Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Networked Systems and Services

Students' Scientific Conference

paper

Levente Márk Maller

Multi-Access Edge Computing and Deep Learning supported Collective Perception in a Cloud-in-the-Loop simulator

SUPERVISORS

Dr. László Bokor – BME-HIT

Péter Suskovics – Ericsson

BUDAPEST, 2022

Table of Contents

Összefoglaló **3**
Abstract **5**
1 Introduction **7**
2 Multi-Access Edge Computing (MEC) **9**
2.1 Vehicle-to-Everything (V2X) communication in MEC systems 11
2.1.1 MEC-based Vehicle-to-Cloud use cases 12
2.1.2 Collective Perception Services (CPS) in MEC systems 14
3 The Cloud-in-the-Loop simulation framework **15**
3.1 Design and implementation of a telco-grade cloud cluster 17
3.1.1 The layout of the devices 17
3.1.2 Software setup 19
3.2 Implementation of a UDP traffic benchmarking tool 20
3.2.1 UDP traffic benchmarking tool client component 21
3.2.2 UDP traffic benchmarking tool server component 22
3.3 Network traffic control 24
4 Design and implementation of automotive use cases into the Cloud-in-the-Loop simulator **26**
4.1 Deep Learning-based automotive use case implementation 27
4.1.1 Client application 27
4.1.2 AI-based edge application component 28
5 Measurements and evaluation **31**
5.1 Evaluating the QoS of the integrated distributed environment 32
5.2 Evaluation of the Deep Learning-based automotive use case implementation 35
6 Summary **40**
References **42**
List of figures, tables and code snippets **45**
List of abbreviations **46**
Appendix **47**
Appendix 1. Python code implementing a deep learning-based object detection application 47

Összefoglaló

A felhő alapú számítástechnika világában az elmúlt évek során egyre nagyobb szerepet kezdett betölteni az edge computing. Ezeknek a rendszereknek a legnagyobb előnye a felhős erőforrások elosztottságában rejlik. A centralizált felhővel szemben az edge architektúráknak a decentralizált, lokális kiszolgálás a feladata, ezáltal nem csak a hardveres erőforrások elosztottsága, de a hálózati terheltség kiegyenlítése (load balancing) is megvalósítható. Az egyik legnagyobb szabványosító szervezet, az ETSI, Multi-Access Edge Computing [1] néven azonosítja a modellt, amelynek célja egy, a jövő hálózataiba hatékonyan, problémamentesen integrálható, szabványos elosztott felhőinfrastruktúra rendszer kialakítása. Fontos kiemelni, hogy az 5. generációs celluláris mobilhálózatok (5G) architektúrális felépítése kézenfekvő alapot nyújt a MEC rendszerekkel való közös működésre [2]. Ötvözve az edge computing koncepció előnyeit az 5G-s rendszerek szolgáltatásaival, rengeteg új használati eset valósítható meg és a korábbi szolgáltatások hatékonyabbá tétele is kivitelezhető. Ennek az egyik legnagyobb haszonélvezője az autóipar lehet, ahol a járműkommunikációs (V2X) szolgáltatásokból eredő nagy mennyiségű adat hatékony feldolgozására van szükség. Ezen a területen a MEC rendszerek legnagyobb előnyét az adja, hogy a járműveken való lokális információfeldolgozás helyett ezek a feladatok kiszervezhetőek az edge erőforrásokra. Így lehetőség nyílik a szenzor adatok kollektív, fúzióval erősített feldolgozására, amely elősegíti a való világ hatékonyabb virtuális leképezését. A több forrásból gyűjtött információ segítségével sokkal pontosabb, megbízhatóbb visszajelzések generálhatók a forgalmi résztvevők részére, ezzel biztonságosabbá és hatékonyabba téve a közlekedést. Az 5G és MEC rendszerek ötvözésével olyan használati esetek implementálására lesz lehetőség a jövőben, mint a valós idejű, nagy felbontású HAD térképek [3] vagy az optimális járműforgalom-menedzsment. A V2X vonatkozású MEC alkalmazásokban a mesterséges intelligencia tölt majd be kiemelkedő szerepet, melynek nagy előnyét a kollektív adatgyűjtés és adatfúzió mellett a hálózati késleltetések csökkentése és a felhasználói eszközök tehermentesítésének lehetősége adja.

Ezeknek a technológiáknak hatékony implementálása komplex feladat, továbbá olyan biztonság-kritikus területen, mint az autóipar, számos követelmények is meg kell felelni. A valós infrastruktúrákon, valós járművekkel végzett vizsgálatokat megelőzően

az egyik legfontosabb feladat a MEC rendszerek tesztelése, validálása. Erre nyújt megoldást a korábban kialakított, azóta számtalan fejlesztésen átesett Cloud-in-the-Loop (CiL) szimulációs keretrendszer [4], amely forgalomszimulátorból kinyert adatok alapján képes egy valós méretű, telco-grade szintű, Kubernetes [5] alapú edge cloud infrastruktúrát vezényelni és azon nagyfelbontású mérési adatokat gyűjteni. Ez a MEC rendszer teljesítményanalízise mellett lehetőséget teremt komplex, járműipari használati eseteket kiszolgáló cloud-native alkalmazások működésének és teljesítményének mélyreható elemzésére is. A dolgozatban a célom a keretrendszert alkalmazva, a járművek által a MEC infrastruktúra felé továbbított videójel alapú szenzoradatfolyamot modellezve, egy valós, mély tanulás alapú objektumdetekciós modellt implementáló, saját fejlesztésű cloud-native alkalmazás gyakorlati vizsgálata autóipari felhasználás szempontjából. A mérések fókuszában az edge architektúra elosztott erőforrásai közötti váltások okozta hatások vizsgálata áll, továbbá a MEC rendszer által kiszolgált felhasználók (járművek) okozta háttérterhelés miatti teljesítménycsökkenés elemzése. A vizsgálatok során különböző járműforgalmi szituációkat implementálva az AI-alapú objektumdetekciós alkalmazás működését analizálom az edge csomópontok terheltségének, a hálózat bizonyos Quality of Service (QoS) paramétereinek, és a cloud native működés különféle mechanizmusainak a szempontjából.

