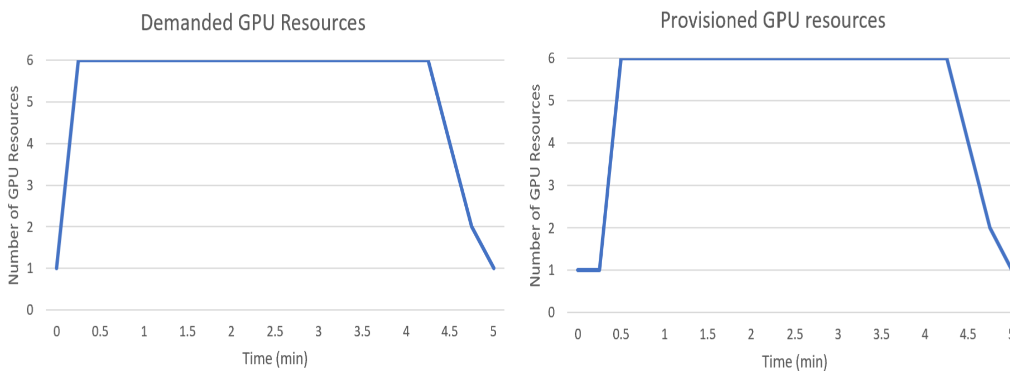
System	Total	Slowest	Fasted	Average	Throughput
A	227.0306s	2.3452s	0.0491s	0.0902s	66.0704
B	1275.8354s	3.4288s	0.0510s	0.5098s	11.7570

Table 6.2: Response Statistics

For system B, requests start queuing up in the activator as the system only uses a single GPU throughout the experiment. As a consequence, higher values for the slowest as well as average response times are observed compared to that of system A.
The total inference response time for system A is roughly six times faster than that of system B.

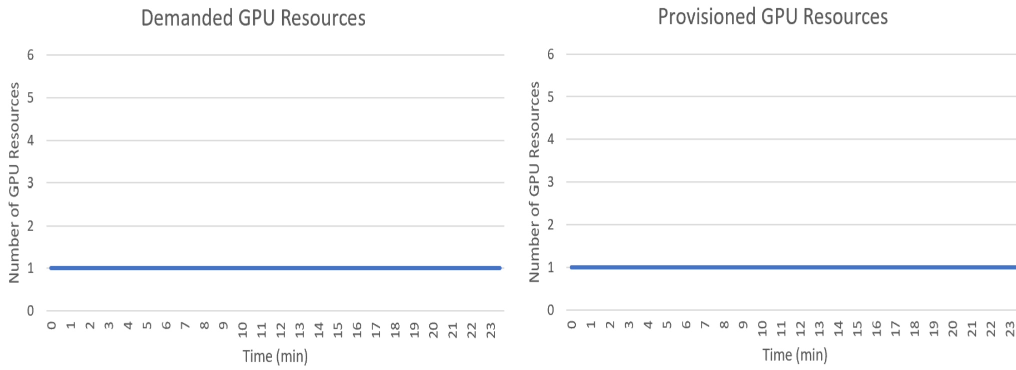
6.1.2 Scaling pattern

This section evaluates the GPU scaling behaviour of both the systems as observed during the experiment. Figures 6.2 and 6.3 show the GPU resources demanded and provisioned by both the systems.



(a) GPU Resources Demanded vs Time (b) GPU Resources Provisioned vs Time

Figure 6.2: Scaling behavior for system A



(a) GPU Resources Demanded vs Time (b) GPU Resources Provisioned vs Time

Figure 6.3: Scaling behavior for system B

In Table 6.3 the calculated autoscaling metrics for both the systems have been shown.

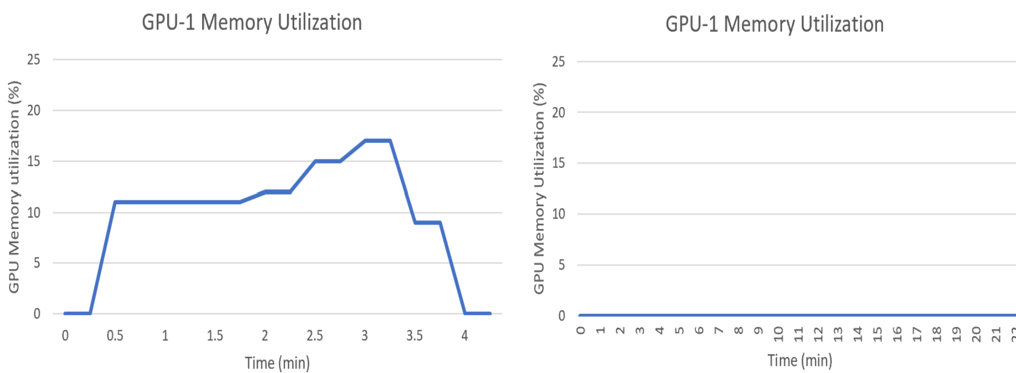
System	θ_U	θ_O	τ_U	τ_O	θ	τ
A	5.5%	0%	6.6%	0%	2.75%	3.30%
B	0%	0%	0%	0%	0%	0%

Table 6.3: Autoscaling Metrics

Since system B demands a single GPU throughout the experiment, it neither over-provides nor under-provides GPU resources at any point. On the contrary, system A reacts to the high volume of incoming requests and scales out to 6 GPU resources. In the process of scaling, it undergoes the cost over-provisioning of GPU resources.

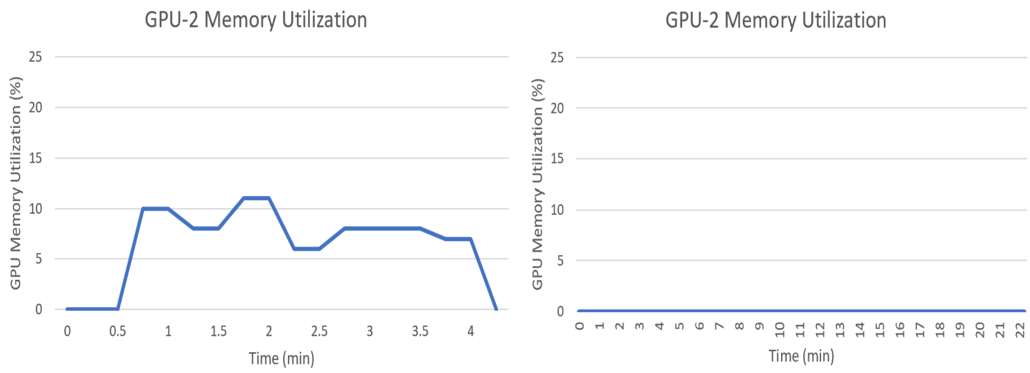
6.1.3 GPU memory utilization

The following graphs depict the GPU Memory utilization values of all six GPUs in the cluster over the duration of the experiment.



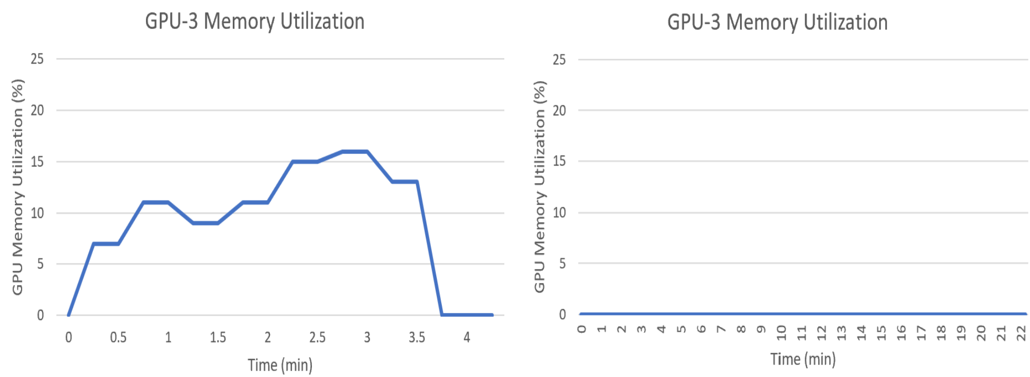
(a) GPU utilization for GPU 1 in system A (b) GPU utilization for GPU 1 in system B

