MULTICLOUD EDGE GATEWAY FOR IOT COMPUTER VISION

by

Samantha Luu

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

2022

Approved by:

_____

Dr. Arun Ravindran

_____

Dr. Hamed Tabkhivayghan

_____

Dr. Ronald Sass

ABSTRACT

SAMANTHA LUU. MultiCloud Edge Gateway for IoT Computer Vision. (Under the direction of DR. ARUN RAVINDRAN)

The last decade has witnessed tremendous advances in cloud computing, IoT, and computer vision. IoT computer vision brings powerful capabilities that enables society to tackle complex problems such as autonomous driving vehicles, smart cities, public safety, and interactive healthcare. However, the field faces challenges of large data streams, complex processing, low latency requirements, and data privacy concerns. Processing at the Edge vastly reduces the data that needs to be sent to the cloud (by a factor of 1000). Additionally, Edge processing results in lowered application latency and sensitive video streams are confined to the privacy perimeter of the end-user (for example, homes, hospitals, etc.). However, current IoT system software infrastructures are designed for low data rate sensor applications and do not satisfy the demanding needs of computer vision-based IoT.

In this thesis, we design and implement an Edge gateway targeted specifically at emerging IoT computer vision applications. The proposed Edge gateway, which we call VEI, enables realization of multiple vision algorithms at the Edge from a single camera stream. Furthermore, unlike existing Edge gateways, VEI is vendor-neutral, and capable of connecting to any Cloud provider. This allows for increased application resilience, lowers costs, and avoids Cloud vendor lock-in. We experimentally evaluate the performance of VEI for canonical object detection applications. Public clouds considered in this work include those from Amazon (AWS) and Google (GCP).

DEDICATION

To my family Sharon Go, Don Luu, Chris Go, and Grace Go Luu.

ACKNOWLEDGEMENTS

I would like to acknowledge and thank Dr. Arun Ravindran for advising and preparing me for my career throughout my 5 years at UNCC. Thank you for training me through this research. Thank you for guiding me through every step and providing the Cloud Native Applications course that change my career goals for the better.

I would like to thank my committee members, Dr. Sass and Dr. Tabhki for not only joining my committee, but for also sharing insight and helping me understand researching in different perspectives.

Finally, I want to acknowledge and thank my family and friends for their infinite amount of love and support throughout my degree.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

API Application Programming Interface

AWS Amazon Web Services

CDF Cumulative Distribution Function

CLI Command Line Interface

CPU Central Processing Unit

CSP Cloud Service Provider

GCP Google Cloud Platform

GPU Graphics Processing Unit

gRPC Google Remote Procedure Call

HTTP Hypertext Transfer Protocol

IoT Internet of Things

JSON JavaScript Object Notation

MQTT Message Queuing Telemetry Transport

RAM Random Access Memory

RPC Remote Procedure Call

SDK Software Development Kit

TLS Transport layer Security

VEI Visual Edge IoT

YOLO You Only Look Once

CHAPTER 1: INTRODUCTION

The last decade has witnessed tremendous advances in the fields of cloud computing and computer vision. Cloud computing makes available on-demand and highly salable computing and storage with a pay-as-you-go model [1]. Deep neural networks with millions of parameters have enabled computer vision to surpass humans in certain tasks such as object detection [2]. Furthermore, advances in wireless networks such as 4G and 5G have ushered in Internet of Things (IoT) applications, where events regarding real-world activities from non-human "things" are sensed, transmitted to the Cloud, persisted in highly scalable data stores, processed using powerful machine learning algorithms, and results transmitted to users via smartphones. An example of such an IoT application is fleet tracking in transit systems to provide passengers with accurate delivery information. With computer vision allowing complex sensing of the environment, the events are sensed as video streams using Internet-enabled cameras. Vision processing allows more powerful applications in a variety of areas including transportation (for example, autonomous vehicles), healthcare (for example, continuous patient monitoring), smart cities (for example, pedestrian safety), and many others [3].

While IoT computer vision brings powerful capabilities that enables society to tackle complex problems, it is beset by technical and social challenges. On the technical side, the amount of data generated by always-on cameras is vast (on the order of 1 TB of data per camera per day). Transmitting this "big data" to the Cloud is expensive and overwhelms network capacity. On the social side, are privacy concerns in transmitting potentially sensitive video data outside the privacy perimeter such as homes, hospitals, and industrial facilities to a distant data center. Edge comput-

ing, where computing is done at or near the edge of the network, has emerged as a promising technology for addressing these concerns. Processing at the Edge vastly reduces the data that needs to be sent to the Cloud (by a factor of 1000). Additionally, processing at the Edge results in lowered application latency through reduced data transmission. Furthermore, sensitive video streams are confined to the privacy perimeter of the end-user [4]. However, the current IoT system software infrastructure is designed for low data rate sensor applications. Edge gateways provided by Cloud vendors (for example, AWS Greengrass [5]) are only able to handle message sizes of 128 KB [6]. Note that a single raw video frame is at least 1 MB in size. An Edge gateway capable of handling the large data sizes associated with computer vision is thus needed. Due to the high costs associated with installing cameras, often multiple vision applications may consume a single camera stream for detecting different events. The Edge gateway must be capable of supporting multiple vision applications. Furthermore, to respond to dynamically changing Cloud service availability, the Edge gateway must be able to readily switch between Cloud Service Providers.

In this thesis, we design and implement an Edge gateway targeted specifically at emerging IoT computer vision applications. The proposed Edge gateway, which we call Vision Edge IoT (VEI), enables the realization of multiple vision applications at the Edge, processing video stream from a single camera. The proposed architecture is scalable to multiple cameras. Furthermore, unlike existing Edge gateways, VEI is vendor-neutral and capable of connecting and dynamically switching between any Cloud provider. We experimentally evaluate the performance of VEI on an Edge server for canonical object detection applications, and utilize multiple Cloud Service Providers.

1.1 Contributions

The thesis makes the following contributions -

- Demonstrates the need for Edge processing for IoT computer vision.

- Proposes VEI - an Edge gateway architecture that enables multiple computer vision algorithms to consume on a single video stream.
- Experimentally evaluates the latency, and resource usage associated with VEI.
- Demonstrates the ability of VEI to scale with multiple vision algorithms.
- Demonstrates the ability of VEI to dynamically switch between Cloud providers, and evaluates the delays incurred in the switch.

1.2 Thesis organization

The thesis is organized as follows - In chapter 2 we provide a brief background of the technologies investigated in the thesis. In chapter 3, we describe VEI, the proposed multi-Cloud Edge gateway for IoT Computer vision. In chapter 4 we present experimental evaluation of VEI. Chapter 5 concludes the thesis and includes suggestions for future work.

CHAPTER 2: BACKGROUND

The following chapter will go over relevant technologies and terminologies that were used in development.

2.1 Computer Vision

Computer vision is a popular technology that has various applications from facial recognition to object detection. The field of computer vision studies different computational methods that can take images and videos to have a system think similarly to that of a human. It is a popular field that stems from artificial intelligence that utilizes both machine and deep learning to better itself [2]. Computer vision requires a lot of data and training for the system to learn and become reliable. For example, using computer vision for facial expression applications will have to run through thousands of images before becoming accurate enough to quickly identify a face. In one assessment, the total number of images used was 35887 of different facial expressions in various settings, lights, and models [7].

Since computer vision is very versatile, the most difficult component is ensuring that it has completed enough training. If the system does not receive enough data, then it could be the creator's fault that it is not accurate enough. Specifically, problems can include image tagging, object, and event identifications in video data [8]. Using fast data-collecting technology, like IoT, can help train computer vision applications and make them more reliable. There are many applications where the data used to train is a live feed, such as cameras to ensure the visually impaired navigate safely [9]. Computer vision systems thus needs to handle large quantities of data, in addition to high processing requirements. Furthermore, for use cases involving real-time sensing

and decision-making, the systems have to provide low latency as well.