Abstract

Edge computing has become increasingly significant in cloud-based computing in recent years. The distribution of cloud resources is one of these systems' main advantages. In contrast to the centralized cloud, edge architectures provide local, decentralized services, making it possible to balance network traffic and distribute hardware resources. One of the largest standardization organizations, ETSI, identifies the model as Multi-Access Edge Computing (MEC) [1], which aims to create a standard distributed cloud infrastructure system that can be efficiently and seamlessly integrated into future networks. It is important to emphasize that the 5th generation cellular mobile network's architectural structure (5G) provides an apparent basis for joint operation with MEC systems [2]. By combining the advantages of the edge computing concept with the services of 5G systems, many new use cases can be implemented, and previous services can be made more efficient. One of the biggest beneficiaries of this could be the automotive industry, where it is necessary to efficiently process large amounts of data resulting from vehicle communication (V2X) services. The primary benefit of MEC systems in this area is the ability to outsource information processing tasks to edge resources rather than performing it locally on vehicles. This makes it possible to process sensor data collectively, enhanced by fusion, which promotes a more effective virtual representation of the real world. With the help of information collected from several sources, much more accurate and reliable feedback can be generated for road users, thus making traffic safer and more efficient. By combining 5G and MEC systems, it will be possible to implement such use cases in the future as real-time, high-resolution HAD maps [3] or optimal road traffic management. In V2X-related MEC applications, artificial intelligence will play a prominent role, the great advantage of which is the possibility of reducing the load on user devices in addition to collective data collection.

Effective implementation of these technologies is a complex task, and several requirements must also be met in a safety-critical field such as the automotive industry. One of the most important tasks before tests on existing infrastructures with actual vehicles is evaluating and validating MEC systems. A solution for this problem is the Cloud-in-the-Loop (CiL) simulation framework [4], which was developed earlier and has since undergone numerous developments. It can orchestrate a real-size, telco-grade level, Kubernetes-based [5] edge cloud infrastructure based on information gathered from a

traffic simulator and performing fine-grained benchmarking and data collection. In addition to the performance analysis of the MEC system, this also creates an opportunity for an in-depth examination of the operation and performance of cloud-native applications serving complex automotive use cases. My goal in this paper is to use the framework and model video signal-based sensor data transmitted by the vehicles to the MEC infrastructure to examine a self-developed cloud-native application implementing an actual, deep learning-based object detection model from an automotive use perspective. The measurements focus on exploring the effects caused by relocations between the distributed resources of the edge architecture and the analysis of the performance degradation due to the background load caused by the users (vehicles) served by the MEC system. By implementing various vehicle traffic situations during the tests, I will analyze the operation of the AI-based object detection application in terms of the load on the edge nodes, particular Quality of Service (QoS) factors, and various mechanisms of cloud-native service provisioning.

1 Introduction

In recent years, cloud-native-based services have gained more and more recognition. The technology is already used in many areas, from information communication through web services to banking systems. Cloud-native applications are easily scalable, deployable, and flexible software packages and, due to their design, can be run in any cloud-based environment. Cloud-based technologies have also expanded into a new area called the edge computing paradigm, the most crucial feature of which is that hardware resources are distributed. As a result, the services are closer to the consumers, thereby speeding up information processing and ensuring high-performance computing for the applications that provide the services. By applying these innovative technologies, new solutions have also emerged, such as the 5G Service-Based Architecture (SBA) [6] approach, which implements the network functions that build up the core network of mobile networks with cloud-native applications. With this approach, modern 5G systems can be integrated with edge cloud deployments, thereby taking benefit of the most significant advantages of the two new technologies. One of the largest European standardization organizations, ETSI, is also actively integrating the two systems [2]. Furthermore, the organization has already achieved several results in unifying edge cloud systems. This model is called Multi-Access Edge Computing (MEC) [1]. Many service areas will be able to utilize the opportunities created by the combination of 5G and MEC systems. One such area is Vehicle-to-Everything (V2X) communication. The application of these technologies opens up many new use cases, such as sensor fusion applications based on collective perception. These rely on deep learning-based object detection technologies, for which the distributed resources of the MEC systems provide an excellent execution environment. Furthermore, with the help of 5G systems, a low-latency, high-data-speed communication can be provided, which ensures a stable connection between user equipments and edge servers. In the first part of this work, I will introduce the above technologies and activities and how they are related to my research.

The effective implementation of these technologies is a complex task, and many requirements must be met. Furthermore, building a real system is a costly task. Before this, it is essential to carry out tests related to the operation and performance of these systems. The apparent basis for this is provided by test systems that can replace expensive real-life tests. Among others, these aspects motivated the design and implementation of the Cloud-in-the-Loop (CiL) simulation framework [4], which currently integrates a traffic simulator called SUMO [7] with a real distributed cloud-based environment. With the help of the framework, the operation of edge cloud systems and cloud-native applications can be investigated and evaluated. Using the CiL simulator, I have already achieved research results [4][8]. Since then, I have made numerous improvements to the system. The framework was integrated into a telco-grade edge cloud environment, the component implementing the orchestration of the whole framework was supplemented with new features, and functions executing more detailed and accurate measurements were also implemented. Furthermore, I also developed new supporting application components compatible with the framework. In Chapter 3 of this paper, I will give a comprehensive presentation of the CiL framework and the improvements I have made to it in the last period. Unlike the measurements carried out in the past, using the improved framework, my goal was to examine a cloud-native application implementing an automotive use case from the Quality of Experience point of view. For this, I implemented a cloud-native, deep learning-based object detection application. The task of the edge application is to process the sensor data transmitted by V2C-capable vehicles and then generate relevant feedback for the vehicles based on the extracted information. The implemented application is discussed in detail in Chapter 4. During the measurements presented in the paper (Chapter 5.), I searched for the answer to how the operation of the application in the edge environment affects its functionality at the application-level and its QoE. To do this, I defined a KPI and specified scenarios that realize various loads on the edge cloud environment and then investigated and evaluated their impact. Furthermore, I also performed measurements examining the Quality of Service (QoS) of the integrated distributed environment.

