ENABLING KUBERNETES FOR DISTRIBUTED AI PROCESSING ON EDGE DEVICES

by

Ronak Vijay Raheja

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Science in
Electrical Engineering

Charlotte

2020

Approved by:

_____
Dr. Hamed Tabkhi

_____
Dr. Arun Ravindran

_____
Dr. Erik Saule

ABSTRACT

RONAK VIJAY RAHEJA. Enabling Kubernetes for distributed AI processing on edge devices. (Under the direction of DR. HAMED TABKHI)

AI processing has been a big area of focus for the research community for quite some years. The growing computation capability of edge devices has allowed Computer Vision to make use of AI techniques with greater efficiency and throughput. This approach has led to the concept of distributed computing to solve AI problems at edge. Distributed computing brings many challenges along with it like complexity, high deployment cost and security. Similar problems have been resolved in the context of Cloud Computing through containerization of the applications. This thesis makes use of the same containerization techniques to provide a simple and effective method for AI developers to create distributed systems with ease. The software framework proposed in this work makes use of Kubernetes for container orchestration.

Containerization allows breaking down the entire application into smaller chunks running in isolated environments which are called microservices. Usage of microservices for distributing tasks across edge devices is a contribution of this thesis. This enables pipelining the application into microservice-level stages that can run on separate edge devices. This thesis explores the configuration of heterogeneous distributed network which incorporate Kubernetes and containerization to explore the benefits of pipelining microservices of previously monolithic applications. By utilizing Kubernetes ability to automatically orchestrate a defined network this thesis also seeks to also explore the networks sustainability and scalability based around the configuration of pipelined microservices, resource availability, and the distribution of these services within the network.

DEDICATION

This work is dedicated to my family, friends, professors and colleagues.

ACKNOWLEDGEMENTS

I would like to thank Dr. Tabkhi for guiding me at every step during my research. I am grateful for all the support provided by him. I thank Dr. Ravindran and Dr. Saule for not just accepting to be on my thesis committee, but also helping me and always teaching me a thing or two whenever I looked up to them for any guidance or suggestions.

I would like to specially thank my colleague Joshua Slycord who has helped me learn more about Kubernetes and Docker. It was his vision to move towards pipelining using Kubernetes for applications like Machine Learning.

I would also like to thank my parents, grandparents and sister for inspiring me all my years to take few important steps in my life that led to this thesis. I thank my friends for their constant love and support which encouraged me to work on my thesis and always go an extra mile.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
API	Application Peripheral Interface
AWS	Amazon Web Services
CCR	Communication to Computation Ratio
CNCF	Cloud Native Computing Foundation
DevOps	Development Operations
DNS	Domain Name System
ECE	An acronym for Electrical and Computer Engineering
GCP	Google Cloud Platform
GPGPU	General Purpose Graphics Processing Unit
IDL	Interface Definition Language
k8s	Kubernetes
L4T	Linux For Tegra
ML	Machine Learning
MLOps	Machine Learning Operations
SDK	Software Development Kit
VM	Virtual Machine
YAML	YAML Ain't Markup Language

CHAPTER 1: INTRODUCTION

The Machine Learning approach for aiding Computer Vision has been a hot topic for several years now. It has been an area of research to constantly better improve Computer Vision using ML for real world applications like object detection, face recognition and so on. They have proven to be better solved with ML or Deep Learning techniques rather than traditional statistical methods.

ML algorithms are based around the training-testing-inference model where training and testing are compute intensive tasks which need huge systems with high compute capabilities to process. All the three tasks have proven to have a need of an accelerator to boost performance due to their embarrassingly parallel nature. Inference for the computer vision applications can now be done at edge due to a vast options of edge devices available which also are bundled with accelerators. Edge computing can be understood as processing data closer to where it is being produced or consumed. Edge computing brings computation and data storage closer to the devices where it is being gathered, instead of relying on a central location that is somewhere far away. This allows lesser probability of being hit by network latency which affects performance and saves cost by reducing the data to be processed by the central servers. However, there are inherent obstacles when moving data such as lower security, bandwidth limitations, and hardware availability which has motivated enterprises to seek improvements to the edge-server model where compute intensive tasks are offloaded to the server.

The use of edge devices for Computer Vision has seen tremendous growth due to the increasing computing capabilities that they bring along with them. Edge devices are often bundled with accelerators like GPU's and FPGA's for achieving different

levels of parallelism in applications that are inherently non-sequential. This allows some good amount of processing to be done at edge for compute intensive applications like AI. However, sometimes the accelerator and the CPU from one edge device is sometimes not enough. This gives rise to the concept of distributed computing on edge devices where a network of such devices work together to complete the individual tasks of the underlying application the system was built for. Distributed computing brings with it many challenges like complexity, high deployment cost and security.

We have seen containerization solve similar problems in the context of Cloud Computing. Containerization is a lightweight alternative to a virtual machine that involves encapsulating an application in a container with its own operating system while still providing with an isolated environment. Containerization has allowed the Cloud Computing world to get the balance of scalability, reliability, portability and security with the use of Kubernetes. This thesis tries to bring the same balance to the world of distributed computing with edge devices. In order to achieve containerization on a distributed system, the application needs to be decoupled into smaller chunks of software called microservices that each serves a specific task required to achieve the processing done by the actual application.

1.1 Motivation

1.1.1 Importance of Edge Computing

Edge Computing allows enterprises more modularity, accessibility and portability. Their smaller form factors allow easier maintenance, deployability and lower power and costs. Edge computing has been even popular in the markets with the advent of 5G Wireless technology. Within no time one will be able to see edge devices bringing services to customers faster than the wireless base stations. This could play a key role in hiding the communication overheads caused with the edge-server distributed computing model and make applications lesser communication bound. To sum up the advantages with edge computing [1], [2], [3]:

- **Reduced Latency**: Edge devices process data locally or in local data centers, thus increasing network performance that leads to reduced latency. This helps maintain higher processing speeds and in turn better performance for the end use or final application and reduces latency to the order of microseconds. The time saved by the reduction of communication helps to lower the communication to computation ratio.
- **Scalability**: Edge devices are more application specific where each device could be designed to serve any specific purpose from computing, storage, interface, analysis or any combination of these. Edge computing offers a far less expensive route to scalability, allowing enterprises to expand their computing capacity through a combination of IoT devices and edge data centers.
- **Cost**: Having huge data servers do a lot of processing work costs more as one has to pay the cloud to compute. There are also typically higher overhead costs for maintaining this type of data center.
- **Reliability**: Processing at the edge, lower data communication reduces dependability on network and thus increasing reliability of the system. Moreover, having multiple devices in the network creates redundant paths for one failure to shut down the system completely. There can be multiple such paths created to ensure a complete functional system at all times.
- **Security**: Edge devices prove to be local layers of protection before any threat affects the entire network. Also, servers prove to be susceptible to these threats with huge amounts of data stored there. However, with edge devices, more data is processed locally and thus reducing risks of such threats.

1.1.2 Distributed Computing at Edge and problems associated with it

One major challenge faced by enterprises and research communities is the inefficient distribution of the workload between edge and server and the communication between these devices on a distributed computing system. These devices are known

to boast of different architectures and inherently different capabilities; both hosts and accelerators. This creates a lot of complexity due to the different versions of tools that they are bundled with. The software and their libraries need to be designed to be compatible to the respective devices. Then comes the typical "it works on my machine" problem. To list out the drawbacks of Distributed Computing at edge:

- **Complexity**: Distributed computing introduces complexity on software, hardware and network levels. Maintaining, deploying and debugging the software with the variance in architectures is difficult and complex.
- **Low software support**: Since it is difficult for programmers to reason about Distributed Computing, there is less support available for it. This is a big drawback because it shoots up the development time of these systems and often such projects are not completed fully.
- **High deployment and development cost**: Complexity and low software support lead to higher development cost. Complexity and the increasing overheads for information exchange between the devices increases the deployment cost.
- **Debugging and troubleshooting**: It gets complex to troubleshoot and diagnose failures across various devices on a network. Data synchronization is another reason why troubleshooting gets difficult.
- **Possible overkill**: Some situations can make it seem like the overall system is using more physical resources for the same application and thereby demanding more engineering efforts for the huge system.
- **Security**: Data is inherently distributed in a distributed system and poses a challenge towards security. Also, the data transfer across multiple devices causes security threats over the network.

1.1.3 Kubernetes approach for Distributed Computing at Edge

Traditional ML programs are written in a monolithic manner that contain huge chunks of code with high dependencies shared between them. These programs could be carefully broken down into loosely coupled components that are called microservices. The microservice architecture allows the introduction of pipelining between the loosely coupled components. This also helps making the application more fault tolerant and boosts its scalability. But it increases complexity of the software architecture and also brings in overheads in creation and management of these microservices.

Kubernetes (K8s) is an open-source system for automating deployment, scaling, and management of containerized applications[4]. This allows orchestration of containerized microservices and reduces the complexity of the software architecture. It works on images built using Docker and puts them into isolated environments that are called containers. A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings [5]. Docker images are created with the help of a Dockerfile that textually specifies the way the image has to be created. It is always built upon other base images or you could always use your own image to build upon.

Docker is a platform for developers and system administrators to build, share, and run applications with containers. The use of containers to deploy applications is called containerization [5]. There are often times many deployment and runtime issues associated with applications such as exposing it on a network, storage and memory and IO management, and controlling access permissions. Using Docker, one could wrap an application so that these issues can be handled out of the application and in way that provides consistence across all the containerized applications. Following is

a list of advantages using Docker:

- **Flexibility**: Docker sports its DockerHub which has a huge list of base images that one could start building upon and mould it in their own way while installing tools and dependencies required for the application that are compatible with that architecture.
- **Standardization**: Docker provides consistency across multiple development and release cycles thereby hiding the deployment and runtime issues associated with complex applications. It provides a standardize infrastructure across to an entire pipeline thereby enabling every team member to work in a consistent environment.
- **Lightweight**: Since containers do not boot separate OS and share the host kernel, they are lightweight in comparison with VMs. When common images have same base images, it reduces disk space too.
- **Rapid Deployment**: Docker manages to reduce deployment to seconds. It can create a container for every process and even does not boot an OS. So, even without worrying about the cost to bring it up again, it would be higher than what is affordable.
- **Ideal for microservice architecture**: Docker helps reduce performance overhead and deploy hundreds and thousands of microservices on the same server since they require lesser resources than other solutions like VMs. The ease in management and scaling of containers is another reason why Docker is a suitable candidate for microservice architecture. It also provides microservices faster start time and deployment.
- **Continuous deployment**: Docker containers are configured to maintain all configurations and dependencies internally; you can use the same container from development to production making sure there are no discrepancies or manual

intervention. It also allows continuous rolling of the containers if they were to be updated or upgraded.