2.2 Cloud Computing

Utilizing the Cloud has become increasingly popular and advantageous as it offers a variety of functions. Cloud computing provides on-demand services and resources that are accessed through the Internet. These providers are Cloud Service Providers (CSPs) that will offer various services including infrastructure, platform, software, and backend as on-demand services. Using these services provides us with the advantage of scaling, increased performance and efficiency, and cost savings. The global Cloud computing market size was valued at USD 368.97 billion in 2021 and is expected to expand at a compound annual growth rate (CAGR) of 15.7% from 2022 to 2030 [1].

There are three different types of cloud computing that are offered: public, private, and hybrid. Public clouds are those provided by a third-party company, for example, AWS from Amazon, GCP from Google, and Azure from Microsoft. Private clouds are those built and owned by an organization that focuses on their internal uses. Hybrid clouds combine public and private clouds to provide on-demand services and data centers [10]. In this thesis, we utilize two public clouds: Amazon Web Services (AWS) and Google Cloud Platform (GCP).

2.2.1 Amazon Web Services

AWS has over 200 services that provide resources and tools to help with machine learning, computing, processing, IoT, etc. The services used and mentioned in this thesis are IoT Core and AWS Greengrass. This is a service that allows IoT devices to be connected to the cloud for communication, processing, or data storage. AWS allows console, programmatic, and command-line interface (CLI) to access services that are in use on a device [11].

IoT Core is the service used to connect IoT devices to the Cloud. Using IoT Core provides a means of communication between multiple devices or between a device

and AWS. These communications can be used for sending data to the Cloud for processing, querying, and updating device statuses. Billions of various messages can be sent through IoT Core as long as the device is connected to the Internet. Messages from IoT core can be sent to other services within AWS for further processing or storage. The device connections and the data sent to and from AWS are secured with mutual authentication and end-to-end encryption. It supports both MQTT and HTTP communication protocols sent over the Transport Layer Security (TLS) [12] [13].

AWS Greengrass is an Edge gateway service that is used to build, deploy, and manage device software. Using Greengrass makes using edge processing easier as AWS provides some template components of device software to watch for any anomalies on the devices [5]. Greengrass can also operate independently on the Edge device even during intermittent connection outages.

As a use case, a large number of sensors that track the temperature throughout the day. Every hour, the data will be sent to AWS using IoT Core. With each sensor sending data to AWS, using Greengrass would allow processing if the network is unreliable. If the sensors needed to be updated, having all of them in a group makes sending updates simple and easy.

2.2.2 Google Cloud Platform

GCP is another well-known public Cloud service provider that has extensive services including those for computer vision, artificial intelligence, and machine learning [14]. The services used in this thesis are Google IoT Core and Google Pub/Sub.

GCP has services that are very similar to AWS, like IoT Core. GCP IoT Core is the same as AWS' where it is used to allow secure connections for IoT devices to connect to the Cloud. GCP IoT Core can manage and ingest data from the connected devices and can connect with other services on GCP [15] [16]. However, unlike AWS IoT Core, the service is not used for messaging. GCP has a separate service for messaging

called Pub/Sub.

Pub/Sub is GCP's messaging platform that uses MQTT or HTTP as its communication protocol. GCP IoT Core connects with Pub/Sub so that devices can send statuses and data to other devices or GCP for further processing. For IoT devices to send messages to Pub/Sub, they need to utilize a gateway, which is specific to GCP. The gateways are used by the MQTT bridge to connect and enable messaging between GCP and the IoT devices [17]. To send data to GCP, the device will first connect to the gateway, then to GCP IoT Core, and then to Pub/Sub. The Pub/Sub service will be where messages converge and republished to other devices or used in other GCP services [16] [18].

2.3 Edge Computing

Edge computing allows for processing to occur at the edge of the network near data sources and thus facilitates low latency processing [19]. The number of devices at the edge of the network has rapidly increased within the last decade, and so has the amount of data produced. Cloud computing has been a reliable source for data processing and the computing power is constantly improving, however, the amount of data that is being sent to the Cloud for processing has caused network congestion issues. Sending an intense amount of data to the Cloud for any service would cause higher latency, decreased efficiency, and increased network pressure [20].

As an example, consider a traffic light system that is constantly aware of vehicles and pedestrians at an intersection. The system will need to measure the distances and speeds of incoming traffic and consider the pedestrians waiting to cross the street. The system will need to be able to stop pedestrians from jaywalking, provide warning signs for incoming traffic, and constantly learn to improve its warnings for safety [20]. Performing the associated computations at the Edge ensures faster results to make real-time decisions. The need for fast and real-time data processing can be used in web applications, smart homes, smart vehicles, and health data management [21].

2.4 Internet of Things

Internet of Things (IoT) are devices that are connected to the Internet and to other devices that we can monitor remotely. It has provided a way for any device to be turned into a "smart device". When we hear about smart devices, we automatically think of something that can be controlled with our voices or phones. IoT offers considerable versatility in that any device can be an IoT device, such as thermostats, cars, ovens, and refrigerators. The device itself must be connected to the Internet. From there, it can be connected to an app on a smartphone, a computer, or to the Cloud [3] [22].

There are four layers to IoT: sensor, gateway, processing, and front-end application. The sensor layer is the physical device that is collecting or producing any data. The gateway layer is responsible for moving the sensor data to the processing layer. The data is then processed at the processing layer and useful information is extracted (analytics). Finally, the front-end application layer is where the user will interact with the data. An example would be a busy parking deck that has sensors in each space to detect parked cars. When a parking space is taken, the sensor will collect that data and send it through the IoT layers to let the end-user know via a smartphone app about available parking spots [23].

CHAPTER 3: MULTICLOUD EDGE GATEWAY

In this chapter, we present the design and implementation of the MultiCloud Edge gateway for IoT computer vision applications. As stated in Chapter 1, current Edge gateways are targeted toward low-bandwidth applications. Computer vision applications at the Edge are characterized by big data, high bandwidth, and have the need for low latency. Furthermore, Edge gateways need to be able to handle multiple computer vision algorithms from a single or multiple camera streams. Finally, we aim to make the Edge gateway interoperable with various Cloud Service Providers with the ability to switch between them.

3.1 Motivation and Related Work

We investigate the suitability of current Cloud IoT infrastructures for computer vision applications. Amazon AWS and Google GCP are among the leading Cloud Service Providers. To facilitate IoT applications, AWS provides two services - AWS IoT Core which runs on the AWS Cloud, and AWS Greengrass, which is an Edge gateway running on the Edge server. Data produced at the Edge is transmitted to the AWS IoT Core either directly, or via AWS Greengrass. Use of Greengrass allows applications to tolerate intermittent network connectivity. From the IoT core, data can readily be transmitted to any AWS service, as mentioned in Chapter 2. Since the current IoT applications involve low bandwidth sensor data (for example, temperature readings), the maximum message size supported by AWS IoT Core (and AWS Greengrass) is 128 KB [6]. Similarly, Google Cloud IoT supports message sizes of 256 KB. A single raw video frame is around 1 MB in size, and thus cannot be handled by existing Cloud IoT infrastructures. Two alternatives exists, with the first

is to upload the video frames to a Cloud object storage such as AWS S3, or Google object storage, for further processing by vision algorithms running on the Cloud. However, use of object storage incurs high costs and latency of 24/7 transmission of large data frames to the Cloud and any other services that would be necessary for processing. The second alternative is to do the vision processing at the Edge, and transmit the results (for example, list of objects detected) to the Cloud. However, scaling to multiple vision applications that consume a single video stream requires an Edge gateway that provides a pub/sub messaging system. As mentioned previously, existing Edge gateways such as AWS Greengrass are limited to small data sizes. The VeerEdge proposed by Dayalan et. al. [24], builds an abstractions over existing CSP Edge gateways to make the Edge vendor-neutral, however it still suffers from the same data size limitations as existing Edge gateways.