2 Multi-Access Edge Computing (MEC)

MEC systems' main advantage comes from the distribution of physical cloud resources. These resources are also located close to the user equipments (UEs) in the network architecture. This way the computational tasks can be executed locally in a decentralized manner, which also creates an opportunity to offload certain tasks that were originally performed on the UEs. Due to their architectural design, MEC systems can also balance the traffic load over the network. By processing data at the edge, these systems can reduce the volume of the data to be sent. Multi-Access Edge Computing is meant to be the enabler of various 5G functionalities in the future [9]. To provide high bandwidth, ultra-reliable connection integrating MEC is the key. One of the most prominent standardization organizations, ETSI [10], also developed a model that defines the joint operation of the 5th-generation cellular mobile networks with MEC systems [2]. One of the most significant innovations in 5G systems is the Service-Based Architecture (SBA), which defines a cloud-native, modular core network. In SBA, all Network Functions (NFs) can communicate over a service-based interface, and the whole architecture relies on network virtualization, and Software Defined Networking (SDN). SBA's cloud-native approach provides an optimal basis for integrating MEC architectures. In [2] ETSI also presents the functionalities of the 5G specification that are key enablers for MEC integration. These functionalities are defined in 3GPP's 5G specification [11]:

1. Local Routing and Traffic Steering: The ability provided by the 5G Core Network to select the traffic in the local area network to be routed to the applications in the distributed cloud.
2. The ability to control User Plane Function (UPF) selection and traffic routing by an Application Function (AF) using the Policy Control Function (PCF) or the Network Exposure Function (NEF).[1]

[1] https://telecompedia.net/5g-core-network-overview/ (Accessed 30 Oct. 2022)

3. The ability to use specific Session and Service Continuity[2] (SSC) modes in different mobility scenarios.

4. The ability provided by the 5G Core Network to support MEC applications to connect Local Area Data Networks (LADN) in an area where they are deployed.

In the SBA all functions (NFs/AFs) expose or accept services. All services are based on request-response or subscribe-notify communication models. The API framework (defined in cit ETSI ISG MEC) of MEC systems implements a similar communication model between its applications to that of the SBA. This makes the information exchange between the entities of the systems remarkably efficient. The nearly identical APIs enable very close integration of the two systems. This way, MEC systems can natively communicate with the 5GC's NFs (e.g., PCF, NEF), control data plane traffic between UEs and MEC applications, and optimize the operation of the distributed cloud resources to provide efficient services (Figure 2.1).

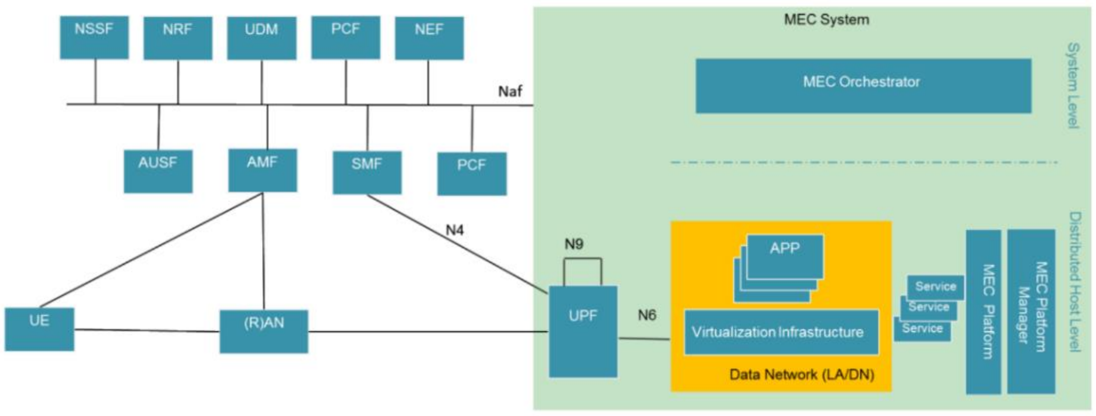


Figure 2.1 5G and MEC integration [2]

Such a deep integration of the two systems creates an opportunity to exploit the main advantages of distributed cloud-based computing and the most important innovations of 5th Generation mobile networks. Thanks to the distributed computing resources of MEC systems, applications can be placed close to user devices. With the modern radio access networks of 5G systems, a reliable, high-bandwidth connection can be established between network entities. Furthermore, through the close cooperation of

2 https://www.cisco.com/c/en/us/td/docs/wireless/ucc/smf/2020-03-0/b_ucc-5g-smf-config-and-admin-guide_2020-03/SMF_chapter_0111.pdf (Accessed 30 Oct. 2022)

the two systems based on cloud-native processes, all services can be managed together; thus, the operation of the applications running in the cloud can be adjusted to the network conditions and the mobility aspects of the UEs. To take advantage of the functions provided by the integrated system, it is also necessary to develop new use cases and applications. The option of offloading processing tasks should be highlighted, the great advantage of which is that load on the UEs can be reduced. The latter is especially beneficial for IoT devices, where support for limited hardware resources can be ensured in addition to energy efficiency. By taking advantage of local processing, it is also possible to distribute the entire network load since, after processing on the edge servers, only the relevant information needs to be forwarded to the other entities of the network. However, one of the most significant advantages is the possibility of joint, fusional processing of data delivered by UEs.

2.1 Vehicle-to-Everything (V2X) communication in MEC systems

One of the most promising areas that will be able to utilize MEC systems is the automotive industry. There are already many areas of V2X communication that provide direct, Wi-Fi-based (802.11p [12]), low bandwidth, and low latency communication between road users (vehicles, pedestrians) and roadside units. Services based on these technologies are reliable but have several limitations. One of their disadvantages is that they are incapable of high-speed data transmission, which is not ideal for forwarding large amounts of data collected by sensors on modern vehicles. Furthermore, if the density of users increases - due to the finite channel capacity - throughput decreases, thereby degrading the quality of the service [13] and endangering functional safety [14][15]. However, these systems can ensure low-latency, reliable communication, which is extremely important in a safety-critical area such as traffic safety. For the Vehicle-to-Cloud (V2C) communication model, providing low latency is currently the critical factor. Yet, MEC systems integrated with 5G can bring a breakthrough in reducing communication delay. Creating an efficient, reliable system is a complex task, so it is conceivable that solutions based on new technologies will serve the needs in a hybrid way with the techniques already used in practice [16]. MEC-based V2C models supported by the 5G system also enable the implementation of many new use cases that were not possible before. Furthermore, it can improve the efficiency of previous use cases developed on other technologies. The most significant advantage of the new approach is that the processing of the environmental sensor data collected by the vehicles can be offloaded to the distributed edge resource of the MEC systems, which provide the best access in terms of network availability. This allows sensor data fusion and simultaneous environmental information processing from multiple sources. This way, the physical environment can be mapped onto the digital representation more accurately and reliably.

