Fachhochschule
Dortmund
University of Applied Sciences and Arts

Fachhochschule Dortmund
University of Applied Sciences and Arts
Embedded Systems Engineering

Edge AI for Automotive with Control Loop-Based QoS Optimization

Research Thesis

Author: Sachin Kumar
Matriculation Number: 7216512

Supervisor: Prof. Dr. Rolf Schuster
Co-Supervisor:
Date: 05/2025

Abstract

The motivation of this research thesis is to enhance the efficiency of automotive applications which are highly dependent on the AI by offloading high computation tasks like lane detection, road segmentation and object detection, to the Edge Nodes. Here, the state-of-the-art (SOTA) algorithm like YOLOP (You Only Look Once for Panoptic Driving Perception) is used as an example DL model, which infers the camera image from Ego Vehicle acting as Edge Client and returns the output results which includes drivable area detection, road segmentation and object detection, is used in EC Nodes. Considering a scenario of automotive application that transmits the image from camera to the EC Node for processing and utilizing the high computational resources to efficiently process the image and sends back the required output results. By using EC Node, the computational load on the Ego Vehicle is reduced drastically. This scenario is emulated using the Emulate Edge Diagnostic Platform (EEDP) which consists of Containerlab, Opentelemetry, Clickhouse, and Event Manager. EEDP provides functionality to generate various scenarios to test the edge client application in different environmental conditions related to CPU stress and network load.

This approach reduces the on-board computation and enhances the overall system performance. The main focus of research thesis is to optimize the Quality of Service (QoS) for the edge application of Ego Vehicle by measuring the E2E Network latency and CPU stress to enable the dynamic switching between different EC Nodes on different Network Operators. The proposed solution improves the efficiency while maintaining high-performance standards in autonomous driving applications.

Furthermore, two solutions, solution A and B have been implemented and tested to achieve the best QoS by minimizing the time to switch EC nodes for both CPU stress and Network load. Both solutions are proposed and used maintain the QoS for the Ego Vehicle application. However, the proposed solution B includes using Zenoh as communication protocol, and Vehicle to Everything (V2X) to communicate between Ego Vehicle, Infrastructure, and EC Nodes, and also control loop algorithm to perform the switching events which can achieve ultra low latency for the automotive application which are dependent on EC Nodes for offloading AI related tasks. While solution B includes using OTLP for getting traces & spans for edge application to switch EC nodes for high CPU stress and network load.

In this thesis, two different scenarios have been tested with EEDP to evaluate the performance of the proposed solutions. The first scenario is about testing the edge application in an environment with two different EC nodes having high computational resources on same network operator. The second scenario is about testing the edge application in an environment with two different EC nodes having high computational resources on different network operators.

The results of both scenarios are compared to evaluate the performance of the proposed solutions. The results show that the proposed solution maintains and improves the Quality of Service (QoS) of the Edge Application, while maintaining high-performance standards of offloading tasks to edge nodes in ADAS and autonomous driving applications. Moreover, the proposed solution enhances overall system performance, and also focuses on measuring E2E Network latency, and CPU stress to optimize Quality of Service (QoS), by enabling dynamic switching between different EC nodes on same network operator as well as on different network operators.

Declaration

I hereby confirm, that I have written the Research Thesis at hand independently, that I have not used any sources or materials other than those stated, and that I have highlighted any citations properly.

Place, Date

Signature

Sachin Kumar

Dortmund, May 8, 2025

Acknowledgement

I would like to express my deepest gratitude to all those who have supported and guided me throughout this project.

First and foremost, I would like to thank my Supervisor, Prof. Dr. Rolf Schuster, for his invaluable guidance, encouragement, and support. I am gratefully indebted to him for his valuable comments on how I could do the research thesis. His expertise and insights have been instrumental in the completion of this research thesis.

I am also grateful to my colleagues and friends for their constant support and for providing a stimulating and collaborative environment. Without their passionate participation and input, this project could not have been successfully conducted.

Finally, I would like to thank my family for their unwavering support and encouragement throughout this journey. Their love and patience have been a constant source of strength.

Sachin Kumar

List of Abbreviations

ADAS	Advanced Driver Assistance Systems
AI	Artificial Intelligence
AV	Autonomous Vehicle
CARLA	Car Learning to Act
CPU	Central Processing Unit
DB	Database
DL	Deep Learning
DNN	Deep Neural Network
E2E	End-to-End
EC	Edge Computing
EC Node	Edge Computing Node
EDP	Edge Diagnostic Platform
EEDP	Emulate Edge Diagnostic Platform
GPU	Graphics Processing Unit
IoT	Internet of Things
ML	Machine Learning
QoS	Quality of Service
SOTA	State-of-the-Art
V2I	Vehicle-to-Infrastructure
V2X	Vehicle-to-Everything
YOLOP	You Only Look Once for Panopitic Driving Perception

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background & Motivation

As the technology is growing rapidly, the latest progression is happening towards Autonomous Vehicles and Advanced Driving Assistant System (ADAS), by integrating various sensors like Camera, Light Detection And Raging (LiDAR), Velodyne, Radio Detection And Ranging (Radar) and Global Positioning System (GPS) with the state-of-the-art (SOTA) deep learning models, which can assist or replace driver by offering a high level of autonomy. The high level autonomy is described as the execution of driving processes that serve self-driving functionality from a source point to the destination without any input or control from a human. Full autonomy can be achieved by integrating these sensors, and communication modules (5G and 6G) with software-level solutions, thus providing the automotive driving features and the advanced driver assistance system.

The current trend involves incorporating ML and DL approaches within autonomous vehicles to provide maximum precision and human-level accuracy. These statistically-based learning algorithms aim to interpret the ego vehicle's surroundings using data from various sensors when provided with impartial data. Based on the characteristics of the provided input, these algorithms classify or predict the output which can be used for autonomous driving. [16] Various autonomous systems require sensor fusion and perception to improve the reliability and efficiency of the whole system. However, these sensor fusion and perception algorithms requires high processing Central Processing Unit (CPU) and Graphical Processing Unit (GPU) resources. These resources can either be on ego vehicle or on cloud/edge.

Various systems e.g. ITS (Intelligent Transport System) and V2X (Vehicle to Everything) have already been proposed that uses the ego vehicle's sensor data as discussed earlier. These systems generally use cloud/edge resources for processing the ego vehicle's sensor data and transmits huge amount of sensor data which has to be processed on either cloud or edge nodes.

1.2 Overview

Recently, the Internet of Things (IoT) is emerging as part of the infrastructure for advanced application which requires huge computation resources by involving connection of many intelligent devices leading to smart cities and other connected communities. [30] This approach enables multiple devices e.g. ego vehicles, cloud and EC nodes to communicate with one another for sharing the data. Moreover, this shared data can also be processed on the cloud or EC nodes instead of ego vehicle. This approach will reduce the computation usage on ego vehicle and save more energy which can be used to cover more distance.

By considering this approach, edge computing enables processing of high computation algorithms like deep learning, considering it has enough resources e.g. GPU. As discussed earlier, Autonomous Vehicle (AV) and Advanced Driver Assistant System (ADAS) require high computation resources for sensor fusion and perception related tasks. In this research thesis, we will consider the perception algorithm as an example using EC node. The perception algorithm requires detecting not only objects which are on the road, but also drivable area and lanes on the road. This crucial information is then used by planning algorithms to avoid obstacle collision and also plan the trajectory of motion that the ego vehicle has to follow to reach the target location.

To achieve these detections various DL models have been trained to process the camera image of ego vehicle. But it requires using multiple models to process the camera images. Therefore, in this thesis, the deep learning model YOLOP (You Only Look Once for Panopitic Driving Perception) as described in the section 3.1, is used which can fast process the image to detect the lanes, drivable area, and also objects from a single camera image. Hence, this type of DL model has wide use in the application of autonomous driving and ADAS.

In order to utilize the resources on the Edge, the huge amount of data generated by the sensors like camera of ego vehicle has to be shared with the EC nodes. Then, EC nodes can process data of these sensors and transmits back the results which contains the detected objects, lanes and drivable area, to the ego vehicle for further processing and planning.

Furthermore, the EC can enable the ego vehicle to make smart decisions by utilizing not only the sensor data from all ego vehicles on the road, but also traffic signals, road congestions, road hazards, and unexpected obstacles in real-time.

1.3 Problem Statement

The problem statement in this thesis focuses on task switching and maintaining Quality of Service (QoS), to make the edge application more efficient and reliable for DL models particularly in the context of AV and ADAS in the edge computing domain. In the latest AV and ADAS, which are going to provide the ability to navigate autonomously on the roads of the smart cities or other environments, these ego vehicles rely on advanced sensor fusion and perception algorithms to make decision for navigation and collision avoidance in urban areas. A crucial aspect of this system is that the ego vehicle can take the navigation actions in real-time while utilizing the EC equipped with high resource CPU and GPU, to offload the processing related to DL algorithms like YOLOP, used as an example model and return response in real-time for further processing of planning the path, avoiding obstacles and also navigating to the target location.

As stated in previous section, the AVs rely heavily on real-time data processing from various sensors like Camera, LiDAR, Velodyne, RADAR, GPS and also communication units to transfer this data. These sensors transmit the large amounts of data for high-performance computing units within the ego vehicle to analyze using intelligent algorithms and AI models. This can result in massive data volumes - up to 20 Terabytes - which are influenced by factors such as driving duration, data rate, and sensor specifications. Generally, camera and velodyne generate huge amounts of data. This sheer amount of data processed on-board, can lead to significant power consumption, while also generating a substantial amount of data exchanged between infrastructure sensors and vehicles for collaborative autonomous applications. Despite advancements in sensory technologies, wireless communication, computing, and AI/ML algorithms, the challenge remains in integrating these innovations. This survey [16] reviews and compares connected vehicular applications, vehicular communications, approximation techniques, and Edge AI methods, with a focus on achieving energy efficiency through newly proposed approximation frameworks and enabling tools. Also, this survey is the first to comprehensively review approximate Edge AI frameworks and publicly available datasets for energy-efficient autonomous driving.

Generally, the AV and ADAS generally requires functionalities like the Lane detection, Object detection and Road segmentation, which requires high computational resources like CPU and GPU to process the images using DL model like YOLOP on the ego vehicle. But, these resources consume high resources from ego vehicle to process these sensor fusion and perception algorithms. Due to which, the ego vehicle consumes a lot of power which can affect the overall performance and energy efficiency of the vehicle. Also, it limits the travel distance of the vehicle, takes a lot of space and generate heat on the vehicle. However, in most of the vehicles, these resources are not scalable and can not be upgraded easily, which can lead to inefficiencies for long term solution. For example, if the ego vehicle has to process another deep learning algorithm which is crucial for driver safety, then it becomes challenge to execute this algorithm when the resources are already limited.

However, this problem can be solved by offloading the high computation tasks related to deep learning to the edge nodes which have high computational resources like CPU and GPU to process and handle these tasks. Despite recent advancements, existing frameworks exhibit notable limitations in efficiently offloading computation to edge nodes while preserving the performance and energy efficiency of autonomous vehicles. Moreover, these frameworks often fail to ensure consistent Quality of Service (QoS) for edge applications and lack the scalability required to support multiple ego vehicles and heterogeneous edge nodes.

This research thesis uses YOLOP dl model as an example application which is used on the EC nodes for autonomous driving vehicles to inference camera images without compromising the performance, and energy efficiency of the ego vehicle. As, there is limited research on frameworks for edge-based YOLOP deployment with real-time constraints. This research thesis also aims to fill this gap by proposing an efficient edge-based YOLOP deployment framework for ADAS and autonomous vehicles. Moreover, the focus is to optimize the E2E performance, quality of service (QoS), and providing real-time video processing for ADAS and AVs by proposing this architectural framework with ego vehicles and EC nodes. Furthermore, this thesis will also evaluate the feasibility and reliability of the proposed framework through various experiment scenarios in different environmental conditions, and then analyze the results.

1.4 Proposed Solutions

The research thesis addresses the challenges explained in subsection 1.3 by developing an architectural framework and using EDP for offloading the high computational tasks related to deep learning models to the edge nodes, providing the edge nodes must have high computational re-

sources e.g. CPU and GPU, to process these tasks and then transmit the results back to the autonomous vehicle, enabling it to operate the navigation and obstacle avoidance functionalities. The framework also integrates the real-time monitoring of CPU stress as well as network load latency with control loop mechanism to maintain the Quality of Service (QoS) for edge application. Moreover, the framework describes a system where CPU stress and network load are continuously monitored, and edge computing tasks are dynamically migrated/offloaded to less loaded networks or less stressed EC nodes when a predefined threshold is exceeded for network load or CPU stress respectively. This control loop mechanism switches the EC nodes to maintain the QoS (Quality of Service) and performance metrics for the edge application.

By utilizing the EC nodes for processing DL models, AV and ADAS can offload high computational tasks related to DL for perception, and use the high performance CPU and GPU on EC nodes while ensuring QoS of application. This approach will reduce on-board resource utilization and also reduces the overall manufacturing cost of AV and ADAS. This thesis proposes **two solutions** and evaluate the solution using framework to maintain QoS for edge application.

1.4.1 Solution A

The proposes solution consists of the following components:

1. **Edge Application:** The edge application is executed on the ego vehicle for capturing the image frames from ego vehicle camera or from video file, and then transmitting them to the EC nodes for further processing using DL model.

2. **EC Nodes:** The EC nodes are handling the image frames transmitted by vehicles using server application which is used to infer the image frames using the YOLOP model and transmit back the result image to the client application.

3. **Control Loop:** The control loop is implemented by utilizing **Edge Analyzer** and **Edge Navigator**, which are used to monitor the traces & spans of the application generated by Opentelemetry [3.2.1] and make decision when CPU stress and network load exceeds the defined threshold, to migrate from one EC node to another which has either low CPU stress or network load respectively.

1.4.2 Solution B

The proposes solution consists of the following components:

1. **Edge Application:** The edge application is executed on the ego vehicle for capturing the image frames from ego vehicle camera, and then transmitting them to the edge nodes for further processing using deep learning model.

2. **Edge Nodes:** The edge nodes are handling the image frames transmitted by vehicles using server application which is used for infer the image frames using the YOLOP model and transmit back the result image to the client application.

3. **Control Loop:** The control loop is implemented by utilizing **Edge Manager** which acts as the manager to handle the scheduling of various events related to CPU stress and network load by analyzing the sensor and vehicle data and make decision when CPU stress and network load exceeds the defined threshold, to migrate from one edge node to another which has either low CPU stress or network load respectively.

4. **Zenoh:**: The zenoh is communication protocol described in the section 3.6 which is used to transmit the sensor and vehicle data from ego vehicle to edge nodes and edge manager.

It is crucial to consider the behavior of the edge application that utilizes EC nodes resources for image processing using YOLOP model. Also, it includes evaluation of different scenario related to CPU stress and network load, and the edge application's performance and Quality of Service (QoS).

These scenarios play an important role in evaluating the feasibility and reliability of the edge application when it offloads processing task to the EC nodes. To simulate the vehicle, the CARLA Simulator described in the section 3.5, is used to get the images from ego vehicle and then transmit these images to EC nodes under normal CPU load and network stress. Once, the processing task is offloaded to the EC node, it becomes essential to measure the E2E latency which includes the network latency as well as the processing time from edge server application to infer the input image

using YOLOP model on GPU, also optimize and determine the QoS (Quality of Service). The entire control loop is implemented to measure the QoS which enables the switching from one EC node to another EC node that has low computation usage and also from one network to another based on the network traffic.

Therefore, the research thesis provides a different solutions and approaches to solve this problem by not only offloading the processing tasks related to DL models to the EC nodes and utilizing its high computational resources e.g. CPU, GPU.

The proposed system mainly focuses on these goals:

- **Quality of Service (QoS):** The system is designed to maintain consistent QoS levels by dynamically adapting to changes in network conditions, workload distribution, and resource availability EC edge nodes, thus supporting critical edge applications with low latency and bandwidth requirements.
- **Reliability:** The solution ensures high reliability suitable for automotive environments by incorporating switching mechanisms, failover strategies with robust communication protocols. It is also scalable, supporting seamless integration of multiple EC nodes.
- **Control Loop:** The architecture facilitates ultra-low-latency switching between EC nodes, whether on the same network or across different networks. This supports stable and responsive control loops crucial for real-time decision-making in autonomous driving and ADAS.

2 Literature Review

A comprehensive literature review of existing research papers in the field of edge computing, artificial intelligence, task allocation, and quality of service (QoS) has been conducted thoroughly, particularly focusing on the utilization of edge computing for AI inferencing tasks, and also other high computation-based applications which require QoS and resource allocation. Mostly, applications which require low latency and high computation resources, can offload their high computation tasks to EC nodes. For example, in the context of automobiles with ADAS or Autonomous System, lane detection, road segmentation and people detection, are high computation tasks which require CPU and GPU resources. However, when deploying edge-based applications for automobiles in the real world, the end-to-end latency plays a crucial role, thus it becomes essential to cope with the changing network and edge node conditions, such as network traffic, network operator and stress related to computation resources.

Edge computing (EC) has emerged as a promising solution to address the challenges of high computation tasks in various domains, including autonomous driving. The integration of edge computing with artificial intelligence (AI) has opened new avenues for real-time data processing and decision-making. Therefore, this literature review focuses on recent advancements in task offloading methods within edge computing.

Task offloading in edge computing is crucial for managing high-computation tasks while maintaining quality of service (QoS). Recent research conducted by [4, 15, 18, 34] focuses on developing methodologies and algorithms to optimize this process. These methodologies aim to enhance the efficiency of task offloading, reduce latency, and ensure that the QoS requirements are met. Corresponding to the specific scenarios, where the edge devices are competing for task offloading with limited communication and computational resources, which can lead to increased latency, reduced performance, and affecting the overall QoS for the application running on the edge devices. Most of the research papers focus on deterministic delay-bounded QoS guarantees as stated in [18] that the offloaded task have to complete before a deadline with 100 percent guarantee. On the other hand, it is not feasible to impose a deterministic delay requirement for all tasks, because of the highly dynamic wireless environment, that can cause drawdown in transient performance. To overcome this issue, Li et al. [18] has proposed a statistical delay-bounded QoS guarantee approach, which allows tasks to complete before a deadline with a specified probability above a given threshold. However, the statistical delay-bounded QoS only focuses on optimizing the tasks for computational resources, and does not consider switching of different EC nodes for offloading the tasks as a way to improve and maintain the QoS. This is important because the EC nodes can have different computational resources and capabilities, and switching between them can help to balance the task load and improve the overall performance of the system. Moreover, it is not suitable for application scenarios which are computational expensive for CPU and GPU resources, such as YOLOP [33] which is used for lane detection and drivable area segmentation in automobile applications, to run on low resource devices.