Figure 6.4: GPU memory utilization of GPU 1 during experiment 1



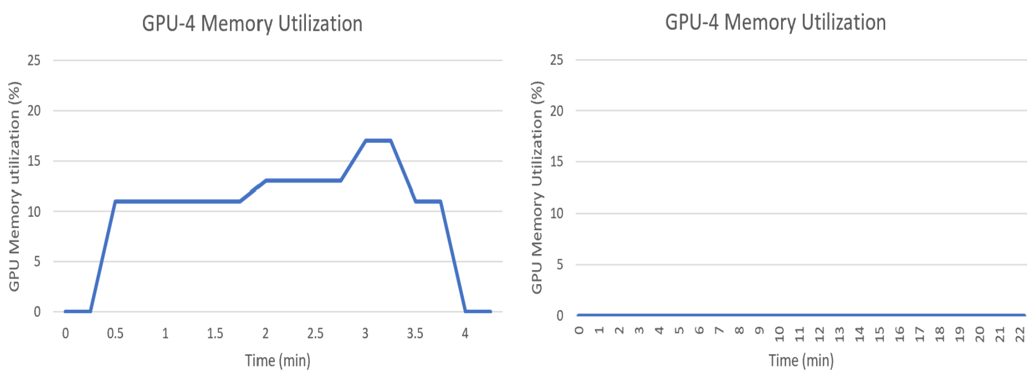
(a) GPU utilization for GPU 2 in system A (b) GPU utilization for GPU 2 in system B

Figure 6.5: GPU memory utilization of GPU 2 during experiment 1



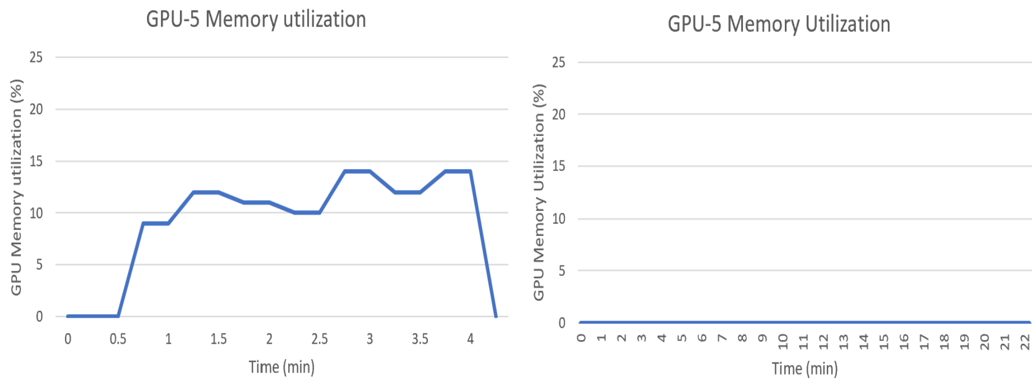
(a) GPU utilization for GPU 3 in system A (b) GPU utilization for GPU 3 in system B

Figure 6.6: GPU memory utilization of GPU 3 during experiment 1



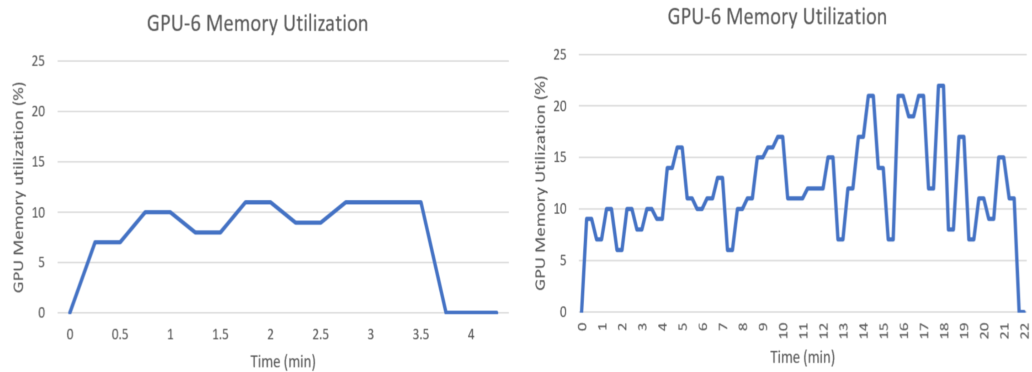
(a) GPU utilization for GPU 4 in system A (b) GPU utilization for GPU 4 in system B

Figure 6.7: GPU memory utilization of GPU 4 during experiment 1



(a) GPU utilization for GPU 5 in system A (b) GPU utilization for GPU 5 in system B

Figure 6.8: GPU memory utilization of GPU 5 during experiment 1



(a) GPU utilization for GPU 6 in system A (b) GPU utilization for GPU 6 in system B

Figure 6.9: GPU memory utilization of GPU 6 during experiment 1

It is observed that all the GPU resources are utilized when the experiment is performed on system A, whereas in system B, a single GPU resource (GPU 6) is utilized . Table 6.4 provides the GPU Activity of each GPU resource in the cluster measured over the duration of the experiment. It can be observed that system B uses the GPU resources more conservatively as compared to system A.

GPU Number	GPU Active Time in System A	GPU Active Time in System B
1	92.5%	0%
2	92.4%	0%
3	92.5%	0%
4	92.5%	99.6%
5	92.5%	0%
6	92.6%	0%

Table 6.4: GPU activity for both the autoscaling systems

6.1.4 Monthly Cost

The monthly estimate generated for running the cluster[1]

System	Monthly Cost
A	$16.57
B	$93.22

Table 6.5: Monthly cost for running the cluster

It is important to note that the cost shown for system B in table 6.5 is inflated, since it is calculated based on the assumption that all 6 GPU resources are pre-booked and thus contribute a fixed cost despite not clocking active usage minutes. If the systems used a GPU-on-demand acquisition model, the monthly cost of maintaining the cluster would reduce to **$15.54**. Table 6.6 shows the cost of maintaining the clusters assuming the GPU resources are pre-booked based on the number of GPUs used by both the systems during the experiment.

System	Monthly Cost
A	$16.57
B	$15.54

Table 6.6: Monthly cost for running the cluster assuming GPUs are pre-booked according to the number of GPUs utilized during the experiment

[1]Obtained from GCP Cost calculator

6.2 Experiment 2 - Loading the system with variable traffic

In this experiment, the systems are loaded with inference traffic of variable intensities. A total of 48000 requests are sent to each system in four phases with varying rates. In the first phase, 16200 requests are sent in batches of 6, with a frequency of 90 requests per second. In the second phase, 9600 requests are sent in batches of 2, with a frequency of 20 requests per second. The third phase sees a rise in the request frequency to 90 requests per second, with 16200 requests sent in batches of 6. In the final phase, 6000 requests are sent with a frequency of 1 request per second.
The following graphs depict the total number of inference requests sent by the load generator measured every 15 seconds over all the four phases.

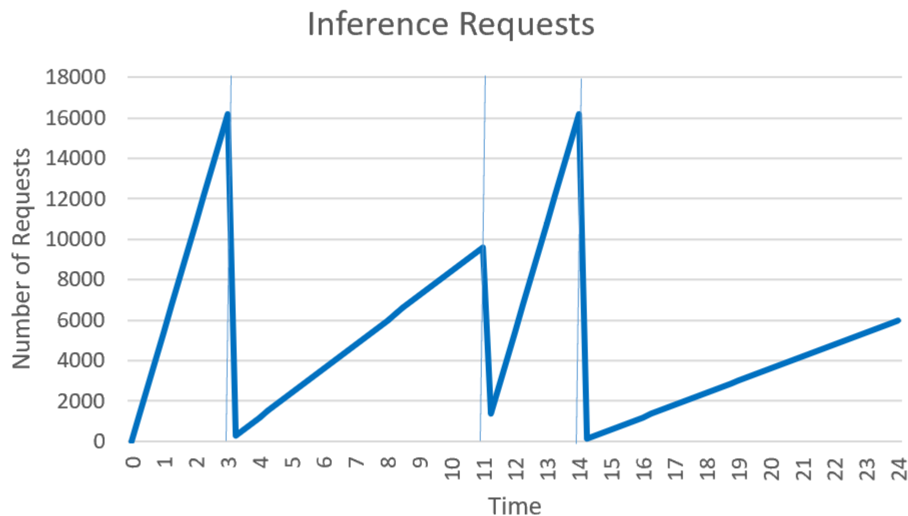


Figure 6.10: Inference Requests sent by the load generator

6.2.1 Inference latency statistics

The inference latency statistics are divided into 4 subsections, representing the four phases.