ETSI is also actively involved in applying MEC systems to V2X technologies. A unique API system was created for efficient integration called the V2X Information Service (VIS) API [17]. This service enables interoperability in V2X communication in environments with multiple vendors, networks, and access types. In the case of MEC-based V2X use cases, it is essential to maintain service continuity, especially in cases where, due to the movement of the vehicle, its service has to be relocated into another MEC system or when it changes mobile networks between countries. With the help of VIS, information can be provided about V2X services, which can be used to prepare vehicles for these handovers and minimize service downtimes. In addition to ETSI, 3GPP [18] is also actively involved in V2X communication solutions supported by mobile network communication. They created a so-called V2X Application Server concept [19]. By integrating it into the core network, these V2X Application Servers, like the VIS, would help interoperability between different domains.

2.1.1 MEC-based Vehicle-to-Cloud use cases

By applying MEC systems, it is possible to implement many new automotive use cases or to develop previously developed solutions further. ETSI has defined four categories for classifying use cases [20]. The first, the *safety* category, encompasses those use cases intended to increase traffic safety and reduce the risk of accidents. An example of such a use case is Intersection Movement Assist, which helps coordination at road intersections, especially when vehicles cannot see each other. Other categories include the *convenience* type of use cases, for example, software updates or telematics. Furthermore, an *advanced driving assistance* category was also defined, which provides for use cases based on sharing large amounts of data with high reliability and low latency. An example of this is the High Definition (Local) Maps use case, which will be able to use the potential of MEC systems effectively. By processing the environmental data transmitted by the vehicles with the help of HD Maps, it is possible to implement continuously updated local maps running on edge servers. These multi-layered maps can contain various information, such as processed environmental information transmitted by vehicles and virtual representations of other traffic participants (Figure 2.2). Object

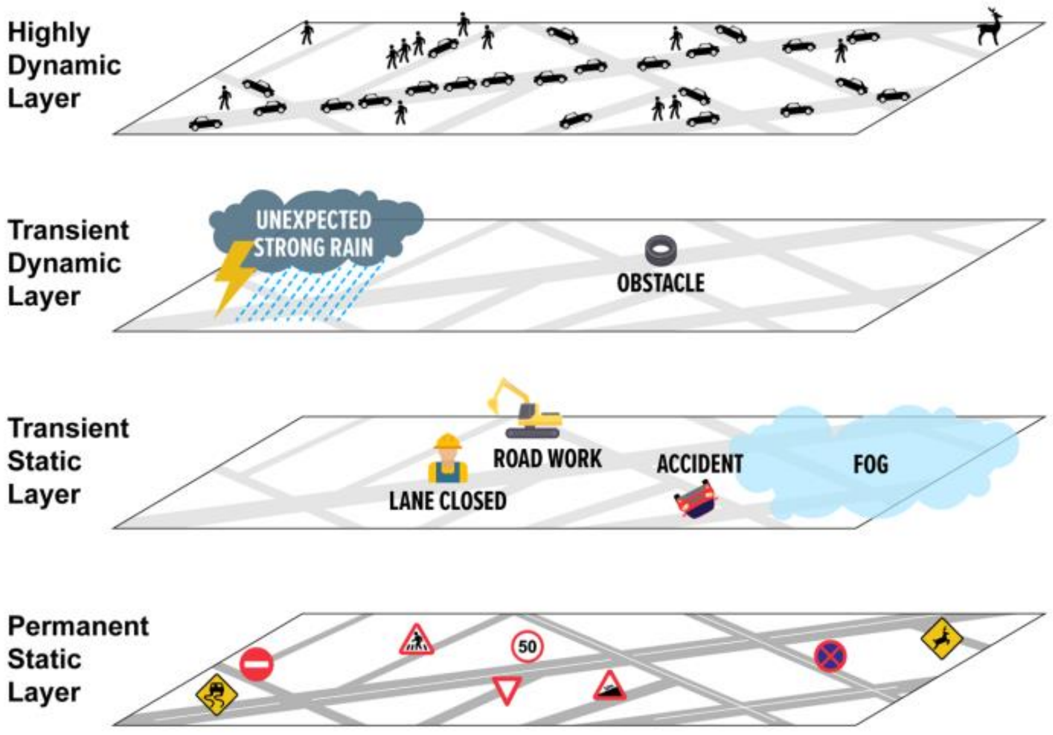


Figure 2.2 Layers of HD maps [21]

recognition AI applications running on edge servers are especially suitable for these tasks. Such applications can process data collected by several vehicles connected to an edge server, thereby implementing sensor fusion. MEC systems supported by 5G networks can efficiently process the data transmitted by vehicles, enabling local HD maps to provide accurate, near real-time information to traffic participants [21]. Furthermore, ETSI also defined a *vulnerable road user* category, which includes use cases meant to increase the safety of pedestrians and cyclists by using smartphones, which provide opportunities for sharing and receiving V2X-related information as dedicated devices. In addition to ETSI, several other organizations research edge cloud-based V2X communication. One of these larger groups is AECC [22], which also defines the use cases that can exploit the potential of these systems. They currently have a Use Case and Requirements [23] documentation, but this is only publicly available at the Table of Contents level. However, detailed research is conducted on the structure and operation of High Definition Map use cases [21].

2.1.2 Collective Perception Services (CPS) in MEC systems

The Collective Perception Service is a concept standardized by ETSI in Europe, the essence of which is that vehicles equipped with sensors should be able to transmit information about detected objects to other road users in a standardized form [24]. The purpose of CPS is to expand the perception of the vehicles. For this, the so-called Collective Perception Messages (CPM) [25] are exchanged in a one-hop geobroadcast manner. These messages also contain information about the devices that record the sensor data (station data), about the sensors themselves (how far they detect, what area they see), and about the detected objects, e.g., how they move (speed, position) or what size and what shape they have (Figure 2.3). CPM messages are transmitted by the devices at a given periodicity (time-triggered way), always providing the latest and most up-to-date/fresh sensor data.

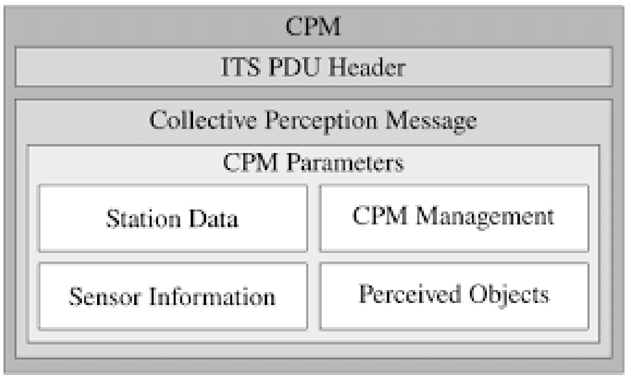


Figure 2.3 Structure of CAM messages [26, p. 4]

This information can be utilized by many automotive use cases and solutions based on MEC systems. Currently, the source of CPM messages is vehicles or RSUs. Later, these messages can also be provided by applications running on edge resources. MEC systems can give an apparent technological basis for implementing Collective Perception Services. The system can produce accurate and reliable information by simultaneously processing sensor data from multiple sources. CPS is also ideally suited to share this information, thus exploiting the potential of these technologies and raising collective perception to a higher level.