Additionally, Cheng et al. [15] surveyed various aspects of computation offloading, including energy consumption minimization and QoS optimization. Boulougaris and Kolomvatsos [4] introduced a proactive, distributed decision-making scheme for task scheduling at the edge, considering multiple parameters to ensure high QoS levels. Zeng et al. [34] presented a QoS-aware task offloading strategy that jointly considers bandwidth resource allocation and task offloading scheduling among multiple edge servers. These studies collectively demonstrate the importance of intelligent resource allocation, proactive decision-making, and consideration of network dynamics in achieving efficient task offloading while maintaining QoS in the EC environments.

Among various applications which can be offloaded to EC nodes like IoT, Smart home, and Smart City, the applications related to AI inferencing are also getting a lot of attention in the automotive industry [13, 19, 29, 31]. Kanwar et al. [29] highlighted the transformative impact of deep learning in the automotive industry, and also discussed how state-of-the-art deep learning models, which are driven by advancements in data availability, computational power, and most importantly algorithmic innovations, are being widely adopted for various applications in self-driving cars, parking and lane-change assist, and safety functions such as autonomous emergency braking. They also propose that virtual sensing for vehicle dynamics application, vehicle inspection/heath monitoring, automated driving and data-driven product development will be key areas that are expected to get the much most attention. Considering these advancements, Hatem et al. [13] have proposed communication protocol like Advanced Autonomous Driving (ADD) Protocol and AutoPilot, which can be used for offloading the edge computing tasks and allocate tasks based on Deep Reinforcement Learning (DRL) based algorithms. However, there is a huge gap between Edge and AI stated by Di et al. [19]. Also, they have proposed methods like quantization/model

compression, hand-crafted models, hardware-aware neural architecture and adaptive deep learning models, that can bridge this gap. Therefore, in this thesis, we will focus on using a similar model called YOLOP (You Only Look Once for Panoptic) 3.1 which is based on hand-crafted, and adaptive deep learning methods.

Next, the thesis explore solutions to implement the framework for offloading the tasks to EC nodes, and also switching between the EC nodes based on the network load and CPU stress. Various researcher and institutes have developed solutions which can be used for deploying the framework using emulators. Some of these solutions are based on emulators like AdvantEdge, GNS3, containerlab, wotemu and emuedge [8, 10, 11, 14, 15, 35].

However, Urwah and Stephan [20, 21] have provided a comprehensive comparison of various emulators and simulators, highlighting their features and limitations. Their research emphasizes the potential of containerlab as a robust tool for emulating edge computing environments. They demonstrated its capability to interface an emulated network with physical hardware in a hardware-in-the-loop (HIL) environment, validating its effectiveness in replicating real-world scenarios and creating realistic digital twins (DTs). Additionally, their findings indicate that containerlab outperforms GNS3 in terms of resource efficiency, with approximately 33% lower vCPU consumption and 12 GiB less memory usage. This makes containerlab a strong candidate for general-purpose emulation, with significant potential for expansion into domains such as cloud computing, IoT, and cellular networks. Future work proposed by the authors includes integrating 5G core network control plane features, refining mobility models to support both 5G and Wi-Fi technologies, and enhancing the graphical user interface (GUI) for improved usability and quality. By considering these conclusions, the Containerlab emulator is selected for this research thesis to implement the framework for offloading the tasks to edge nodes and switching between them based on network load and CPU stress.

With the rapid advancement of the Internet of Things (IoT), the interconnectivity of devices has led to the emergence of communication paradigms such as Vehicle-to-Everything (V2X), Vehicle-to-Vehicle (V2V), and Vehicle-to-Infrastructure (V2I) communication technologies that play a pivotal role in enhancing the efficiency of EC in automotive applications, particularly for offloading tasks. Moreover, these technologies enable seamless communication between vehicles, infrastructure, and other connected devices, facilitating real-time data exchange and decision-making. [17, 27]. Hence, the thesis moves in the direction of V2X implementation, which can be used for offloading the tasks to EC nodes and also switch between the EC nodes based on the predetermined requirements and thresholds to meet the desired QoS for the application. Sanghvi et al. [27] studies the modern cellular V2X (C-V2X) communication technology which can be used for autonomous connected vehicles (ACVs) enabling automated, responsive, safe, and effective driving through sensor-driven integration, and communication between peer ACVs, and road side infrastructure (RSI) through V2V and V2I communication.

In summary, the literature review highlights the critical role of EC in addressing high-computation tasks e.g. AI Inferencing, particularly in automotive applications which require low latency and high resource efficiency to maintain the QoS service for the applications. It also emphasizes advancements in task offloading methodologies for QoS optimization, and the integration of AI inferencing models like YOLOP.

Furthermore, it also identifies the potential of emulation tools such as Containerlab for implementing edge computing frameworks and explores the significance of V2X communication technologies in enhancing task offloading and maintaining QoS. These insights collectively guide the research direction towards developing efficient, adaptive, and scalable edge computing solution which can be emulated for automotive applications like AI inferencing which require high computation power and also allocation of the tasks to maintain the QoS.

3 Background

This section describes the tools which are used in this thesis to design the architectural framework solution that is used to emulate infrastructure with edge client, EC nodes, routers, and network operators, where the edge client offloads image processing tasks to the server running on the EC Node having access to the GPU.

These tools are listed below and described broadly in subsections:

1. YOLOP
2. OpenTelemetry & Clickhouse
3. Containerlab
4. Edge Diagnostic Platform
5. CARLA Simulator
6. Zenoh Communication Protocol

3.1 YOLOP

YOLOP is a State-Of-The-Art algorithm, used as multi-task deep learning model in a panopitic driving system, to generate autonomous driving directives only from images/frames captured by a camera. For autonomous driving, the network must make judgments at a fast speed, with high precision, and in real time. [33]

Various methods have been developed to detect traffic objects, drivable area segmentation, and lane detection, but these methods are used separately. For instance, object detection is handled by Faster R-CNN and YOLOv4, while semantic segmentation is performed by ENet and PSPNet. Lane detection is accomplished by SCNN and SADENet. Although these methods exhibit excellent performance, processing each task sequentially takes longer than handling them simultaneously. [32]

The purpose of the YOLOP model is to perform traffic object detection, drivable area segmentation, and lane detection simultaneously. This is the first in the work that can process three visual perception tasks simultaneously in real-time on an embedded device like Jetson TX2 (23 FPS) while maintaining excellent accuracy. Therefore, a multi-task network is more suitable in this situation, that can accelerate the image analysis process by handling multiple tasks simultaneously rather then sequentially and share the same feature extraction backbone. [32]

To solve this problem, a simple and efficient network architecture as shown in figure 3 is designed to do these tasks, while obtaining high precision and fast speed. The lightweight CNN as the encoder is used to extract features from the image and then these feature maps are fed to three decoders to complete their respective tasks. [32]

Figure 1: Input Image for YOLOP Model [32]

Figure 2: Output Image from YOLOP Model [32]

The figure 1 shows the input image from ego vehicle's camera that is driving on the urban area and figure 2 shows the processed image which contains drivable area painted in green color, lanes in blue color, and detected vehicles in red color 2d box.

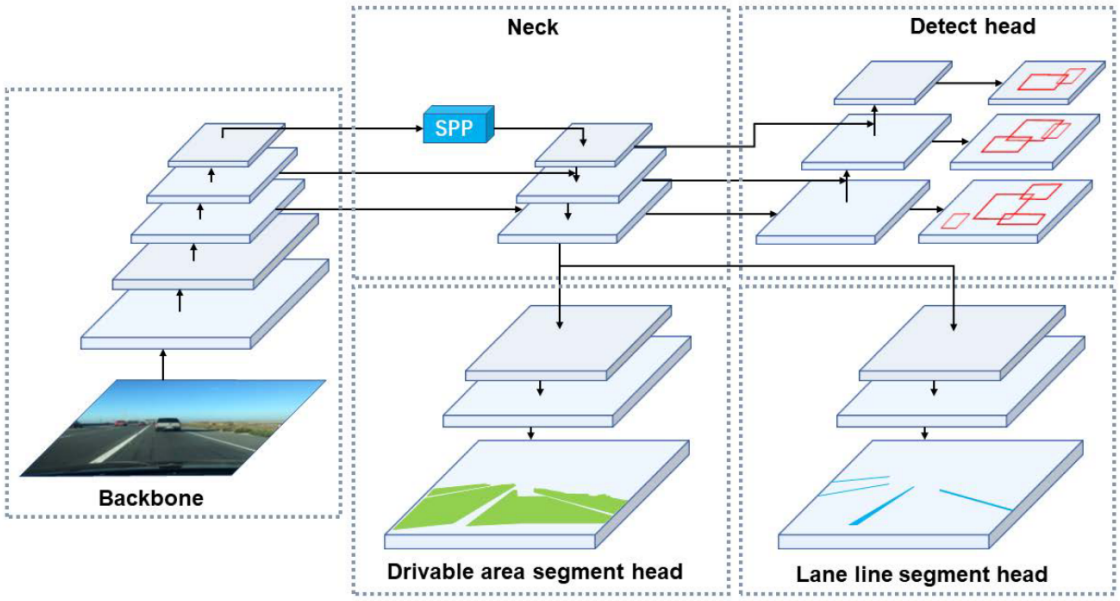


Figure 3: The architecture of YOLOP. YOLOP shares one encoder and combines three decoders to solve different tasks. The encoder consists of a backbone and a neck. [32]

3.1.1 YOLOP Client

The YOLOP Client application is developed which reads the image frame from either video file or camera of ego vehicle in CARLA Simulator, and then it serializes the image frame to the bytes using python library pickle before sending the data to the YOLOP Server using TCP Socket Communication. After that it waits to receive the processed image from the edge server, and shows the processed image in the window with FPS of images on the top right.

3.1.2 YOLOP Server

The YOLOP Server application uses the TCP socket Communication to open the port on the server which waits for the ego vehicle clients, to connect and then receives the image frames from the client as input. These received data then deserialized to the original image and then converted to the tensors to transfer it to GPU to infer the lanes, drivable area, and objects using the YOLOP Model. After that, processed image is transmitted to the client application.

3.2 Opentelemetry & Clickhouse

Opentelemetry is an Observability framework and toolkit designed to create and manage telemetry data such as traces, metrics, and logs. It is focused on the generation, collection, management, and export of telemetry. It provides observability, which is the ability to understand the internal state of a system by examining its outputs. In the context of software, this means understanding a system's internal state by examining its telemetry data, which includes traces, metrics, and logs. The Opentelemetry Collector offers a vendor-agnostic implementation for receiving, processing, and exporting telemetry data. In addition, it removes the need to run, operate, and maintain multiple agents/collectors to support open-source telemetry data formats (e.g. Jaeger, Prometheus, etc.) to multiple open-source or commercial back-ends. To make a system observable, it must be instrumented. That is, the code must emit traces, metrics, or logs. The instrumented data must then be sent to an observability backend. [24]

3.2.1 Opentelemetry

The growing importance of edge & cloud computing, microservices architectures, and complex business requirements has significantly increased the need for observability in software and infrastructure. Opentelemetry addresses these needs by adhering to two fundamental principles:

1. Data Ownership: You maintain control over the data you generate, ensuring no vendor lock-in.
2. Unified APIs and Conventions: You only need to learn a single set of APIs and conventions, providing flexibility in modern computing environments.

One major thing to consider when using Opentelemetry is the performance overhead. It intercepts an application's operations and collects a significant amount of data, which requires additional CPU and memory resources. This can directly impact application performance, leading to:

1. Performance Degradation
2. Operational Cost

These effects can be significant and should be considered when integrating Opentelemetry into an application. [26]

Also, Opentelemetry supports a wide range of programming languages with APIs and SDKs. In this project, the Python programming SDK is used to integrate it into the client and server applications. To instrument code encompasses the topics such as:

1. Automatic Instrumentation: This method involves the automatic integration of instrumentation libraries into the application, often through the use of operators like the Opentelemetry Operator for Kubernetes.
2. Manual Instrumentation: This approach requires manually writing code to instrument specific parts of the application, providing more detailed and customized telemetry.
3. Exporting Data: This involves the process of sending collected telemetry data to external systems for analysis and visualization, such as data warehouses or monitoring platforms. [23]

Manual Instrumentation is selected for the YOLOP client and server application, which can transmit the traces and span.

3.2.2 OTLP Collector

OTLP Collector is another component of the Opentelemetry, which is responsible for receiving, processing, and exporting telemetry data. It is a vendor-agnostic implementation that supports open-source telemetry data formats like Jaeger, Prometheus, etc., and multiple open-source or commercial back-ends. The OTLP Collector removes the need to run, operate, and maintain multiple agents/collectors. [25]

OTLP Collector is used in two ways in this project:

1. **Central Collector:** It is an OTLP Collector which is used to receive the traces and spans from the Local Collector (another OTLP Collector), and then send it to the Clickhouse Database. The Central Collector is running on a separate server, which has enough resources to run the Central Collector and Clickhouse Database. The edge applications send the traces and spans to the Local Collector, which then sends it to the Central Collector. It then processes the data and sends it to the Clickhouse Database.
2. **Local Collector:** It is another OTLP Collector which is used to receive the traces and spans from the edge application thought the Opentelemetry SDK which supports various programming algorithms, and then send it to the Central Collector.

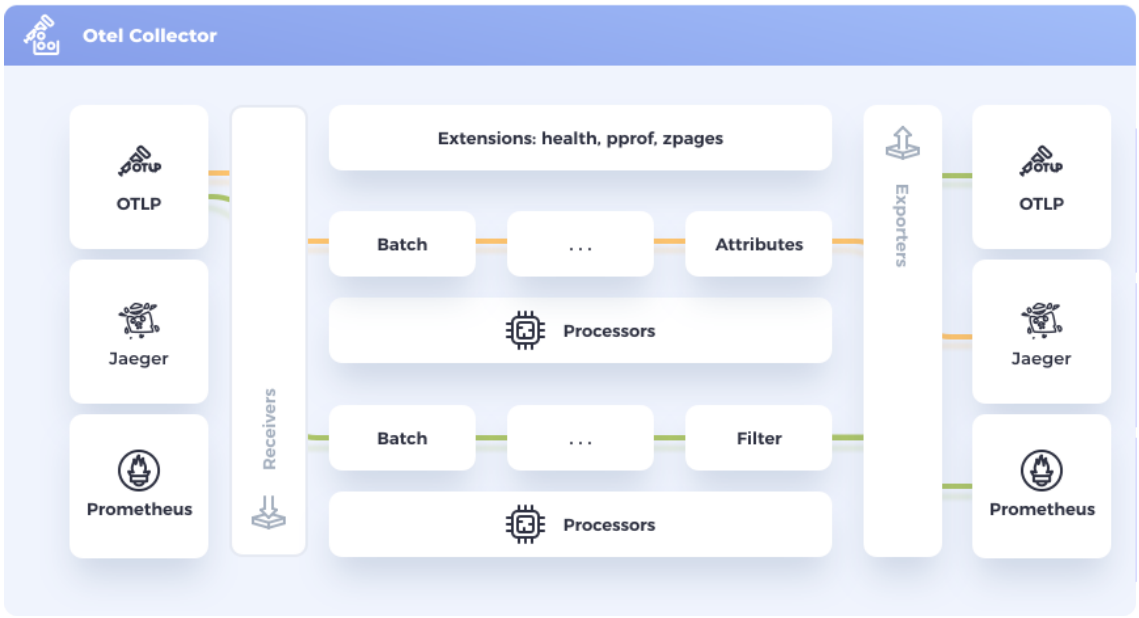


Figure 4: OTLP Collector [25]

3.2.3 Host Metrics

Host Metrics are the system metrics such as CPU, disk, and memory usage that are collected from the host machine where the edge application is running. These metrics are collected using the Opentelemetry SDK, and then send to the Local OTLP Collector, which then send it to the Clickhouse using the Central OTLP Collector. The Host Metrics are collected to analyze the CPU Load on the edge applications e.g. YOLOP Client and YOLOP Server. [22]

3.2.4 Clickhouse

Clickhouse is the real-time database that truly shines at scale, and counts on blazing performance when low latency really matters. [6] It is a high-performance, open-source column-oriented database management system specifically designed for online analytical processing (OLAP) workloads. It stands out for its remarkable speed and efficiency in handling massive volumes of data, often delivering query results in milliseconds. The key to ClickHouse's speed lies in its columnar storage format, which stores data by columns rather than rows. This allows for highly efficient data compression and minimizes the amount of data read from the disk during query execution, as only the relevant columns are accessed. Additionally, ClickHouse employs advanced indexing techniques, such as sparse primary indexes and data-skipping indexes, which further accelerate

query performance by reducing the search space. OLAP systems, like ClickHouse, are optimized for complex analytical queries that involve aggregations, multi-dimensional analysis, and large-scale data processing, as opposed to OLTP (Online Transaction Processing) systems that are designed for frequent updates and real-time transactions. [12]

What makes ClickHouse unique is its ability to handle both real-time data ingestion and analytical queries concurrently, without compromising on performance. This is achieved through its innovative architecture that supports distributed and parallel processing, allowing it to scale horizontally across multiple nodes. ClickHouse also offers robust support for SQL, making it accessible to users familiar with traditional relational databases. Moreover, its compatibility with various data formats and integration with popular data processing frameworks, such as Apache Kafka and Hadoop, enhances its versatility. Another distinctive feature is its ability to perform on-the-fly aggregations and complex transformations, enabling real-time analytics and reducing the need for pre-aggregated data. These features, combined with its open-source nature and the ability to run on commodity hardware, make ClickHouse a cost-effective and powerful solution for organizations grappling with big data challenges. Its blend of speed, scalability, and ease of use positions ClickHouse as a leading choice for modern data analytics and business intelligence applications. [12]

3.2.5 Edge Analyzer

Edge Analyzer is an application that analyzes the traces and spans of the edge application to determine the CPU stress and also network load. This analysis is crucial for diagnosing the edge application (client & server). It utilizes the Clickhouse Python library to connect with the Clickhouse database and retrieve the spans and traces of the edge client and server application. Furthermore, it analyzes these traces and spans, to evaluate the end-to-end latency for each image frame.