- **Phase 1**
  Table 6.7 and 6.8 represent the latency distribution and general statistics related to the inference response times for the requests sent in phase 1.

System	10%	25%	50%	75%	90%	95%	99%
A	0.0570s	0.0819s	0.0978s	0.1001s	0.1243s	0.1379s	0.1613s
B	0.4982s	0.0819s	0.0978s	0.1001s	0.1243s	0.1379s	0.1613s

Table 6.7: Latency distribution

System	Total	Slowest	Fasted	Average	Throughput
A	264.7642s	2.2597s	0.0492s	0.0973s	61.1865
B	2703.0474s	4.6491s	0.2962s	0.9986s	5.9932

Table 6.8: Response Statistics

- **Phase 2**
  Table 6.9 and 6.10 represent the latency distribution and general statistics related to the inference response times for the requests sent in phase 2.

System	10%	25%	50%	75%	90%	95%	99%
A	0.0515s	0.0519s	0.0524s	0.0536s	0.1140s	0.1275s	0.1582s
B	0.2367s	0.3003s	0.7090s	1.1229s	1.3049s	1.4023s	1.5156s

Table 6.9: Latency distribution

System	Total	Slowest	Fasted	Average	Throughput
A	501.6552s	2.0061s	0.0502s	0.0648s	19.1367
B	3690.1597s	3.8326s	0.1373s	0.7687s	2.6015

Table 6.10: Response Statistics

- **Phase 3**
  Table 6.11 and 6.12 represent the latency distribution and general statistics related to the inference response times for the requests sent in phase 3.

System	10%	25%	50%	75%	90%	95%	99%
A	0.0553s	0.0798s	0.0976s	0.0999s	0.1111s	0.1228s	0.30368s
B	0.6995s	0.7992s	1.1001s	1.2972s	1.4004s	1.4955s	1.6028s

Table 6.11: Latency distribution

System	Total	Slowest	Fasted	Average	Throughput
A	255.4326s	2.1345s	0.0494s	0.0938s	63.4218
B	2850.4794s	5.5516s	0.2378s	1.0544s	5.6833

Table 6.12: Response Statistics

- **Phase 4**

  Table 6.13 and 6.14 represent the latency distribution and general statistics related to the inference response times for the requests sent in phase 3.

System	10%	25%	50%	75%	90%	95%	99%
A	0.0514s	0.0519s	0.0526s	0.0538s	0.0757s	0.1069s	0.1318s
B	0.0519s	0.0531s	0.2946s	0.9055s	1.2989s	1.3989s	1.5126s

Table 6.13: Latency distribution

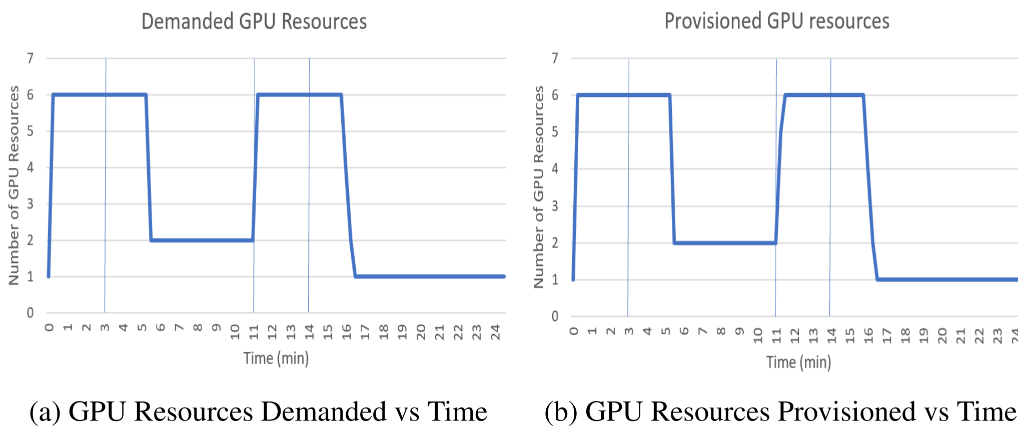
System	Total	Slowest	Fasted	Average	Throughput
A	602.4522s	0.1537s	0.0503s	0.0584s	9.9593
B	3235.4530s	4.2545s	0.0504s	0.5228s	1.8545

Table 6.14: Response Statistics

It is observed that the inference response times to serve the requests sent during each phase is much faster when system A is used. This is again due to the inability of system B to scale up the number of GPUs during the phases with high intensities of requests. As a consequence, a single GPU processes all the requests sequentially, leading to requests waiting in the activator for longer durations of time.

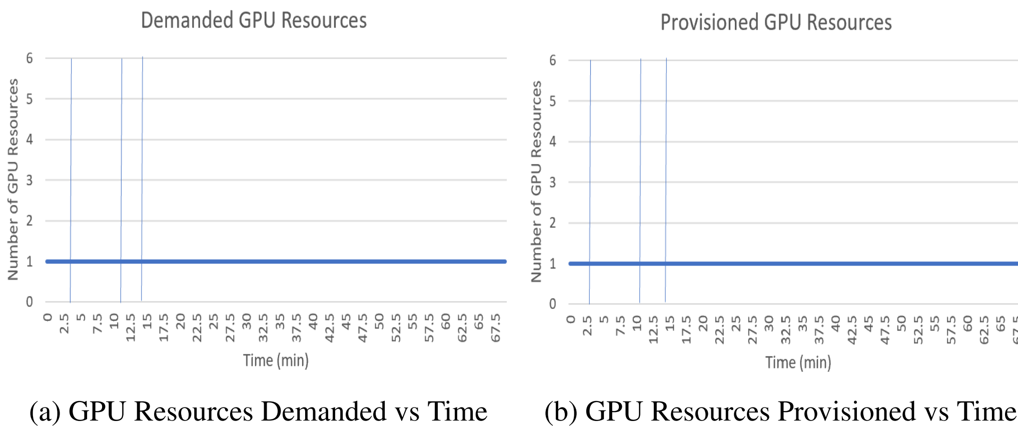
6.2.2 Scaling pattern

This section evaluates the GPU scaling behaviour of both the systems as observed during the experiment. Figures 6.11 and 6.12 show the GPU resources demanded and provisioned by both the systems as measured every 15 seconds.



(a) GPU Resources Demanded vs Time (b) GPU Resources Provisioned vs Time

Figure 6.11: Scaling behavior for system A



(a) GPU Resources Demanded vs Time (b) GPU Resources Provisioned vs Time

Figure 6.12: Scaling behavior for system B

System A closely monitors the incoming traffic. It scales the number of GPU resources, up and down, matching the varying intensities introduced by each phase. The scaling, however introduces

under-provisioning costs. On the other hand, system B does not scale the GPU resources to the varying intensities of each phase since the GPU memory utilization for every request remains more or less the same (and below the set threshold). As a result, a single GPU resource is used to handle the entire inference load.
Table 6.15 depicts the autoscaling metrics calculated for both the systems during experiment 2.

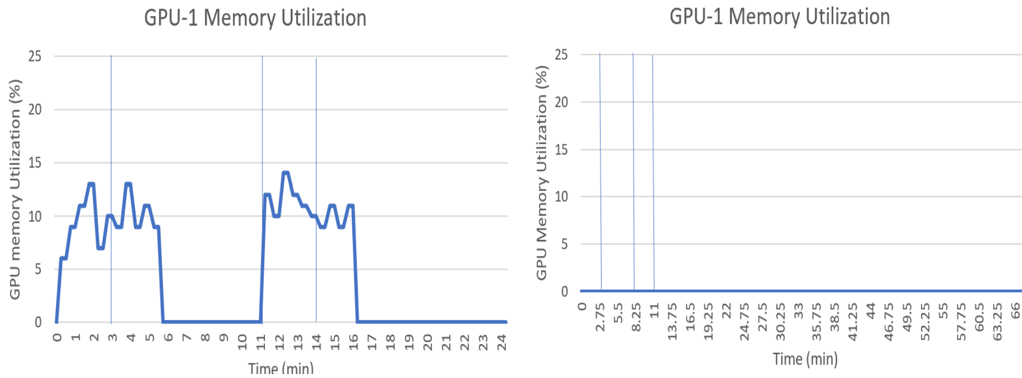
System	θ_U	θ_O	τ_U	τ_O	θ	τ
A	0.17%	0%	1.02%	0%	0.08%	0.51%
B	0%	0%	0%	0%	0%	0%

Table 6.15: Autoscaling Metrics for experiment 2

Since no scaling happens when system B is used, no under-provisioning or over-provisioning of GPU resources is observed. On the contrary, system A undergoes under-provisioning of GPU resources.