3 The Cloud-in-the-Loop simulation framework

The Cloud-in-the-Loop (CiL) simulation framework [4][8] results from several years of research and development and provides the basis for the achievements presented in this work. Based on information extracted from a simulator that models the behavior of user devices, the framework can orchestrate a closely integrated, real, distributed cloud-based environment and run and test real cloud-native applications in it. In the current construction of the framework, it is configured to examine V2C use cases; accordingly, the simulation environment is provided by a very versatile, multi-modal traffic simulation software called SUMO [7]. The orchestration of the distributed cloud-based environment is realized by the Kubernetes (k8s) widely used, open-source platform, which already plays a prominent role in operating SBA-based 5G Core networks nowadays [27] and can also be used excellently for managing edge cloud systems. The CiL framework consists of three main system components (Figure 3.1):

- Automotive traffic simulator: The simulation environment is realized by the vehicle traffic simulator called SUMO. The software can model real, large-scale road networks and simulate detailed, high-precision traffic models. Furthermore, detailed information on the behavior of each simulated object and vehicle can be extracted, and their run-time configuration is also provided.
- CiL Orchestrator: The orchestration component is a software developed in Java, which is one of the essential elements of the framework. This is where the control of the distributed cloud-based environment is realized based on the information extracted and processed from the simulator. It enables the implementation of various automotive use cases and the logic and algorithms controlling the distributed environment. Furthermore, this component ensures detailed data collection during the measurements.
- Distributed cloud environment: The distributed cloud environment is realized by a Kubernetes (k8s) platform-based real-scale hardware cluster. The nodes of k8s represent the edge servers that form a cluster. The software also enables cloud-native applications that implement automotive use cases on distributed resources.

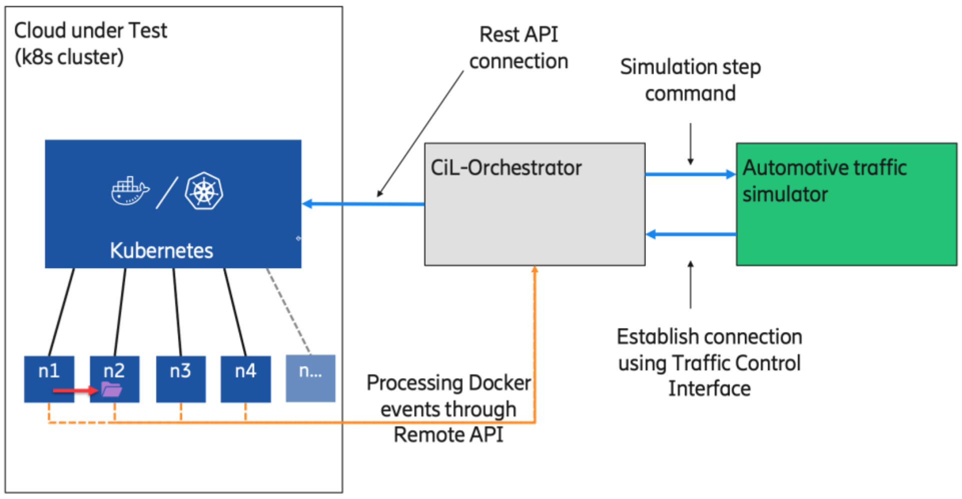

Cloud under Test
(k8s cluster)
Kubernetes
n1
n2
n3
n4
n...
Rest API connection
Simulation step command
CiL-Orchestrator
Automotive traffic simulator
Establish connection using Traffic Control Interface
Processing Docker events through Remote API

Figure 3.1 The architecture of the Cloud-in-the-Loop simulation framework

An important aspect of MEC systems is examining the effect of switching between edge resources resulting from the mobility of user devices, which affects the system's operation in many ways. In such cases, relocating applications that provide services to user devices may also be necessary, and service continuity must also be ensured. The system must also manage the redirection of the network connection giving access to the servers. In addition to relocation, the performance and operation of the system are also affected by the network and resource load, which largely depends on the number of devices served, the type of applications providing the service, and the data traffic generated. With the help of the CiL framework, based on the information extracted from the simulator, the operation of real applications can be examined, and fine-grained benchmarking and data collection can also be performed. Accordingly, many improvements have been made to the system and its supporting applications since I published my initial results in [4][8]:

- Telco-grade level, real distributed edge infrastructure was integrated into the framework (See in detail: 3.1).
- Specialized client and server application components were implemented to test UDP traffic and generate/evaluate QoS metrics (See in detail: 3.2).
- Client applications were designed and deployed that model services running on user equipments and generate UDP test traffic directly forward packets to edge applications running on the corresponding node. K8s networking no longer performs the entire traffic management as before (See in detail: 3.3).

- Simulation control and cloud system control were parallelized with appropriate thread management.
- A deep learning-based application component was designed and implemented, modeling an automotive use case featuring AI-supported object detection for MEC-level sensor fusion and related applications (See in detail: 4.1).

3.1 Design and implementation of a telco-grade cloud cluster

This chapter thoroughly describes the design and implementation of the telco-grade test infrastructure that was integrated into the Cloud-in-the-Loop framework. The leading goal of the system is to evaluate V2C applications' operations and functionalities on an actual, high-performance server cluster. To test these functionalities, it was essential to plan the structure of the servers and arrange the software components accordingly. The orchestration and management of the distributed environment are realized by the Kubernetes (k8s) platform, a powerful software to control multi-node cloud systems dealing with the orchestration and management tasks of the distributed environment. Besides the fact that k8s allow for fully functional cluster deployment, it is essential to evaluate how the software manages and operates services and functionalities that support the V2C concept and its use cases.