- **Isolation**: Docker ensures your applications and resources are isolated and segregated. Docker makes sure each container has its own resources that are isolated from other containers.

One could end up with dozens or even millions of containers when using with microservices. Deployment, management, scheduling and load balancing are other tasks tied with containerized microservices which leads to the need for a container orchestrator. Kubernetes is an open source orchestrator for deploying and managing containerized applications at scale. Following are the advantages of Kubernetes:

- **Container Orchestration**: Kubernetes provides with features such as storage orchestration, automated roll outs and rollbacks, service discovery and load balancing, automatic bin packing, self-healing, secret and configuration management, endpoint slices, IPv4/IPv6 dual-stack, batch execution and horizontal scaling to aid container orchestration.
- **Ease of use**: Kubernetes is declarative in nature. Describing state of the cluster, details of containers to work with and nodes to assign jobs to can be easily done in Kubernetes with the help of YAML files.
- **Portability**: Kubernetes can be deployed on any and every infrastructure ranging from bare metal to virtual machines to public and private cloud environments. Being open source, it provides with higher flexibility too.
- **Scalability**: Kubernetes provides with Horizontal infrastructure scaling, autoscaling, manual scaling and replication controllers. It favors decoupled architectures that aids scaling of the system. Scaling is fairly easy due to the immutable, declarative nature of Kubernetes which was explained earlier.
- **Ideal for microservice architecture**: In contrast to monolithic applications whose constituent parts are not reusable and modular, Kubernetes encourages

writing code as microservices. Their small size and loose coupling make them easy to test and deploy in rapid fashion.

- **Service discoverability**: Kubernetes provides IP addresses for each pod, assigns a DNS name for each group of pods, and then load-balances the traffic to the pods in a set. This creates an environment where the service discovery can be abstracted away from the container level.

1.1.4 Contributions

This thesis presents a framework using Kubernetes clusters for distributed AI processing on edge devices. This framework provides with the scalability and portability that is essential for a good distributed system while maintaining ease of use in terms of job deployment. It enables the use of microservice approach which comes bundled with the idea of containerization. The contributions of this work are as follows:

- A scalable framework for the use of Docker and Kubernetes in the context of distributed edge computing.
- Integration of gRPC into microservices for data communication across pods and nodes on the Kubernetes cluster.
- Pipeline architecture for processing streaming applications in stages and thus boosting throughput of the system.

CHAPTER 2: BACKGROUND AND RELATED WORK

2.1 Kubernetes

Kubernetes is a software system that addresses the concerns of deploying, scaling and monitoring containers [4]. It was released by Google as open-source software, which is now managed by the Cloud Native Computing Foundation (CNCF). It is often associated with Google Cloud Platform and Amazon Web Services to run on their clusters. Kubernetes provides with:

- **Storage orchestration**: Kubernetes allows you to automatically mount the storage system of your choice, whether from local storage, a public cloud provider such as GCP or AWS, or a network storage system such as NFS, iSCSI, Gluster, Ceph, Cinder, or Flocker.
- **Automated roll-outs and rollbacks**: You can describe the desired state for your deployed containers using Kubernetes, and it can change the actual state to the desired state at a controlled rate. For example, you can automate Kubernetes to create new containers for your deployment, remove existing containers and adopt all their resources to the new container.
- **Service discovery and load balancing**: No need to modify your application to use an unfamiliar service discovery mechanism. Kubernetes gives Pods their own IP addresses and a single DNS name for a set of Pods, and can load-balance across them.
- **Automatic bin packing**: You provide Kubernetes with a cluster of nodes that it can use to run containerized tasks. You tell Kubernetes how much CPU and memory (RAM) each container needs. Kubernetes can fit containers onto your nodes to make the best use of your resources.

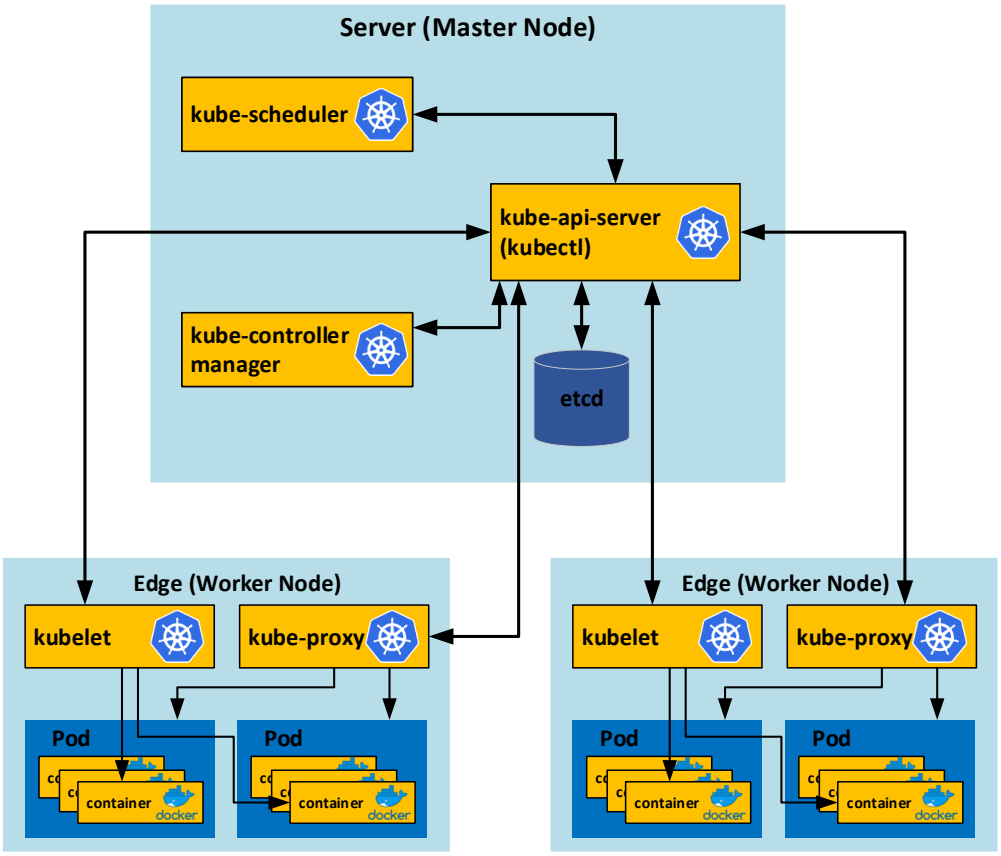

Figure 2.1: Kubernetes Edge-Server Architecture

- **Self-healing**: Kubernetes restarts containers that fail, replaces containers, kills containers that do not respond to your user-defined health check, and does not advertise them to clients until they are ready to serve.
- **Secret and configuration management**: Kubernetes lets you store and manage sensitive information, such as passwords, authorization tokens, and ssh keys. You can deploy and update secrets and application configuration without rebuilding your container images, and without exposing secrets in your stack configuration.
- **Endpoint slices**: Scalable tracking of network endpoints in a Kubernetes cluster. Kubernetes offers a more scalable and extensible alternative to endpoints.

- **IPv4/IPv6 dual-stack**: Allocation of IPv4 and IPv6 addresses to Pods and Services
- **Batch execution**: In addition to services, Kubernetes can manage your batch and CI workloads, replacing containers that fail, if desired.
- **Horizontal scaling**: Scale your application up and down with a simple command, with a UI, or automatically based on CPU usage.

Kubernetes is a production-ready, open source platform designed with Google's accumulated experience in container orchestration, combined with best-of-breed ideas from the community [4]. Before going to Kubernetes objects, here are a few basic Kubernetes concepts:

- **Container**: A lightweight and portable executable image that contains software and all of its dependencies.
- **Container Runtime**: The container Runtime is the software that is responsible for running containers.
- **Node**: A node is a worker machine in Kubernetes.
- **Cluster**: A set of machines, called nodes, that run containerized applications managed by Kubernetes. A cluster has at least one worker node and at least one master node.
- **Controllers**: In Kubernetes, controllers are control loops that watch the state of your cluster, then make or request changes where needed. Each controller tries to move the current cluster state closer to the desired state.
- **Kubectl**: A command line tool for communicating with a Kubernetes API server.
- **Sysctl**: Sysctl is a semi-standardized interface for reading or changing the attributes of the running Unix kernel.
- **Workload**: A workload is an application running on Kubernetes.

- **Kubelet**: An agent that runs on each node in the cluster. It makes sure that containers are running in a pod
- **Container**: Runtime interface (CRI): The CRI is an API for container runtimes to integrate with kubelet on a node.

Kubernetes contains abstractions to represent the state of your system: deployed containerized applications and workloads, their associated network and disk resources, and other information about what your cluster is doing. These abstractions are represented by objects in the Kubernetes API. The basic Kubernetes objects include:

- **Pod**: A Pod is the basic execution unit of a Kubernetes application. It is the smallest and simplest unit in the Kubernetes object model that you create or deploy. A Pod represents processes running on your Cluster.
- **Service**: An abstract way to expose an application running on a set of Pods as a network service.
- **Volume**: The on-disk files in a Container are ephemeral, giving rise to the Volume abstraction that is simply a directory on disk or in another Container.
- **Namespace**: Namespaces are intended for use in environments with many users spread across multiple teams, or projects

2.1.1 NVIDIA Jetson Xavier at edge

NVIDIA Jetson is the leading platform for AI at the edge. Its high performance, low power computing for deep learning and computer vision makes it an ideal platform for compute-intensive applications. Xavier is the latest addition to the Jetson family of developer kits by NVIDIA designed for robots, drones and other autonomous machines. It boasts of an 512-core NVIDIA Volta GPU with Tensor cores included along with an 8-core ARM v8.2 64-bit CPU. NVIDIA provides an SDK for its Jetson family which it calls it JetPack SDK. JetPack provides the developer kits with the latest OS image, libraries and APIs, samples, and documentation, as well as developer tools.

Once the SDK is loaded onto the Xavier, it boots a L4T based Ubuntu 18.04 (or the present available stable Ubuntu kernel) customized for 64-bit ARM architecture. NVIDIA L4T provides the bootloader, Linux kernel, necessary firmwares, NVIDIA drivers and sample filesystem. It contains prebuilt images for CUDA, TensorRT, cuDNN, OpenCV and other tools needed for AI software development. The kit has a small form factor of 100x87x16mm with various power modes ranging 15W through 30W.