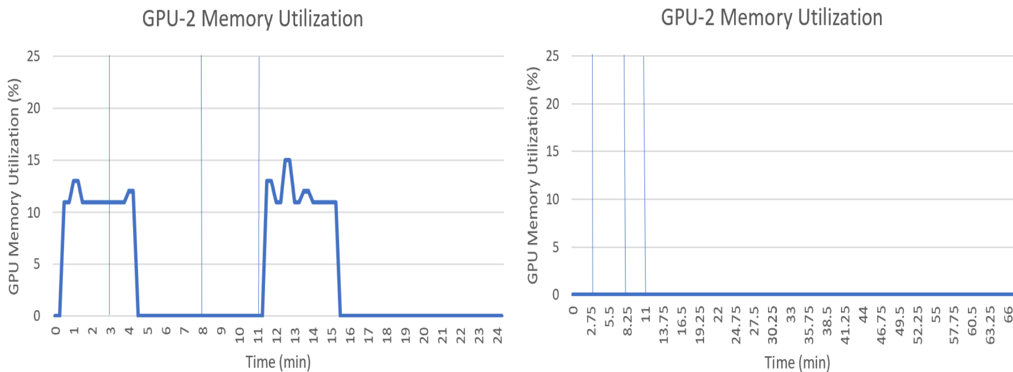
6.2.3 GPU Memory utilization

The following graphs depict the GPU memory utilization values of all six GPUs in the cluster over the duration of the experiment.



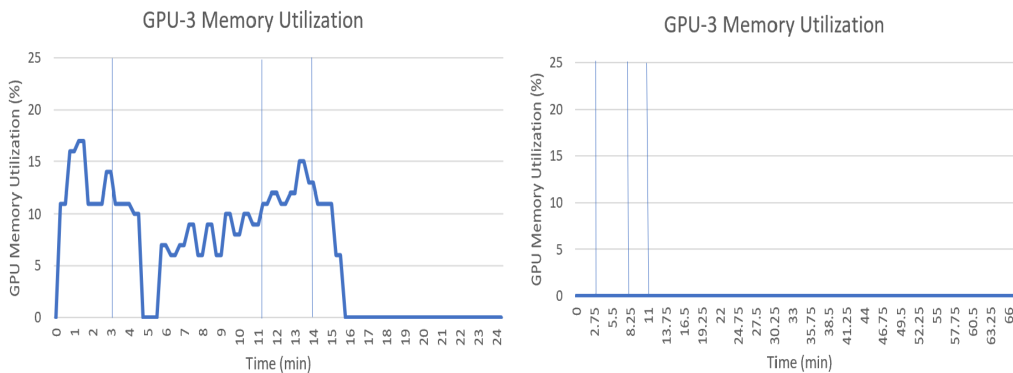
(a) GPU utilization for GPU 1 in system A (b) GPU utilization for GPU 1 in system B

Figure 6.13: GPU memory utilization of GPU 1 during experiment 2



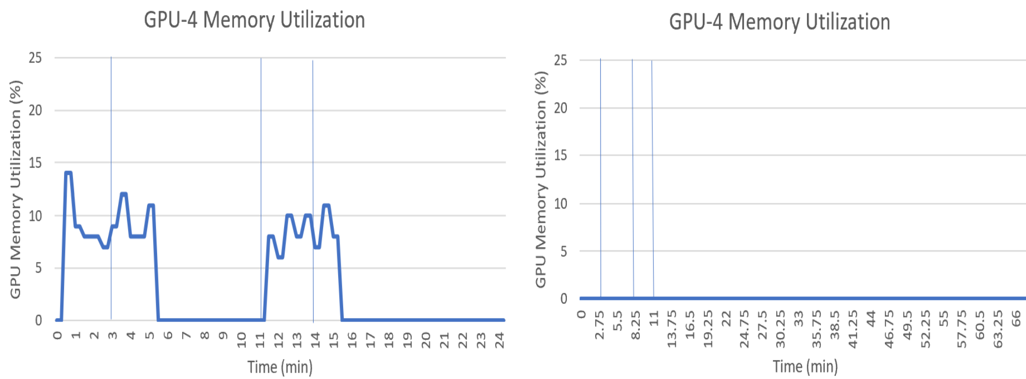
(a) GPU utilization for GPU 2 in system A (b) GPU utilization for GPU 2 in system B

Figure 6.14: GPU memory utilization of GPU 2 during experiment 2



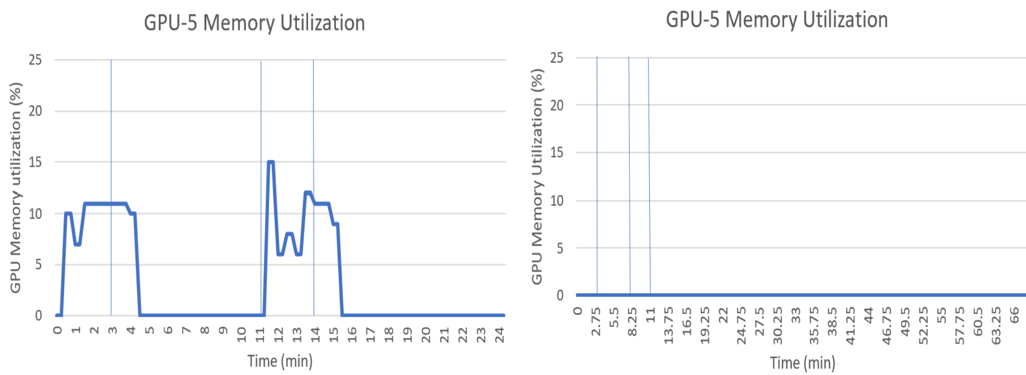
(a) GPU utilization for GPU 3 in system A (b) GPU utilization for GPU 3 in system B

Figure 6.15: GPU memory utilization of GPU 3 during experiment 2



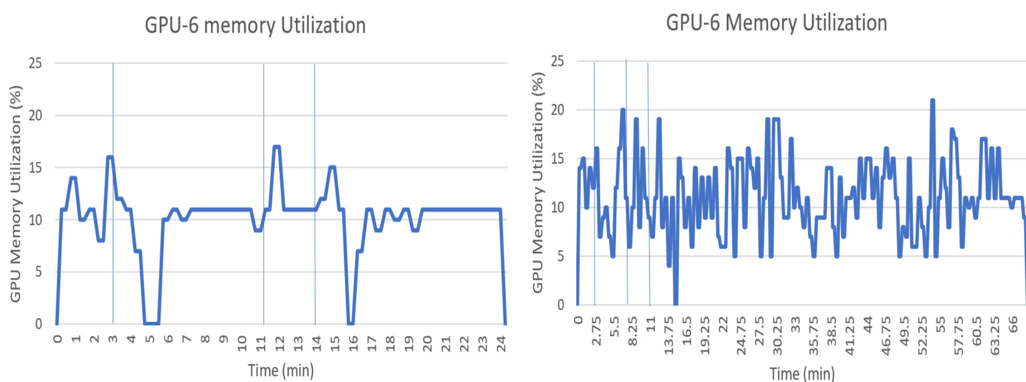
(a) GPU utilization for GPU 4 in system A (b) GPU utilization for GPU 4 in system B

Figure 6.16: GPU memory utilization of GPU 4 during experiment 2



(a) GPU utilization for GPU 5 in system A (b) GPU utilization for GPU 5 in system B

Figure 6.17: GPU memory utilization of GPU 5 during experiment 2



(a) GPU utilization for GPU 6 in system A (b) GPU utilization for GPU 6 in system B

Figure 6.18: GPU memory utilization of GPU 6 during experiment 2

Similar to the previous experiment 1, it is observed that all the GPU resources are utilized when the experiment is performed on system A, whereas a single GPU resource (GPU 6) is utilized for system B.
Table 6.16 provides the GPU Activity of each GPU resource in the cluster measured over the duration

of the experiment. It can be observed that system B uses the GPU resources more conservatively as compared to system A.