3.1.1 The layout of the devices

The first important task of the cluster design was planning the network and hardware elements that implement an actual cloud hardware environment based on the Edge Cloud paradigm. To evaluate various V2C scenarios and use cases, the k8s cluster was deployed with three worker nodes and a master node, which also runs the main software components of the Cloud-in-the-Loop framework. The deployment also includes a server entirely dedicated to running applications that realize and model V2C applications. Each server is interconnected through a high-performance switch with two 10 Gbit ports, an Operation and Maintenance, and a traffic port. These links can also be aggregated to create a 20 Gbit/s bonded connection between the servers. The selection and arrangement of the devices were aimed at developing an actual distributed cloud deployment that would be possible to be integrated into a real network. According to this intention, the hardware is capable of solving demanding computational tasks and serves a significant number of clients. Thus, this hardware platform enables the investigation and evaluation of use cases and validation of the concept on a telco-grade level. The cluster design also allows the integration of 5G functionalities in the future to which a high-performance server is prepared. The hardware environment consists of 6 devices (Figure 3.2). The details of the hardware components are shown in Table 3.1.

Device model	Role	Hardware description	Networking
Dell R630	Kubernetes master node, Running the CiL Orchestrator and the traffic simulator component	**CPU**: 28 cores (56 threads), 2.4 GHz, 3.3 GHz w/ Turbo **RAM**: 128 GB, 2.4 GHz **Storage**: 4 x 372 GB SSD	2 x 10 Gbit/s
Dell R630	Edge server 1 (k8s: worker1 node)	**CPU**: 24 cores (48 threads), 2.5 GHz, 3.3 GHz w/ Turbo **RAM**: 128 GB, 2133 MHz **Storage**: 4 x 372 GB SSD	2 x 10 Gbit/s
Dell R630	Edge server 2 (k8s: worker2 node)	**CPU**: 24 cores (48 threads), 2.5 GHz, 3.3 GHz w/ Turbo **RAM**: 128 GB, 2133 MHz **Storage**: 4 x 372 GB SSD	2 x 10 Gbit/s
Lenovo x3650	Edge server 3 (k8s: worker3 node)	**CPU**: 20 cores (40 Threads), 2.3 GHz **RAM**: 144 GB, 2133 MHz **Storage**: 5 TB SSD/HDD	2 x 10 Gbit/s
Lenovo x3650	Application server: running client and central server application components	**CPU**: 20 cores (40 Threads), 2.3 GHz **RAM**: 144 GB, 2133 MHz **Storage**: 5 TB SSD/HDD	2 x 10 Gbit/s
Dell R640	Not in use currently. Its future task: Running Cloud-Native 5G network functions	**CPU**: 40 cores (80 Threads), 2 GHz, 3.7 GHz w/ Turbo **RAM**: 384 GB, 2666 MHz **Storage**: 8 x 480 GB SSD	2 x 10 Gbit/s

Table 3.1 The specification of the hardware environment

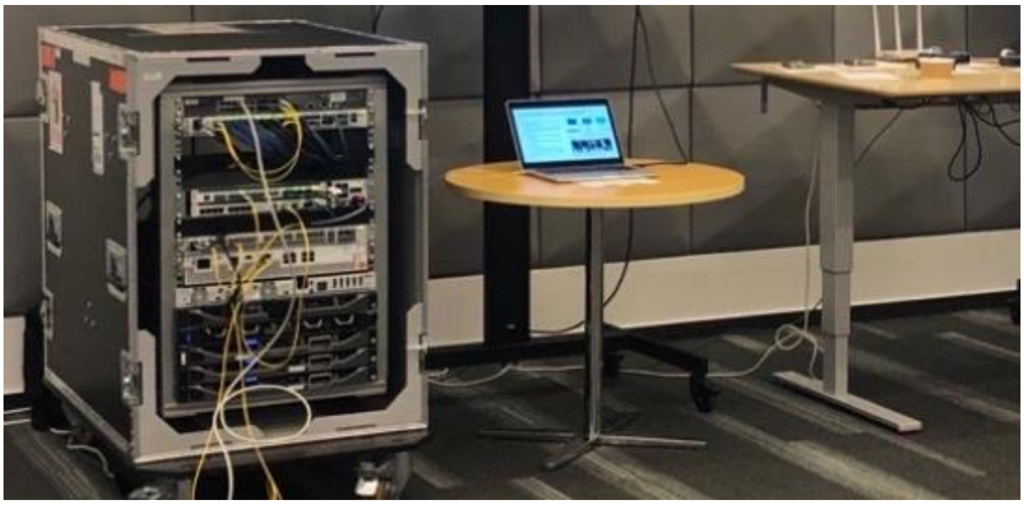


Figure 3.2 The actual hardware environment (on the left)

3.1.2 Software setup

To run the Cloud-in-the-Loop frameworks and the Kubernetes platforms' components, selecting an operating system that also properly utilizes the hardware's capabilities was necessary. All of the devices were installed with Ubuntu 18.04., a very stable and reliable version with good compatibility with various software used in the framework. With a functioning OS, the next step was to install Kubernetes, which required multiple preparatory steps. These include disabling the SWAP function (to make the software work properly), configuring the iptables and firewalls, and, most importantly, installing the Docker engine (version 20.10.7) on the nodes. Kubernetes is constantly tweaked and frequently updated with new functions. Hence, it was essential to install the newest version (version 1.23.3) of the software to be up-to-date with the framework and be able to produce measurement results with state-of-the-art technology. In the Kubernetes environment, Pod-to-Pod networking [28] is a critical part of cluster networking. It enables communication between the pods that encapsulate the containerized applications. It is a necessary function for the platform to operate. This type of communication can be realized by different technologies that are implemented mainly through third-party software components. In this implementation, the system is installed with the Container Network Interface (CNI) plugin called Calico [29]. Unlike other CNI plugins, Calico realizes the communication in the network layer (layer 3), using BGP routing protocol instead of relying on network virtualization. This way, additional encapsulation of the packets can be avoided, resulting in better performance [30]. The plugin also supports network policies, which can be very useful for regulating the network traffic between pods. After installing all the necessary software, the cluster was ready to install the components of the simulation framework.

3.2 Implementation of a UDP traffic benchmarking tool

One of the most critical aspects of benchmarking edge cloud systems is evaluating the effects of network and hardware resource load resulting from serving user devices. Furthermore, it is important to consider that since it is a distributed system, this load is distributed among the individual resources according to the users and the implemented load-balancing algorithms. During the planning of MEC services and systems, it is a vital aspect to be able to ensure the QoS required by the requirements in all cases. This is particularly important in areas with strict requirements, such as the automotive industry. Degradation of service quality is caused by errors and outages in application-level network traffic. To investigate this in the CiL framework, it became necessary to integrate an application that can be appropriately scaled and is capable of generating network traffic.