2.2 Related work

Tao et al [6] present a survey on the virtualization frameworks using VM for edge computing. They talk a lot about the challenges that VM frameworks face for aiding edge computing. Large resource footprint of VMs, large data size of VMs and security issues are the main challenges listed after studying the industrial projects like OpenStack , KubeEdge, and OpenEdge. Other research projects studied by them include Paradrop, AirBox, Middleware for IoT Clouds, FocusStack, Amino, Lightweight Service Replication for Mobile-Edge Computing, SOUL and LAVEA. They shed light on the complexity that rises from using VMs due to added OS based overheads like placement and scheduling and security issue that the developer needs to constantly focus on.

Kubernetes for distributed edge devices is a less explored topic in the research community. KubeEdge [7] proposed by Xiong et al from Huawei introduce an infrastructure in edge computing environment, to mainly extend cloud capabilities to the edge. They present a network infrastructure for communication between cloud and edge devices with the help of Kubernetes. However, it does not provide a pipeline architecture and their results do not show any improvement over existing architectures. KubeEdge was built as a new distribution of Kubernetes and since then we have had multiple distributions like microk8s and k3s who have brought Kubernetes in better ways at edge.

Vayghan et al [8] propose an architecture for deploying microservice based applications with Kubernetes in a private cloud. Their architecture enables high availability with Kubernetes for microservice based applications. They tackle the pod failure and node failure scenarios through their architecture in the cloud environment.

CHAPTER 3: Enabling Kubernetes on Edge

3.1 Overview

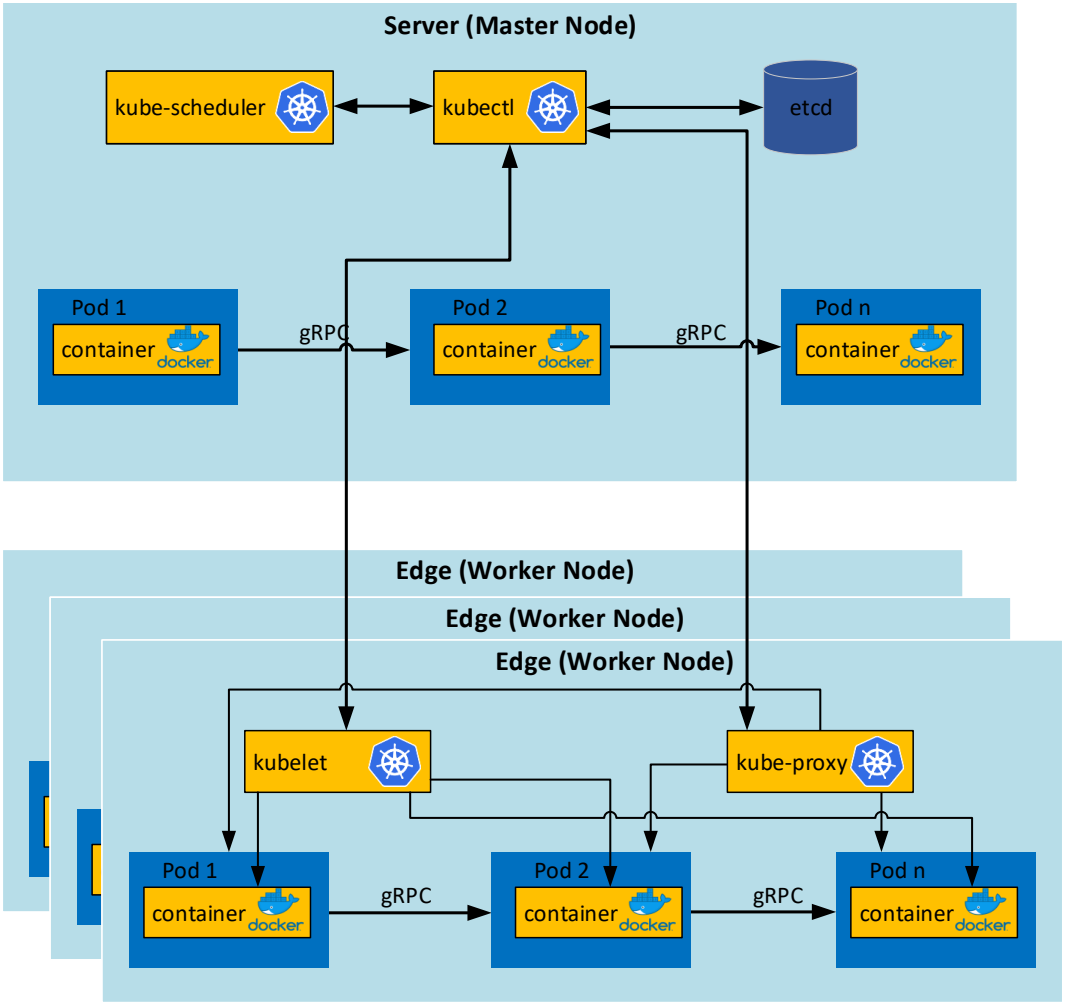

Figure 3.1: Kubernetes Approach for Distributed Computing

Each microservice is basically a pod in the Kubernetes cluster in this approach. Each pod is associated with one or more Docker containers. These containers hold the

decoupled software components from the Monolithic application. Kubernetes allows use of persistent storage volumes and also communication protocols using DNS for transferring or sharing data across containers within or across pods. The pods can be configured to run on a single node or across different nodes depending on the system architecture design. Persistent storage volumes cannot be shared across nodes as they are tied to the local network of a node. Hence, usage of communication protocols is a must for Distributed Computing. As we want to provide means to balance load between Edge and Server, there would be pods running on both these nodes; with provision for having multiple worker edge nodes in the system. Due to the ease provided by DNS, the application programmer does not need to track the IP address of each producer or consumer for data transfer. Using YAML files, architects can configure the communication path across containers with the help of service name tied to them. Not only does the DNS help for communication, it also helps in deploying containers to worker and master nodes. Again, containers can be tied to specific nodes by configuring the YAML files for Kuberenetes service and Deployment. This allows a lot of ease in Distributed Computing where jobs can be deployed to any device on the cluster with just configuring one value in YAML file. The thesis revolves around using this approach to perform Distributed AI processing on edge devices.

A communication protocol has to be established for the microservices to communicate with a basic requirement that it is compatible with Docker and Kubernetes. This thesis makes use of gRPC, Google's Open Source RPC framework that transports using HTTP2 [9]. Kubernetes and Docker have wide support for gRPC which makes it a bigger reason to choose it as the communication protocol.

As discussed, this thesis revolves around pipelined microservice approach for Machine Learning applications. One such example of a Machine Learning application is 2d Convolution. The application can be easily decoupled into convolution, ReLU and MaxPool blocks that also run as different smaller applications i.e. microservices

along with a producer block that sends image pixels to convolution block. Each of them would be containerized using Docker to be deployed into a Kubernetes cluster such that they communicate over gRPC.

3.2 GRPC for commmunication between containers

Like many RPC systems, gRPC is based around the idea of defining a service, specifying the methods that can be called remotely with their parameters and return types. By default, gRPC uses protocol buffers as the Interface Definition Language (IDL) for describing both the service interface and the structure of the payload messages. It is possible to use other alternatives if desired [9]. The use of protocol buffers allows implementation in various languages like C++, Java, Python etc. This implies that one could have a server written in C++ and separate clients written in Python and Java. This also means we can have microservices based on different technologies.

GRPC aids creation of distributed services with its client-server model where a client can directly call procedures on a server application on the same or different machine like a call to a local object. As the RPC in its name suggests, gRPC is based around the idea of defining a service, specifying the functions that can be called remotely with their parameters and return types. The server implements this interface and runs a gRPC server to handle client calls. On the client side, the client has a stub that provides the same methods as the server.

Protocol buffers [10] are a flexible, efficient, automated mechanism for serializing structured data. The protocol buffer message types are defined in .proto files where the serializing structure is specified for communiaction. Each protocol buffer message is a small logical record of information, containing a series of name-value pairs.

A sample *protobuf.proto* file is shown in listing 3.1. *GetData* here is a streaming RPC where the client sends a request *DataRequest* to the server and gets a stream to read a sequence of messages back in *DataReply*. Multiple keys can be defined during the implementation in the server and client files which could correspond to every data

request from the client and the corresponding reply be sent from the server. For the context of this example, each *DataRequest* message contains a string *key* and integer *datacount* that specify the kind and size of data to be sent. The streaming message *DataReply* is then sent by the server in bytes of data where bytes is a specific data type in protocol buffers. The bytes array is similar to a string concept and gRPC allows sending chunks of bytes as huge as 4MB in one packet which allows data communication speeds upto 100mbps.

Listing 3.1: Sample protobuf.proto file

```
syntax = "proto3";
package conv;

service GetDataService {
  rpc GetData (DataRequest) returns (stream DataReply) {}
}

message DataRequest {
  string key = 1;
  int64 datacount = 2;
}

message DataReply {
  bytes data = 1;
}
```

3.3 Containerization using Docker on NVIDIA Xavier edge devices

Docker allows a programmer to specify the software and dependencies for building a container using a configuration file called as Dockerfile. An example of a Dockerfile is shown in file *Dockerfile.server* at [11]. An interesting concept in Docker is that application images can be built on top of other base images. Line 1 tells Docker to

use grpc/cxx:1.12.0 as the base image which includes a C++ installation of gRPC tagged at version 1.12.0. COPY allows copying of software program and other files from the local machine to the container file system. WORKDIR sets the working directory that the container will run in. RUN allows execution of commands using the tool specified in the current image (in this case it is the base image grpc/cxx:1.12.0) as new layers. EXPOSE is used to expose a run-time port for the Docker container (in this case it is the port that serves a gRPC port). All this is built into an image during creation. A running instance of the container looks at the command specified along with CMD (in this case it runs the conv executable). Docker allows working with local images or one could push them onto a public or private registry in Docker Hub [12] and retrieve those images from the hub on the device where it needs to be used.

For running docker on NVIDIA Jetson Xavier, install jetpack versions 4.2.1 and higher. Docker images can be built using the base image nvcr.io/nvidia/l4t-base:r32.2.1 where L4T implies Linux for Tegra relating to the NVIDIA Tegra processor series. Unlike the grpc/cxx image, the l4t-base does not include c++ compiler and other frequently used tools, but it allows installation of these tools like one would on a linux ubuntu machine. A Dockerfile to use on Xavier or any of the Jetson boards is shown in file *Dockerfile.xavier* at [11]. It uses the l4t-base image and installs the toolchain required to compile the code with g++ and protoc compilers and installs their required dependencies line 3 through 19.