3.2 Requirements

Our proposed Edge gateway for vision applications, called VEI (Vision Edge IoT) needs to meet the following requirements -

- Needs to handle large message sizes, corresponding to the size of individual video frames
- Should minimally contribute to overall latency
- Needs to enable multiple vision applications with low resource overhead
- Should be able to tolerate intermittent connectivity to the Cloud
- Should support multiple Cloud Service Providers and dynamically switch between them fairly rapidly
- Should be secure from unauthorized data access

While many of the requirements stated above are readily apparent, we elaborate on the need to support multiple vision applications and multiple Cloud Service Providers.

Consider the use case of a Department of Transportation for a large city that seeks to alleviate traffic during busy hours by installing cameras in multiple traffic intersections. They would want to stream the data from all the cameras for processing at the Edge and generate real-time results for fast decision-making such as rerouting traffic or changing the traffic light duration in real-time. Since camera installations are expensive, other departments in the city could also use the camera data for purposes such as public safety. Different vision algorithms would thus be consuming the same camera data to generate different data.

The need to be able to interface the Edge gateway to multiple cloud providers is needed to (a) prevent vendor lock-in, (b) take advantage of dynamic pricing models in the Cloud, (c) increase resilience in case of unavailability of a particular Cloud, and (d) support the ability of different vision applications running at the Edge while connected to different Cloud Service Providers.

3.3 APIs

We first designed the APIs that VEI needs to support. The three APIs that VEI supports are listed below.

- PublishImage
  - Inputs: Topic and Image Stream
- SubscribeImage
  - Input: Topic
  - Output: Image Stream
- PublishToCloud
  - Inputs: CSP, Topic, and Data Stream

The *PublishImage* API is invoked by the camera to send a stream of video frames to VEI. The topic identifies the particular image stream, which is typically the camera ID. The *SubscribeImage* API is invoked by the vision application to subscribe to a particular image stream identified by the topic. Note that a particular application may subscribe to multiple topics corresponding to multiple cameras. The *PublishToCloud* API is invoked by the vision application to publish the output of the vision processing to a particular Cloud Service Provider (CSP) with the data stream identified by a topic name (for example, the application name).

3.4 Architecture

The VEI architecture is composed of three main components: the pub/sub system, a data store for persistence, and the multicloud mux. Figure 3.1 provides the architectural block diagram of VEI (shaded in green), along with the external clients (camera, vision processing algorithm, and Cloud Service Providers).

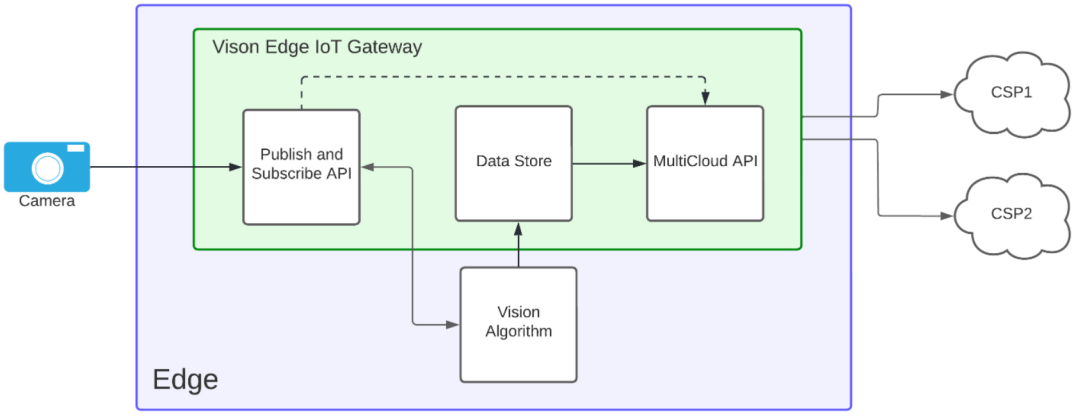

Figure 3.1: VEI architecture. VEI components are shown in the green box. The camera, VEI, and the vision application all physically reside on an Edge server.

3.4.1 Pub/Sub System

A pub/sub system is a type of messaging system that implements a publish-subscribe pattern of communication between data producers and consumers. A producer publishes data to a particular topic. At least one consumer subscribes to pull

data from one or more topics. A pub/sub system decouples producers from consumers. Producers are not aware of the consumers subscribing to the topic they publish to. If the producer or consumer fails, the pub/sub system buffers the data (often in persistent storage), so that operations can resume when the publisher or subscriber is restored.

3.4.2 Data Store

The data store can persist the data generated by the vision processing application. This data can optionally be consumed by an analytics application at the Edge or could be streamed to a Cloud Service Provider for further processing. The persistence provided by the data store allows VEI to recover data following a crash and prepare for intermittent loss of connectivity to the Cloud. Note that if the pub/sub system provides persistence, the data store is optional, unless specialized querying patterns for Edge analytics needs to be supported. While the choice of the data store depends on the analytics use case, a time series database would be a good choice for many IoT analytics applications.

3.4.3 MultiCloud Mux

The multicloud mux implements the *PublishToCloud* API. Essentially, the multicloud mux is a wrapper around different Cloud service APIs. The mux establishes a connection to the specified CSP, performs authentication, and starts transmitting data. When switching to another CSP, the mux aborts the connection of the existing CSP, establishes a new one with the new CSP, and starts streaming data. The Cloud service APIs are invoked programmatically using the programming language-specific Software Development Kits (SDKs).

3.5 Implementation

The following subsections will go over how we implemented VEI using specific open-source resources.

3.5.1 gRPC

To implement APIs, we needed a fast communication protocol that could handle the large data size of images. Remote Procedure Calls (RPC) is a high-level communication protocol where calls are made between a client and a server. These calls mimic local function calls. We specifically used gRPC, which is an open-source RPC framework from Google that is language-neutral and allows for highly efficient communication. Figure 3.2 provides an example of a gRPC server and client with two clients programmed in different languages than the server.

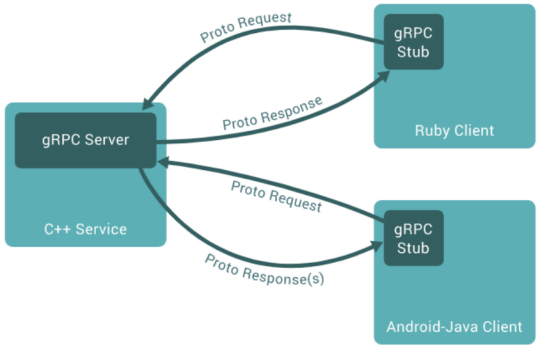

Figure 3.2: Example gRPC Block Diagram

gRPC was chosen because of its ability to stream data and its support for authentication [25] [26]. It is imperative that the technologies used in the Publish and Subscribe APIs have streaming abilities as it results in lower latency as compared to individual pushing and pulling of data.

3.5.2 NATS

We utilized NATS for the pub/sub system. NATS is an open-source messaging infrastructure that is designed for high bandwidth applications. Figure 3.3 is a basic example of the NATS publish/subscribe system using topics, also known as subjects.

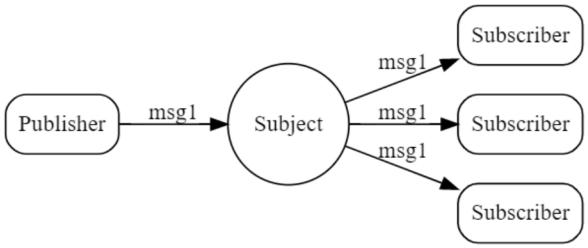

Figure 3.3: NATS Example

The *PublishImage* API internally invokes the NATS Publish API via a NATS client. As shown in Figure 3.4, the NATS client publishes to the NATS server under the named topic. Similarly, the *SubscribeImage* API invokes the NATS Subscribe API via a NATS client, and subscribes to images from the named topic on the NATS server.