GPU Number	GPU Activity in System A	GPU Activity in System B
1	43.29%	0
2	32.98%	0
3	59.79%	0
4	37.11%	99.27%
5	23.98%	0
6	92.7%	0

Table 6.16: GPU activity over the duration of the experiment for both the autoscaling systems

6.2.4 Monthly Cost

The monthly cost estimates calculated for running the cluster for both the systemss is shown in table 6.17.

System	Monthly Cost
A	$105.41
B	$297.82

Table 6.17: Monthly cost for running the cluster

As seen before in experiment 1, it is important to note that the cost shown for system B is inflated, since it is calculated based on the assumption that all 6 GPU resources are pre-booked. If system B pre-booked a single GPU resource or the system used a 'GPU-on-demand' acquisition model, the monthly cost of maintaining the cluster would reduce to **$49.64**. Table 6.18 shows the cost of maintaining the cluster assuming that the GPU resources are pre-booked based on the number of GPUs used by both the systems during the experiment.

System	Monthly Cost
A	$105.41
B	$49.64

Table 6.18: Monthly cost for running the cluster assuming GPUs are pre-booked according to the number of GPUs utilized during the experiment

6.3 Experiment 3: Reducing the user-defined threshold for average GPU memory utilization

In this experiment the threshold for the average GPU memory utilization is set to 6 % for system B. Both experiments 1 and 2 are then run on this system to evaluate the performance in the two different environments.

6.3.1 Load the system with constant traffic

As done previously, the systems are loaded with 15000 requests, with a request frequency of 90 requests per second. The following is a graph depicting the total number of inference requests sent by the load generator measured every 15 seconds.

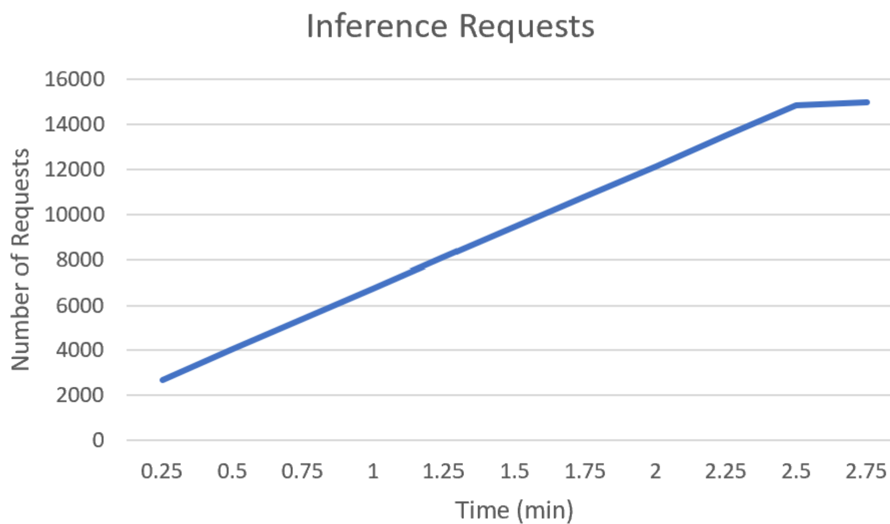


Figure 6.19: Inference Requests sent by the load generator

6.3.1.1 Inference Latency Statistics

Tables 6.19 and 6.20 represent the latency distribution and general statistics on the observed response times.

10%	25%	50%	75%	90%	95%	99%
0.0519s	0.0550s	0.0845s	0.1261s	0.1873s	0.2826s	0.5964s

Table 6.19: Latency distribution

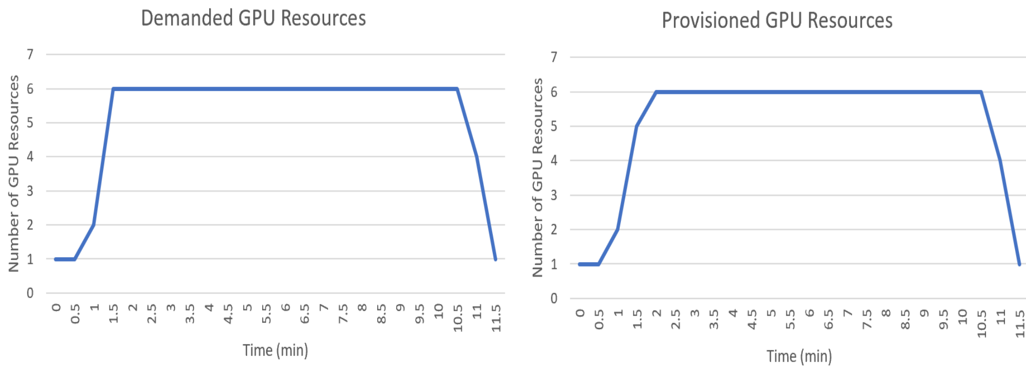
Total	Slowest	Fasted	Average	Throughput
306.8517s	4.3214s	0.0495s	0.1200s	48.8835

Table 6.20: Response Statistics

With a decrease in the threshold value, the total inference response time becomes approximately four times faster than before. System A still shows a faster response time compared to system B. However, the difference in the response times between the two systems have drastically reduced.

6.3.1.2 Scaling Behaviour

This section evaluates the GPU scaling behaviour of both the systems as observed during the experiment. Figure 6.20 shows the GPU resources demanded and provisioned by the system as measured every 15 seconds.



(a) GPU Resources Demanded vs Time (b) GPU Resources Provisioned vs Time

Figure 6.20: Scaling behavior of the system during experiment 1

Table 6.21 depicts the autoscaling metrics calculated for both the systems during the experiment.

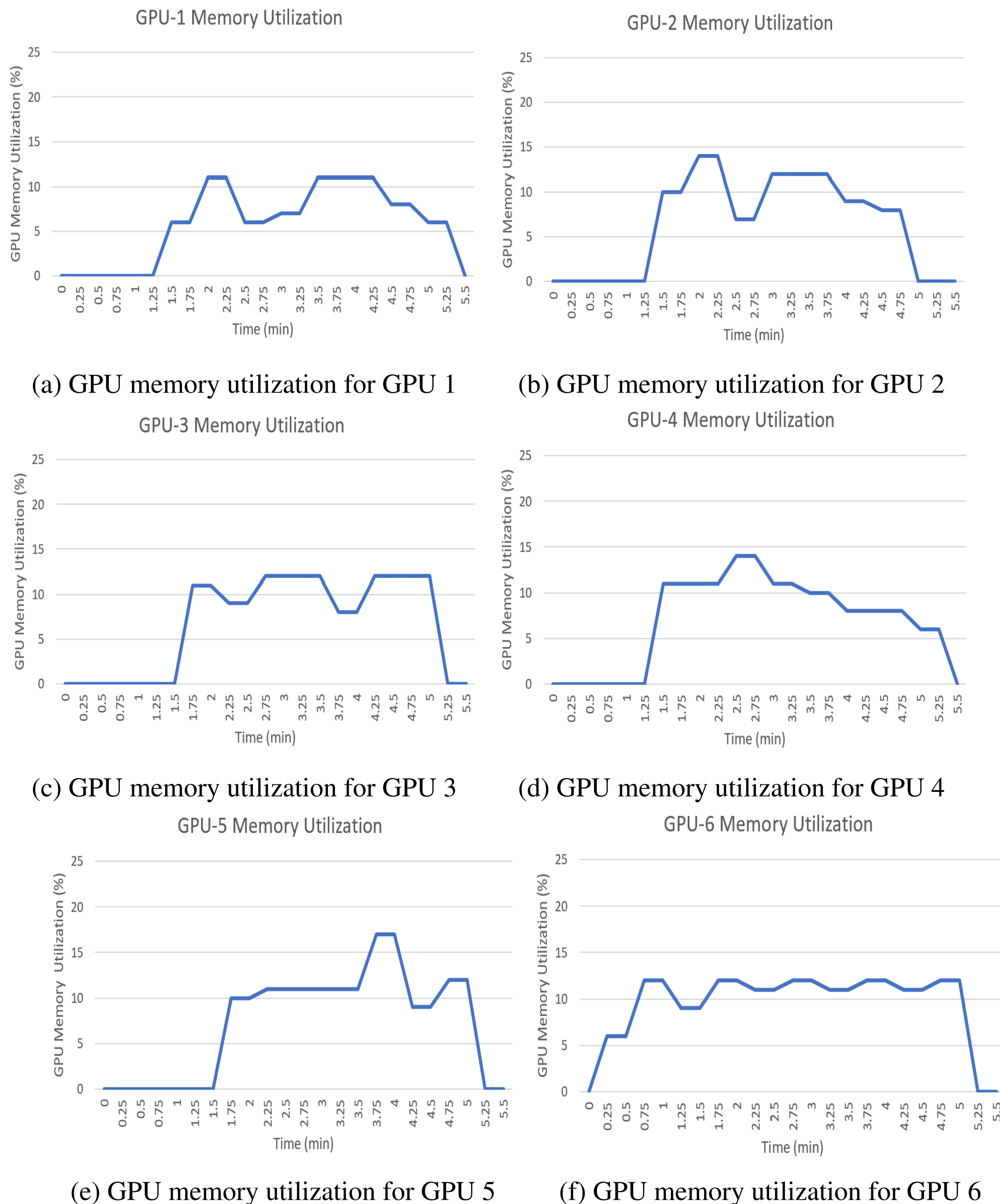
θ_U	θ_O	τ_U	τ_O	θ	τ
0.76%	0%	4.56%	0%	0.38%	2.27%