3.4 Kubernetes on NVIDIA Xavier edge devices

Kubernetes (k8s), just like Linux, has a wide list of distributions available like microk8s, minikube, k3s etc. Minikube runs a single-node Kubernetes cluster inside a Virtual Machine (VM) for users trying to get acclimated to k8s. However, it has low support for going over one node, i.e you could run a k8s cluster on one machine within a VM and cannot join other machines to the cluster since they have to be on

the VM too. However, k8s documentation relates more to minikube, so a beginner could start off with this distribution. Also, it does not have support to run on edge devices. Microk8s is a lightweight version of k8s that has stripped away some features and allows enabling them as plugins. It has been claimed to have good support for edge devices and IoT, along with support for workstations and GPGPU's. It is an excellent choice for an edge-server model, but it has been confirmed from many users that the ports on microk8s mess up the IP tables on that device and it gets difficult to debug communication between services.

K3s is another lightweight version of k8s that fits the entire binary within less than 40MB. Like microk8s, it offers a reduced feature k8s which could be added as addons on-the-go. K3s is maintained by Rancher who thrives on adding constant support and helping tools for Kubernetes and Kubeflow and hence is able to give frequent updates of k3s. The port configurations on k3s are straight forward and one could use service names to connect to server ports on gRPC. K3s provides with a convenient installation script that is configured to run k8s as a service. This service reboots after its process crashes or is killed and also if the machine reboots. K3s is built on a master worker node concept where a regular installation can be done by the command on line 1 in listing 3.2. For installing it on worker nodes, make sure they run Docker and run the command on line 2 on the worker node. Replace the *mynodetaken* with the the token retrieved on the master with the command on line 4. Replace *myserver* with the IP address of the master. An important point to note that has not been mentioned in any documentation online is to not access the cluster from the worker node, but only from the master node. Verify that the master recognizes a worker node has joined the cluster with the command on line 5.

Listing 3.2: K3s installation on master and worker

```
$ curl -sfL https://get.k3s.io | sh -
$ curl -sfL https://get.k3s.io | K3S\_URL=https://myserver:6443 \
    K3S\_TOKEN=mynodetoken sh -
$ cat /var/lib/rancher/k3s/server/node-token
$ k3s kubectl get nodes
```

3.4.1 Deployment of an Application in Kubernetes

Lets start with a small C program to print the current time every second on the console. To be able to run it in the k8s cluster, we need to write a Dockerfile for it. For this purpose, we can base it on an alpine image as shown in file *Dockerfile.time* at [11]. Alpine is a lightweight Linux distribution based on musl libc and busybox which offers images that are as small as 8MB in the Docker world. The *LABEL* field is to specify the author preceded by 'maintainer'. Next line copies the source file into the Docker container. Now that the base image and source file has been specified to Docker, the only necessary detail remaining is the tool chain. For the C file in linux we need gcc, which needs to be added onto the alpine image by the command *add build-base*. For specifying the command to run the executable, use *CMD* as shown in line 10.

Kubernetes makes use of *YAML* scripts to specify details about the workload to the cluster. Look at the file time-app-deployment.yaml at [11] for reference. The important kinds to specify in the script are *Deployment* and *Service*. Deployment is an API object that manages replication and assignment and updating of pods associated with it. The name under metadata is the name of the deployment. The number of replicas of pods that we want the application to work the time-app on is 1. The details for the image it needs to run with goes under *containers*. One could have more than one containers with distinct names associated with a single pod or replicas of that pod.

We have the source code and its Dockerfile in place. Next is to build the Dockerfile using command on line 1 in listing 3.3. To confirm that the image has successfully been built, run the command on line 2. Now that we have the image in place, make sure that k3s is running kubernetes on your node using command on line 3. The docker image created is not accessible by k8s and we need to import it into the k8s namespace using commands on lines 4 and 5. To confirm that the previous command ran successfully, run the command on line 6. The image is now accessible in the k8s namespace. To associate the image with a pod and run it on the cluster, create the deployment and service for time-app using command on line 7. Line 8 shows the command to view all pods, deployments and services active in the Kubernetes cluster.

Listing 3.3: Building a pod in Kubernetes from a Docker image

```
$ docker build -f Dockerfile.time -t time-image:latest .
$ docker images
$ k3s kubectl get nodes
$ docker save time-image > time-image.tar
$ k3s ctr -n k8s.io image import time-image.tar
$ k3s ctr -n k8s.io images ls
$ k3s kubectl create -f time-app-deployment.yaml
$ k3s get all
```

3.5 Distributed AI processing on edge

This section talks about leveraging distributed AI processing on edge with the help of Kubernetes. It has been established that k3s is an ideal candidate for this approach and the prerequisite for using this approach is the installation of Docker *v19* and higher, along with k3s *v0.9.1* installed. The version *0.9.1* has proven stable on ARM architecture which is prevalent among edge devices and hence that specific version is recommended for all the nodes on the cluster. The server node needs the k3s-server installed and edge nodes need k3s-agent installed onto them.

The monolithic code needs to be decoupled into smallest possible software components that can process the data and also communicate with other components. The motivation section reasons the use of gRPC for this communication and hence all the decoupled components need to have software that can actually send and/or receive data as per need. The resulting microservices can be deployed on the Kuberenetes cluster to run on the desired edge nodes with the YAML files as deployments and services.

3.5.1 Microservice based pipeline architecture

The microservice approach along with gRPC communication enables the pipeline architecture where the number of stages in the pipeline is the number of number of microservices in the cluster when working with streaming applications and this thesis does aim to aid the streaming applications in AI. Figure 3.2 shows how pipelining is achieved using containerized microservices in a Kuberentes cluster which lets say is agnostic of the nodes it runs the containers on.

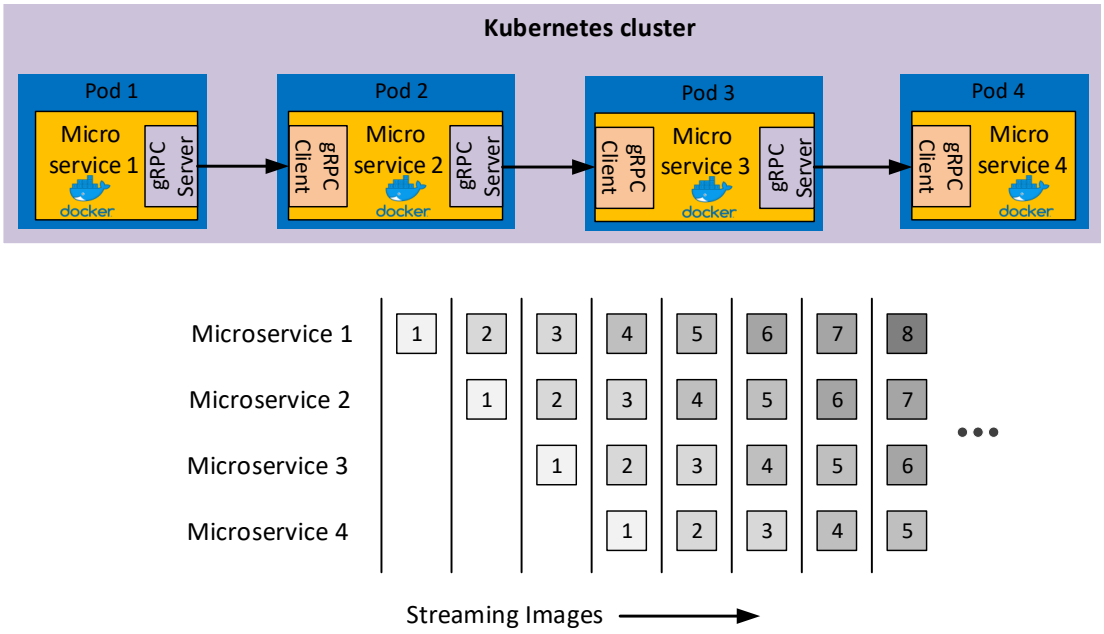

Figure 3.2: Pipelining using containerized microservices in Kuberentes cluster

Deploying each microservice as a separate pod also allows assigning a separate

service name to it. Pods 1, 2 and 3 contain implementation of gRPC server for the example in figure 3.2. The server implementation requires sourcing the remote services on specific port numbers on the local host. Client implementation in pods 2, 3 and 4 does not need to connect to IP addresses of the corresponding server but instead connect using the service name of respective deployments on the cluster. This pipeline structure can support a distributed system too due to DNS feature provided by gRPC and supported by Docker and Kubernetes where the clients just need to connect to their servers through service names.

CHAPTER 4: EXPERIMENTS AND RESULTS

4.1 Experimental setup

For evaluating and testing our system, a heterogeneous system has been setup as shown in table 4.1 with and x86 based server and ARM based edge device which is an NVIDIA Jetson AGX Xavier. Both the devices need basic tools Docker v19.03.5 and k3s Kubernetes v0.9.1 installed on them. Master (or server as known to k3s) has been installed on the Server and worker (or agent as known to k3s) has been installed on the edge Xavier device. CUDA version 10.0+ is another basic dependency to build images in docker.