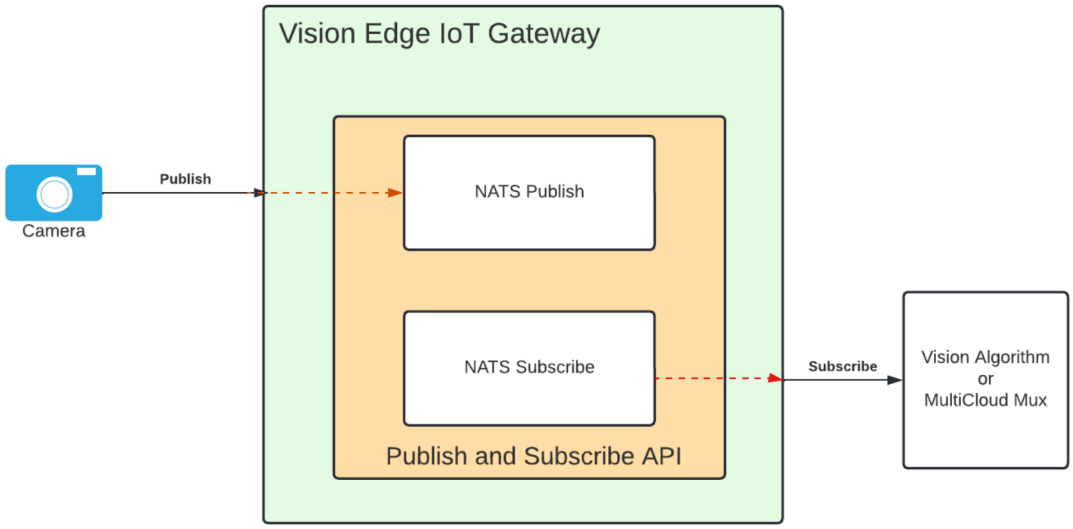

Figure 3.4: NATS Publish and Subscribe

NATS supports message persistence via file store on the Edge server storage. This allows the application to snapshot important images, and recover images after crash failures.

3.5.3 Data Store

In our implementation, we did not use a data store, as NATS provided built-in persistence. As stated previously, the data store is optional if the the pub/sub system provides persistence. However, if we were to implement a data store, we would have used the open-source time-series data store, InfluxDB, due to the time-series nature of video frame analysis data generated by the vision application. InfluXDB is a NoSQL datastore, where different time-series data streams are organized as buckets, queried via InfluxQL - a SQL-like query language. InfluxDB persists data on the Edge server storage medium for crash recovery. On loss of connectivity to the Cloud, the *PublishToCloud* API could retransmit the missing data fetched from InfluxDB. Figure 3.5 shows the diagram of the implementation without the data store.

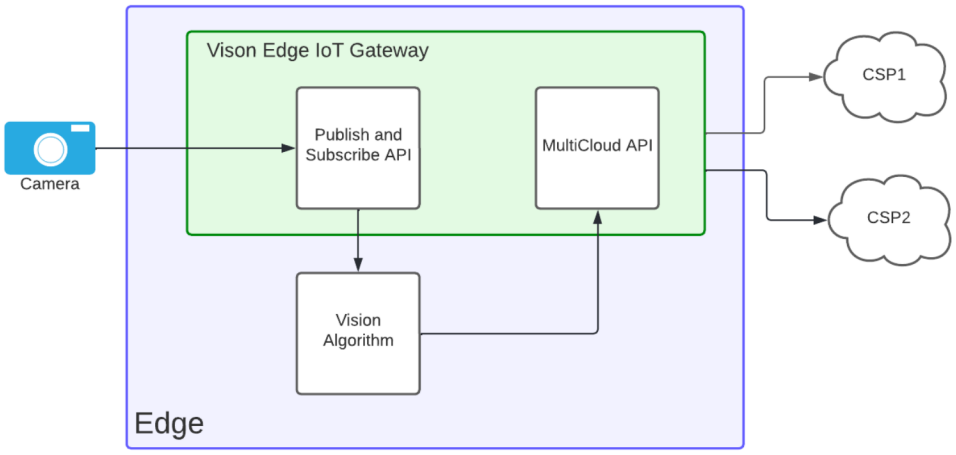

Figure 3.5: VEI implementation without a data store. The pub/sub system implemented has persistence included, therefore data store was not necessary

3.5.4 MultiCloud Mux

As previously stated, our goal is that the Edge gateway can work with multiple CSPs. With the plethora of CSPs, they all have their proprietary SDK to push and pull data from the Cloud. Rather than manually calling API calls from the CSP SDKs directly, the mux compiles all of the necessary SDKs and function calls within the

mux API. In our implementation, we utilize Amazon Web Services (AWS) and Google Cloud Platform (GCP). The multicloud mux publishing API allows specification of the CSP, the topic - which is wherein the Cloud the data should go to, and the actual data.

3.5.5 Docker Containers

All VEI components including NATS, data store, and the computer vision applications are run as Docker containers on the Edge. Containers are lightweight OS-level virtual machines. Docker uses containers to build self-contained containers with all dependencies included [27]. For reduced latency along the VEI pipeline, the applications need to be containerized such that the resource footprint is minimized. The containers could be managed using a container orchestration platform such as Kubernetes. Containerization facilitates a microservice architecture [28] and enables DevOps [29] practices of continuous integration, and continuous deployment. This allows quick roll of out of new features and bug fixes in VEI.

CHAPTER 4: EVALUATIONS AND RESULTS

In this chapter, we experimentally evaluate the functionality and performance of VEI on an Edge server using YOLO V3, a well-known deep learning-based object detection computer vision algorithm. The Edge server used in our evaluations is a Lenovo Ideapad laptop with the Intel i7 dual-core processor and 16 GB of memory. The operating system is Linux (Ubuntu 18.04, kernel version 4.15). The Cloud Service Providers used are Amazon AWS, and Google GCP. The laptop camera is used to capture live images.

Note that while the above described Edge server is sufficient for our purposes for evaluating VIE, a realistic setup would have the computationally expensive YOLO running on a GPU equipped Edge server.

4.1 VEI Latency

We first establish a latency baseline in running YOLO at the Edge and with the results transmitted to the Cloud (AWS IoT Core Python SDK). No Edge gateway is employed, so only a single vision algorithm can be run at the Edge. Figure 4.1 shows the block diagram of the experimental setup. To measure end-to-end latency, 1000 samples are measured starting from the images captured by the camera (1 frame every 4 seconds), processed by YOLO, and then transmitted by the AWS IoT client to finally arrive at AWS IoT Core. Figure 4.2 shows the latency Cumulative Distribution Function (CDF) from 1000 image frames. The 95th percentile latency is observed to be 1.61 seconds. The individual components of the latency are analyzed as shown in the pie chart of Figure 4.3. The largest latency component of 97.1% is due to YOLO, with the remaining 2.9% latency due to that of the network from the Edge to

the Cloud. As noted above, running YOLO on GPUs rather than the Intel i7 CPUs would greatly reduce the YOLO processing time.