Additionally, these end-to-end latency is also used to trigger the switching of edge nodes, when the CPU stress is more than the defined threshold or network load is greater than threshold. Based on these analysis results, an HTTP request is sent to the Edge Navigator application to switch to different edge node either same network or different network.

3.2.6 Edge Navigator

Edge Navigator is an application that is responsible for assigning the different edge nodes to the edge client. As the edge client application starts, it requests the Edge Navigator to assign a server IP, that will be used by the edge client to send the image frame from ego vehicle and get the processed image result from edge server application. Also, the analyzer will send the request to the edge client, when edge analyzer application triggers the switching event to change the edge node on either on same network or different network.

3.3 Containerlab

Containerlab is an innovative open-source tool that revolutionizes the creation and management of network emulation environments. Designed for network engineers, researchers, and students. It leverages container technology to provide a lightweight, scalable, and flexible platform for deploying complex network topologies. Its declarative approach allows users to define lab topologies using simple YAML files, abstracting away the complexities of container orchestration and network configuration as shown in the figure 5. [8]

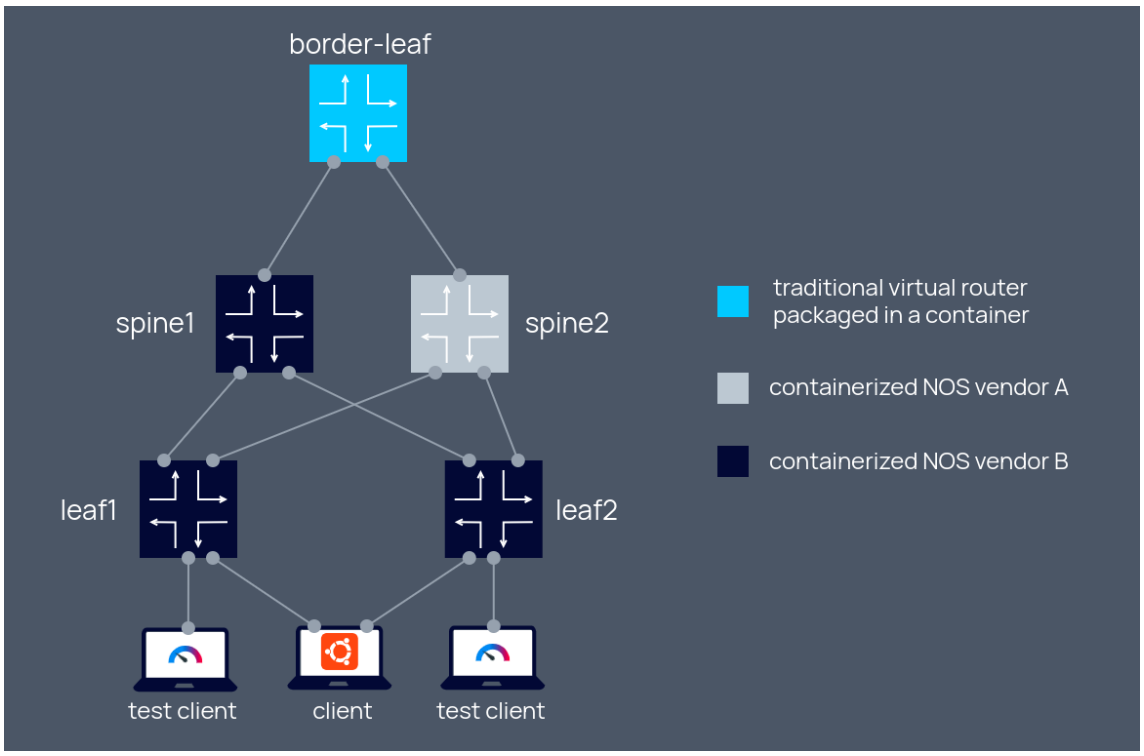


Figure 5: Containerlab Topology. [8]

It supports a wide range of network operating systems and virtual network functions, enabling multi-vendor setups that closely mimic real-world network architectures. With features like automated TLS certificate provisioning, topology visualization, and integration with virtualization platforms, Containerlab streamlines the process of building, testing, and validating network designs. This powerful tool not only facilitates rapid prototyping and experimentation but also serves as an invaluable asset for continuous integration and delivery pipelines in network automation workflows. [8]

It provides a CLI for orchestrating and managing container-based networking labs. It starts the containers, builds virtual wiring between them to create lab topologies of users' choice, and manages the lab lifecycle. [8]

3.4 Edge Diagnostic Platform

The Edge Diagnostic Platform provides tools to diagnose the behavior of the edge application based on the network and cpu load. It relies on the Opentelemetry, Clickhouse, and Edge Diagnostic Orchestrator to provide the functionalities. It is also used to generate the events and scenarios related to stress on the server and load on the network.

3.4.1 Network Manager

The Network Manager is a custom python script that is responsible for generating the network load between various endpoints. For example, once the configuration of the routers, load clients and servers is done. It can send http request to start the python application which generates the network traffic to constraint the network and also change the bandwidth of the containers running the application as YOLOP Client and YOLOP Server.

3.4.2 Stress Manager

The Stress Manager is another python application that is deployed on the edge node which is used for putting stress on the CPU using stress-ng tool. Basically, it provides a simple HTTP server functionality where different parameters like no of CPUs, % of load and time has to be provided to generate the desired load on the edge nodes.

3.4.3 Event Manager

The Event Manager is the custom script which is the part of EDP orchestrator that is used to generate the events related to the network load and CPU stress, which are then sent to the Network Manager for network load between the routers of that particular network, and Stress Manager for putting the stress on the CPU using stress-ng respectively to generate different scenarios for the client edge application.

3.5 CARLA Simulator

CARLA (Car Learning to Act) is an open-source autonomous driving simulator designed to support the development, training, and validation of autonomous urban driving systems. It provides a flexible and modular API built also supporting python, on top of the Unreal Engine, which offers realistic environments that can emulate real-world towns, cities, and highways as shown in the figure 6. It employs a client-server architecture, in which the server runs the Unreal Engine simulation and handles physics calculations, and clients control the logic of actors and set world conditions. Major key features of CARLA include:

1. Realistic environments with high-quality assets
2. Customizable sensor configurations, including cameras, LiDAR, and radar
3. Traffic management system for recreating urban-like scenarios
4. Support for multiple clients and scalability
5. Integration with popular machine learning frameworks
6. Ability to simulate various weather conditions and time of day

Figure 6: A street in Town 2, shown from a third-person view in four weather conditions. Clockwise from top left: clear day, daytime rain, daytime shortly after rain, and clear sunset. See the supplementary video for recordings from the simulator. [9]

Also, CARLA's versatility makes it a valuable tool for researchers and developers in the field of autonomous driving as well as ADAS, allowing them to test and evaluate their algorithms in a safe, controlled virtual environment before deploying them in real-world scenarios. [9]

3.6 Eclipse Zenoh: Communication Protocol

The Eclipse Zenoh protocol represents a paradigm shift in distributed communication systems through its unified approach to data in motion, data at rest, and computations. It is designed for the cloud-to-microcontroller continuum, and Zenoh's architecture enables efficient communication across heterogeneous networks while it maintains minimal overhead [5,7,36]. This section analyzes its core components, protocol stack implementation, and performance characteristics critical for modern cyber-physical systems.

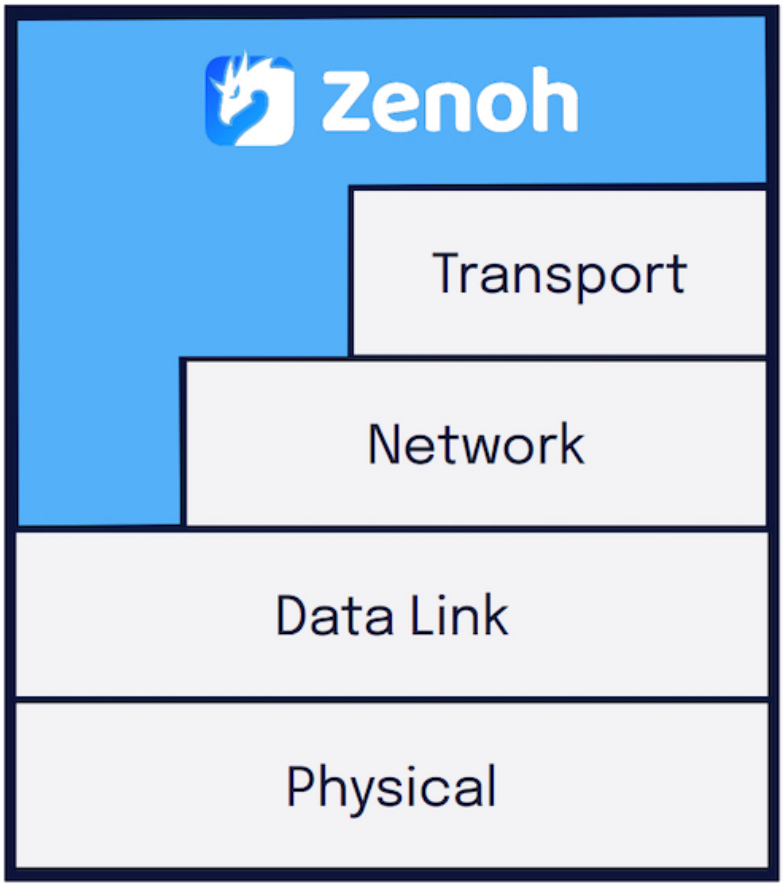


Figure 7: Zenoh's place alongside the various layers in the OSI mode, allowing efficient communication between constrained devices. [5]

Zenoh operates through a dual-layered abstraction model that decouples logical communication patterns from physical network topologies as shown in the figure 7. At its foundation lies a *key-space* organization where resources are represented as hierarchical key-value pairs using slash-separated identifiers (e.g., `/sensors/temperature/room42`) [7]. This structure enables native support for content-based routing and querying across distributed systems, with wildcard patterns (`*`) and recursive wildcards (`**`) providing flexible subscription capabilities [36]. The protocol's wire format achieves remarkable efficiency through binary encoding with a fixed 5-byte header, supporting payloads ranging from sensor readings to multimedia streams [5, 7].

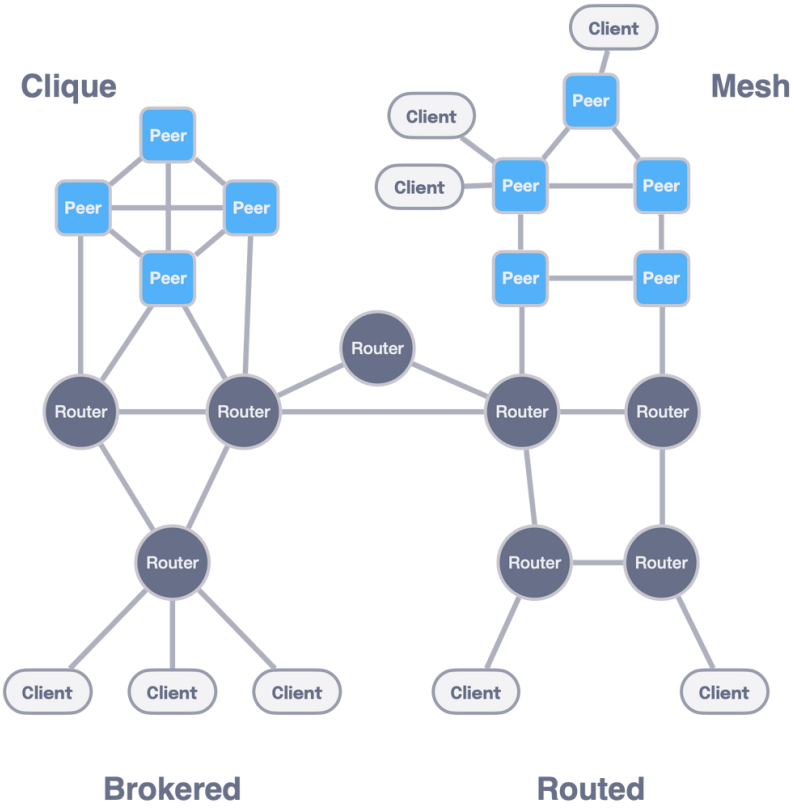


Figure 8: Zenoh supported topology [7]

However, the zenoh protocol doesn't impose any topological constraints on how application may communicate. As shown in the figure 8, it supports a wide range of topologies, including peer-to-peer over complete connectivity graphs as well as over arbitrary mesh, including routed communication and both routers and peers, can broker communication for clients. This flexibility allows for efficient communication across constrained devices, edge nodes, and cloud data centers, making it an ideal choice. Finally, it is worth to mention that zenoh's routers are software-based and can run very efficiently on a Raspberry Pi. [7].

4 Solutions

The goal of the thesis is to develop an architecture framework, which can be used for edge applications which require heavy computation for tasks related to AI and other fields in autonomous driving vehicles and also Advanced Driver Assistance Systems (ADAS), while it is maintaining the QoS for the edge application. To achieve this goal, we propose an architecture topology 9 with the help of Containerlab, which is used to emulate the infrastructure with multiple components connected together to emulate real world infrastructure of two network operators with EC nodes. This infrastructure is used to implement the control loop mechanism to maintain the QoS of edge application with two different solutions. These solutions will be used for maintaining the QoS for the edge application by using control loop feedback mechanism that allocates the resources for tasks that needs to be offloaded to the EC nodes. First, the proposed architecture framework is described, followed by the two solutions that are used to achieve the goal, and then the two distinct scenarios which are covered by the framework to test and evaluate the proposed solutions. It is important to note that both solutions assume the switching events are managed on the edge client (i.e., the ego vehicle), not on a distance EC node, by utilizing a set of custom-developed tools, as detailed in the following subsections 4.2 and 4.3.

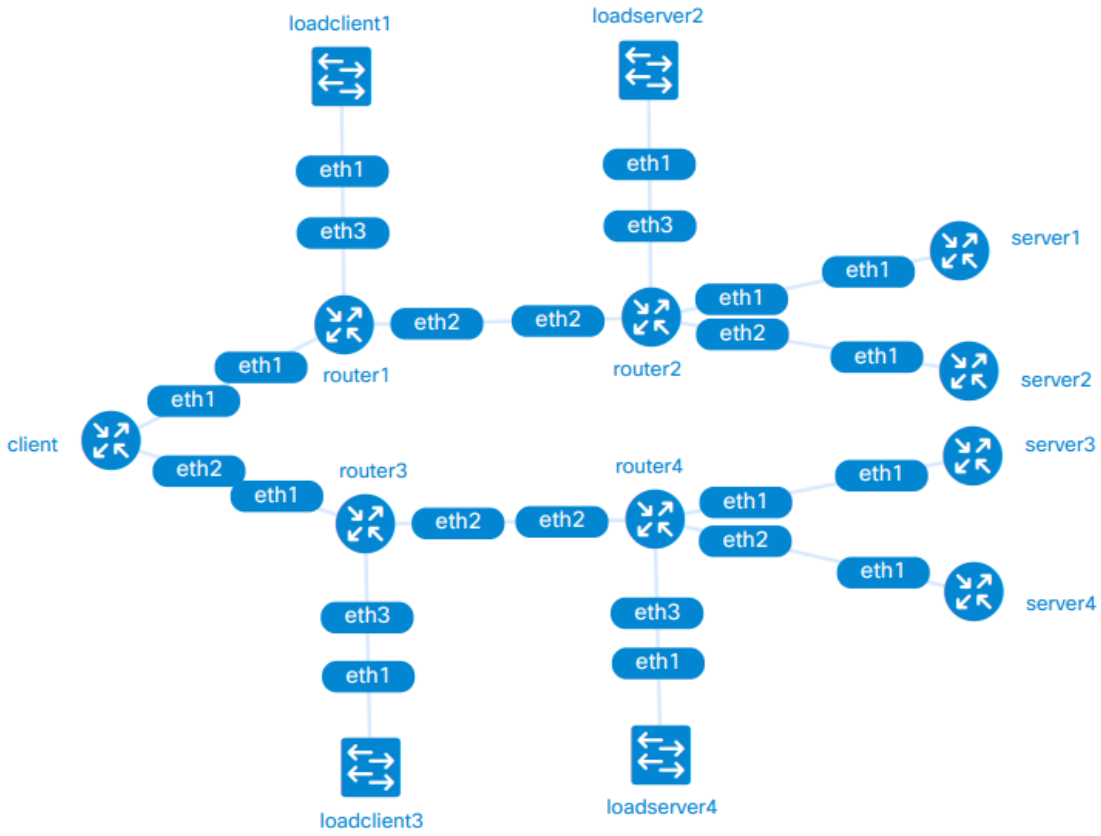


Figure 9: Proposed Architecture Framework with Containerlab Topology for the edge application

4.1 Proposed Architecture Framework

The proposed architecture framework is developed by using Containerlab 3.3 which is used to create different topologies using docker containers. Each docker container emulates a component. These components are described below:

1. **Ego Vehicle:** The ego vehicle is emulated vehicle which is used to run the edge application that is used to stream the camera image data to the EC nodes and receives the processed image from the EC nodes.

2. **EC Node:** The EC node is the emulated edge server that is used to run the server application to receive the camera image data from the ego vehicle and process it using the AI model e.g. YOLOP, then transmit it the ego vehicle.

3. **Router:** The router is emulated router container which is used to create the network topology with various docker container to form the architecture. These routers are also used to connect the ego vehicle container and the EC node container.

4. **Load Client and Server:** The load client and server are another docker containers which are used to put the network load on the network between the EC nodes and also ego vehicle container by transmitting huge amount of data.

4.2 Solution A

This solution builds upon the architectural framework presented in Section 4.1 which employs a multi-layer approach to collect, analyze, and offload the computationally intensive AI task to the EC nodes as illustrated in Figure 10. Furthermore, the solution consists of four distinct layers as follows:

1. **Devices Layer:** This is the foundational layer, that establishes connectivity among various entities including ego vehicles, EC nodes, and auxiliary devices e.g. routers, load client and servers that participate in data exchange and processing operations.

2. **Data Collection Layer:** This layer is responsible for the systematic collection and storage of data originating from the devices. It uses open-source tools such as Signoz, OTLP, and Clickhouse DB to facilitate efficient data collection, and data storage.