Table 4.1: Experimental setup

Server CPU	Intel Xeon 32-core 64-bit CPU ES-2650
Server GPU	5120-core NVIDIA Titan-V GPU with 640 Tensor Cores
Edge CPU	ARM based 8-core 64-bit NVIDIA Carmel CPU
Edge GPU	512-core NVIDIA Volta GPU with Tensor cores

The aim is to decouple a monolithic application into microservices and run them across both the server and Xavier. A convolution application has been chosen for functionality tests and evaluating results to prove the performance in terms of latency and throughput. The code for the convolution application can be found in file *conv.c* at [11]. It can be observed from the code that it can be decoupled into four components: master, convolution, reLU and maxpool. Containerizing them would grant four microservices. A gRPC module has to be included in each of the microservices following a client-server topology between each link joining the adjacent microservices. While using gRPC, make use of *byte arrays* as the data type because other data types are inefficient in terms of speed of communication. These microservices process streaming images and send the processed image to the next microservice

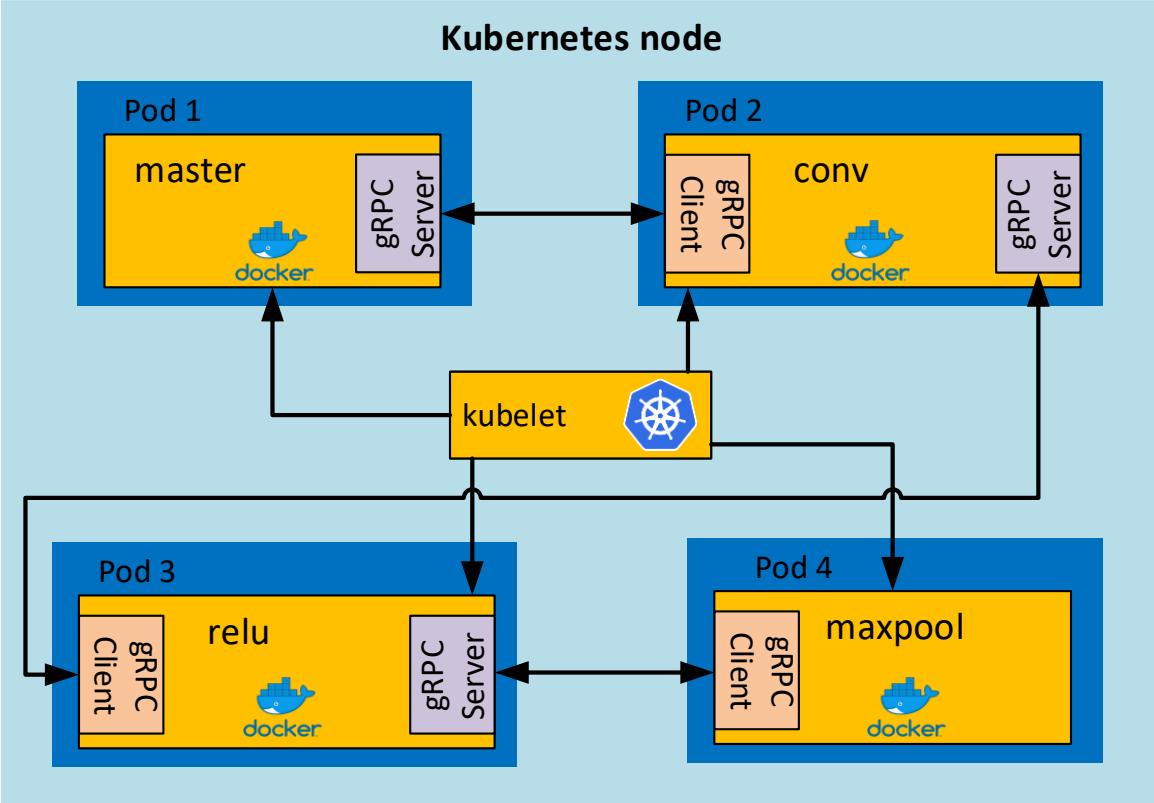

Figure 4.1: Test Setup with containerized microservices on one node

in the pipeline.

Each of the microservices access timers within and store the time taken to receive, process and send each image. The total time taken to stream 1000 images is then published as a print log. The average time taken to process one image is observed by dividing the total time into 1000 equal parts and this is used to infer latency and throughput of the system. The communication and computation time are noted too. The tests are run on various image sizes namely 128x128, 256x256, 512x512 and 1024x1024.

Above setup is replicated on both the edge and server devices. Experiments are performed first with all microservices on the server as shown in figure 4.1. Next, all microservices run on the edge as shown in figure 4.1, and the latency and through-

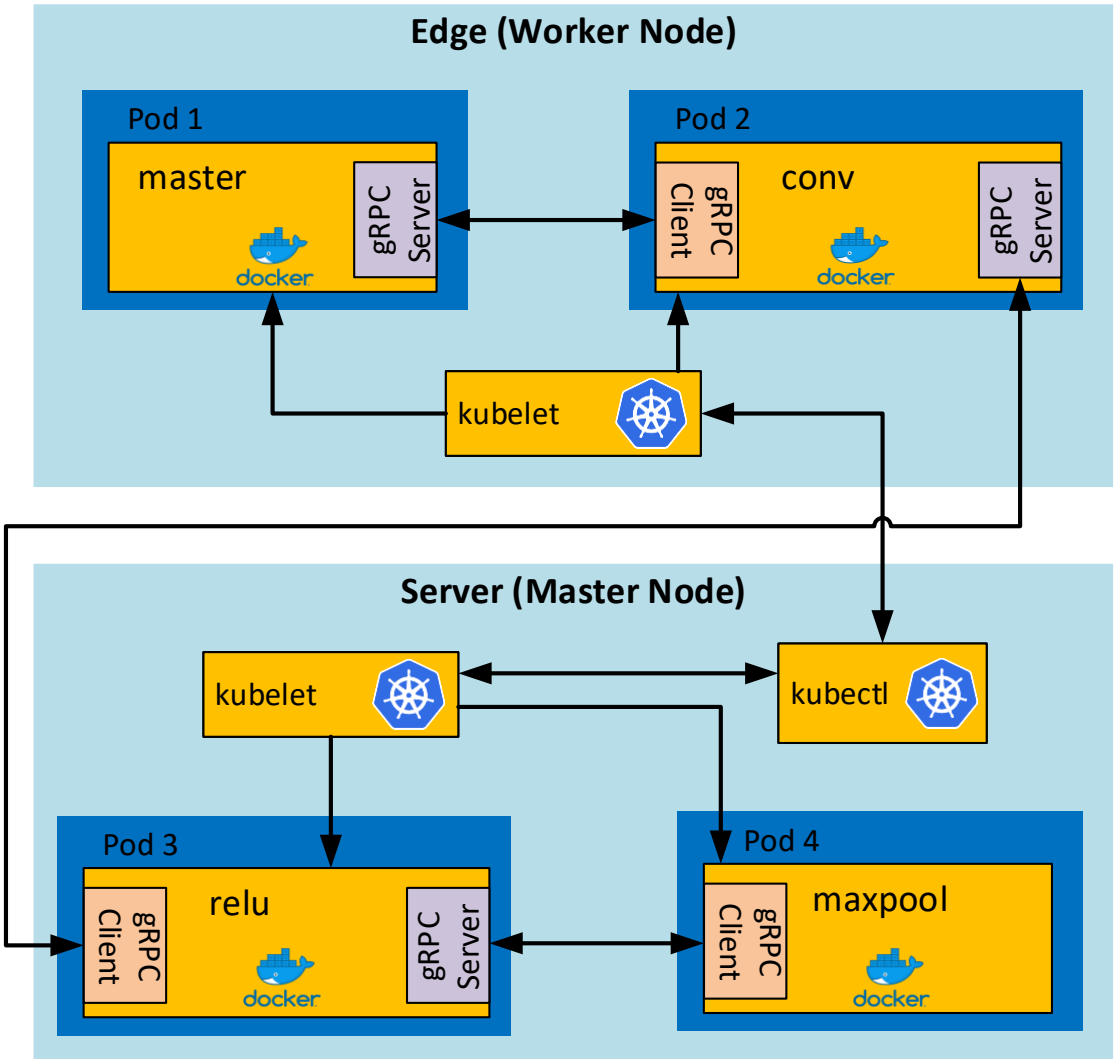

Figure 4.2: Test Setup with containerized microservices on distributed nodes

put results are noted. Further, to test how the system performs with distributed computing, microservices are run on both edge and server exclusively such that one microservice either runs on edge or on server which is the setup with distributed nodes as shown in figure 4.2. This setup shows the pipeline effect on throughput for distributed computing.

4.2 Results

4.2.1 Results on Server

These results are generated for testing out functionality and performance of microservices on the server alone as shown in figure 4.1. Figure 4.3 reflects the comparison of end-to-end latency between the monolithic and microservice approach on four different image sizes for processing one image completely. Monolithic represents the convolution application which is not containerized whereas microservice represents the convolution decoupled into four components which are containerized using Docker, communicate over gRPC and run in the Kubernetes cluster. As seen in the figure, the microservice approach worsens the end-to-end latency of the application as compared to the monolithic approach. The computation time remains approximately unchanged in the microservice approach but the communication time hits the latency and thereby lowering performance. An important observation that can be drawn from this figure is that the communication time does not increment as exponentially as the computation time does due to usage of byte arrays in gRPC. Byte arrays allow up to 4MB of data to be sent in one chunk across the channel.

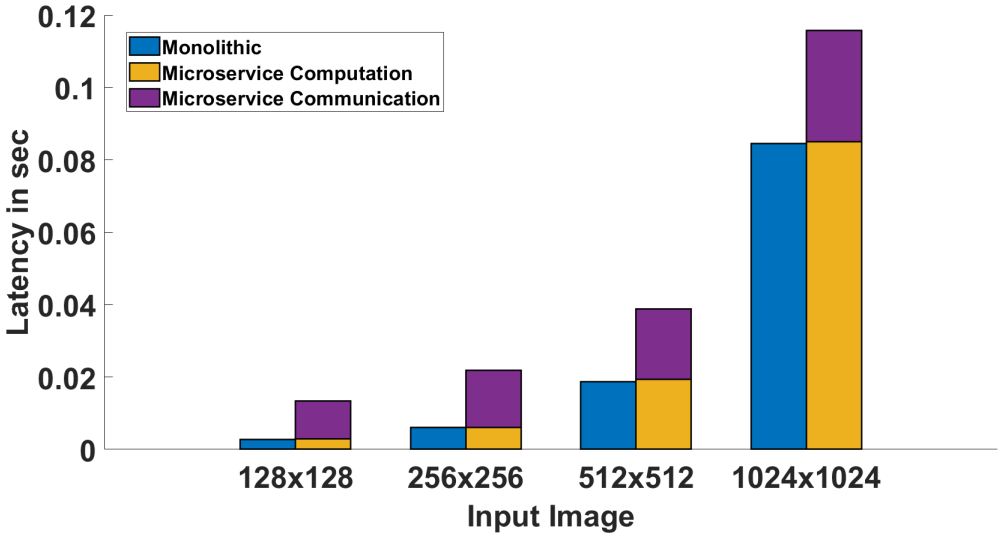

Figure 4.3: End-to-end Latency Comparison - Server

Figure 4.4 depicts the communication to computation ratio (CCR) introduced by the microservice approach for different sizes of input images. The CCR exponentially decreases as the image size increments due to exponential increase in computation time and minimal increment in communication time. CCR drops below 1 for image size of 512x512 and way below 0.5 for image size of 1024x1024.