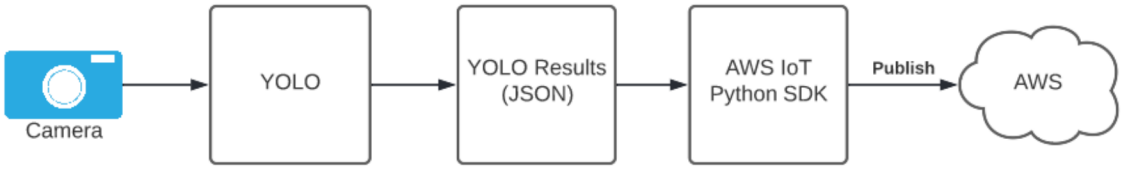

Figure 4.1: Setup for testing Edge processing

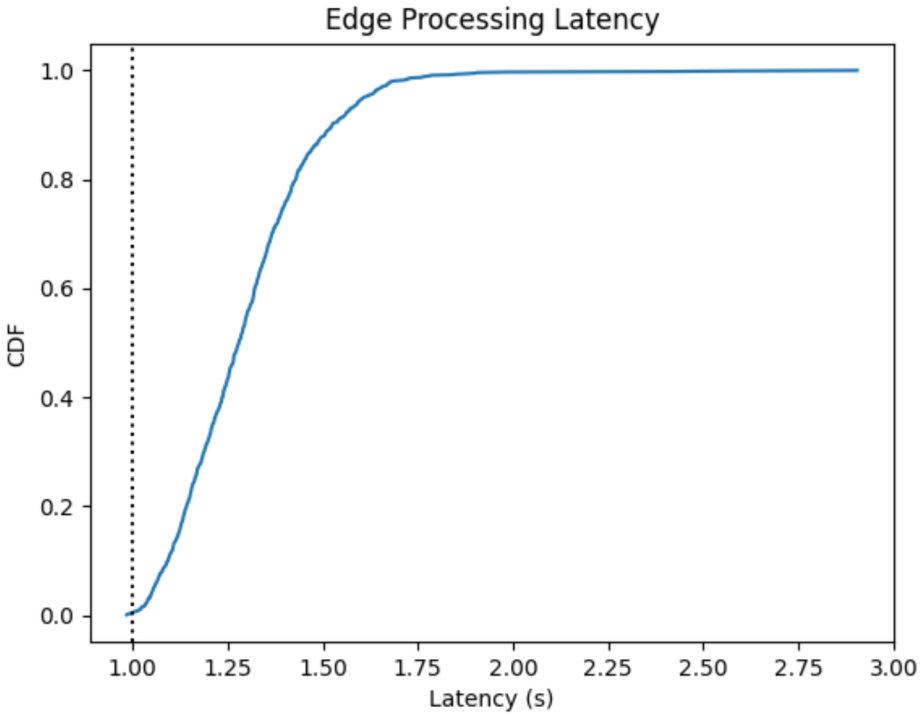

Figure 4.2: Latency CDF for Edge processing

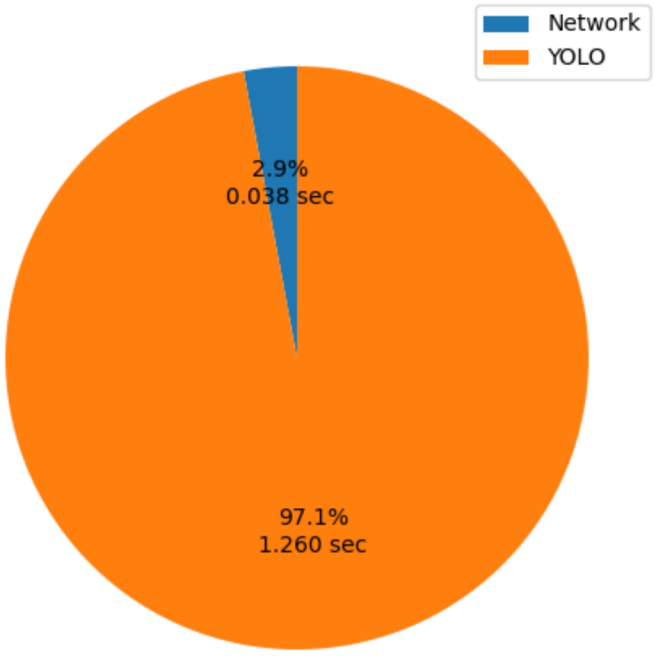

Figure 4.3: Component breakdown for Edge processing

We next characterize the latency of VEI with the experimental setup shown in Figure 4.4. Image frames from the camera are published to VEI, consumed by YOLO, and then published to the Cloud. Latencies are measured for operations that involve only VEI (excludes YOLO processing). Figure 4.5 plots the latency CDF for VEI. The 95th percentile latency is observed to be 6.29 milliseconds or 0.00629 seconds. VEI thus adds 0.39% additional latency. We also measure the increase in CPU and memory utilization with VEI. Compared to the baseline setup, VEI increases CPU utilization by 1.6% and memory utilization by 1.1%. Note that an end-to-end measurement is completed in the baseline setup of Figure 4.1 would be misleading for our setup since the increased CPU and memory utilization by VEI impacts the performance of YOLO running on the CPU.