On the other hand, it can also create QoS metrics based on the generated data. In this way, the load caused by various automotive use cases can be modeled on the actual edge cloud system integrated with the framework, and the system's performance can also be examined in terms of service quality. The performance and operation of the systems can be well examined from the point of view of the packet loss metrics of the traffic generated by the applications, a UDP-based network test traffic is ideal for such measurements. To this end, I previously integrated the iperf [31] application into the framework, resulting in the generation of incorrect metrics during measurements in the edge cloud environment. It has become necessary to use a tool that is prepared for adequately handling errors occurring during operation in MEC systems and for proper representation in metrics. The basis of the self-developed UDP traffic benchmarking tool was an open-source application [32] developed in python, which primarily focuses on measuring network latency. I made several modifications to the application, preparing it for proper operation in a distributed environment. During the measurements, the application was running in two replicas:

- Client-side application (3.2.1): This component compiles and forwards the packets with the appropriate information, which is used to evaluate the generated data traffic and detect network errors.

- Server-side application (3.2.2): This component receives the network traffic generated by the client and ensures the processing of the information extracted from the packets and the generation of QoS metrics based on them.

3.2.1 UDP traffic benchmarking tool client component

During the measurements, UDP data transmission sessions (with a definable time interval and data rate) provide the basis for the QoS metrics. Therefore, it was important to develop a concept in which the components can differentiate and identify these sessions regardless of the implemented use case and the network paths of the edge cloud infrastructure. The functionality on the client side is defined by the *Client* class of the python code, in the constructor of which the UDP socket on the sending side is created based on the IP addresses and port numbers specified as parameters. The essential element of the class is the *send* function, which is responsible for the assembly and forwarding of the UDP packets. The size of the payload is determined within the function, which at the application level only contains zero bytes. The application-level header includes the information required for the functionality of the tool. In addition, so that the application can differentiate each session, its current session number is read from a separate file. During the initialization of the *send* function, the session's start time is also determined, as well as the number of packets to be transmitted and the length of the packet transmission periods based on the frequency (or bandwidth) passed as a parameter (Snippet 3.1).

```
_payload_size = packet_size - HEADER_SIZE
_fill = b''.join([b'\x00'] * (_payload_size))

if os.path.exists(self.path):
    with open(self.path) as fp:
        session_id = fp.read()
        with open(self.path, 'w') as fp2:
            fp2.write(str(int(session_id) + 1))
if not os.path.exists(self.path):
    session_id = 1
    with open(self.path, 'w') as fp2:
        fp2.write(str(session_id))

start_time = time.time_ns()
total_packets = frequency * running_time
print(int(total_packets))
running_time = running_time * 1e9
period = 1 / frequency

total_packets = round(total_packets)
total_packets_bytes = int(total_packets).to_bytes(6, 'big')
```

Snippet 3.1 Part of the Python code that implements the UDP benchmarking tool (send)

The packets sent in each data transmission session are assembled and transmitted in a while loop of the function. Each package receives a unique serial number and contains the given session's identifier. Similarly, the number of packets belonging to a given session is also transmitted so that all the properties of a given session can be extracted from a single packet on the receiving side. Furthermore, the code can also set the dynamic transmission period time. As a result, the data rate can be provided with an error rate within a maximum of 1% [32]. Each session is closed by a single packet containing 0, which indicates the last packet on the receiving side (Snippet 3.2).

```
while True:
    index_bytes = self.packet_index.to_bytes(4, 'big')
    current_time = time.time_ns()
    time_bytes = current_time.to_bytes(8, 'big')
    session_id_bytes = int(session_id).to_bytes(5, 'big')
    send_nums = self._udp_socket.sendto(
        index_bytes + time_bytes + session_id_bytes + total_packets_bytes + _fill,
        (self.remote_ip, self.to_port))
    self.log.append([self.packet_index, current_time, send_nums])

    ...

    if (current_time - start_time
    ) > running_time or self.packet_index == total_packets:
        break

    self.packet_index += 1

    if dyna:
        prac_period = (running_time - (current_time - start_time)) / (
                total_packets - len(self.log)) * (
                              len(self.log) /
                              (frequency *
                               (current_time - start_time) * 1e-9)) * 1e-9
        prac_period = period if prac_period > period else prac_period
    else:
        prac_period = period

    time.sleep(prac_period)


self._udp_socket.sendto((0).to_bytes(4, 'big'),
                        (self.remote_ip, self.to_port))
self._udp_socket.close()
```

Snippet 3.2 Part of the Python code that implements the UDP benchmarking tool (generating packets)

3.2.2 UDP traffic benchmarking tool server component

The functionality on the receiving side is implemented by the Server class of the python code. Similarly to the client side, the constructor ensures the creation of the UDP socket based on the IP address and port number passed as parameters. The listening class takes care of the reception of the packets, in which the information transmitted by the packets is read, from which the program later generates the metrics. The code appends the data extracted from packets belonging to a specific session to a log list. To differentiate between sessions, it reads the identifiers (session_id) from the first packet of each transmission and only evaluates if the last packet containing a single 0 arrives. However, in some cases, the last packet may not arrive due to packet loss. In such cases, as soon as a packet for the next session arrives, it performs the evaluation and starts logging the data of the new transfer (Snippet 3.3).

```
def listen(self, buffer_size, verbose, sync):

    …

    self.first_packet = True
    self.current_session_id = 0
    while True:
        msg, _ = self._udp_socket.recvfrom(buffer_size)
        recv_time = time.time_ns()
        packet_index = int.from_bytes(msg[:4], 'big')
        send_time = int.from_bytes(msg[4:12], 'big')
        session_id = int.from_bytes(msg[12:17], 'big')
        total_packets = int.from_bytes(msg[17:23], 'big')
        old_latency = latency
        latency = round((recv_time - send_time) * 1e-9 - self.OFFSET, 6)
        jitter = abs(latency - old_latency)
        recv_size = len(msg)

        …

        if packet_index == 0:
            server.evaluate()
            break

        if self.first_packet:
            self.current_session_id = session_id
            self.first_packet = False
        if not self.current_session_id == session_id:
            server.evaluate()
            self.first_packet = True
            self.log.clear()

        self.log.append(
            [packet_index, latency, jitter, recv_time, recv_size, total_packets])
```