Table 6.21: Autoscaling Metrics

It is observed that the low threshold for the average GPU memory utilization triggers an early scale up to 6 GPU resources. The behavior is similar to that of system A. However, it takes a longer duration for system B to scale down due to the cool down period introduced by the HPA (Section 2.3.1).

6.3.1.3 GPU Utilization

The following graphs depict the GPU memory utilization values of all six GPUs in the cluster over the duration of the experiment.



(a) GPU memory utilization for GPU 1

(b) GPU memory utilization for GPU 2

(c) GPU memory utilization for GPU 3

(d) GPU memory utilization for GPU 4

(e) GPU memory utilization for GPU 5

(f) GPU memory utilization for GPU 6

Figure 6.21: GPU memory utilization of all GPU resources during the experiment

It is observed that system B starts utilizing all 6 GPU resources, similar to system A. Table 6.22 shows the GPU activity of all the GPU resources in the cluster during the experiment.

GPU Number	GPU Activity
1	72.72%
2	63.63%
3	63.63%
4	72.72%
5	63.63%
6	90.90%

Table 6.22: GPU activity over the duration of the experiment

It is observed that system B does not use the GPU resources in a conservative manner anymore. It's behaviour is similar to that of system A.

6.3.1.4 Monthly Cost

Table 6.23 provides the monthly cost estimate calculated for running the cluster for both the systems.

System	Monthly Cost
A	$16.57
B	$ 22.41

Table 6.23: Monthly cost for running the cluster

The difference between the cost estimates has reduced compared to before. However, system A still has a lower cost.

6.3.2 Loading the system with variable traffic

As done previously in Experiment 2, the systems are loaded with inference traffic of variable intensities. A total of 48000 requests are sent to each system in four phases with varying rates. In the first phase, 16200 requests are sent with a frequency of 90 requests per second. In the second phase, 9600 requests are sent with a frequency of 20 requests per second. The third phase sees a rise in the request frequency with 16200 requests sent at 90 requests per second. In the final phase, 6000 requests are sent with a frequency of 1 request per second.
The following graph depicts the total number of inference requests sent by the load generator measured every 15 seconds over all the four phases.

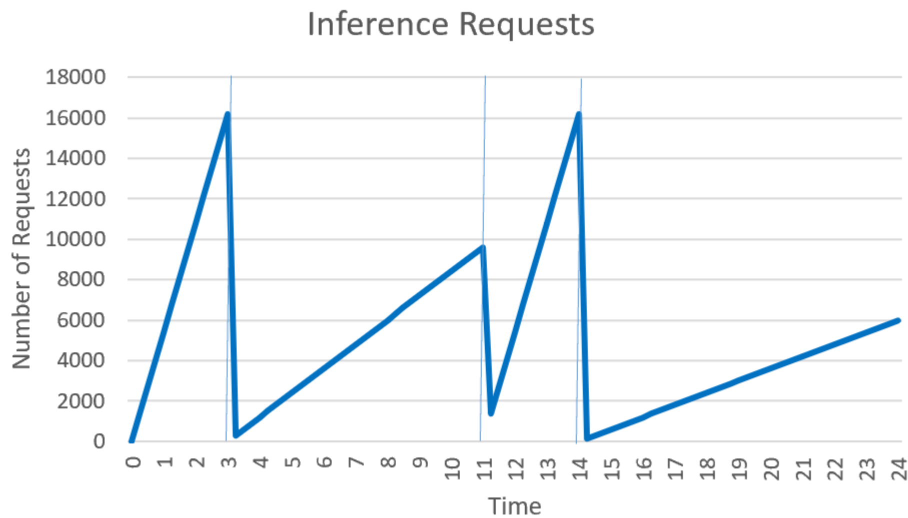


Figure 6.22: Inference Requests sent by the load generator

6.3.2.1 Inference Latency Statistics

- **Phase 1**

  Tables 6.24 and 6.25 represent the latency distribution and general statistics on the observed response times for the requests sent in phase 1.

10%	25%	50%	75%	90%	95%	99%
0.0523s	0.0699s	0.1126s	0.1630s	0.2143s	0.3657s	0.5990s

Table 6.24: Latency distribution

System	Total	Slowest	Fasted	Average	Throughput
A	264.7642s	2.2597s	0.0492s	0.0973s	61.1865
B	387.9200s	4.4531s	0.0495s	0.1407s	41.7612

Table 6.25: Response Statistics

- **Phase 2**

  Tables 6.26 and 6.27 represent the latency distribution and general statistics on the observed response times for the requests sent in phase 2.

10%	25%	50%	75%	90%	95%	99%
0.0542s	0.0550s	0.0562s	0.0956s	0.1596s	0.1901s	0.2363s

Table 6.26: Latency distribution

System	Total	Slowest	Fasted	Average	Throughput
A	501.6552s	2.0061s	0.0502s	0.0648s	19.1367
B	529.9146s	0.3521s	0.0497s	0.0817s	18.1161

Table 6.27: Response Statistics

- **Phase 3**

  Tables 6.28 and 6.29 represent the latency distribution and general statistics on the observed response times for the requests sent in phase 2.

10%	25%	50%	75%	90%	95%	99%
0.0518s	0.0571s	0.0982s	0.1430s	0.1783s	0.1982s	0.2386s

Table 6.28: Latency distribution

System	Total	Slowest	Fasted	Average	Throughput
A	255.4326s	2.1345s	0.0494s	0.0938s	63.4218
B	294.5161s	0.3613s	0.0493s	0.1072s	55.0055

Table 6.29: Response Statistics

- **Phase 4**

  Tables 6.30 and 6.31 represent the latency distribution and general statistics on the observed response times for the requests sent in phase 2.

10%	25%	50%	75%	90%	95%	99%
0.0519s	0.0550s	0.0845s	0.1261s	0.1873s	0.2826s	0.5964s

Table 6.30: Latency distribution

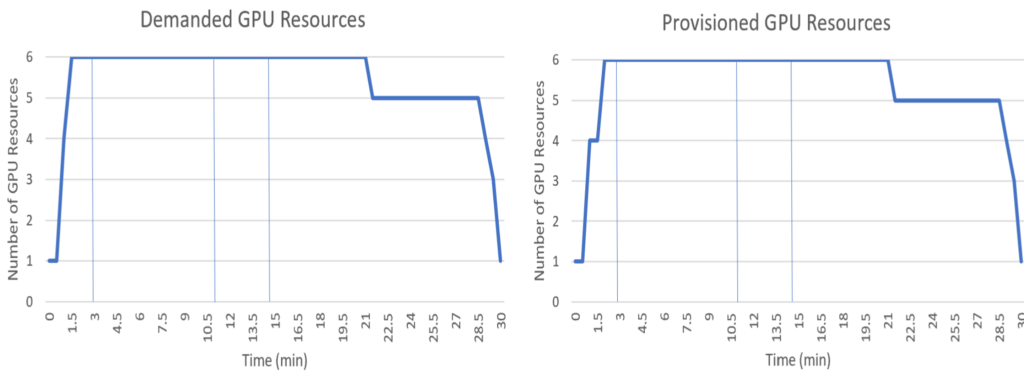
System	Total	Slowest	Fasted	Average	Throughput
A	602.4522s	0.1537s	0.0503s	0.0584s	9.9593
B	613.1551s	0.2520s	0.0500s	0.0632s	9.7855

Table 6.31: Response Statistics

It is observed that the inference response times for system B have reduced and are much closer to that of system A. This is because system B scaled up the number of GPU resources to 6 during the high intensity phases.