3. **Data Analyzer/Navigator Layer:** This is the critical layer of this solution as it is responsible for analyzing the data collected from the Clickhouse DB and make the decision responsible for switching the task on different EC nodes. It mainly consists of two applications, namely Edge Analyzer and Edge Navigator. The Edge Analyzer is responsible for analyzing the data collected from the Clickhouse DB, while the Edge Navigator is responsible for making decisions about switching the task on different EC nodes.

4. **EDP Orchestrator Layer:** This layer is used to send the trigger events related to CPU stress and network load through the EDP framework, load client/server, and event manager. These events are crucial for implementing various scenarios to test the application under different conditions, such as network load or CPU stress.

Moreover, an Edge AI Framework is proposed for offloading the heavy computation tasks from Ego vehicle to the edge nodes. It utilizes the traces & spans generated by the application running on edge nodes to analyze the performance and the QoS of the application itself. The main objective of this framework is to optimize the end-to-end performance, quality of service, and real-time video processing for ADAS and autonomous driving vehicles using control loop mechanism.

This solution of the research thesis involves utilizing **Containerlab**, a tool for creating topologies of Docker containers to define the infrastructure. This infrastructure consists of a vehicle, edge nodes, routers, and load servers. The vehicle hosts the client application, while the edge nodes run the server application. Routers are employed to model network operators, and load servers are used to simulate computational load on the edge nodes.

This approach leverages edge computing to address the challenge of efficiently processing and transmitting data. Specifically, data processing from the vehicle is offloaded to the edge nodes, allowing the vehicle to perform tasks such as object detection, lane detection, and drivable area detection. These computations are executed on the edge nodes using the YOLO model. This infrastructure provides a structured architectural methodology designed for vehicle-based applications.

The multiple scenarios have been executed in order to evaluate the methodology. In the first scenario, the images are sent to the edge node from the vehicle and then the edge node process the image using YOLOP model and then returns the image to the vehicle itself. In this scenario, the edge node is stressed using the Stress NG which puts the CPU cores under stress and then the behavior of the application is monitored using the traces and spans data. Then based on this data the Trigger is sent to the control. And the navigator of the application to change the server to a different edge node.

In the second scenario, the network is loaded by load servers which are connected to the routers, and they provide a way to put the network under load. This way the clients can't communicate with the edge node and the overall latency increases. Resulting into the bad performance and QoS. To overcome this bad performance, the edge node is disconnected and a different edge node on different network operator which is using another set of routers to define a separate network

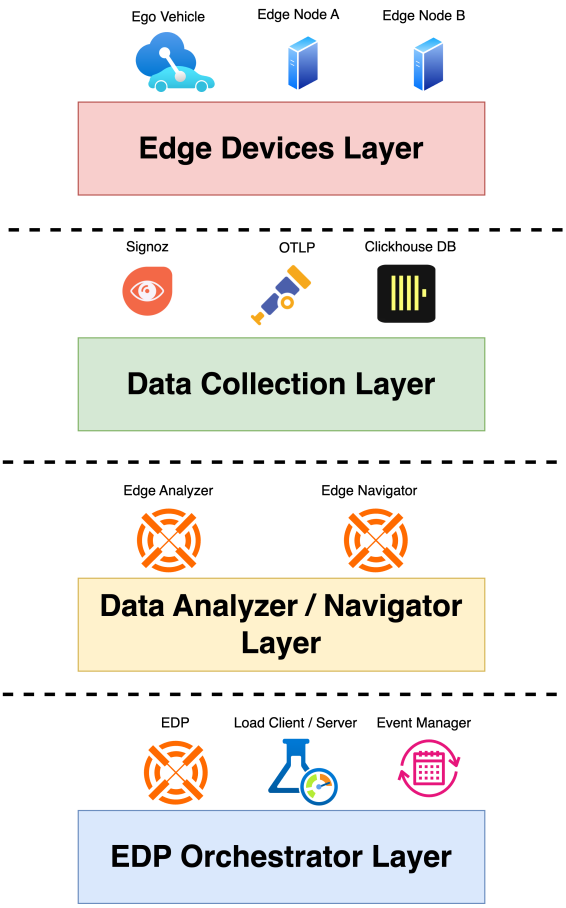


Figure 10: Layers of Solution A [1–3, 6, 24]

operator is used. The network Operator then gives the access to another edge node which is used for processing the images and then return to the Client Application.

The traces & spans data is gathered using the click house database which is running on a separate virtual machine and recording the data using the open telemetry which provides APIs to record the traces and spans. These traces and spans can be exported for later evaluation of the experiments. Which can be used to evaluate the performance of the application.

Moreover, EDP provides a way to record the times it takes to trigger the switching. So these switching have been performed multiple times in order to get the results for the scenario and then these results have been plotted to understand the behavior and the switching of the application to a different edge node.

The Sequence Diagram of this solution is shown in the figure 11, which shows the flow of the data from the client to the YOLOP Server and then back to the client, including sending the traces and spans to the Clickhouse though Local OTLP Collector and Central OTLP Collector. The Edge Analyzer is responsible for analyzing the traces & spans, and then sending the change server request to the Edge Navigator. The Edge Navigator sends another trigger request with new IP address for changing the edge node either on the same network or different network, based on the CPU and Network Load.

4.2.1 Overall Solution

In this section, the data is gathered using the click house database which is running on a separate virtual machine and recording the data using the open telemetry which provides API for different programming languages to record the traces and spans. These traces and spans can be exported for later evaluation of the experiments which can be used to evaluate the performance of the application.

To collect this data containing spans and traces from Clickhouse DB in real-time. The SQL query is executed to get the latest data from the database and then this data is used to evaluate the performance of the application. To optimize the query execution time, the query is executed with the latest time to reduce the computation time for the Clickhouse DB.

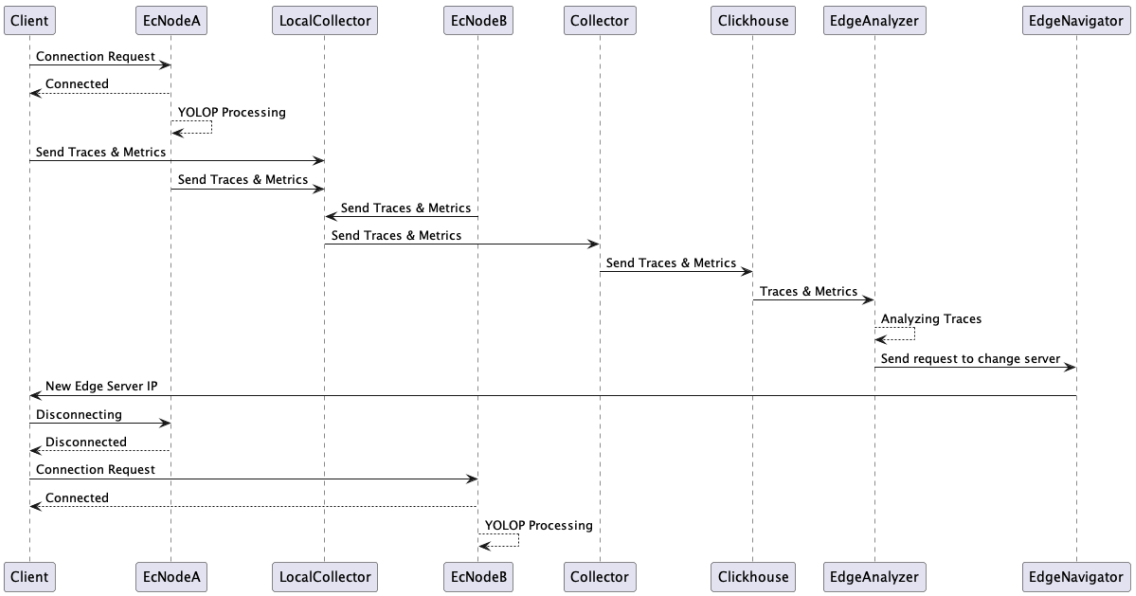


Figure 11: Sequence Diagram of Solution A

Edge Analyzer is another edge computing application running on the edge client, designed to process and analyze data collected using Open Telemetry (OTLP). This application plays a crucial role in monitoring and managing computational resources within an edge computing environment. The data gathered through OTLP is stored in a ClickHouse DB, which serves as a high-performance storage system optimized for handling large-scale telemetry data. Once the data is stored in Clickhouse DB, the Edge Analyzer retrieves CPU utilization metrics for each individual core from the ClickHouse database, then then performs a comparative analysis by checking whether the measured CPU utilization exceeds a predefined threshold value. If the CPU utilization surpasses this threshold, the application proactively triggers a control loop mechanism. The algorithm to implement the control loop is defined in equations 1, 2 and 3. This control loop is responsible for dynamically managing computational workloads by redistributing tasks across available edge nodes based on the threshold. By offloading the workload to a different EC node, the system ensures optimized resource allocation, prevents potential performance degradation, and enhances overall system efficiency.

$$U_{\text{core}} = \frac{U_{\text{total}}}{N_{\text{cores}}} \tag{1}$$

$$T = 0.80 \tag{2}$$

$$C = \begin{cases} \text{True}, & \text{if } U_{\text{core}} > T \quad \text{(Trigger to change the server)} \\ \text{False}, & \text{if } U_{\text{core}} \leq T \quad \text{(No action required)} \end{cases} \tag{3}$$

Where:

- U_{core}: CPU utilization per core
- U_{total}: Total CPU utilization
- N_{cores}: Total number of CPU cores
- T: Threshold value (0.80)
- C: Condition for server action

4.2.2 Switching EC Nodes

Here, another application known as Edge Navigator, is used to trigger the switching of the client application, so that it can connect to different edge node which has low CPU stress or low network load. This process is performed by Edge Navigator by using all the information about the CPU stress and Network load, related to the edge nodes and the network operators. Furthermore,

the Edge Navigator perform the decision of planning about which edge node should the client application connect, to offload the AI processing. The Edge Navigator is performing the control loop mechanism that checks if either the CPU stress is exceeded by certain threshold, or the network load is exceeded by certain threshold then it sends the trigger event to the Client app to change the server to a different edge node. Moreover, the Edge Navigator provides a REST API to trigger this event, so that the client application can change to different EC node.

Since, the client application is using the TCP Socket communication to send the image as buffer to the edge node, the client can simply disconnect the TCP server port and send request to joint another TCP server port which is decided by Edge Navigator. The process of disconnecting from one edge node and connecting to another edge node takes only few milliseconds (ms) based on the network load.

4.2.3 Performance Evaluation

The performance evaluation of the experiments for various scenarios is done by using the Edge Diagnostic Platform (EDP), which also provides functionality to record the time events in microseconds required to trigger switching between edge nodes on same network as well as different network. To ensure accuracy and reliability, these switching operations have been performed multiple times under different conditions for each scenario, by capturing various parameters such as system load, network latency, and processing time. The collected results are then analyzed and plotted to gain insights into the behavior of the system during the switching process, identifying potential delays and optimizations necessary for improved performance.

The performance of the application is primarily evaluated based on the CPU stress and network load of the edge node, which serves as a key metric for assessing computational efficiency and network usage. CPU utilization is continuously monitored through traces and spans data, which is gathered by using Open Telemetry (OTLP) and also stored in a Clickhouse database for further analysis. Additionally, memory consumption and network bandwidth usage are considered to provide a comprehensive performance assessment. The overall switching time is visualized using python plotting libraries like Matplotlib, allowing for a detailed measurement of the average end-to-end latency incurred when transitioning the application between edge nodes. These insights help in understanding the effectiveness of the edge deployment strategy and optimizing the system for seamless application mobility.

4.3 Solution B

Unlike the Solution A, which is based on collecting the span & traces in the Database and then analyzing this data to make the switching, the Solution B using the real-time messages called **heartbeats** to get the information related to the Network latency and CPU usage, which are two crucial parts for the Control loop mechanism that will be used to allocate the tasks on various EC nodes and also, on different network operators providing the latencies in the network latency and load on the CPU. The multi-layer approach is modified to remove the Data Collection Layer to make the solution more efficient and reliable as shown in the figure 13. These modified layers are described below:

1. **Devices Layer:** The devices layer is the foundational layer which is used to run the edge client and EC node. The edge client is running on the ego vehicle, and is responsible for transmitting the heartbeat to the edge manager and camera images to EC node, whichever will be allocated. Similarly, the EC node is responsible for transitioning the heartbeat to the edge manager and receive the camera images from the edge client, and then processing it based on the edge application. After that, the EC node is also for transmitting the processed data back to the edge client.

2. **Data Analyzer/Navigator Layer:** This is the critical layer of this solution as it is responsible for allocating the EC nodes and also switch to different EC nodes based on the conditions related to network load and CPU stress.

3. **EDP Orchestrator Layer:** This layer is used to send the trigger events related to CPU stress and network load through the EDP framework, load client/server, and event manager. These events are crucial for implementing various scenarios to test the application under different conditions, such as network load or CPU stress.

The Sequence Diagram of this solution is shown in the figure 12, which shows the flow of the data from the client to the YOLOP Server and then back to the client, including sending the

traces and spans to the Clickhouse though Local OTLP Collector and Central OTLP Collector. The Edge Analyzer is responsible for analyzing the traces & spans, and then sending the change server request to the Edge Navigator. The Edge Navigator sends another trigger request with new IP address for changing the edge node either on the same network or different network, based on the CPU and Network Load.

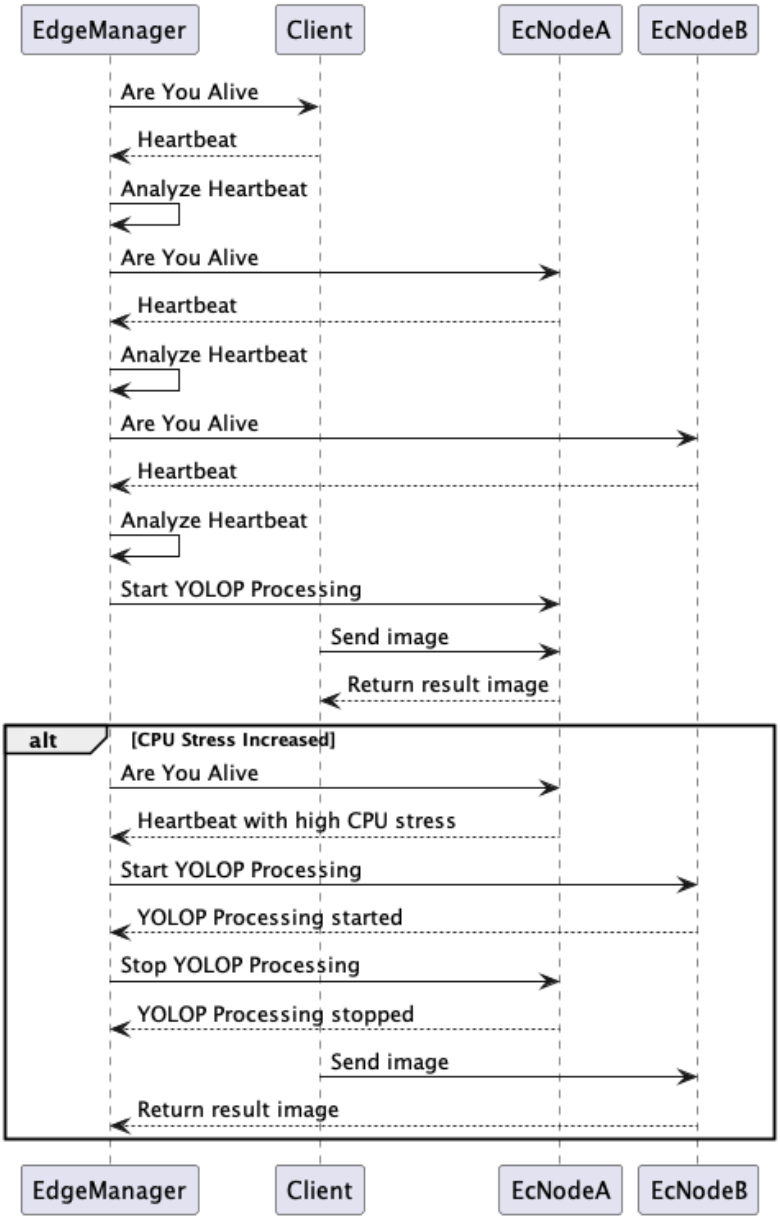


Figure 12: Sequence Diagram of Solution B

To accomplish this, Zenoh (communication protocol) 3.6 is used to implement the V2X (Vehicle to Everything) communication for Edge Client, EC Nodes, and Edge Manager. The main idea of using this approach is that the EC Nodes are running on the nearest locations and then providing the information related to network latency (through timestamp) and CPU stress to the Edge Manager as shown in the figure 14 and 15. More details about the messages and the data structure are described in below subsections.

4.3.1 Overall Solution

In this solution, the edge manager is continuously publishing the message with timestamp in UTC at 100 ms on the topic named **are you alive**, then the ego vehicle and EC nodes subscribes to this message to get the timestamp and use the same timestamp which is sent by edge manager to prepare the information message and publish it on the topic **heartbeats** as shown in the figure 16. This heartbeat message is subscribed by edge manager that contains the information related

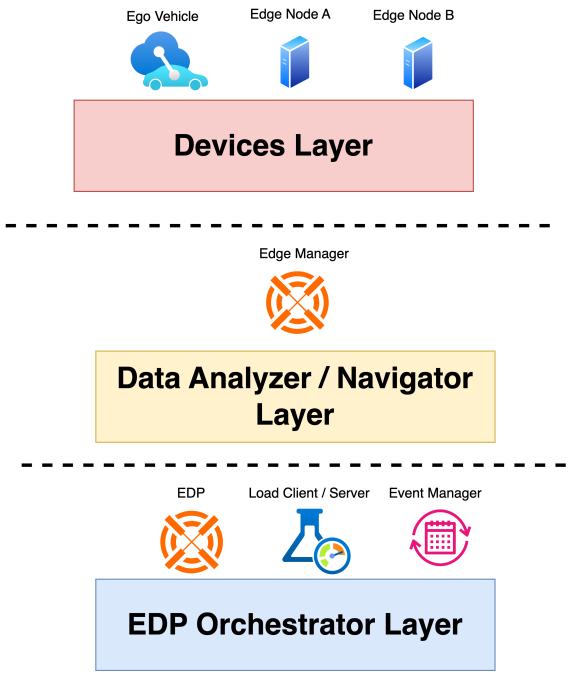


Figure 13: Layers of Solution B [1–3, 6, 24]

to device name, timestamp, and CPU usage. This whole process of sending the data related to device name, timestamp and CPU usage is called heartbeat. Hence, each device either edge client or EC node can be considered alive or not based on threshold defined for the heartbeat timestamp by edge manager. With this, all the devices edge client, EC node, Edge manager are connected to each other, and forming a Vehicle-to-Everything (V2X) communication. After that, edge manager uses this information find the E2E latency by using the timestamp received from heartbeat, and then compare it with current timestamp. Based on this E2E latency, the edge manager starts the processing of the ego vehicle images on EC nodes which includes using Deep Learning model like YOLOP to inference the image, and then publish this result image which includes the road segment, drivable area detection and object detection data. Also, the Edge Manager does the planning to switch the EC nodes using the vehicle heartbeat message sent earlier. Furthermore, when the CPU load of EC node is greater than the defined threshold of edge manager, it will trigger two events. First event will first start the processing of inference of images on another EC node and the other event will stop the processing of the image on the previous EC node. This whole architecture of the communication between various devices, e.g. edge clients, edge servers and edge managers, is using the Zenoh as the communication protocol, since this protocol is highly efficient and reliable as described in the section 3.6.