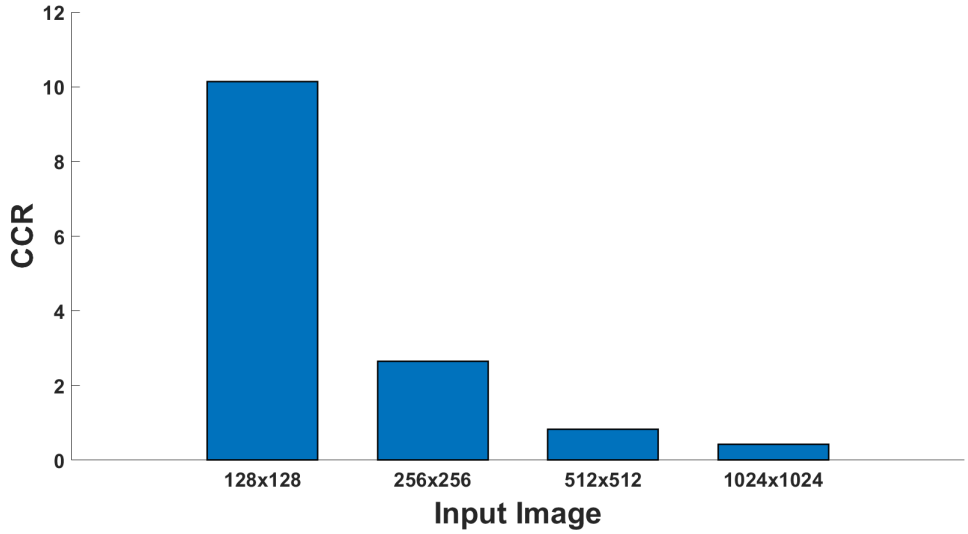

Figure 4.4: Communication to Computation Ratio - Server

Table 4.2 shows the pipeline effect on throughput as seen on the server for monolithic and microservice approaches. Throughput is being discussed in terms of frame rate for streaming 1000 images of size 720p (1280x720) while performing 2d convolution on the same. The microservice approach increases throughput of the system by 76.58% while running the application as four microservices. Figure 4.5 shows experimental results for throughput on different configurations of pipeline. 2-stage pipeline indicates that the monolithic application which can still be decoupled to four stages but would be decoupled to two stages instead. Similarly, 3-stage pipeline runs 3 microservices and 4-stage pipeline runs 4 microservices.

Table 4.2: Pipeline effect on Throughput - Server

Throughput observed for monolithic convolution	15.9 fps
Throughput observed for microservice convolution	27.9 fps
Increase in throughput with microservice convolution	76.58%

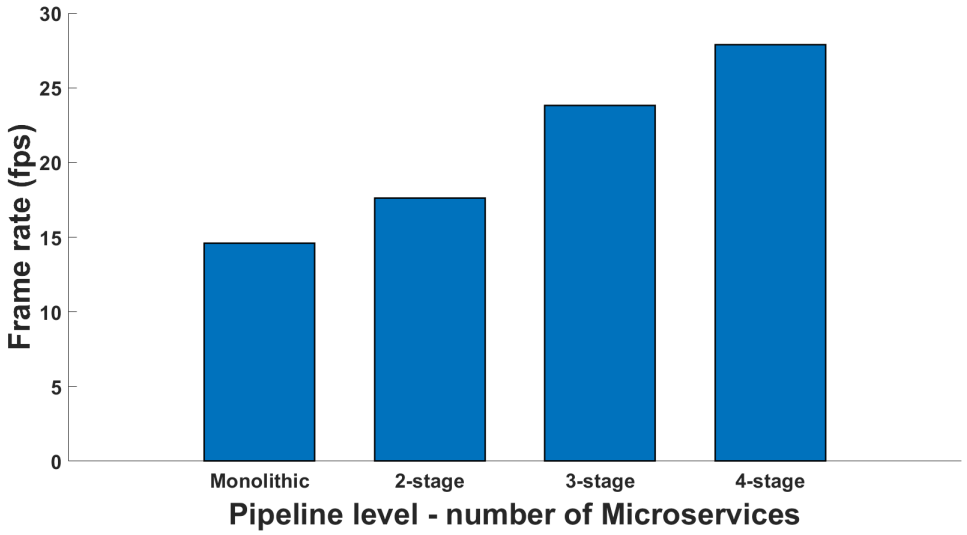

Figure 4.5: Pipeline effect on Throughput with increasing number of micrservices on server

4.2.2 Results on Edge - NVIDIA Xavier

These results are generated for testing out functionality and performance of microservices on the edge alone as shown in figure 4.1. Figure 4.6 reflects the comparison of end-to-end latency between the monolithic and microservice approach on four different image sizes for processing one image completely. Monolithic represents the convolution application which is not containerized whereas microservice represents the convolution decoupled into four components which are containerized using Docker, communicate over gRPC and run in the Kubernetes cluster. As seen in the figure, the microservice approach worsens the end-to-end latency of the application as compared to the monolithic approach. The computation time remains approximately unchanged in the microservice approach but the communication time hits the latency

and thereby lowering performance.

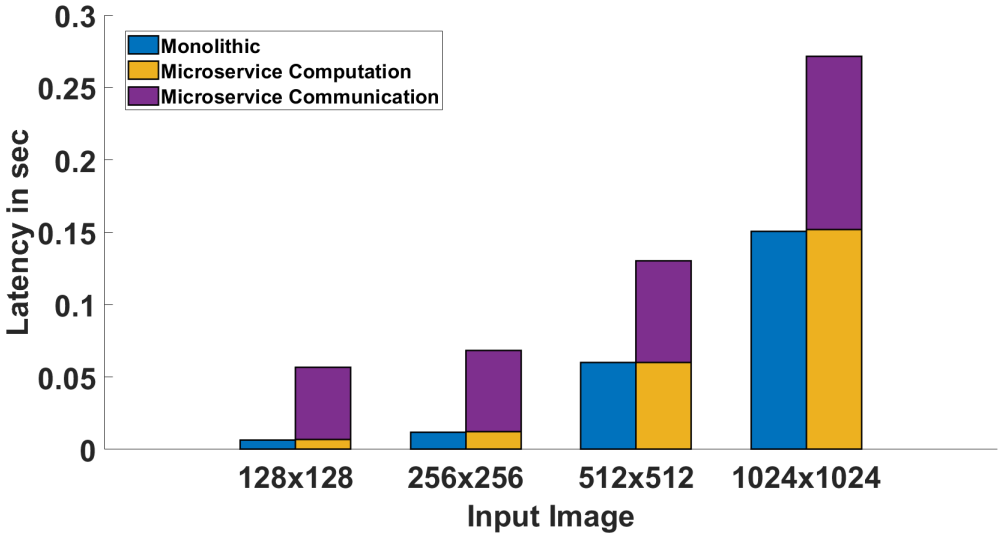

Figure 4.6: End-to-end Latency Comparison - Edge

Figure 4.7 depicts the communication to computation ratio (CCR) introduced by the microservice approach for different sizes of input images. The CCR exponentially decreases as the image size increments due to exponential increase in computation time and minimal increment in communication time up to image size of 512x512. CCR does not drop as exponential as it did for the server results because computation time does not increase as much on edge as it did on server with increment in image size.

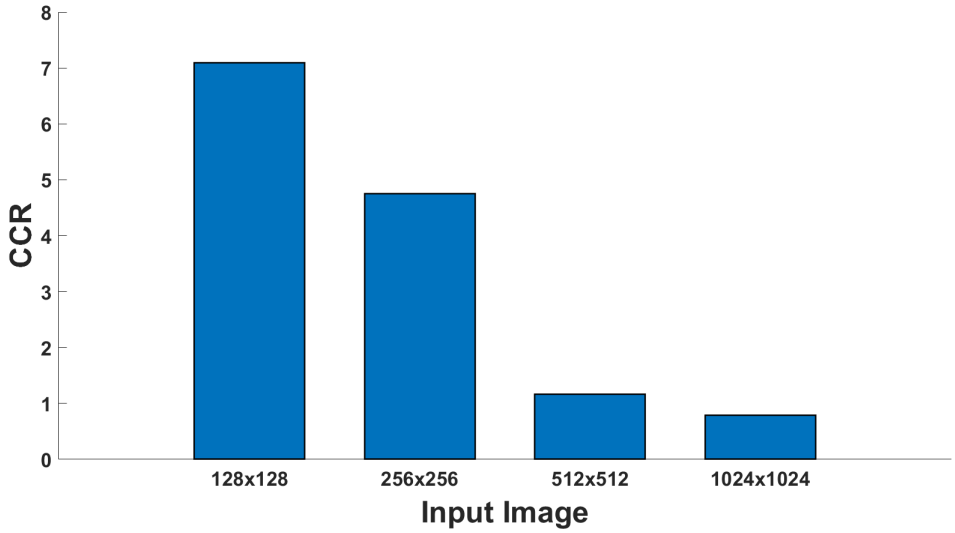

Figure 4.7: Communication to Computation Ratio - Edge

Table 4.3 shows the pipeline effect on throughput as seen on the edge for monolithic and microservice approaches. Throughput is being discussed in terms of frame rate for streaming 1000 images of size 720p (1280x720) while performing 2d convolution on the same. The microservice approach increases throughput of the system by 103.94% while running the application as four microservices. Figure 4.8 shows experimental results for throughput on different configurations of pipeline. 2-stage pipeline indicates that the monolithic application which can still be decoupled to four stages but would be decoupled to two stages instead. Similarly, 3-stage pipeline runs 3 microservices and 4-stage pipeline runs 4 microservices.