6.3.2.2 Scaling Behaviour

This section evaluates the GPU scaling behaviour of the system as observed during the experiment. Figure 6.23 shows the GPU resources demanded and provisioned by the system as measured every 15 seconds.



(a) GPU Resources Demanded vs Time (b) GPU Resources Provisioned vs Time

Figure 6.23: Scaling behavior

Table 6.32 depicts the autoscaling metrics calculated for both the systems during experiment.

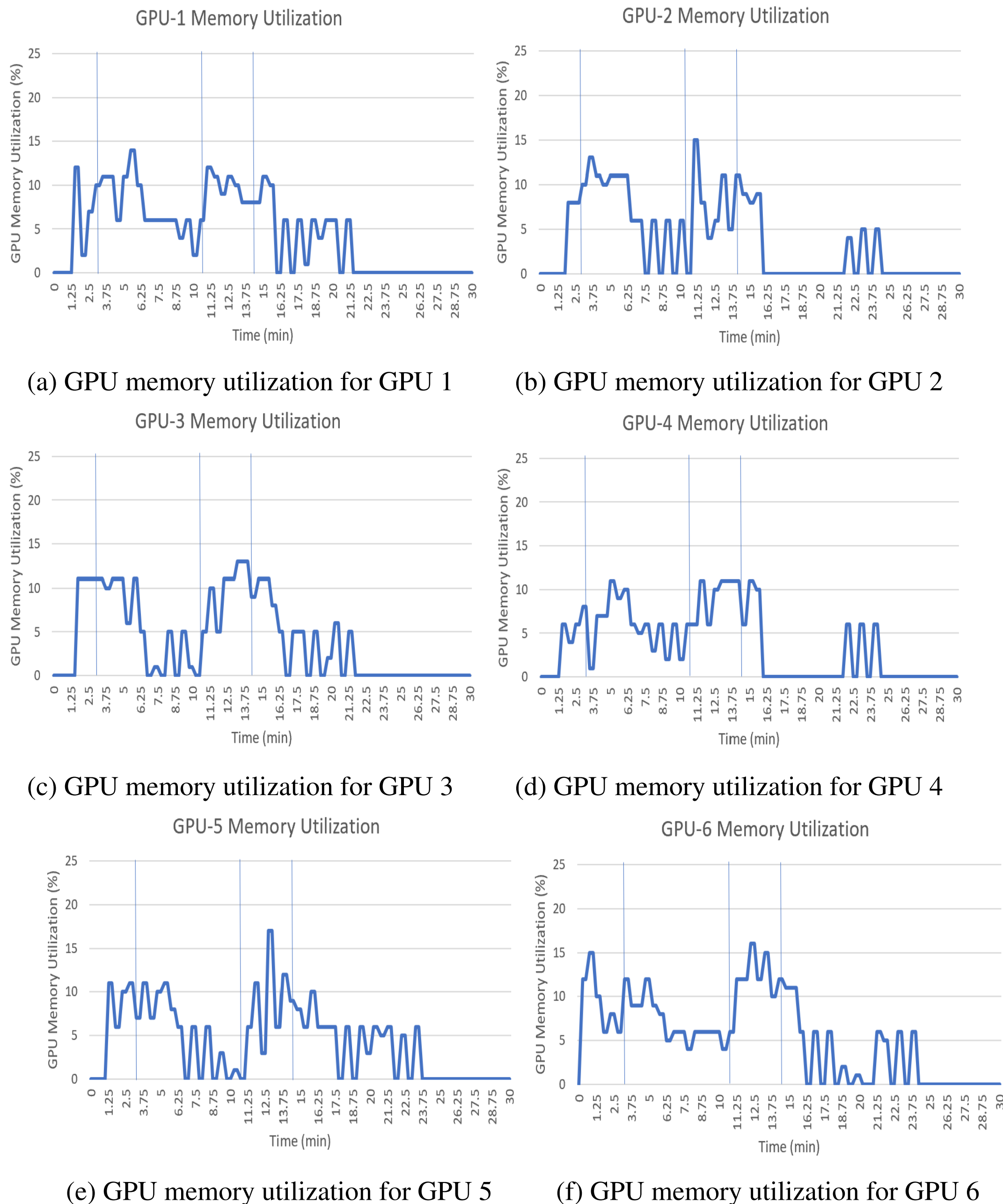
θ_U	θ_O	τ_U	τ_O	θ	τ
1.77%	0%	3.28%	0%	0.885%	1.64%

Table 6.32: Autoscaling Metrics

Similar to the previous experiment, the low threshold triggers the scaling up of the number of GPU resources used by the inference system from 1 to 6. In addition, we note that while system A would scale down in phase 2 as seen in Figure 6.11, the number of GPU resources in system B remain at 6 during phase 2. This is attributed to the 'cool down' period built into the HPA logic (Section 2.3.1), wherein after scaling up, the HPA waits for a duration of 5 minutes before making a decision to scale down. As a consequence, by the time it gets an opportunity to decide on whether to scale down, the system is loaded with the high intensity of requests in phase 3 forcing the system to maintain the use of 6 GPU resources. The GPU resources gradually scale down in phase 4 (low-intensity).
A certain degree of under-provisioning of GPU resources is observed during the experiment.

6.3.2.3 GPU Utilization

The following graphs depict the GPU memory utilization values of all six GPUs in the cluster over the duration of the experiment.



(a) GPU memory utilization for GPU 1

(b) GPU memory utilization for GPU 2

(c) GPU memory utilization for GPU 3

(d) GPU memory utilization for GPU 4

(e) GPU memory utilization for GPU 5

(f) GPU memory utilization for GPU 6

Figure 6.24: GPU memory utilization of all GPU resources during the experiment

All 6 GPU reources are utilized in system B. Table 6.33 displays the GPU activity of all 6 GPU resources during the experiment.

GPU Number	GPU Utilization
1	60.83%
2	45%
3	53.33%
4	53.33%
5	60%
6	66.66%

Table 6.33: GPU Activity over the duration of the experiment

As seen in the previous experiment, system B does not utilize the GPU resources in a conservative manner anymore.

6.3.2.4 Monthly Cost

Table 6.34 presents the monthly cost estimates calculated for running the cluster for both the systems.

System	Monthly Cost
A	$105.41
B	$106.16

Table 6.34: Monthly cost for running the cluster

The cost for maintaining the cluster for system B is similar to that of system A.

CHAPTER 7

Result Discussion

In both Experiments 1 and 2, where the defined threshold for the average memory utilization (30%) is sufficiently greater than the observed memory utilization, longer inference response times are observed for the GPU utilization based autoscaling approach.

System	Total Response Time
A	227.03s
B	1275.83s

Table 7.1: Inference Response Time in Experiment 1

System	Total Response Time
A	1442.45s
B	4075.43s

Table 7.2: Inference Response Time in Experiment 2

In Experiment 1 and 2, the GPU utilization based autoscaling approach uses a single GPU resource to handle the incoming inference workload. The approach utilized the GPU resources in a more conservative manner as compared to the inflight requests. On the contrary, all 6 GPU resources are utilized for the duration of the experiments in the case of the inflight requests based autoscaling approach.

Since the inflight requests based autoscaling approach scales the number of GPU resources during both experiment 1 and 2, it experiences a certain degree of under-provisioning of GPU resources. This can be seen by the values described in Table 6.3 and 6.15.