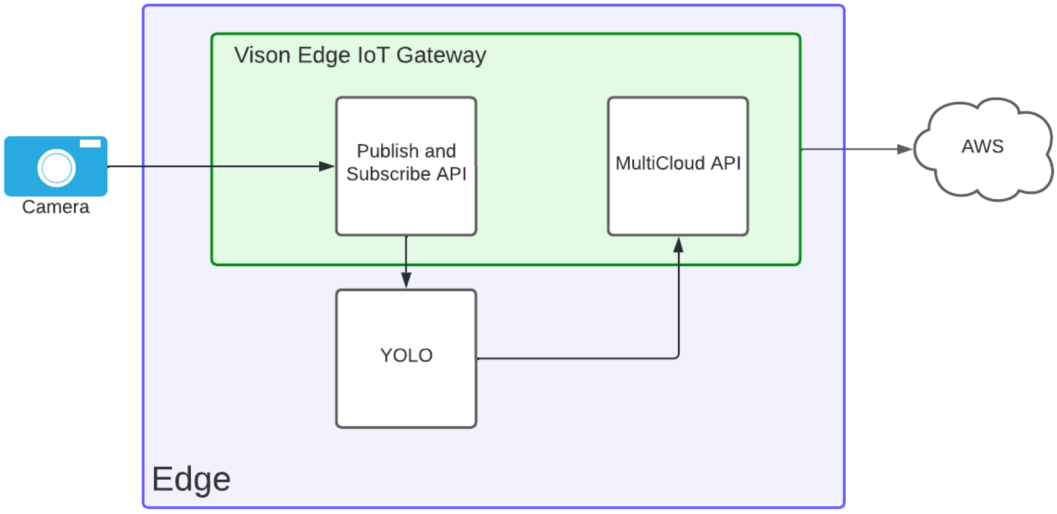

Figure 4.4: Setup for testing VEI

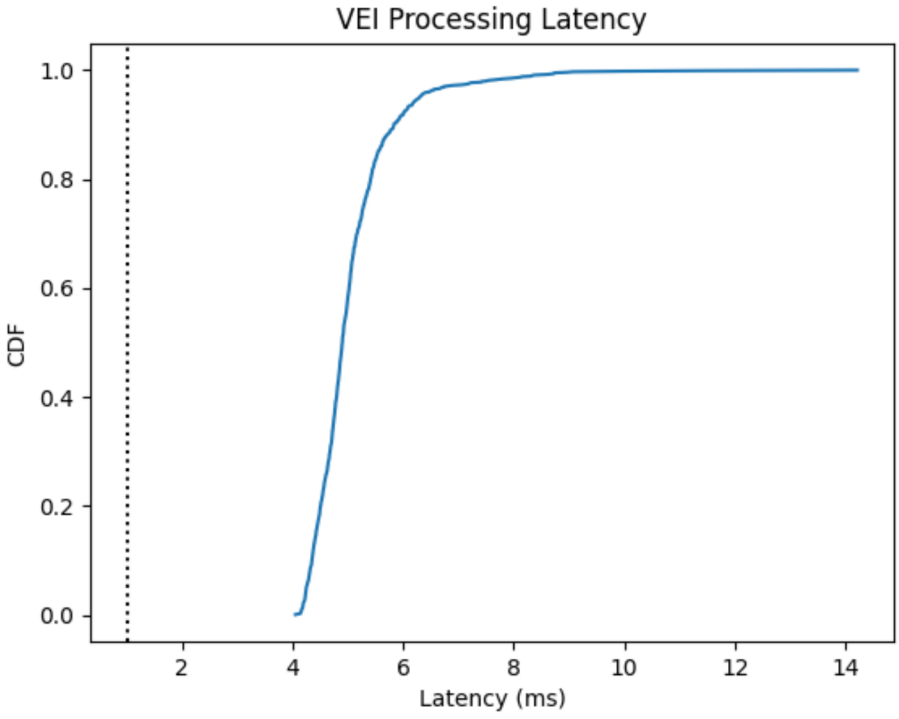

Figure 4.5: Latency CDF for VEI processing

4.2 Application Scaling with VEI

The key advantage of an Edge gateway is the ability to support multiple computer vision processing applications that can subscribe to a single image stream. To evaluate VEI's capability to support multiple vision processing applications, we scale the number of independent YOLO instances that consume the image stream as shown in the setup of Figure 4.6. Due to resource limitations on our Edge server, and our interest in VEI latency and resource usage, we emulate the object detection algorithm through a lookup table of detected objects. Figure 4.7 shows the average VEI latency that would contribute to the end-to-end latency as the number of object detection instances increases. An increase of 45.36% in average latency is observed as the number of object detection instances is scaled from 1 to 4. Figure 4.8 plots the scaling in memory and CPU utilization (as a relative percentage). The CPU utilization increases by 3.7% and memory utilization increases by 4.45% as the number of object detection instances are scaled from 1 to 4.