4.3.2 Switching EC Nodes

The eclipse zenoh provides a lot of features which has to be integrated in the system architecture to make it more robust and reliable. The edge client and server application both should use zenoh as the communication protocol. The edge client sends the data in the form of json which includes the key and value. The data structure defined for devices ego vehicle and edge nodes is described in the figure 14 and 15 as the heartbeat to the Edge Manager. Similar to solution A, this solution also utilizes the equations 1, 2 and 3 for switching the EC nodes based on the CPU stress threshold.

```
{
  "name": "edge_client1",
  "timestamp": "2022-01-01T12:00:00",
  "cpu_load": 0.10
}
```

Figure 14: Example JSON data representing an edge client with its name, timestamp, and CPU load.

```
{
  "name": "edge_server1",
  "timestamp": "2022-01-01T12:00:05",
  "cpu_load": 0.20
}
```

Figure 15: Example JSON data representing an edge server with its name, timestamp, and CPU load.

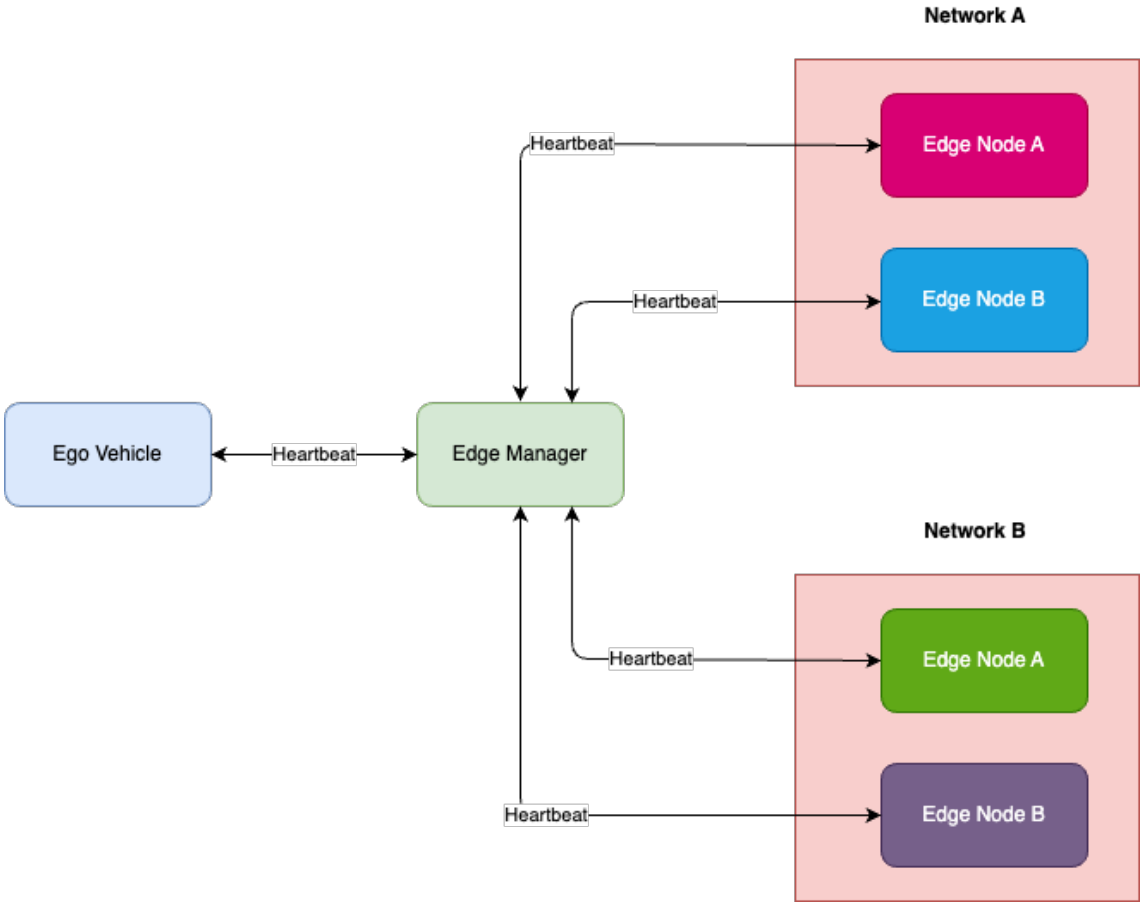


Figure 16: Architecture for V2X using Zenoh [1]

The heartbeat messages which are transmitted from various devices, are used by the edge manager to evaluate the real time E2E latency as well as the CPU stress. The workflow of heartbeat include the following steps:

1. Edge manager publishes the topic heartbeat with the payload containing timestamp.
2. Edge client/server subscribes to the topic heartbeat and sends the same timestamp to the edge manager and also adds the current CPU stress.
3. Edge manager receives the message from the edge client/server and then calculates the network latency as well as CPU stress.

4.3.3 Performance Evaluation

This subsection also describes the performance evaluation of the solution B, which is done by using EDP, that provides functionality to record the time of events in microseconds (μs) required to trigger various events for switching the processing between EC nodes on same network as well as different network operators. To ensure accuracy and reliability of EC node switching, these switching operations have been performed multiple times under different conditions for each scenario, by capturing various parameters such as CPU load, overall network latency, and processing time. The collected results are then analyzed and plotted to gain insights into the behavior of the system

during the switching process, and also to identifying potential delays and optimizations necessary for improved performance.

The performance of the edge application is primarily evaluated based on the CPU stress and network load of the edge application, which serves as a key metric for assessing computational efficiency and network usage. Moreover, CPU utilization and network latency are continuously monitored through heartbeats sent by the edge client and edge nodes to the edge manager. The overall switching time is visualized using similar solution as in solution A. These plots give insights into the effectiveness and reliability of the Solution B for seamless application mobility.

5 Experiment Design

In this section, the detailed implementation of experiment design for evaluating, is described, in which the client and server applications are deployed on different systems (docker containers) using Containerlab across different CPUs and emulated network providers. The client application switches the task offloading from one EC node to another EC node in case there is high CPU stress on it and also one network operator to another network operator in case of high network load on one network operators.

5.1 Development Setup

The system requirements for the development setup are as follows:

1. The system should have a minimum of 8 GB RAM.
2. The system should have a minimum of 4 CPU Cores.
3. The system should have a minimum of 1 GPU.
4. The system should have a minimum of 80 GB Disk Space.

A separate system is used for running the **Signoz** and **Clickhouse** for storing the traces and spans.

5.1.1 Client Server Architecture

The client and server application architecture follow an implementation where the client app uses either camera of vehicle in CARLA simulator or already provided video file, to stream the frame of images to the server app which is using YOLOP DNN model. Then, the server application inference the received images to detect the drivable area, lanes and also objects. The client and server app are developed using python including libraries like OpenCV for computer vision and PyTorch for inferencing the deep learning models.

The client communicate with the server using TCP socket. The approach follows a method where the client first sends the size of data packets of image which gets serialized and then waits for the acknowledgement that the packet has been received and then sends the actual image. This approach is used to ensure that there is no data loss between client app and server app. In similar fashion, server app also transmits the size of data packets of image in the processed image and wait for the acknowledgement that it has been received by client and then sends the final processed image in the serialized bytes.

For inferencing the YOLOP model on server app, PyTorch library is used. It provides the helping function which can be used to load the pre-trained YOLOP model and then transfer that model to GPU as it can provide high performance when doing inferencing compared to CPU. The YOLOP model is used to detect the objects in the image and then draw the bounding boxes around the detected objects.

The following image shows the latency when using the YOLOP model for inferencing the images on CPU and GPU. The latency is calculated by taking the time difference between sending the image to the server and receiving the processed image back from the server. When using GPU, the fps is 30 while when using CPU, the fps is 5-8. This shows huge improvement in performance when using GPU for inferencing the images.

5.1.2 Containerlab Setup

The containerlab is used to deploy the client and server applications on an infrastructure containing various routers, emulating different network operators as well as edge devices. This infra is defined using a topology.yaml file which contains the details of the routers, edge devices, network operators and the client.

The setup of the development is as follows:

1. The containerlab topology is used with 1 client, four routers with four load servers having two network operators, and then having 4 EC nodes. Two on each operator.
2. As the containerlab uses the docker containers, each container can be passed with the CPU and Memory limits. Therefore, each EC node is passed with 1 CPU Core and 500 MB Memory limits.

3. The client is passed with 1 CPU Core and 500 MB Memory limits.
4. The client is connected to the EC node using the network operator.
5. Since the application is required to use GPU, the GPU is passed through to all the EC node containers.
6. The routers are configured to emulate two different network operators.

5.1.3 Edge Diagnostic Platform Setup

The **Edge Diagnostic Platform** is used to collect the traces and spans from the client and server applications. The traces are collected using the OpenTelemetry SDK which is integrated into the client and server applications. The traces are then sent to the OpenTelemetry Collector which is running on a separate system. The OpenTelemetry Collector is configured to send the traces to the Signoz which is a distributed tracing system. The traces are stored in the Clickhouse database which is used to query the traces and spans.

5.1.4 Architecture

The architecture of the project that contains the implementation of the client, which transmits the image frames from the vehicle, YOLOP Servers which inference the images using the YOLOP model, and also four routers are used to define two different networks as shown in the figure 17.

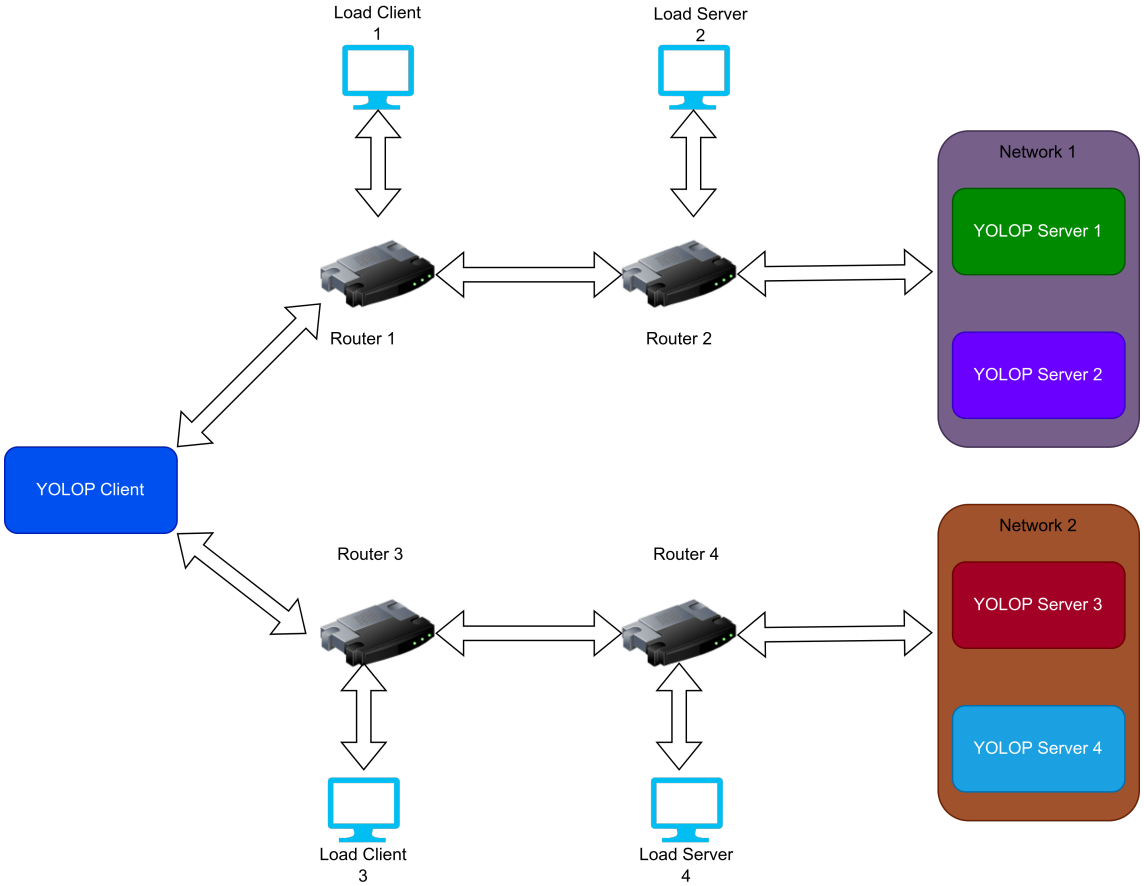


Figure 17: Architecture of the project [1]

Moreover, Each component shown in figure 17 are explained in more detail in the subsection 4.1.

5.2 Experiment Setup

To begin the experimental setup, a Docker image must be built containing all the necessary components required for the experiments. These components include Stress-NG for generating synthetic CPU and memory stress, a Local Collector for host-level metric collection, networking utilities

such as iproute2 and ping, an Event Manager to orchestrate the triggering of stress events, and the YOLOP model for image inference tasks.

Following the successful construction of the Docker image, the Signoz observability platform and ClickHouse database are initiated. These tools are responsible for collecting and storing distributed tracing data—specifically spans and traces—from the client and server applications, which are subsequently analyzed during the evaluation phase.

Subsequently, the Local Collector service is launched to gather host-level metrics, including CPU and memory usage, from the EC nodes involved in the deployment.

Once the monitoring components are operational, the Edge Diagnostic Platform (EDP) is deployed. The EDP facilitates the controlled generation of stress conditions on EC nodes, including both CPU-intensive tasks and network load simulations.

Finally, upon successful deployment of EDP and Signoz, Containerlab is launched to instantiate the complete network topology. This topology comprises clients, load servers, routers, and EC nodes, forming the infrastructure required for the experimental study.

5.3 Experiment Flow

The experiment flow is as follows:

1. **Prepare Edge Client:** The edge client is the application running on ego vehicle which sends the image to the EC node.

2. **Prepare EC nodes:** In the EC nodes, the YOLOP server application should be executed as defined in the Containerlab so that the client sends the image to the EC node to inference it.

3. **Prepare Edge Analyzer and Navigator:** Edge Analyzer and Edge Navigator applications are started on the ego vehicle container which allocates the EC node for the client application. This decision is taken with the help of edge analyzer which plays an important role in analyzing the traces & spans.

4. **Prepare Edge Manager:** Edge Manager is another application which runs on the ego vehicle container that uses the real-time messages transmitted by edge devices and EC nodes to perform the allocation and de-allocation of the EC nodes for the edge offloaded tasks.

5. **Prepare Event Manager:** Event Manger is the core part of EDP, which is used to read the json file that defines the scenarios and then executes these scenarios to perform different task on the EC node and network operators.

5.4 Solution A

Here, the experiments are performed for Solution A and different scenarios related to CPU stress and network laod are used to evaluate the performance and QoS of the edge client application to perform the EC node switching if there is a high CPU stress on the EC node or high network load on the network operator.

5.4.1 CPU Stress Experiment

The Stress Load Experiments are performed by putting the EC node nodes on the load. To perform the CPU stress experiment, the scenario is defined in the JSON file which is used by the event manager of EDP to trigger the CPU stress on the EC nodes based on the values defined in the JSON file.

As shown in the scenario file in fig 23, the CPU stress is put to 80% on the EC nodes named **svr101** and **svr102** for 15 seconds respectively, which results in the CPU load on the EC nodes to reach 80% for 15 seconds, which affects the application performance and QoS.

Simultaneously, the Edge Analyzer application is getting the hostmetrics from the clickhouse DB using the SQL query to monitor the CPU load on the EC nodes. Once the EC node CPU load reaches to threshold 80%, it sends the signal to the Edge Navigator to switch to a different EC node which has a lower CPU load.

Until the whole scenario is completed, the client keeps sending the images to the EC node, but the EC node keeps changing based on the CPU load on the EC nodes.

5.4.2 Network Load Experiment

The Network Load Experiments are performed by putting the network that the client is using to connect with the EC node to send the images.

Since the Network Load part relies on the analyses of spans & traces data, if the network bandwidth is too bad, and then, it takes a lot of time for the traces to show the E2E latency of the client server application. Therefore, it takes a lot of time to trigger the switching to different EC node on different network operator.

In fig 24, the scenario file for the network load experiment is shown. The network load is put as 5 MB and bandwidth of the network route is decreased to 5mb for 15 seconds. With this way, we achieve 100% network load on the Ethernet route defined in the Containerlab, which results in the network load and affects the application performance and QoS.

5.5 Solution B

In this subsection, the experiments are performed by using tools defined for Solution B and different scenarios are used to evaluate the performance and QoS of the client application to perform the EC node switching if there is a high CPU stress on the EC node or high network load on the network operator. The difference with this solution compared to Solution A is that in the solution, the setup is using the zenoh router to connect the client, edge manager and server applications. It provides a simple way to discover the topics, publish, and also subscribe to the topics.

5.5.1 CPU Stress Experiment

The CPU Stress Experiments are performed similar to Solution A by putting the EC node nodes under the stress by using stress-ng. To perform the CPU stress experiment, the scenario is defined in the JSON file which is read by the event manager of EDP to trigger the CPU stress on the EC nodes based on the values defined in JSON file shown in the figure 23. When the CPU stress reaches 80%, then Edge Manager receives the CPU usage information from heartbeat message. After that, Edge manager triggers the switching of the processing from EC node A to another EC node B that has low CPU usage. Then, Event manager waits for time defined in the JSON file and trigger the CPU stress on EC node B. This process continuously runs until the whole scenario with multiple experiments is complete.