Table 4.3: Pipeline effect on Throughput - Edge

Throughput observed for monolithic convolution	7.11 fps
Throughput observed for microservice convolution	14.5 fps
Increase in throughput with microservice convolution	103.94%

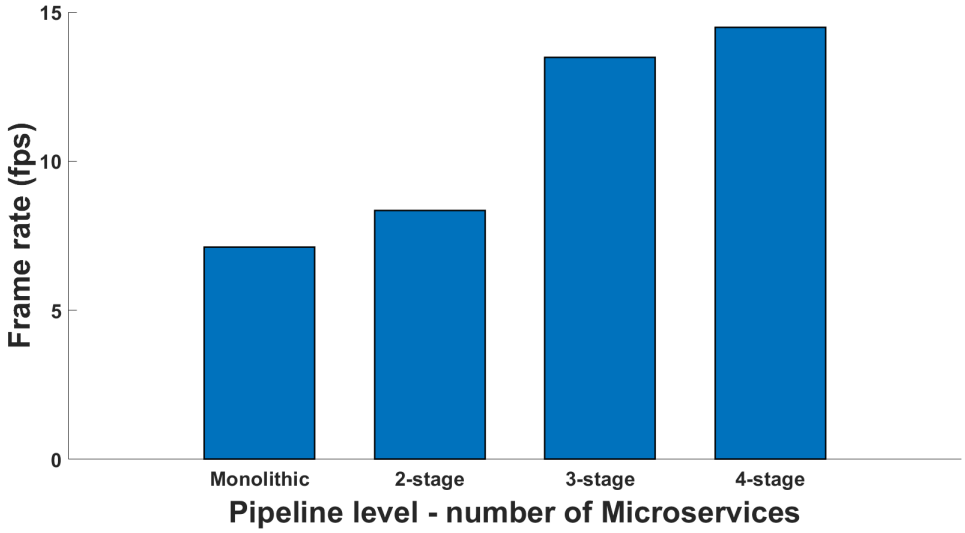

Figure 4.8: Pipeline effect on Throughput with increasing number of micrservices on Xavier

4.2.3 Distributed Computing Results - Edge Server model

Having studied the system characteristics for server and edge through these results, this section talks about results for distributed computing on the edge server model. For this section, both edge and server hosts as the node for the microservice pods as shown in figure 4.2.

4.2.3.1 Results for different Edge-Server configurations

These results are drawn while maintaining a 4-stage pipeline architecture on the edge server model. Each of the following results focus on either end-to-end latency or throughput while processing convolution on a specific input image size with 4 microservices running on the Kubernetes cluster. A key point to note is that convolution is being performed on a stream of input images which would be fed from the camera in the real world. Hence, for a distributed system, the first microservice has to run on the edge device. Also, it does not make sense to run the second microservice on server and then the third on edge, this would merely worsen the latency due to higher communication time between edge and server. For understanding the results in the

following sections this terminology has been followed:

- **SSSS**: All microservices run on Server.
- **ESSS**: Microservice 1 runs on Server while microservices 2, 3 and 4 run on Edge.
- **EESS**: Microservices 1 and 2 run on Server while microservices 3 and 4 run on Edge.
- **EEES**: Microservices 1, 2 and 3 run on Server while microservice 4 runs on Edge.
- **EEEE**: All microservices run on Edge.

4.2.3.2 Latency results for different Image sizes

Figures 4.9, 4.10, 4.11 and 4.12 show the end-to-end latency comparison across all five configurations of a 4-stage pipelined microservice architecture for convolution application as explained in the section 4.2.3.1. The results in these figures vary due to the varying image size that the microservices process and, as a result, the varying size of data that needs to be transferred across pods in the cluster. ESSS, EESS and EEES are the configurations that matter from the distributed system perspective. As can be seen, processing and data communication on the edge is far slower than processing on the server. However, through more experiments, it has been noted that there is very minimal reduction in communication speed between edge and server as compared to the communication speed between two edge devices. Hence, the latency seen on EEES and EEEE configurations is approximately the same for all image sizes.

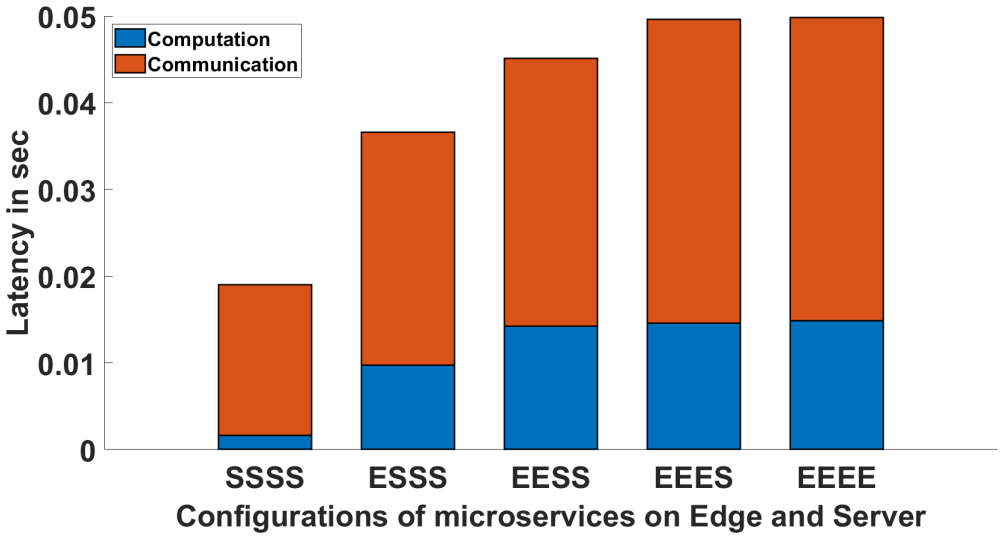

Figure 4.9: End-to-end Latency Comparison for 128x128 image

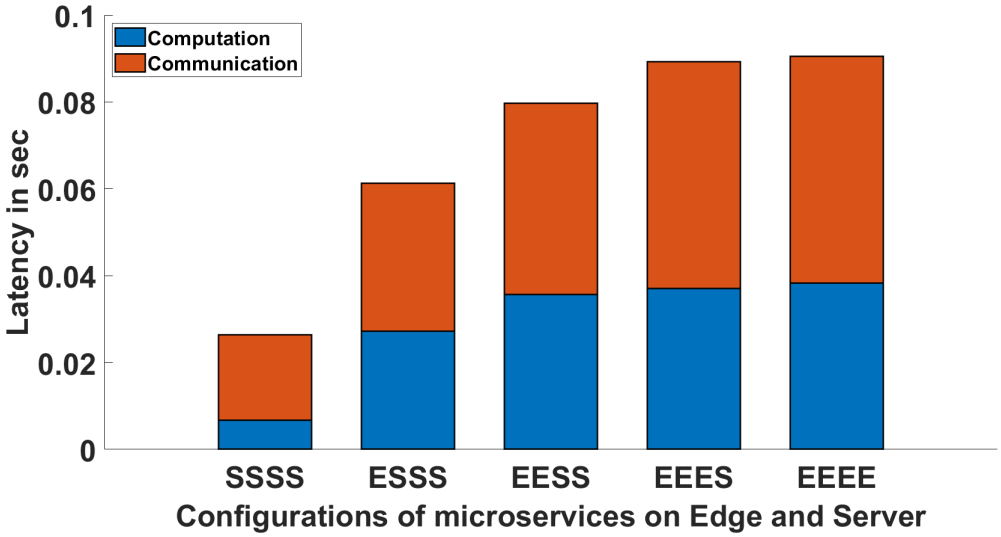

Figure 4.10: End-to-end Latency Comparison for 256x256 image

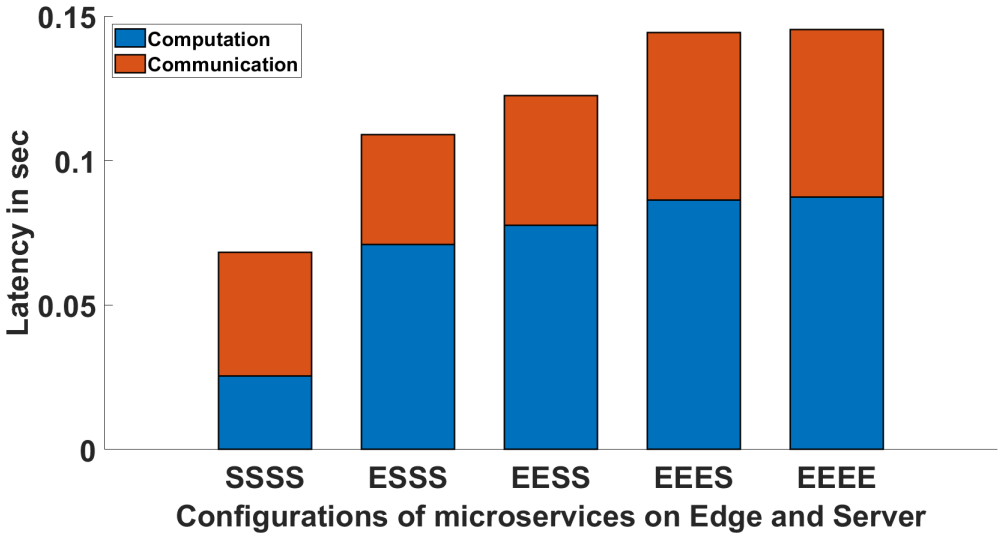

Figure 4.11: End-to-end Latency Comparison for 512x512 image

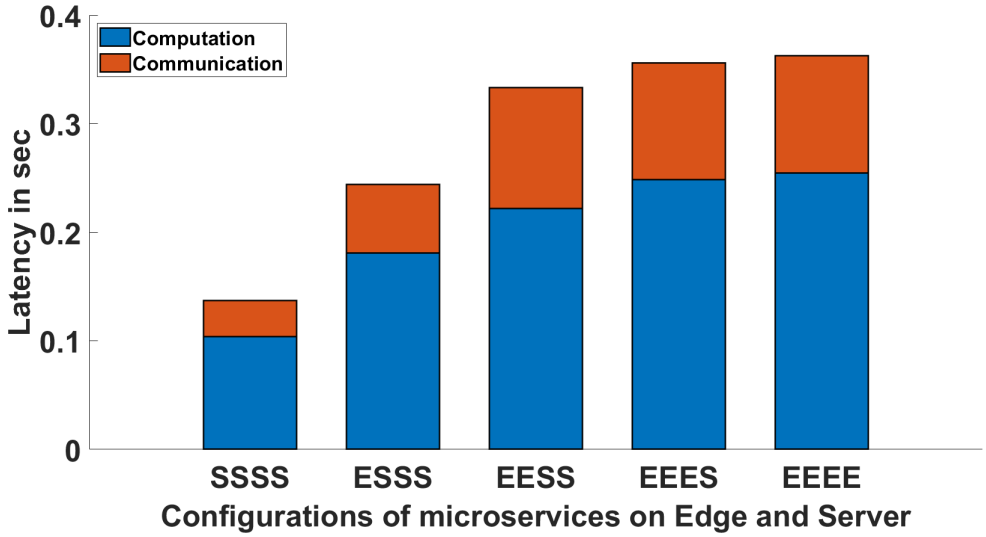

Figure 4.12: End-to-end Latency Comparison for 1024x1024 image

4.2.3.3 Throughput results for different Image sizes

Figures 4.13, 4.14, 4.15 and 4.16 show the throughput latency comparison across all five configurations of a 4-stage pipelined microservice architecture for convolution application as explained in the section 4.2.3.1. As seen before in section 4.2.3.2, the results in these figures vary due to the varying image size that the microservices pro-

cess and, as a result, the varying size of data that needs to be transferred across pods in the cluster. As the image size increases, there is less variation seen in throughput on the different configurations. Even though the convolution application can be decoupled into four microservices, it is not ideal decoupling where the middle two microservices (convolution and ReLU) are more compute intensive ones which proves to be a bottleneck for the pipeline.