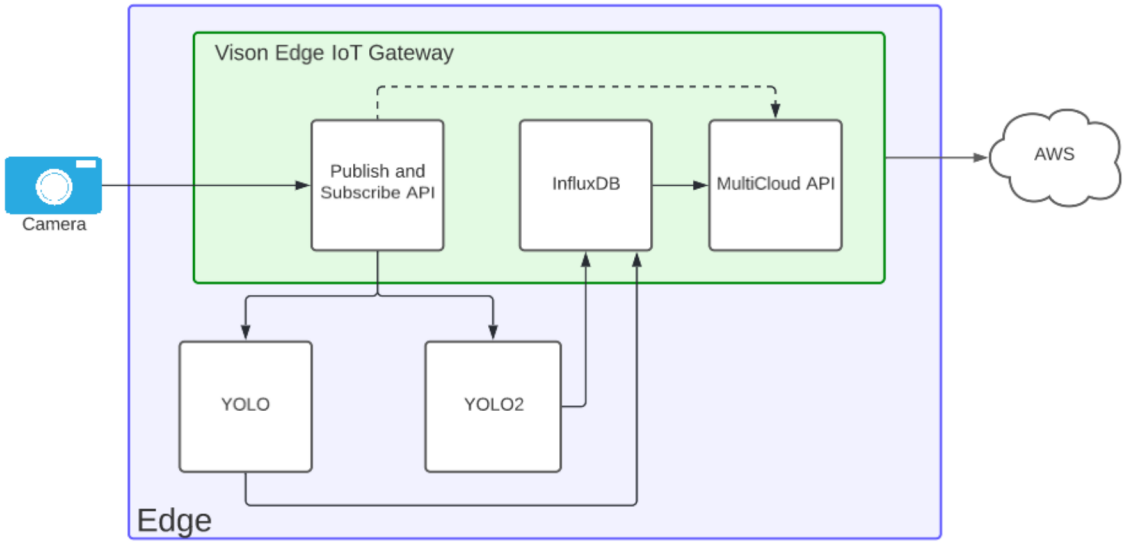

Figure 4.6: Setup for testing multiple vision applications with VEI

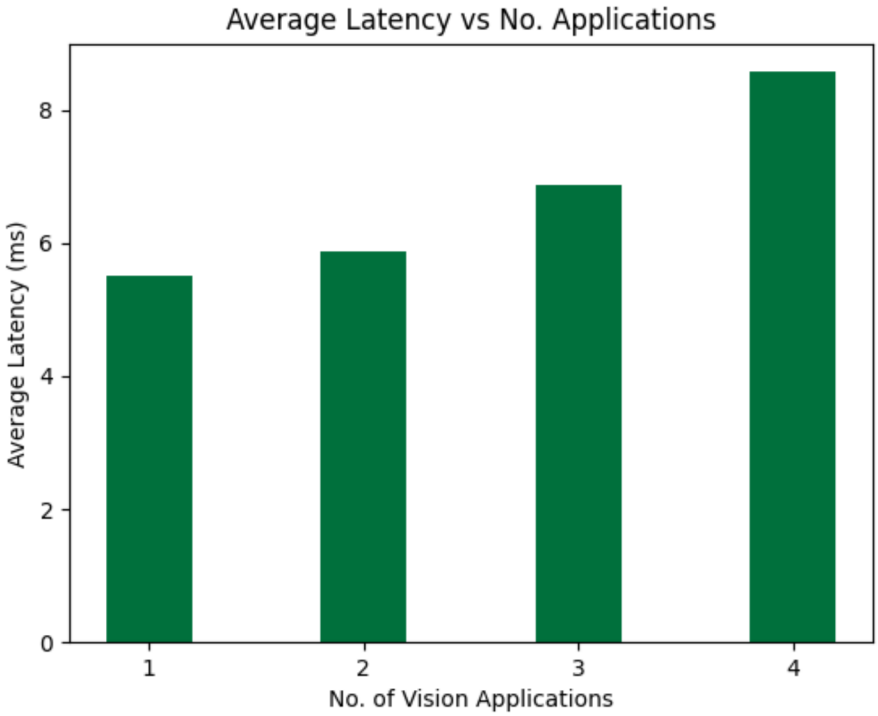

Figure 4.7: Average latency measurements with multiple vision applications

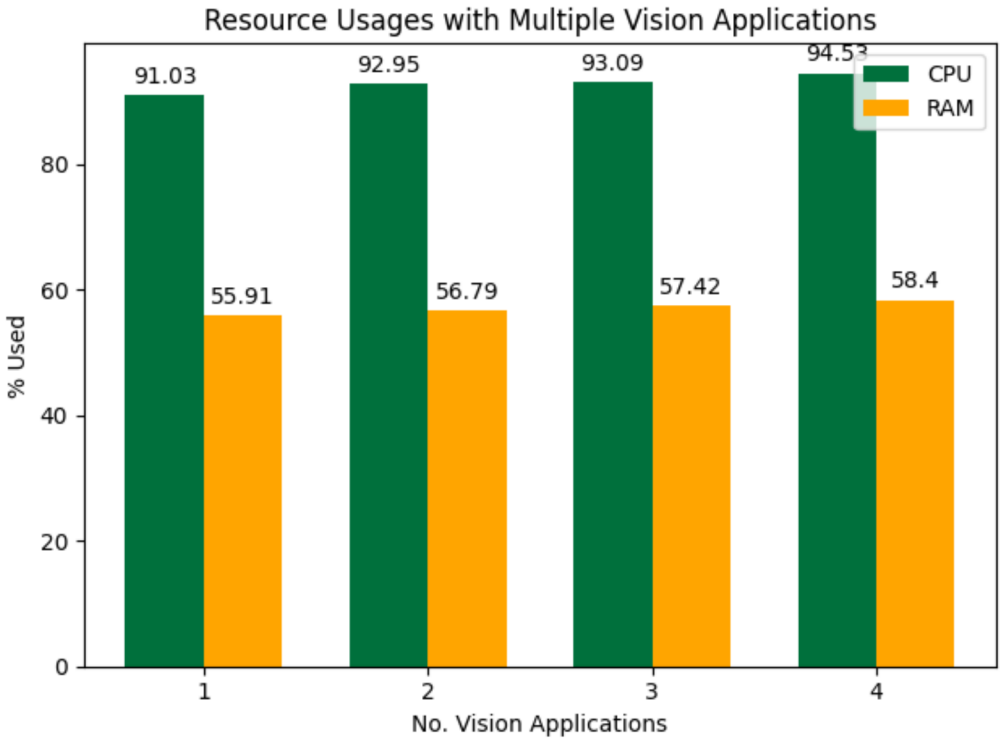

Figure 4.8: Resource breakdown for multiple vision applications

4.3 Dynamic MultiCloud Support

VEI is designed for dynamically switching between Cloud Service Providers for reasons mentioned in Chapter 3. The metric we seek to evaluate is the excess latencies incurred in switching from one cloud service provider to another. Our experimental setup is shown in Figure 4.9. Object detections are sent to one Cloud Service Provider (AWS) and then switched to another (GCP). A total of 10 images are sent through where the first 4 are sent to AWS, and the remaining are switched to send to GCP. Figure 4.10 shows the extra latency incurred in switching from AWS to GCP. The switch from AWS to GCP results in an excess latency of 45.5 % while the steady-state results show a decreased delay of 50 %. Whether such delays are tolerable depends on the particular use case depending on real-time requirements. For example, counting pedestrians crossing an intersection versus determining that a pedestrian is in danger from an imminent vehicular crash.

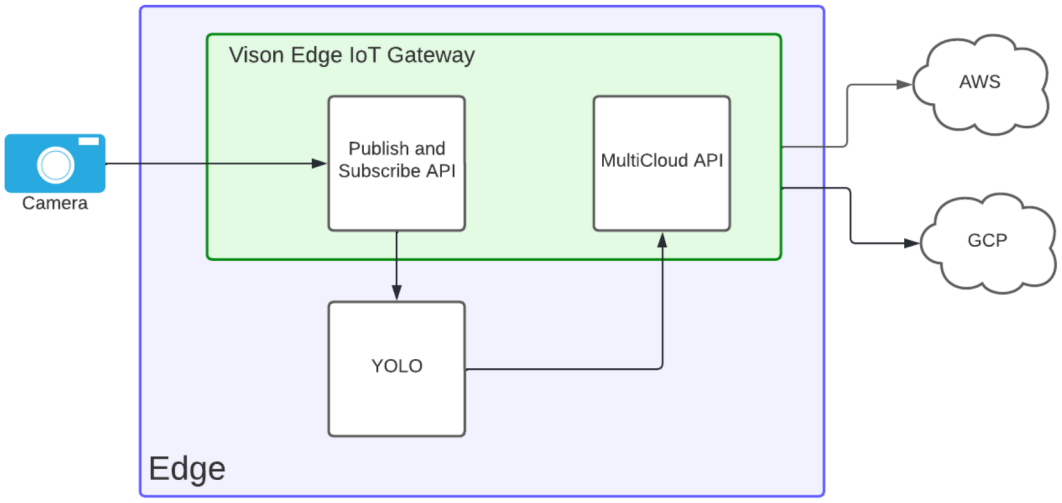

Figure 4.9: Setup for testing multiple CSPs with VEI

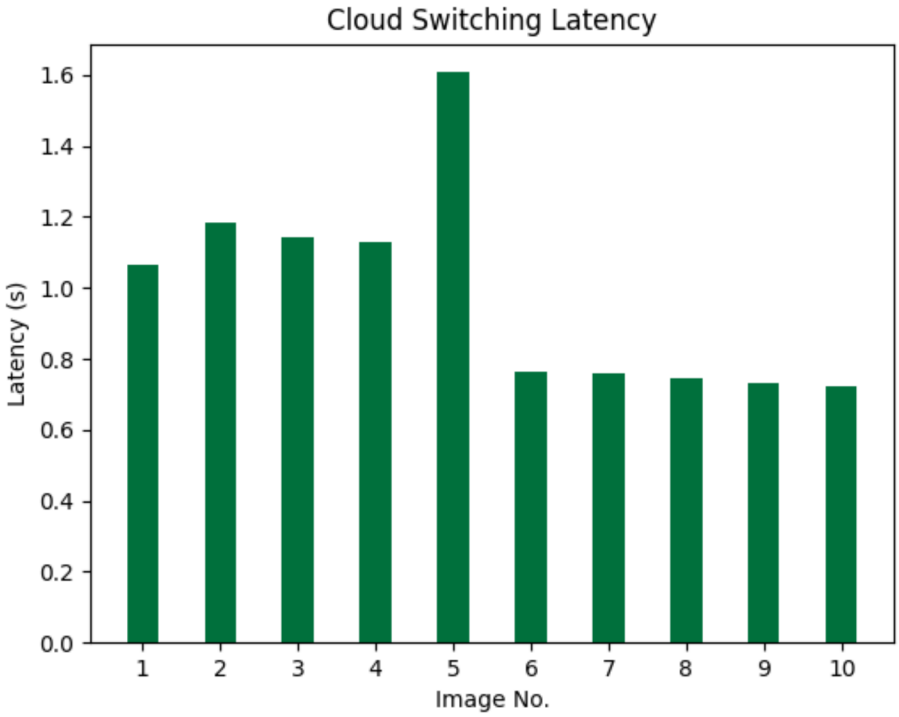

Figure 4.10: End-to-end latency measurements with AWS and GCP. Images 1 - 4 are sent to AWS. Images 5 - 10 are sent to GCP. We note an initial 45% spike in latency, before settling down to its steady state value.

CHAPTER 5: CONCLUSION AND FUTURE WORK

In this chapter we summarize our work, and comment on future extensions.

In this thesis we have identified the need for performing vision processing at the Edge for IoT applications involving computer vision. We note that existing solutions from Cloud vendors such as Amazon and Google are geared towards low data rate sensor applications, and are unable to address the "big data" challenge presented by video streams. We propose VEI, an Edge gateway for vision applications, that is not only able to handle video frames, but also support multiple vision applications, and provide dynamic access to multiple Cloud service backends. We experimentally evaluate the latency and resource usage of VEI across different use-case scenarios. Experimental results indicate that VEI is a viable solution for IoT vision Edge processing.

Several future extensions of our work are possible. The experimental setup can be made more realistic through a GPU powered Edge platform, freeing up CPU resources for exclusive use by VEI. Benchmarking of VEI could be done with more diverse vision algorithms with different processing requirements. We could also incorporate data store, and Edge analytics in our experimental evaluation of VEI. Cloud service providers could be extended to include Microsoft Azure IoT. The ability of VEI to handle multiple camera streams could also be investigated.

REFERENCES

[1] A. Elmorshidy, "Cloud computing: A new success model," in *2019 7th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*, pp. 7–12, 2019.

[2] R. A. Gheorghiu, V. Iordache, and V. A. Stan, "Computer vision application to determine crowdedness in public transport stations," in *2021 13th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, pp. 1–4, 2021.

[3] AWS, "What is IoT?." Accessed Apr. 16, 2022 [Online].

[4] IBM, "What is edge computing?." Accessed Apr. 15, 2022 [Online].

[5] AWS, "AWS IoT Greengrass." Accessed Apr. 15, 2022 [Online].

[6] AWS, "AWS IoT Core Pricings." Accessed Apr. 17, 2022 [Online].