5.5.2 Network Load Experiment

The Network Load Experiments are also performed similarly by putting load on the network by using the event manager of EDP with JSON file which is used by the event manager to trigger the network load on the network operator through the load client and load server based on the values defined in the file shown in the figure 24.

6 Results and Discussion

The result shows the capability of the EDP which can be used for various edge application running on Ego vehicle providing the control loop based switching with different servers to maintain the QoS for the edge application. This EC node switching is based on CPU stress on the EC node which is processing the offloaded tasks like inference the images with YOLOP DL model to detect the lanes, segment drivable area and detect objects, and even, and also on network latency on different network operators. For example, the network latency is terrible for the edge application to process the images based on the predefined threshold, it should switch to more reliable network operator. The results from solutions A and B are described in below subsections.

6.1 Solution A

The experiment is performed multiple times for the stress test on the edge node. The results show that the average time required for the client application to switch between two edge nodes is 2-3 seconds as shown in the figure 18. This result includes the time of the trigger at which the request from the EDP was sent to the edge node to increase the CPU load on it. The time taken by the server application which processes the images with YOLOP model to detect the lanes, segment drivable area and detect objects, increases gradually from the 40% to 80% and the traces of the client server communication is analyzed by the Edge Analyzer application which then triggers the Edge Navigator application which assigns the client application to a different edge node based on the stress on the edge node.

The control loop of the whole application is used to maintain the QoS of the client application running on the Ego vehicle. The QoS is maintained by switching to different edge node on the same network if the CPU stress on one of the edge node is too high. But it is also very important to know the E2E latency for the client server application to switch between two edge nodes.

For example, when the EDP trigger is sent to the edge node to increase the CPU load to 80%, the stress-ng application running on the edge node increases the CPU load to 80%, but it takes some time for the CPU to actually affect the application and reach the required CPU usage. In the meantime, the hostmetrics of the CPU, and RAM are collected by OpenTelemetry and send to the Clickhouse DB. Now, the Edge Analyzer application is analyzing the CPU usage and if the CPU of an edge node reaches 80%, then the client application is switched to a different edge node by triggering the Edge Navigator application, which sends the request to the client to switch to a different edge node. This whole End-to-End latency for client server application to switch to different edge node takes an average of around 2-3 seconds approximately and the plot is shown below 18:

However, it is not possible to perform the network load experiment on the edge application to trigger the edge node switching for different network operators as the traces of network latency are not available when there is huge traffic on the network. More details are shared in the section 1.

While implementing the Edge Diagnostic Platform to run various experiments for different scenarios related to cpu stress and network load. The proposed solution has some limitations and challenges which are discussed below:

1. **Network Load Calculation** : The network load calculation is based on the traces and spans data which is collected by the OpenTelemetry SDK. The network load calculation is based on the latency of the client server communication. If the network latency is too bad, then it takes a lot of time to calculate the network load and then switch to a different network operator. This can be improved by using the ping response time to calculate the network latency in real time and then switch to a different network operator based on the network latency.

2. **Stress Load Calculation**: The stress load calculation is based on the CPU load on the edge nodes. The CPU load is calculated in real time and then the client switches to the edge node with the lower CPU load. The switching time for the CPU load is less than the switching time for the Network load. The average switching time for the CPU load is less than 2-3 second, whereas the switching time for the Network load is more than 10 second. This can be improved by using the stress-ng tool to trigger the CPU stress on the edge nodes and then calculate the CPU load in real time and then switch to the different edge node based on the CPU load.

3. **End-to-End Latency**: The end-to-end latency for the client server application to switch between two edge nodes is around 2-3 seconds. This latency can be improved by using the

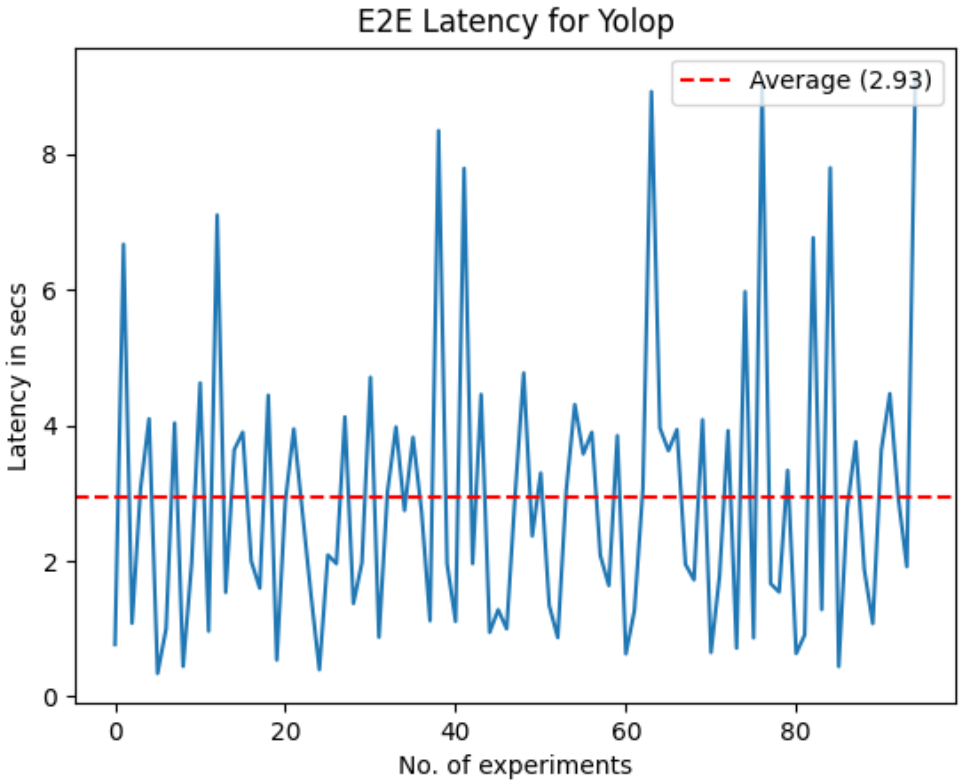


Figure 18: E2E Latency for switching between two edge nodes

stress-ng tool to trigger the CPU stress on the edge nodes and then calculate the CPU load in real time and then switch to the different edge node based on the CPU load.

When testing the various scenarios, the end-to-end latency for the switching is around 2-3 seconds which includes doing 100 experiments. The initial guess for the delay is thought to be stress-ng as it is responsible for triggering the switching by sending the hostmetrics which contains CPU, Memory and Network usage of the system to the Clickhouse DB. Later, this data is fetched from the DB and then the CPU usage for each core is calculated. When performing this whole process, it generates extra latency for the edge analyzer application to make the switching when the stress load is increased.

Therefore, the experiment is performed which includes triggering the stress load on the CPU using stress-ng tool through subprocess (another python library) and simultaneously fetching the CPU usage from Clickhouse DB using clickhouse-connect python library used to execute the SQL query and get the data from the table which is storing the CPU usage. This data is then used as CPU usage for the Edge Analyzer application which is used to trigger the switching. After the running experiment for 50 iterations, the result is as expected, the average latency to fetch the CPU usage, when it reaches 80% of the usage is 2000-3000 ms which is huge for an automotive application which has to offload the processing for deep learning model task.

Considering these results, the main cause of end-to-end latency for the switching between different edge nodes when the CPU stress is applied is Clickhouse DB and Hostmetrics. Also, by giving more thought to this approach, it doesn't make sense to store the cpu usage in the DB and then fetch it if it is giving huge latency for an application which require ultralow latency and quickly switch to different edge server when there is huge CPU stress on the edge node.

However, when performing the experiment of checking the time to reach 80% of CPU stress from the edge node server application without using Clickhouse DB, the results shows that the latency is quite low around 15 ms as shown in Figure 19 which is surprising. This means that the application which is running on the edge node can detect the CPU stress much faster

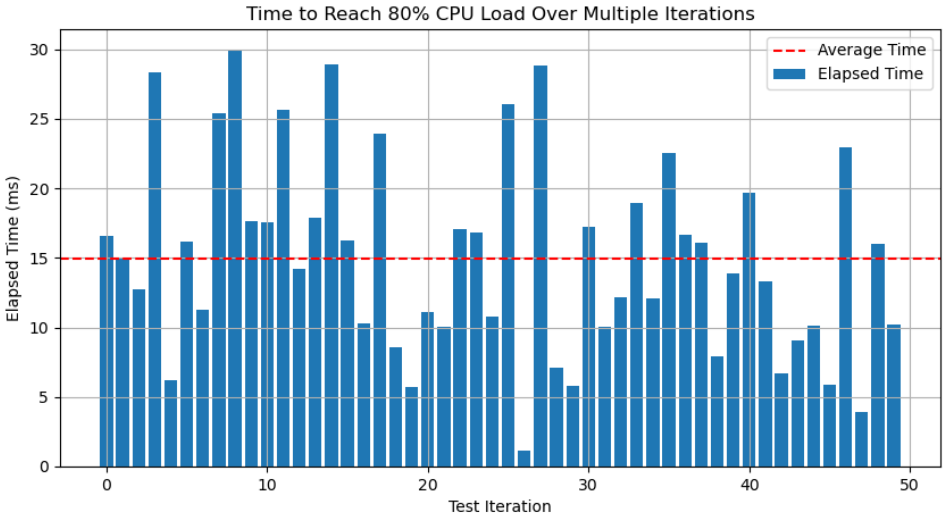


Figure 19: Time to reach 80% of CPU usage when trigger the stress with Stress Ng

Another experiment is also performed for testing the time to check the CPU stress as in the Figure 20, the stress-ng event is triggers to put the stress of 80% on the CPU Core 0 for 10 seconds. This experiment is executed every 15 seconds and then the CPU usage is recorded simultaneously. The result generated from this experiment shows that the trigger event and CPU stress both reach 80% at the same interval of time.

By doing these two experiments, the conclusion is that the stress-ng is not causing any delays in the E2E latency for the edge nodes switching tasks, but the latency caused by Clickhouse DB which provides the CPU stress data to the Edge Analyzer application. Therefore, the decision is made to remove the Clickhouse DB from the architecture and directly send the CPU stress data to the Edge Manager application which will be used to trigger the switching of the edge nodes.

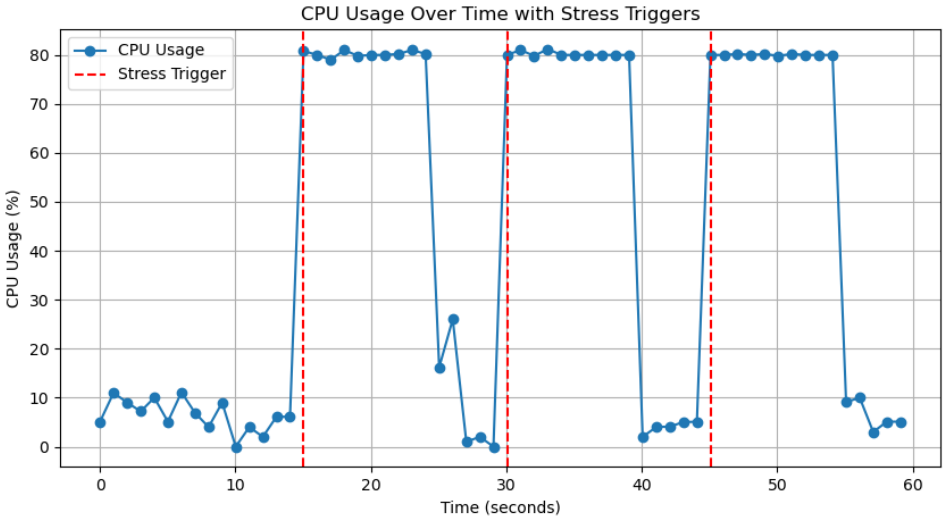


Figure 20: CPU usage over time with Stress Triggers

6.2 Solution B

This section describes the results from solution B, which are executed to switch the EC nodes for two different scenarios for network load and CPU stress.

Considering these evaluations, the decision is made to reimplement the architecture and applications which can be improved to reduce this 2-3 E2E latency further for switching between different edge nodes which are either on the same network as well as on different network.

Furthermore, it shows the results for switching of EC nodes using control loop-based mechanism

which takes the CPU stress of EC nodes as input to trigger the switching of these EC nodes for offloading of the edge client application. With this solution, E2E latency for switching to different EC node when EDP triggers the event to stress CPU is 185 ms as shown in the Figure 21.

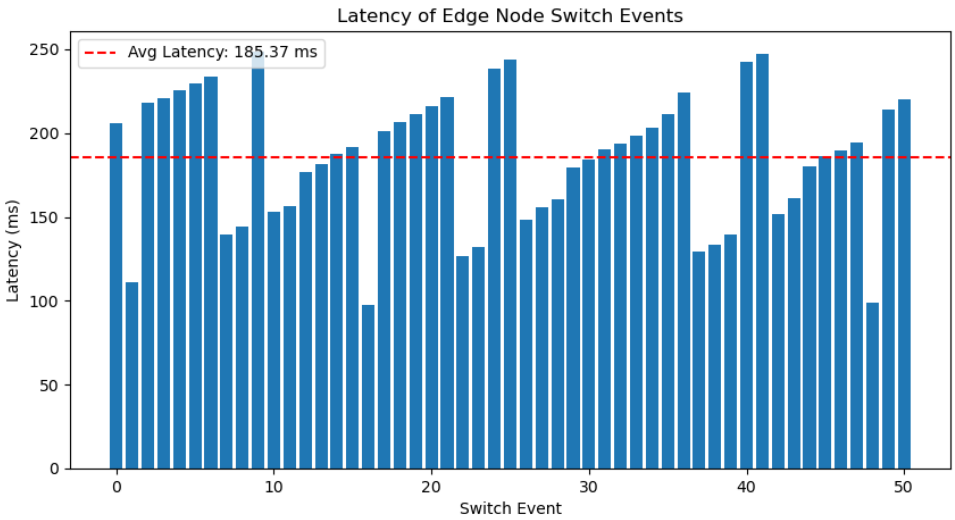


Figure 21: CPU usage over time with Stress Triggers

Also, it describes the implementation of the edge manager which is the core of this solution, and is also used to trigger the switching of EC node from one network operator to another network operator, whenever there is an E2E network latency exceeds the threshold for the client application to maintain the QoS. With this experiment, the average network latency is 372 ms as shown in the Figure 22. This latency includes the 200 ms which is introduced as threshold, that the edge manager uses, to check the heartbeat of EC node server, and edge client to make sure that they are alive or not. If this threshold of E2E network latency is exceeded by EC node or ego vehicle, then edge manager switch the processing to different EC node which has low latency than threshold defined.

Similarly, this solution also has some drawbacks and limitations which are discussed below:

1. **Lack of diagnostic capability**: This solution doesn't provide a way to get the diagnostic information about the edge application. Therefore, it becomes difficult to analyze the fault in the infrastructure or the edge application.

2. **No adaptive response**: The current solution doesn't account for anomalies or changes in the heartbeat message, causing it to switch states immediately upon detection, rather than adapting to changing conditions. It also doesn't provide a way to adaptively respond to the changes in the network load or CPU stress.

3. **Fixed location of Edge Manager**: The edge manager is deployed on the edge client (ego vehicle) which is used to trigger the switching of EC nodes by measuring the CPU stress and network latency. However, the solution doesn't consider the network latency of edge client when it is communicating with the edge manager at different location. This can lead to delays in switching the EC nodes when the network latency is high.

6.3 Comparison of Solutions A and B

This section describes the major difference between the two solutions A and B. The solution A and B are similar in terms of the architecture and the edge applications used to perform the switching of EC nodes for CPU stress and network load scenarios on different network operators. Both solutions are using the same edge application which is transmitting the images to the EC node to process the images with YOLOP DL model and then EC node sends the response back to the client application. Also, the EDP Event Manger is used to trigger various events related to the CPU stress and network load based on the scenarios defined in the file as shown in the figure 14 and 15. The only difference is that the solution A uses Clickhouse DB to store the spans and traces data for the CPU usage and also network latency, whereas the solution B uses process called

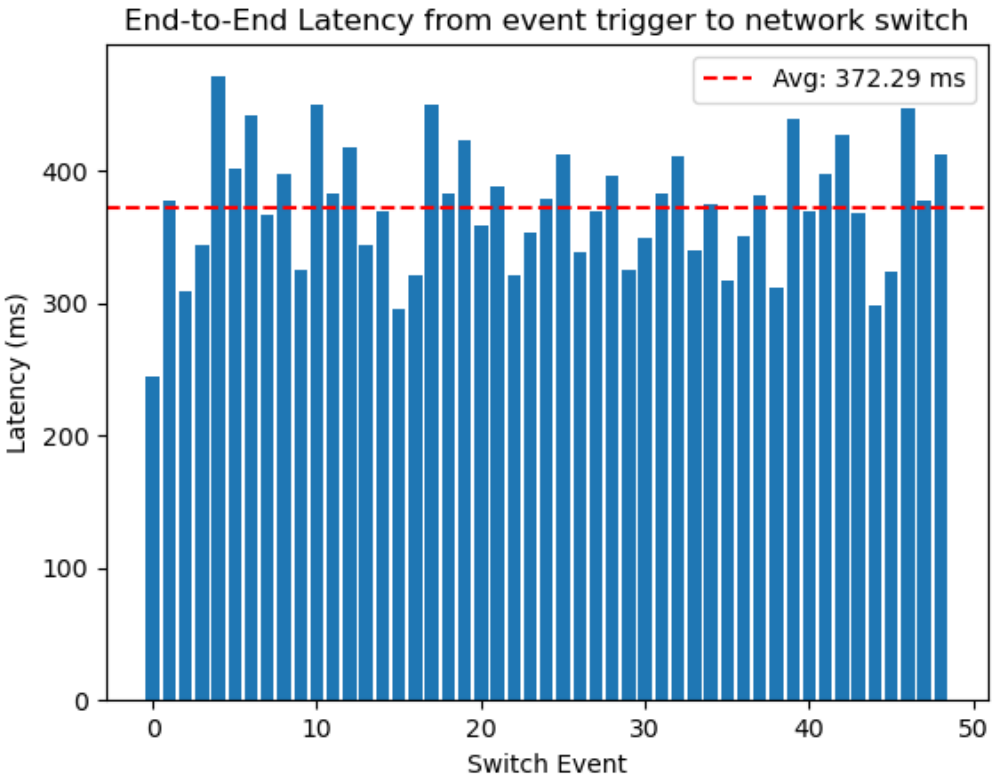


Figure 22: Network switch E2E latency

heartbeat which is used to check the liveliness, CPU usage and network latency of the EC nodes and edge client application. This difference can be visualized from the differences in the layer based architecture for solution A shown in figure 10 and for solution B shown in figure 13, which basically removes the data collection layer from the solution A layers and combine the Edge Analyzer and Edge Navigator to form Edge Manager, which is processing the real-time information messages from Edge Devices, compared to logging the traces & spans into DB and then fetching this data to analyze and perform the switching of EC nodes.