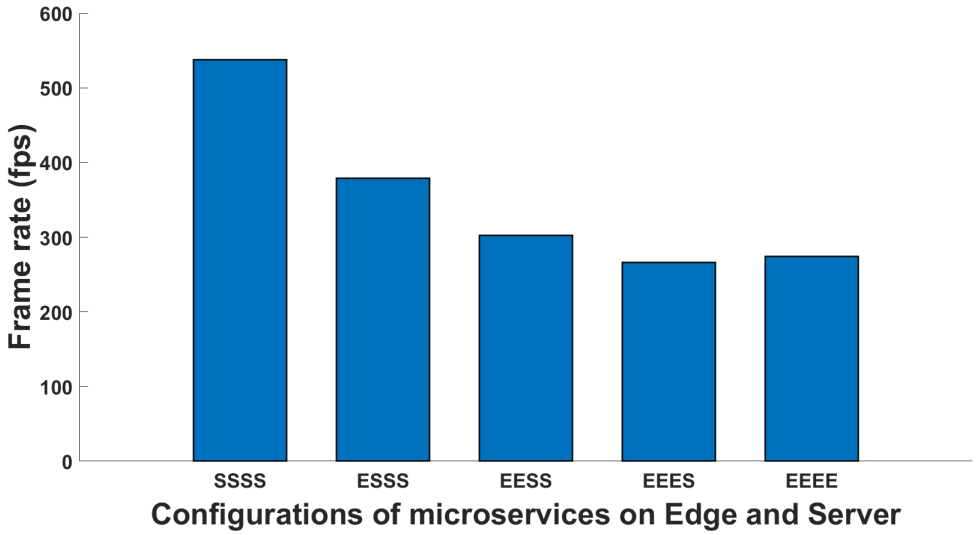

Figure 4.13: Throughput Comparison for 128x128 image

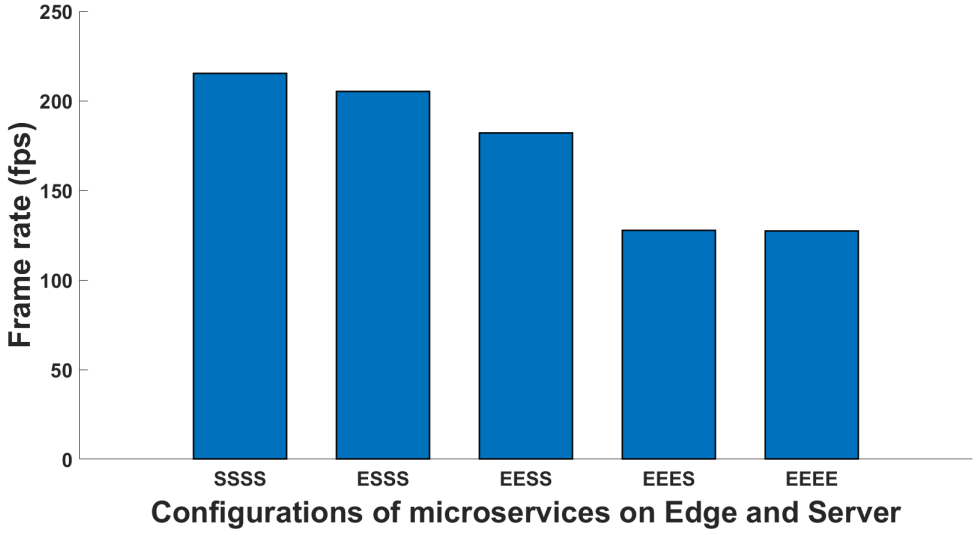

Figure 4.14: Throughput Comparison for 256x256 image

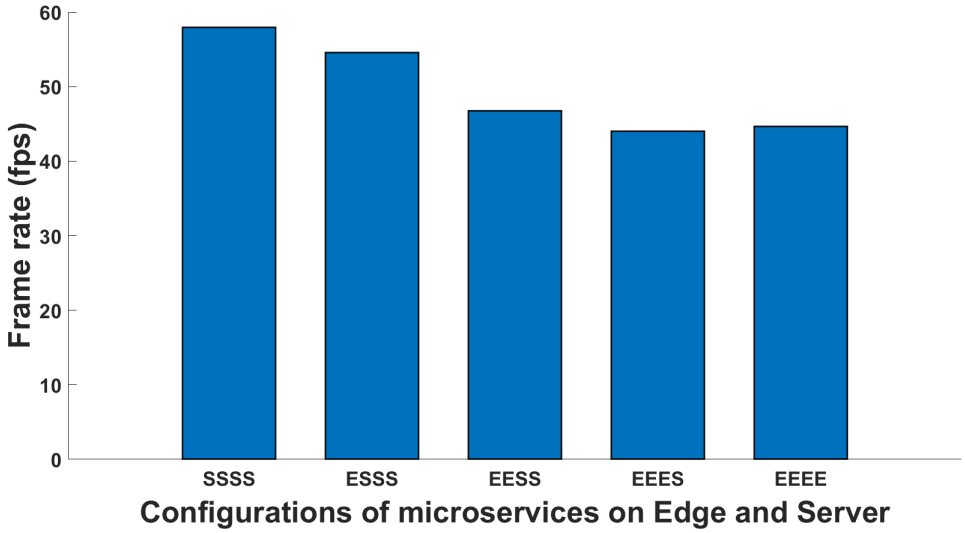

Figure 4.15: Throughput Comparison for 512 image

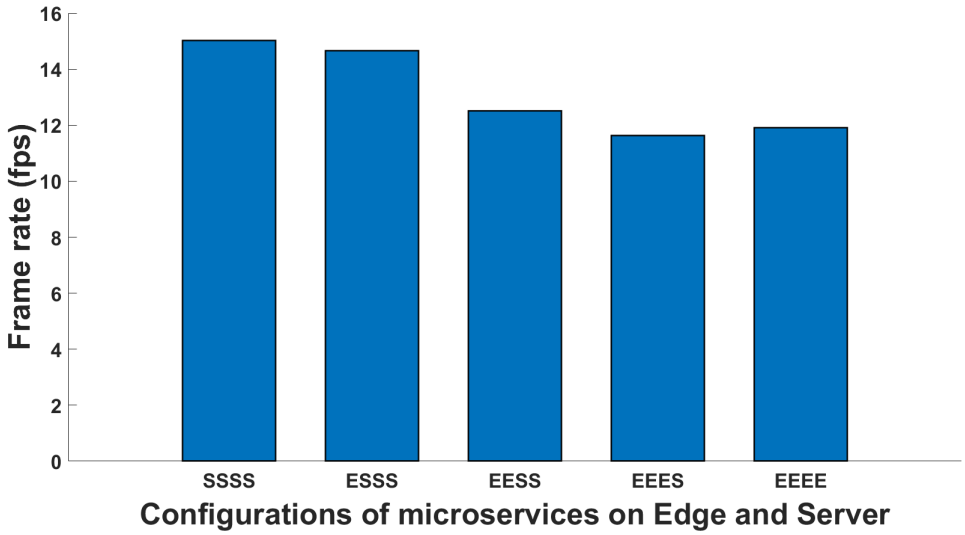

Figure 4.16: Throughput Comparison for 1024x1024 image

CHAPTER 5: CONCLUSION AND FUTURE WORK

5.1 Conclusion

This work talks about leveraging Kubernetes for distributed computing to aid AI processing for an edge server system architecture. It implicitly leverages Docker for building containers as microservices to be deployed as pods in a Kubernetes cluster. The experimental results have proven that this approach helps increase the throughput for a 2D convolution on a single node cluster. However, the bigger picture that the results cannot clearly draw is the ease provided by this approach for computer vision applications on a distributed system. It could be observed from the YAML configuration files how easily they allow a developer to associate nodes to run particular pods. It is this ease which allows exploring scalability aspect of Kubernetes for distributed systems.

5.2 Future Work

The work here has been proven to solve the problems associated with distributed systems while providing better performance in terms of throughput due to pipelining. However, it has only been proven with a toy example in this thesis. Future work involves running a real world application using the same architecture. Moreover, this architecture was built with a lot of simplicity which does not enable other important features of Kubernetes like automatic rollout updates and smart deployment of pods onto nodes so as to run them on most available node with highest computing.

REFERENCES

[1] S. Dey and A. Mukherjee, "Implementing deep learning and inferencing on fog and edge computing systems," in *2018 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pp. 818–823, March 2018.

[2] B. P. Rimal, D. Pham Van, and M. Maier, "Mobile-edge computing versus centralized cloud computing over a converged fiwi access network," *IEEE Transactions on Network and Service Management*, vol. 14, pp. 498–513, Sep. 2017.

[3] R. Bruschi, F. Davoli, P. Lago, and J. F. Pajo, "A multi-clustering approach to scale distributed tenant networks for mobile edge computing," *IEEE Journal on Selected Areas in Communications*, vol. 37, pp. 499–514, March 2019.

[4] Kubernetes, "Kubernetes." https://kubernetes.io.

[5] I. Docker, "Docker empowering app development for developers." https://www.docker.com.

[6] Z. Tao, Q. Xia, Z. Hao, C. Li, L. Ma, S. Yi, and Q. Li, "A survey of virtual machine management in edge computing," *Proceedings of the IEEE*, vol. 107, pp. 1482–1499, Aug 2019.

[7] Y. Xiong, Y. Sun, L. Xing, and Y. Huang, "Extend cloud to edge with kubeedge," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pp. 373–377, Oct 2018.

[8] L. Abdollahi Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Deploying microservice based applications with kubernetes: Experiments and lessons learned," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pp. 970–973, July 2018.

[9] gRPC, "grpc.io." https://grpc.io/docs/guides/concepts.

[10] I. Google, "Protocol Buffers developer guide." https://developers.google.com/protocol-buffers/docs/overview.

[11] R. Raheja, "Enabling kubernetes for distributed ai processing on edge devices." https://github.com/RonakRaheja/kubernetes-distributed-edge-processing.

[12] I. Docker, "Docker hub." https://hub.docker.com.