In Experiment 3 where the defined threshold for the average memory utilization is reduced to 6 %, it is observed that the behaviour of the GPU utilization based autoscaling system is similar to that of the inflight requests based autoscaling system. Similar inference response times, GPU activity as well as monthly costs for maintaining the clusters are observed.

Since the inbuilt autoscaling algorithm of the HPA handles thrashing by introducing a cool-down

period, where the resources aren't scaled down for at least five minutes following a scaling operation, substantially lesser degrees of under-provisioning of GPU resources is observed.

Further, as seen by the corresponding monthly cost estimations for each system (Sections 6.1.4 6.2.4), it is observed that in cases of an on-demand-GPU acquisition strategy, the GPU-utilisation based metric system, at higher utilisation target thresholds, is cheaper than the latency-favouring inflight requests based approach. This is indicative that such an autoscaling test simulation can potentially help companies make informed decisions about the outer bounds of cloud compute costs projections for either solution. This is of course, subject to other factors such as the system's multi-processing capabilities, and real-time environment characteristics.

CHAPTER 8

Conclusion

The thesis explored a possible alternate approach to GPU-autoscaling based on GPU memory utilization and compared its performance to the existing inflight based autoscaling approach currently provided by most inference serving solutions in the market. This was done by first designing and implementing an autoscaling component which scales on the basis of the average GPU memory utilization metric, and then integrating it with an ML inference system.
The autoscaling algorithm used by the approach is that provided by the in-built Kubernetes resource, Horizontal Pod Autoscaler. Following the implementation, the autoscaling component was integrated into an inference serving architecture described in Chapter 4. The final system was compared with another inference serving system having a similar architecture, differing only in the autoscaling component, which is an autoscaler that scales on the basis of the number of inflight requests.

The thesis conducts a comparison of two systems that mark different ends of the austoscaling objectives. It inspects the boundaries of their behaviours to better inform the design of any autoscaling system. The main findings of this thesis are summarized below:

- **Implementation and integrating custom solutions for GPU utilization based autoscaling which can inform industry applications:** Designed and implemented a custom solution which can be adapted and reused for testing or estimating costs by simulating behaviour of proposed ML inference deployment systems under varying conditions of incoming request traffic and system configurations.

- **Latency in serving requests:** Inflight-request based autoscaling, by virtue of definition of its scaling objective, offers lower latency than GPU based autoscaling. However, it does so at the expense of availing more GPU resources and potentially operating them at suboptimal levels. On the other end of the spectrum, optimising solely on basis of GPU utilisation metrics can result in more aggressive conservation of resources at the cost of slower processing and response times.

- **Managing GPU-use efficiency:** A tradeoff is observed between optimising for GPU utilisation at device level, versus response latency of the ML inference service. Thus, it is found that scaling logic based on GPU use metrics such as GPU memory utilization, more efficiently uses

a single resource to serve the entire request load, whereas a latency favouring system will use more resources while under-utilising each GPU in the stack so as to process requests quicker by concurrent processing. Thus, if a certain use-case can afford higher latency times, but requires minimal resource count, it can choose an approach closer to the GPU utilisation metric based autoscaling paradigm.

- **Resource Sharing flexibility:** Inflight-requests based autoscaling offers low inference times which is a requirement for several business cases, particularly in services that are part of customer facing applications. However it frequently requires the acquisition of a number of GPU resources to maintain these response speeds. This leaves the scarce GPU resources less flexible for sharing between different ML inference services.

- **Monetary Cost:** The cost for maintaining the clusters vary according to the pricing model and GPU acquisition strategy for cloud compute. If considering a pay-by-second model for *a fixed, pre-booked number of GPUs*, [1], the costs for maintaining a cluster running an inflight-request based autoscaling system appear lower compared to its GPU-based autoscaling counterpart, for performing inference on the same workload. On the other hand, if following an *on-demand-GPU acquisition strategy*, GPU-utilisation based autoscaling appears to be significantly cheaper than the latency favouring inflight requests based scaling approach. However, it is non trivial to determine exact cost functions, due to other influencing factors such as deployment strategy, cloud pricing model as well as the efficiency of the Deep learning logic deployed (parallelisation capability).

- **Notes on limits and convergence:** In case of GPU-metric based autoscaling, the study finds that lowering the value of GPU memory utilisation threshold to enable early triggering of the scaling decisions, causes the system to begin to mimic the behaviour of its inflight requests based autoscaling counterpart. In other words, we observe that lowering average GPU utilization threshold in the GPU-metric based scaling paradigm converges to the performance of the inflight request based autoscaling system. This allows delivering faster inference serving times as well as utilizing the available GPU resources in the cluster.

8.1 Future Work

The study, while systematically investigating foundational questions in the design of a GPU-based autoscaling strategy, also identifies interesting regions for possible future work beyond the thesis.

- The GPU utilization is usually low for serving a single model. An approach to increase the GPU utilization would be to run multiple models in parallel in the same GPU. Currently, Nvidia Triton inference server [Nvidia(2021)] allows placing multiple models in the same container but there is a need for a smarter solution which can intelligently place multiple models in the same container based on current GPU memory utilization.

- Exploring the performance of the GPU utilization based autoscaling system on a more complex metric which combines CPU utilization with GPU utilization can provide a more accurate measure to base the scaling decision on.

[1] https://cloud.google.com/products/calculator

- Incorporating predictive autoscaling to further improve the performance of the autoscaling solution. In Predictive autoscaling, an ML model is used to predict the future values for the autoscaling metrics.

Bibliography

[Al-Haidari et al.(2013)] Fahd Al-Haidari et al. Impact of cpu utilization thresholds and scaling size on autoscaling cloud resources. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, volume 2, pages 256–261. IEEE, 2013. 7

[bol(2021)] bol. Hpa: how it can save you headaches and money. Technical report, 2021. URL https://techlab.bol.com/hpa-how-it-can-save-you-headaches-and-money/. Aug, 2021. 7

[Casalicchio(2019)] Emiliano Casalicchio. A study on performance measures for auto-scaling cpu-intensive containerized applications. *Cluster Computing*, 22(3):995–1006, 2019. 7

[Core(2021)] Seldon Core. Seldon core documentation. Technical report, 2021. v1.1.0, access Aug 2021. 7, 8

[Cortex(2021)] Cortex. Cortex documentation. Technical report, 2021. URL https://docs.cortex.dev/. v0.40, access Aug 2021. 7, 8

[Cox et al.(2020)] Clive Cox et al. Serverless inferencing on kubernetes. *arXiv preprint arXiv:2007.07366*, 2020. 7

[Google(2015)] Google. Borg: The predecessor to kubernetes. Technical report, 2015. URL https://kubernetes.io/blog/2015/04/borg-predecessor-to-kubernetes/. April, 2015. 10

[Google(2021)] Google. Gcp gpu pricing. Technical report, 2021. URL https://cloud.google.com/compute/gpus-pricing. access Aug 2021. 25

[Herbst et al.(2016)] Nikolas Herbst et al. Ready for rain? a view from spec research on the future of cloud metrics. *arXiv preprint arXiv:1604.03470*, 2016. 24

[Kaiser(2019a)] Caleb Kaiser. Benchmarking openai's gpt-2 on gpus vs. cpus. Technical report, 2019a. URL https://bit.ly/3yd7aFq. 6

[Kaiser(2019b)] Caleb Kaiser. Implementing request-based autoscaling for machine learning workloads. Technical report, 2019b. URL https://bit.ly/3z8VpRA. 6, 7

[KFServing(2021)] KFServing. Kfserving documentation. Technical report, 2021. URL https://www.kubeflow.org/docs/components/kfserving/. v0.1.3, access Aug 2021. 7, 8, 23

[Kubernetes(2021)] Kubernetes. Kubernetes documentation. Technical report, 2021. URL https://kubernetes.io/docs/. v1.10, access June 2021. 10, 11, 12, 23

[Mell et al.(2011)] Peter Mell et al. The nist definition of cloud computing. 2011. 10

[Nvidia(2021)] Nvidia. Nvidia triton inference server documentation. Technical report, 2021. URL https://developer.nvidia.com/nvidia-triton-inference-server. Aug, 2021. 51

Appendix

DCGM currently supports the following products and environments:

- All K80 and newer Tesla GPUs
- NVSwitch on DGX-2
- All Maxwell and newer non-Tesla GPUs
- limited DCGM functionality is available on non-Tesla GPUs