Hence, the solution A uses the OpenTelemetry SDK to collect the spans and traces data from the edge node server application and then store it in the Clickhouse DB. The solution B uses the Zenoh communication protocol to send the CPU usage and network latency data to the edge manager application which is used to trigger the switching of EC nodes.

Although, both solutions use the same architectural framework which is based on EDP and Containerlab, that is used to define the topology of the infrastructure with various EC nodes, routers to define separate network, load clients/servers and also the edge client. Also, the edge analyzer and navigator applications of Solution A are deployed on the edge client and similarly, the edge manager application of Solution B is also deployed on the edge client itself.

Therefore, the **Solution B** is more efficient than the **Solution A** as it provides the real-time response for the edge application to switch the EC nodes and network operators. The average switching time for CPU load is 185 ms and for network load is 372 ms as shown in the table 1. While the solution A takes around 2-3 seconds to switch between two edge nodes which is huge for an automotive application, to offload the processing for deep learning model task and network latency is difficult to measure in a constrained network as the traces and spans has to be sent to the Clickhouse DB. However, the solution A latency can also be improved by reducing the latency involved in fetching the resources of EC nodes from the Clickhouse DB, which is introduced by the Hostmetrics application. Also, the Solution B can be used for the edge applications, which require real-time response to switch the EC nodes and network operators based on the CPU stress and network load, without caring about the cause of latency. While, the solution A can be used for the edge applications which require the diagnostic information about the edge application and also the infrastructure to analyze the fault in the system. This diagnostic information can further be used to improve the performance and robustness of the edge application over time.

Scenario	Solution A (ms)	Solution B (ms)
Switching Time for CPU Stress	2000–3000	185
Switching Time for Network Load	>10000	372

Table 1: Comparison of scenarios for Solution A and B

As shown in table 1, Solution B provides the real-time response for the edge application to switch the EC nodes which ensures the proper functioning of the application and maintains the QoS of the edge application, on different network operators based on the CPU load and network load,

7 Conclusion and Outlook

This research thesis provides architectural framework for maintaining the QoS for edge application by switching the processing tasks from one EC node to another. These processing tasks are using AI based Edge applications in the context of automotive, which can be used to offload the high computation tasks to EC nodes. Moreover, the framework provides a control loop-based mechanism to allocate the offloading of tasks, and also switch the offloading to another EC node if resources of EC node is not sufficient or the network latency is high.

To further enhance the capabilities of EC, The Containerlab when integrated with the EEDP, provides a simple and efficient mechanism for emulating various real-world scenarios related to network load as well as CPU stress. This functionality is very crucial for testing applications, validating their behavior under diverse conditions, and also diagnosing failures. Through the use of spans & traces of function callbacks, and host metrics, the system enables monitoring and performance optimization. These insights allow developers to analyze and address inefficiencies, leading to improved system reliability, robustness, and QoS. Moreover, it is using Zenoh and V2X to communcate with all edge devices including edge client and EC nodes which can share the real-time data as heartbeats that can be used to estimate the network latency and also CPU usage for individual EC node. This crucial information is then used for allocating and switching the processing to different EC nodes.

In this study, different scenarios related to CPU stress and Network load are tested for client-server application. The edge application mainly involves a client that transmits images to an EC node, where a DL model inferences the image and transmits back the result to the client. When CPU or network load is triggered by the EDP, and reaches beyond a predefined threshold, the client dynamically switched between EC nodes and network operators based on resource availability. This dynamic allocation ensured continued service reliability and efficiency, preventing bottlenecks in computational and network resources.

For Solution A, the E2E switching latency for CPU stress is observed to be significantly lower than that of network load switching as shown in the table 1. This is because for measuring the network latency of whole trace of the application require the execution of all the spans which consists of span for sending the image and span for receiving the result image. These spans take a lot of time, when the network is loaded heavily with the network traffic using load client and server provided by EDP. Specifically:

- The average switching time for CPU stress is less than 2–3 seconds.
- The average switching time for network load exceeded 10 seconds.

Moreover, the switching time for CPU stress can be attributed to real-time CPU stress monitoring which is done by fetching data from Clickhouse DB, allowing the edge client to rapidly transition to an EC node that has lower computational usage. In contrast, network load switching is inherently slower due to its reliance on spans and traces, which require time to complete the span for transmitting and receiving the images in constrained network conditions.

Whereas, in Solution B, the switching latency of CPU stress is less than the switching latency of network load as shown in the table 1, as the threshold for switching in CPU scenario is 100 ms and for network load is 200 ms. Thus average switching time for CPU stress is 185 ms, which provides the real-time response for the edge application to switch an EC node having CPU stress less than predefined threshold. Furthermore, for network load, the average switching time is 372 ms, which also provides real-time response for the edge application to connect with another EC node on different network operator, having E2E network latency less than threshold. These results are as follows:

- The average switching time for CPU stress is 185 ms.
- The average switching time for network load is 372 ms.

The switching time for CPU stress can be attributed to real-time CPU stress by using heartbeat message from EC node, allowing edge client to rapidly switch to an EC node with lower computational stress. Also, network load can also be monitored by heartbeat message from EC node, which provides the E2E network latency between edge client and EC node. This allows the edge client to rapidly switch to an EC node with lower network load.

In summary, the proposed architectural framework provides both solutions to maintain the performance and QoS of the edge application in distributed systems. In Solution A, the Edge

Analyzer and Navigator applications serve as critical elements to fetch the spans & traces of application, whereas Solution B relies predominantly on the edge manager for monitoring heartbeats of edge client and EC nodes. These components facilitate seamless transitional processes between EC nodes and network operators. The EDP implements dynamic switching protocols in response to computational and network load thresholds, thus preserving edge application stability and responsiveness. This maintenance of QoS is particularly significant in automotive industry applications, specifically within ADAS and autonomous driving technologies, where performance, reliability and QoS are paramount for safety and efficiency. While Solution A offers enhanced diagnostic capabilities through the use of spans & traces, thereby facilitating the identification and resolution of communication or computation failures, but Solution B lacks such diagnostic support and focuses solely on triggering EC node switching when predefined thresholds for network load and CPU utilization are exceeded. However, these solutions doesn't consider scenarios where all EC nodes are having high computation or all network operators have high traffic load, which are not in the scope of this thesis. This could be an interesting field for future research.

8 Future Work

This research thesis has presented an architectural framework for testing edge applications under various conditions such as network load and CPU stress, emulating real-world infrastructure to uphold performance and Quality of Service (QoS) in the context of automotive edge computing. While the implemented framework demonstrates promising results, several areas remain for further exploration and enhancement.

1. **Optimizing switching algorithms**: Future research should also focus on optimizing the switching algorithms and integrating advanced predictive models to anticipate CPU and network load fluctuations. Additionally, empirical results, including performance plots, should be incorporated to illustrate CPU stress and network load variations when `stress-ng` and `Load Client` are triggered, providing deeper insights into system behavior under CPU stress and network load conditions. With this, the switching latency for e2e application can be further reduced, and the QoS can be maintained.

2. **Anomaly detection for Resource Load Variation**: Another critical direction involves implementing anomaly detection mechanisms to mitigate false triggers caused by transient spikes in CPU and network usage. For example, system-level tasks may induce high CPU usage that does not necessitate edge application migration. Failure to distinguish such events from actual performance degradation may lead to unnecessary switching, causing service delays. Integrating anomaly detection can enhance decision-making robustness in dynamic edge environments.

3. **C++ based inference for DL model**: Moreover, Transitioning from Python to C++ for DL model inference offers substantial performance gains. Because, using C++ for model inference can significantly enhance performance compared to Python, as C++ is a compiler based language, which generally results in faster execution times due to optimized machine code. Additionally, C++ provides better control over system resources, such as memory and CPU usage, allowing for more efficient utilization of hardware capabilities. This can be particularly beneficial for real-time applications where low latency is crucial. Furthermore, many DL libraries, such as TensorRT, Libtorch, ONNX and OpenVINO, offer C++ APIs that are optimized for high-performance inference on various hardware platforms, including GPUs and specialized accelerators. By leveraging these libraries, the inference speed and overall system performance can be greatly improved. This research paper [28] provides a comprehensive performance on execution times of various frameworks.

4. **Model Quantization for Edge Devices**: Another area where the performance can be improved is the model quantization. The model quantization is a technique that reduces the size of the model and improves the inference time. This can be done by reducing the precision of the weights and activations from 32-bit floating-point to lower bit-width representations, such as 8-bit integers. This can lead to significant improvements in performance, especially on edge devices with limited computational resources.

5. **Selective Data Transmission for Reduced Latency**: To further optimize latency, only essential output data—such as object coordinates for detection, lane boundaries, or drivable area indicators, should be transmitted instead of the complete inference images. This selective transmission approach should minimize both data transfer and processing overhead, thereby improving responsiveness and making the system more viable for real-time applications in autonomous driving and ADAS.

6. **Validation on Real-World Edge Devices**: Finally, comprehensive validation on actual edge hardware is required to assess the system's real-world performance and QoS under practical scenario conditions. The same stress and network load scenarios should be applied to ensure reliability and resilience. Results should be benchmarked against existing platforms, such as EDP and Containerlab, to quantify improvements and identify further optimization opportunities.

References

[1] Draw.io icons, 2025. URL: https://app.diagrams.net.

[2] Flat icons, 2025. URL: https://www.flaticon.com/authors/flat-icons.

[3] Signoz icon, 2025. URL: https://signoz.io/.

[4] Georgios Boulougaris and Kostas Kolomvatsos. A qos-aware, proactive tasks offloading model for pervasive applications. In *2022 9th International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 24–31, 2022. doi:10.1109/FiCloud57274.2022.00011.

[5] All About Circuits. New network protocol zenoh slashes energy required to send data, 2024. URL: https://www.allaboutcircuits.com/news/new-network-protocol-zenoh-slashes-energy-required-to-send-data/.

[6] Clickhouse. Clickhouse, 2024. URL: https://github.com/ClickHouse/ClickHouse.

[7] Angelo Corsaro. Zenoh: Unifying communication, storage, and computation from edge to cloud, 2024. URL: https://www.linkedin.com/pulse/zenoh-unifying-communication-storage-computation-from-angelo-corsaro/.

[8] Roman Dodin. Containerlab, 2024. Accessed: Jun. 18, 2024. URL: https://github.com/srl-labs/containerlab.

[9] Alexey Dosovitskiy, Germán Ros, Felipe Codevilla, Antonio M. López, and Vladlen Koltun. CARLA: an open urban driving simulator. *CoRR*, abs/1711.03938, 2017. URL: http://arxiv.org/abs/1711.03938, arXiv:1711.03938.

[10] Andrés García Mangas, Francisco José Suárez Alonso, Daniel Fernando García Martínez, and Fidel Díez Díaz. Wotemu: An emulation framework for edge computing architectures based on the web of things. *Computer Networks*, 209:108868, 2022. URL: https://www.sciencedirect.com/science/article/pii/S1389128622000767, doi:10.1016/j.comnet.2022.108868.

[11] GNS3. Gns3: The software that empowers network professionals, 2025. Accessed: April. 01, 2024. URL: https://www.gns3.com/.

[12] Göransson, Adrian and Wändesjö, Oskar. Evaluating ClickHouse as a Big Data Processing Solution for IoT-Telemetry, 2022. Student Paper.

[13] Hatem Ibn-Khedher, Mohammed Laroui, Mouna Ben Mabrouk, Hassine Moungla, Hossam Afifi, Alberto Nai Oleari, and Ahmed E. Kamal. Edge Computing Assisted Autonomous Driving Using Artificial Intelligence. In *2021 International Wireless Communications and Mobile Computing (IWCMC)*, pages 254–259, June 2021. doi:10.1109/IWCMC51323.2021.9498627.

[14] InterDigital Communications Inc. Advantedge: Mobile edge emulation platform, 2025. Accessed: April. 01, 2025. URL: https://interdigitalinc.github.io/AdvantEDGE/.

[15] Congfeng Jiang, Xiaolan Cheng, Honghao Gao, Xin Zhou, and Jian Wan. Toward computation offloading in edge computing: A survey. *IEEE Access*, 7:131543–131558, 2019. doi:10.1109/ACCESS.2019.2938660.

[16] Dewant Katare, Diego Perino, Jari Nurmi, Martijn Warnier, Marijn Janssen, and Aaron Yi Ding. A survey on approximate edge ai for energy efficient autonomous driving services. *IEEE Communications Surveys & Tutorials*, 25(4):2714–2754, 2023. doi:10.1109/COMST.2023.3302474.

[17] Muhammad Jalal Khan et al. Advancing c-v2x for level 5 autonomous driving from the perspective of 3gpp standards. *Sensors (Basel, Switzerland)*, 23(4):2261, 2023. URL: https://www.mdpi.com/1424-8220/23/4/2261, doi:10.3390/s23042261.

[18] Qing Li, Shangguang Wang, Ao Zhou, Xiao Ma, Fangchun Yang, and Alex X. Liu. Qos driven task offloading with statistical guarantee in mobile edge computing. *IEEE Transactions on Mobile Computing*, 21(1):278–290, 2022. doi:10.1109/TMC.2020.3004225.

[19] Di Liu, Hao Kong, Xiangzhong Luo, Weichen Liu, and Ravi Subramaniam. Bringing AI To Edge: From Deep Learning's Perspective. *Neurocomputing*, 485:297–320, May 2022. arXiv:2011.14808, doi:10.1016/j.neucom.2021.04.141.

[20] Urwah Muslim and Stephan Recker. A comparative analysis of digital twins for advanced networks. In *2024 IEEE 7th International Conference and Workshop Óbuda on Electrical and Power Engineering (CANDO-EPE)*, pages 281–286, 2024. doi:10.1109/CANDO-EPE65072.2024.10772762.

[21] Urwah Muslim and Stephan Recker. Demo: Emulation platform to build digital twins of edge computing environments. In *2024 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 512–514, 2024. doi:10.1109/SEC62691.2024.00062.

[22] OpenTelemetry. Host metrics, 2024. Version. URL: https://docs.datadoghq.com/opentelemetry/integrations/host_metrics/?tab=host.

[23] OpenTelemetry. Language apis and sdks, 2024. Version. URL: https://opentelemetry.io/docs/languages/.

[24] OpenTelemetry. Opentelemetry, 2024. Version. URL: https://opentelemetry.io/.

[25] OpenTelemetry. Opentelemetry collector, 2024. URL: https://opentelemetry.io/docs/collector/.

[26] OpenTelemetry. Opentelemetry perf testing, 2024. Version. URL: https://opentelemetry.io/blog/2023/perf-testing/.

[27] Jainam Sanghvi, Pronaya Bhattacharya, Sudeep Tanwar, Rajesh Gupta, Neeraj Kumar, and Mohsen Guizani. Res6Edge: An Edge-AI Enabled Resource Sharing Scheme for C-V2X Communications towards 6G. In *2021 International Wireless Communications and Mobile Computing (IWCMC)*, pages 149–154, June 2021. doi:10.1109/IWCMC51323.2021.9498593.

[28] Murat Sever and Sevda Öğüt. A performance study depending on execution times of various frameworks in machine learning inference. In *2021 15th Turkish National Software Engineering Symposium (UYMS)*, pages 1–5, 2021. doi:10.1109/UYMS54260.2021.9659677.

[29] Kanwar Bharat Singh and Mustafa Ali Arat. Deep learning in the automotive industry: Recent advances and application examples, 2019. URL: https://arxiv.org/abs/1906.08834, arXiv:1906.08834.

[30] Yaozhong Song, Stephen S. Yau, Ruozhou Yu, Xiang Zhang, and Guoliang Xue. An approach to qos-based task distribution in edge computing networks for iot applications. In *2017 IEEE International Conference on Edge Computing (EDGE)*, pages 32–39, 2017. doi:10.1109/IEEE.EDGE.2017.50.

[31] Dilip Kumar Vaka. The Convergence of AI, ML, and IoT in Automotive Systems: A Future Perspective on Edge Computing. 11(05):25535–25549. URL: https://ijecs.in/index.php/ijecs/article/view/4673, doi:10.18535/ijecs/v11i05.4673.

[32] Dong Wu, Man-Wen Liao, Wei-Tian Zhang, Xing-Gang Wang, Xiang Bai, Wen-Qing Cheng, and Wen-Yu Liu. Yolop: You only look once for panoptic driving perception. *Machine Intelligence Research*, pages 1–13, 2022.

[33] S D Yashwanth, Srimadh V Rao, Rakshit, Yashass P Meharwade, and Ramesh Kivade. Autonomous driving using yolop. In *2022 IEEE North Karnataka Subsection Flagship International Conference (NKCon)*, pages 1–6, 2022. doi:10.1109/NKCon56289.2022.10127088.

[34] Luyuan Zeng, Wushao Wen, and Chongwu Dong. Qos-aware task offloading with noma-based resource allocation for mobile edge computing. In *2022 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1242–1247, 2022. doi:10.1109/WCNC51071.2022.9771629.

[35] Yukun Zeng, Mengyuan Chao, and Radu Stoleru. Emuedge: A hybrid emulator for reproducible and realistic edge computing experiments. In *2019 IEEE International Conference on Fog Computing (ICFC)*, pages 153–164, 2019. doi:10.1109/ICFC.2019.00027.