[7] H. Jung, S. Lee, S. Park, B. Kim, J. Kim, I. Lee, and C. Ahn, "Development of deep learning-based facial expression recognition system," in *2015 21st Korea-Japan Joint Workshop on Frontiers of Computer Vision (FCV)*, pp. 1–4, 2015.

[8] P. Kaur and R. Kumar, "Human based computation: A solution to computer vision problems," in *2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*, pp. 630–633, 2018.

[9] S. Rao and V. M. Singh, "Computer vision and iot based smart system for visually impaired people," in *2021 11th International Conference on Cloud Computing, Data Science Engineering (Confluence)*, pp. 552–556, 2021.

[10] A. Ghazizadeh, "Cloud computing benefits and architecture in e-learning," in *2012 IEEE Seventh International Conference on Wireless, Mobile and Ubiquitous Technology in Education*, pp. 199–201, 2012.

[11] AWS, "Cloud computing with AWS." Accessed Apr. 15, 2022 [Online].

[12] AWS, "AWS IoT Core." Accessed Apr. 15, 2022 [Online].

[13] J. Waterman, H. Yang, and F. Muheidat, "Aws iot and the interconnected world â aging in place," in *2020 International Conference on Computational Science and Computational Intelligence (CSCI)*, pp. 1126–1129, 2020.

[14] Google Cloud, "Why Google Cloud." Accessed Apr. 15, 2022 [Online].

[15] Google Cloud, "Google Cloud Internet of Things." Accessed Apr. 15, 2022 [Online].

[16] D. Gupta, S. Bhatt, M. Gupta, O. Kayode, and A. S. Tosun, "Access control model for google cloud iot," in *2020 IEEE 6th Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing, (HPSC) and IEEE Intl Conference on Intelligent Data and Security (IDS)*, pp. 198–208, 2020.

[17] Google Cloud, "Using gateways," Apr. 14, 2022 [Online].

[18] Google Cloud, "Google Cloud Pub/Sub." Accessed Apr. 15, 2022 [Online].

[19] W. Shi and S. Dustdar, "The promise of edge computing," *Computer*, vol. 49, no. 5, pp. 78–81, 2016.

[20] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[21] S. Yi, Z. Hao, Z. Qin, and Q. Li, "Fog computing: Platform and applications," in *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, pp. 73–78, 2015.

[22] S. Tanwar, P. Patel, K. Patel, S. Tyagi, N. Kumar, and M. S. Obaidat, "An advanced internet of thing based security alert system for smart home," in *2017 International Conference on Computer, Information and Telecommunication Systems (CITS)*, pp. 25–29, 2017.

[23] S. G. H. Soumyalatha, "Study of iot: understanding iot architecture, applications, issues and challenges," in *1st International Conference on Innovations in Computing & Net-working (ICICN16), CSE, RRCE. International Journal of Advanced Networking & Applications*, no. 478, 2016.

[24] U. K. Dayalan, R. A. K. Fezeu, N. Varyani, T. J. Salo, and Z.-L. Zhang, "Veeredge: Towards an edge-centric iot gateway," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 690–695, 2021.

[25] gRPC, "Why gRPC?." Accessed Apr. 16, 2022 [Online].

[26] F. Paolucci, A. Sgambelluri, M. Dallaglio, F. Cugini, and P. Castoldi, "Demonstration of grpc telemetry for soft failure detection in elastic optical networks," in *2017 European Conference on Optical Communication (ECOC)*, pp. 1–3, 2017.

[27] Docker, "What is a Container?." Accessed Apr. 18, 2022 [Online].

[28] L. Abdollahi Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Microservice based architecture: Towards high-availability for stateful applications with kubernetes," in *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, pp. 176–185, 2019.

[29] P. Agrawal and N. Rawat, "Devops, a new approach to cloud development amp; testing," in *2019 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)*, vol. 1, pp. 1–4, 2019.

APPENDIX A: List of API Scripts

A.1 Publish API

```
func (s *server) Publish(stream VEI.VEI_PublishServer) error
{
 autoReply := &VEI.AutoResponse{}

 //start new connection with nats
 var nc, err = nats.Connect(nats.DefaultURL)
  if err != nil {
     panic(err)
   }
  defer nc.Close()

 //Streaming loop
 for {
 //start receiving streaming messages from client
  req, err := stream.Recv()
  //check if the stream has finished
  if err == io.EOF {
    //close nats and stream connection and send autoReply
    nc.Close()
        return stream.SendAndClose(autoReply)
  }
  //Grab individual data from req
  publishSubj := req.GetSubj()
  publishData := req.GetData()
```

```
    //Let client know that data was successfully published
    autoReply.AutoResp = fmt.Sprintf(
                            "Successfully published data to %s",
                            publishSubj)

    //Publish data using nats
    nc.Publish(publishSubj, publishData)
    nc.Flush()
  }
}
```

A.2 Subscribe API

```
func (s *server) Subscribe
                (in *VEI.SubscribeParams,
                stream VEI.VEI_SubscribeServer) error {
  subscribeSubj := in.GetSubj()

  var nc, _ = nats.Connect(nats.DefaultURL)

  //Using NATs
  wg := sync.WaitGroup{}
  wg.Add(1)
  msgNum := 0 //message number

  if _, err := nc.Subscribe(subscribeSubj, func(m *nats.Msg)
  {
```

```
    msgNum++
    responding_data := VEI.ImageData{Data: m.Data}
     if err := stream.Send(&responding_data); err != nil {
       log.Printf("send error %v", err)
      }
  }); err != nil {
    log.Fatal(err)
  }
  //Wait for messages to arrive
  wg.Wait()
  //close the connection
  nc.Close()
  return nil
}
```

A.3 MultiCloud Mux – Configure and Send to AWS

```
def configAWS(clientID):
        #AWS MQTT PARAMS
        cli = AWSIoTMQTTClient(clientID)
        cli.configureEndpoint(iotCoreEndpoint, port)
        cli.configureCredentials(rootCAPath,
                                 privateKeyPath, certPath)
        cli.configureAutoReconnectBackoffTime(1, 32, 20)
        cli.configureOfflinePublishQueueing(-1)
        cli.configureDrainingFrequency(50)
        cli.configureConnectDisconnectTimeout(360)
        cli.configureMQTTOperationTimeout(360)
```

```
        cli.connect()
        clients[clientID] = cli

#AWS PUBLISHING FUNCTION
def AWS_Pub(clientID, pubTopic, msg):
        clients.get(clientID).publish(pubTopic,msg, 0)
```

A.4 MultiCloud Mux – Configure and Send to GCP

```
def GCP_Pub(payload):
 token = {
  "iat" : datetime.datetime.now(tz = datetime.timezone.utc),
  "exp" : datetime.datetime.now(tz = datetime.timezone.utc) +
          datetime.timedelta(minutes=20),
  "aud" : projectID,
 }

 with open(private_key_file, "r") as f:
  private_key = f.read()

 jwtToken = jwt.encode(token, private_key, algorithm="RS256")

 clientID = "projects/{}/locations/{}/
␣␣␣␣␣␣␣␣␣␣␣␣registries/{}/devices/{}".format(
   projectID, region, registryID, deviceID
 )
```

```
gcpMQTTCli = mqtt.Client(client_id= clientID)

gcpMQTTCli.username_pw_set(
username = "unused", password = jwtToken
)

gcpMQTTCli.tls_set(ca_certs = ca_cert_path,
                   tls_version = ssl.PROTOCOL_TLSv1_2)
gcpMQTTCli.on_connect = on_connect
gcpMQTTCli.on_publish = on_publish
gcpMQTTCli.on_disconnect = on_disconnect
gcpMQTTCli.on_message = on_message
gcpMQTTCli.connect("mqtt.googleapis.com", 443)

topic = "/devices/{}/events".format(deviceID)
gcpMQTTCli.publish(topic, payload=(str)(payload), qos=1)
```