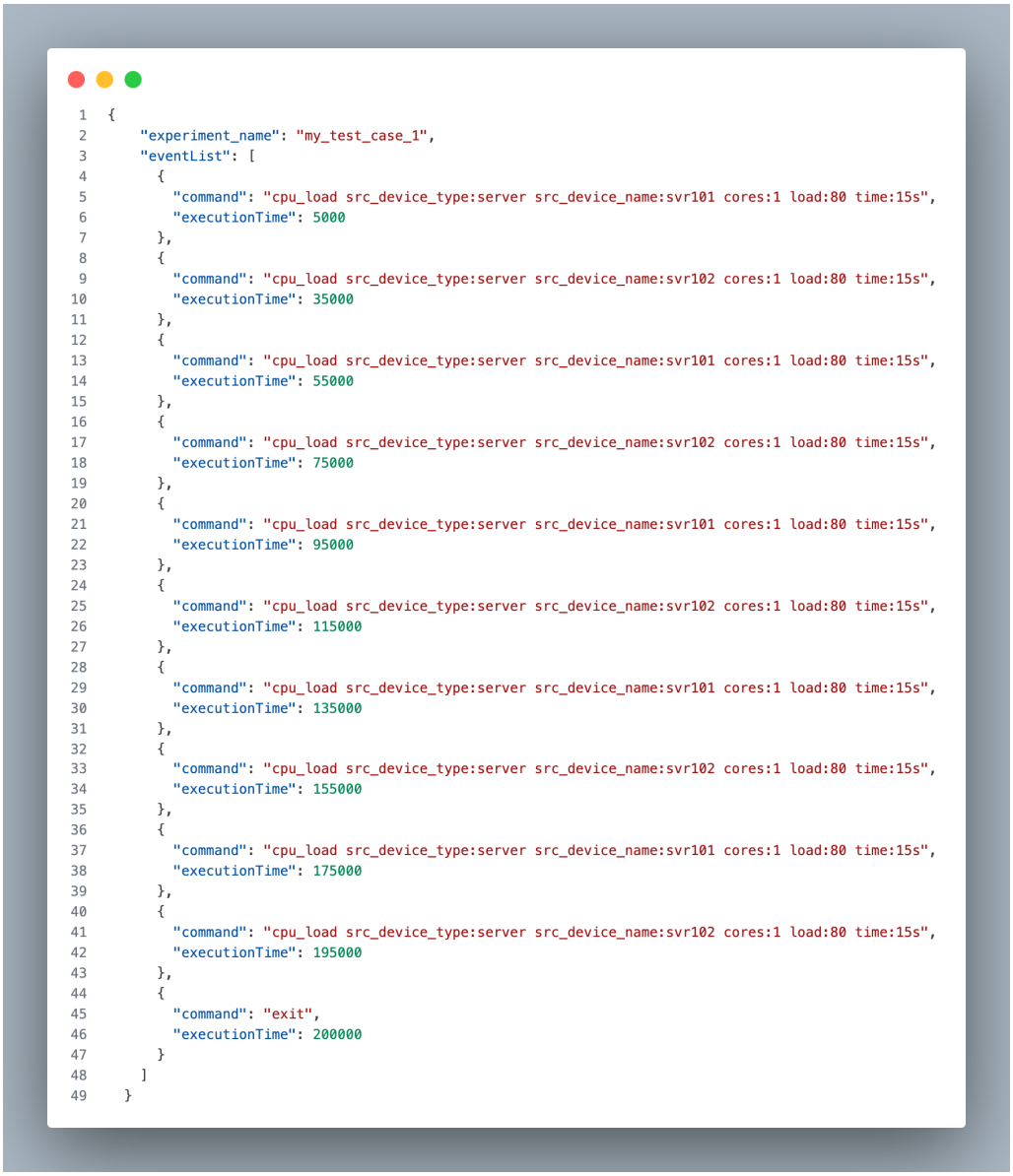
[36] Eclipse Zenoh. Zenoh, 2024. URL: https://zenoh.io/.

A Appendix

Appendix contains the supplementary material that is not included in the main body of the thesis. This includes additional figures, tables, and code snippets that support the findings and discussions presented in the thesis.

The code snippets and data presented in this appendix are part of a larger project hosted on GitHub. The complete source code, along with detailed documentation, can be accessed at the following repository: github.com/sachinkum0009/edge-ai-research-thesis. This repository provides additional context and implementation details for the experiments and solutions discussed in this thesis.

A.1 Additional Figures



```
{
    "experiment_name": "my_test_case_1",
    "eventList": [
      {
        "command": "cpu_load src_device_type:server src_device_name:svr101 cores:1 load:80 time:15s",
        "executionTime": 5000
      },
      {
        "command": "cpu_load src_device_type:server src_device_name:svr102 cores:1 load:80 time:15s",
        "executionTime": 35000
      },
      {
        "command": "cpu_load src_device_type:server src_device_name:svr101 cores:1 load:80 time:15s",
        "executionTime": 55000
      },
      {
        "command": "cpu_load src_device_type:server src_device_name:svr102 cores:1 load:80 time:15s",
        "executionTime": 75000
      },
      {
        "command": "cpu_load src_device_type:server src_device_name:svr101 cores:1 load:80 time:15s",
        "executionTime": 95000
      },
      {
        "command": "cpu_load src_device_type:server src_device_name:svr102 cores:1 load:80 time:15s",
        "executionTime": 115000
      },
      {
        "command": "cpu_load src_device_type:server src_device_name:svr101 cores:1 load:80 time:15s",
        "executionTime": 135000
      },
      {
        "command": "cpu_load src_device_type:server src_device_name:svr102 cores:1 load:80 time:15s",
        "executionTime": 155000
      },
      {
        "command": "cpu_load src_device_type:server src_device_name:svr101 cores:1 load:80 time:15s",
        "executionTime": 175000
      },
      {
        "command": "cpu_load src_device_type:server src_device_name:svr102 cores:1 load:80 time:15s",
        "executionTime": 195000
      },
      {
        "command": "exit",
        "executionTime": 200000
      }
    ]
}
```

Figure 23: Scenario file for CPU Stress Experiment

```
{
    "experiment_name": "test_case_2",
    "eventList": [
      {
        "command": "network_load src_device_type:router src_device_name:router1 interface:eth2 load:5 time:15s",
        "executionTime": 5000
      },
      {
        "command": "network_bandwidth src_device_type:router src_device_name:router1 interface:eth2 bandwidth:5 time:15s",
        "executionTime": 7000
      },
      {
        "command": "network_load src_device_type:router src_device_name:router3 interface:eth2 load:5 time:15s",
        "executionTime": 55000
      },
      {
        "command": "network_bandwidth src_device_type:router src_device_name:router3 interface:eth2 bandwidth:5 time:15s",
        "executionTime": 57000
      },
      {
        "command": "network_load src_device_type:router src_device_name:router1 interface:eth2 load:5 time:15s",
        "executionTime": 115000
      },
      {
        "command": "network_bandwidth src_device_type:router src_device_name:router1 interface:eth2 bandwidth:5 time:15s",
        "executionTime": 117000
      },
      {
        "command": "network_load src_device_type:router src_device_name:router3 interface:eth2 load:5 time:15s",
        "executionTime": 135000
      },
      {
        "command": "network_bandwidth src_device_type:router src_device_name:router3 interface:eth2 bandwidth:5 time:15s",
        "executionTime": 137000
      },
      {
        "command": "network_load src_device_type:router src_device_name:router1 interface:eth2 load:5 time:15s",
        "executionTime": 175000
      },
      {
        "command": "network_bandwidth src_device_type:router src_device_name:router1 interface:eth2 bandwidth:5 time:15s",
        "executionTime": 177000
      },
      {
        "command": "exit",
        "executionTime": 200000
      }
    ]
}
```


Figure 24: Scenario file for Network Load Experiment

A.2 Supplementary Tables

EDP Trigger	Edge Analyzer	Edge Client Switch	E2E Latency (ms)
1719519135765000	1719519136533170	1719519136625370	768.17
1719519165764310	1719519172430530	1719519172436470	6666.22
1719519185761790	1719519186843690	1719519186856600	1081.9
1719519205762180	1719519208826750	1719519208848700	3064.57
1719519225762820	1719519229856650	1719519229917820	4093.83
1719519443204230	1719519443546460	1719519443577280	342.23
1719519473202650	1719519474204060	1719519474232680	1001.41
1719519493199350	1719519497230790	1719519497254770	4031.44
1719519513200070	1719519513647000	1719519513689120	446.93
1719519533200420	1719519535161470	1719519535281560	1961.05
1719519553200830	1719519557822670	1719519557891270	4621.84
1719519573200130	1719519574165180	1719519574219340	965.05
1719519593199740	1719519600296500	1719519600371960	7096.76
1719519613201060	1719519614737290	1719519614813430	1536.23
1719519633198900	1719519636835310	1719519636937590	3636.41
1719519804921070	1719519808817930	1719519808893380	3896.86
1719519834923090	1719519836920090	1719519837058490	1997
1719519854920340	1719519856518580	1719519856525620	1598.24
Total			**2711.45**

Table 2: E2E Latency to switch for CPU stress test with Solution A

EDP Trigger	Edge Client Switch	E2E Latency (ms)
1738911465266534	1738911465472227	205.693
1738911480282267	1738911480393342	111.075
1738911495297189	1738911495514965	217.776
1738911510314076	1738911510534975	220.899
1738911525329686	1738911525555187	225.501
1738911540345357	1738911540574808	229.451
1738911555361319	1738911555594918	233.599
1738911570376183	1738911570515421	139.238
1738911585391805	1738911585535873	144.068
1738911600407367	1738911600655678	248.311
1738911615423013	1738911615575739	152.726
1738911630438660	1738911630595377	156.717
1738911645440306	1738911645617217	176.911
1738911660456237	1738911660637845	181.608
1738911675471179	1738911675658692	187.513
1738911690487374	1738911690678915	191.541
1738911705502138	1738911705599642	97.504
1738911720518898	1738911720719951	201.053
1738911735534075	1738911735740457	206.382
1738911750549707	1738911750761012	211.305
1738911765565311	1738911765781380	216.069
1738911780580125	1738911780801536	221.411
1738911795595150	1738911795721478	126.328
1738911810610475	1738911810742530	132.055
1738911825625148	1738911825863387	238.239
Total		**185.36**

Table 3: E2E Latency to switch for CPU stress test with Solution B

EDP Trigger	Edge Client Switch	E2E Latency (ms)
1739054882936299	1739054883181633	245.334
1739054897937411	1739054898314921	377.51
1739054912939045	1739054913247984	308.939
1739054927939913	1739054928284286	344.373
1739054942940487	1739054943411974	471.487
1739054957942014	1739054958343604	401.59
1739054972943403	1739054973385196	441.793
1739054987945024	1739054988311506	366.482
1739055002945790	1739055003343602	397.812
1739055017947392	1739055018272363	324.971
1739055032948853	1739055033398970	450.117
1739055047950059	1739055048332846	382.787
1739055062950768	1739055063368063	417.295
1739055077952193	1739055078296303	344.11
1739055092953097	1739055093322161	369.064
1739055107954583	1739055108249670	295.087
1739055122955977	1739055123276628	320.651
1739055137956731	1739055138406262	449.531
1739055152957396	1739055153340480	383.084
1739055167958083	1739055168381618	423.535
1739055182958856	1739055183318255	359.399
1739055197960203	1739055198348641	388.438
1739055212960900	1739055213281594	320.694
1739055227961734	1739055228314870	353.136
1739055242963252	1739055243342278	379.026
Total		**372.6498**

Table 4: E2E Latency to switch for Network load test with Solution B

```
def switch_edge_nodes(self):
    while True:
    time.sleep(0.1)
    if self.switch_trigger_flag or not self.switch_trigger_flag2:
        continue
    if self.switch_edge_node not in self.devices.keys():
        continue

    liveliness_edge_server_latency = (datetime.now(timezone.utc) -\
        self.devices[self.switch_edge_node].timestamp).total_seconds() * 1000
    network_latency = self.devices[self.switch_edge_node].latency
    if  network_latency > NETWORK_LATENCY or\
        liveliness_edge_server_latency > NETWORK_LATENCY:
        unix_time = int(datetime.now(timezone.utc).timestamp() * 1000000)
        self.switch_trigger_start.append(unix_time)
        self.switch_trigger_start_file.write(str(unix_time) + "\n")

        timestamp = time.time_ns()
        payload = {
            "device_name": self.switch_edge_node,
            "process": False,
            "timestamp": timestamp
        }
        self.switch_edge_node_pub.put(json.dumps(payload))
        self.switch_edge_node = "edge_server_3" if self.switch_edge_node == "edge_server_1"\
            else "edge_server_1"
        payload = {
            "device_name": self.switch_edge_node,
            "process": True,
            "timestamp": timestamp,
            "namespace": "client1",
            "encryption_key": "",
        }
        self.switch_edge_node_pub.put(json.dumps(payload))

        print(f"switching is triggered as the network latency is {network_latency}")
        print(f"sswitching to server {self.switch_edge_node}")
        self.switch_trigger_flag = True
```

Figure 25: Code snippet for switching EC nodes for Solution B

```
self.are_you_alive_pub : Publisher = self.session.declare_publisher("are_you_alive")
self.i_am_alive_sub : Subscriber = self.session.declare_subscriber("heartbeat",\
                                                self.i_am_alive_cb)

def i_am_alive_cb(self, sample: ZBytes):
    payload = json.loads(sample.payload.to_string())
    recv_time = datetime.now(timezone.utc)
    latency = (recv_time -\
     datetime.fromisoformat(payload['timestamp'])).total_seconds() * 1000 #ms
    device = Device(**payload)
    if device.name not in self.devices.keys():
        self.devices[device.name] = device
    else:
        self.devices[device.name] = device

def check_devices_alive(self):
    while True:
        payload = {
            "timestamp": datetime.now(timezone.utc).isoformat(),
        }
        self.are_you_alive_pub.put(json.dumps(payload))
        time.sleep(self.interval)
```

Figure 26: Code snippet for checking heartbeat of EC nodes for Solution B

```
def send_change_ip_req1(self):
    edge_navigator_url = "http://127.0.0.1:5000/change_ip1"
    data = {'host': '127.0.0.1', 'port': 5000}
    unix_time = int(datetime.now(timezone.utc).timestamp() * 1000000)
    self.file.write(str(unix_time)+"\n")
    print(unix_time)
    try:
        response = requests.post(edge_navigator_url, json=data)
        if response.status_code == 200:
            ip_address = response.text
            print("Received IP address:", ip_address)
            return ip_address
        else:
            print("Failed to get IP address. Status code:", response.status_code)
    except requests.RequestException as e:
        print("Error occurred:", e)
        return "error"


def send_change_ip_req2(self):
    edge_navigator_url = "http://127.0.0.1:5000/change_ip2"
    data = {'host': '127.0.0.1', 'port': 5000}
    unix_time = int(datetime.now(timezone.utc).timestamp() * 1000000)
    self.file.write(str(unix_time)+"\n")
    print(unix_time)
    try:
        response = requests.post(edge_navigator_url, json=data)
        if response.status_code == 200:
            ip_address = response.text
            print("Received IP address:", ip_address)
            return ip_address
        else:
            print("Failed to get IP address. Status code:", response.status_code)
    except requests.RequestException as e:
        print("Error occurred:", e)
        return "error"

def send_change_network_req(self):
    edge_navigator_url = "http://127.0.0.1:5000/change_network"
    data = {'host': '127.0.0.1', 'port': 5000}
    response = requests.post(edge_navigator_url, json=data)
    unix_time = int(datetime.now(timezone.utc).timestamp() * 1000000)
    try:
        self.file.write(str(unix_time)+"\n")
        if response.status_code == 200:
            ip_address = response.text
            print("Received IP address:", ip_address)
            return ip_address
        else:
            print("Failed to get IP address. Status code:", response.status_code)
    except requests.RequestException as e:
        print("Error occurred:", e)
        return "error"
```

Figure 27: Code snippet for Edge Analyzer for Solution A

```
def fetch_data(self, frame=None):
    start_unix_time = int(datetime.now(timezone.utc).timestamp() * 1000000)

    query_result = self.db_client.query(
        "WITH latest_usage AS ("
        "    SELECT server_id, MAX(timestamp) AS latest_timestamp "
        "    FROM server_data_db.server_data "
        "    WHERE server_id IN ('server1', 'server2') "
        "    GROUP BY server_id"
        ") "
        "SELECT sd.server_id, sd.cpu_usage "
        "FROM server_data_db.server_data sd "
        "JOIN latest_usage lu "
        "ON sd.server_id = lu.server_id AND sd.timestamp = lu.latest_timestamp;"
    )

    avg_metric_latency_dict = {metric: latency for metric, latency\
     in zip(*query_result.result_columns)}

    server1_cpu_usage = avg_metric_latency_dict.get('server1', 0)
    server2_cpu_usage = avg_metric_latency_dict.get('server2', 0)
    print("cpu usage server 1: {} ms".format(server1_cpu_usage))
    print("cpu usage server 2: {} ms".format(server2_cpu_usage))

    if self.server_switch:
        self.i += 1
        if self.i > 20:
            self.server_switch = False
            print("server switch reset")
            self.i = 0

    if (server1_cpu_usage > CPU_USAGE) and not self.server_switch1:
        self.send_change_ip_req1()
        self.server_switch1 = True
        self.server_switch2 = False


    if (server2_cpu_usage > CPU_USAGE) and not self.server_switch2:
        self.send_change_ip_req2()
        self.server_switch1 = False
        self.server_switch2 = True

    stop_unix_time = int(datetime.now(timezone.utc).timestamp() * 1000000)

    print((stop_unix_time-start_unix_time)/1000000)
```

Figure 28: Code snippet for Edge Analyzer to fetch CPU usage for Solution A

```
def send_change_ip_req(self):
    edge_navigator_url = "http://127.0.0.1:5001/change_ip"
    self.server_index += 1
    if self.server_index > 1:
        self.server_index = 0
    data = self.list_of_servers_on_network1[self.server_index]
    try:
        response = requests.post(edge_navigator_url, json=data)
        if response.status_code == 200:
            ip_address = response.text
            print("Received IP address:", ip_address)
        else:
            print("Failed to get IP address. Status code:", response.status_code)
    except requests.RequestException as e:
        print("Error occurred:", e)

def send_change_network_req(self):
    edge_navigator_url = "http://127.0.0.1:5001/change_ip"
    self.server_index += 1
    if self.server_index > 1:
        self.server_index = 0
    data = self.list_of_servers_on_network[self.server_index]
    try:
        response = requests.post(edge_navigator_url, json=data)
        if response.status_code == 200:
            ip_address = response.text
            print("Received IP address:", ip_address)
        else:
            print("Failed to get IP address. Status code:", response.status_code)
    except requests.RequestException as e:
        print("Error occurred:", e)
```

Figure 29: Code snippet for Edge Navigator to switch EC node for Solution A