Snippet 3.3 Part of the Python code that implements the UDP benchmarking tool (listen)

The evaluation and the calculation of the QoS metrics are performed by the *evaluate* function, which is called from the *listen* function. Among the other results, the data rate of the packets transmitted in the given session (based on the number of transmitted packets) and packet loss (based on the ratio of the number of packets received on the server side to the total number of transmitted packets) are calculated here (Snippet 3.4).

```
def evaluate(self):

    …

    cycle = (self.log[-1][3] - self.log[0][3]) * 1e-9
    latency_list = [row[1] for row in self.log]
    latency_max = max(latency_list)
    latency_avg = sum(latency_list) / len(latency_list)
    var = sum(pow(x - latency_avg, 2)
              for x in latency_list) / len(latency_list)
```

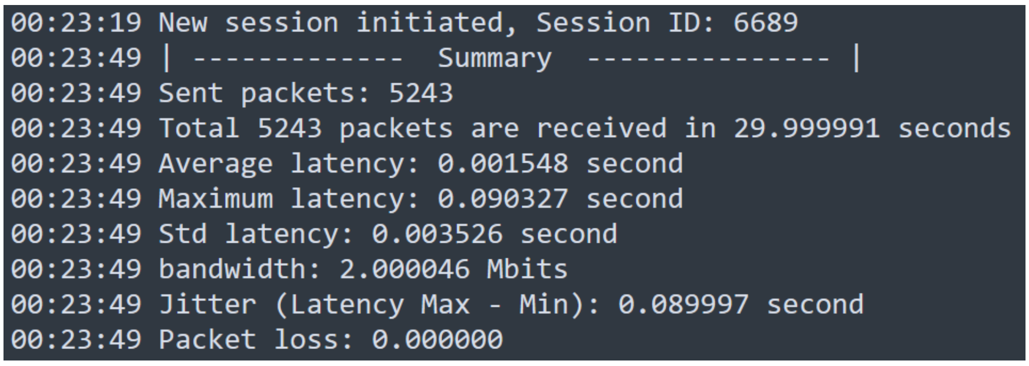
```
latency_std = math.sqrt(var)
jitter = max(latency_list) - min(latency_list)

bandwidth = sum([x[4] + 32 for x in self.log]) / cycle

packet_loss = (int(self.log[0][5]) - len(latency_list)) / int(self.log[0][5])
print('| -------------  Summary  --------------- |')
print('Sent packets: ' + str(self.log[0][5]))
print('Total %d packets are received in %f seconds' %
      (len(self.log), cycle))
print('Average latency: %f second' % latency_avg)
print('Maximum latency: %f second' % latency_max)
print('Std latency: %f second' % latency_std)
print('bandwidth: %f Mbits' % (bandwidth * 8 / 1024 / 1024))
print('Jitter (Latency Max - Min): %f second' % jitter)
print('Packet loss: %f' % packet_loss)
self.log.clear()
```

Snippet 3.4 Part of the Python code that implements the UDP benchmarking tool (evaluation)

During the measurements, a bash script logs the calculated and summarized session statistics (Figure 3.3) on the server side. The aggregated log file is processed by a post-process python script, which generates statistics from the QoS metrics and implements the results' representation.

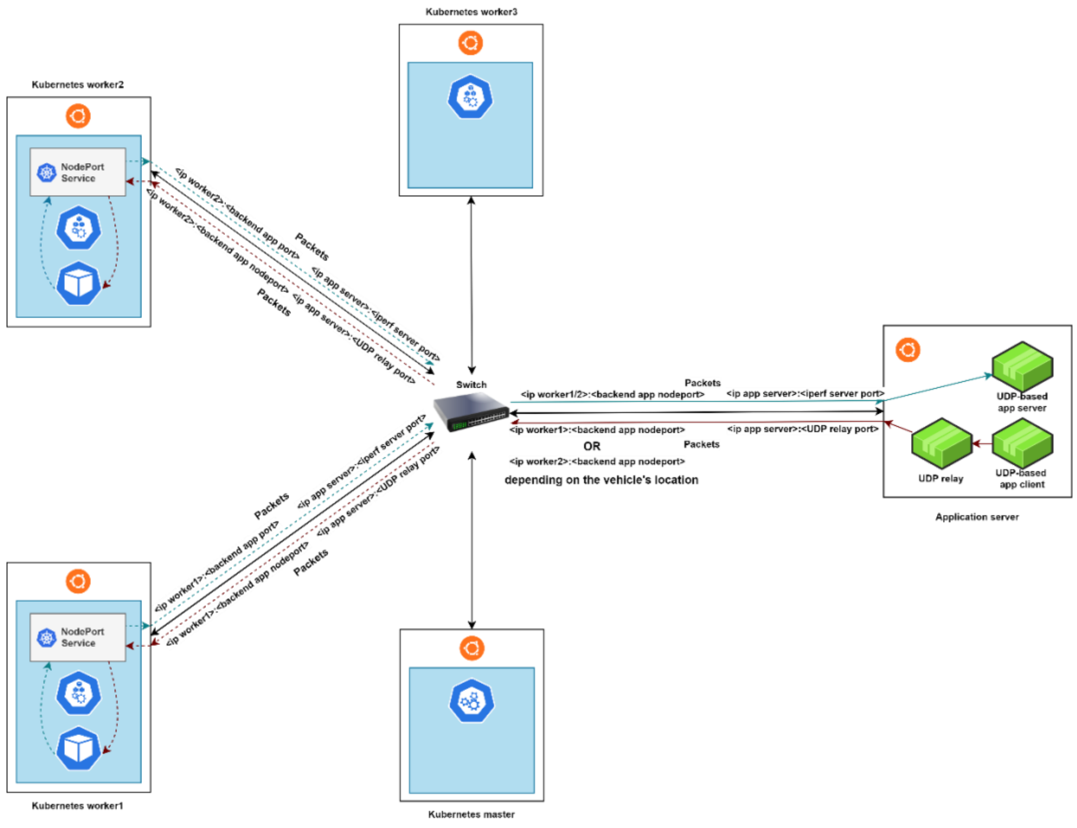


```
00:23:19 New session initiated, Session ID: 6689
00:23:49 | -------------  Summary  --------------- |
00:23:49 Sent packets: 5243
00:23:49 Total 5243 packets are received in 29.999991 seconds
00:23:49 Average latency: 0.001548 second
00:23:49 Maximum latency: 0.090327 second
00:23:49 Std latency: 0.003526 second
00:23:49 bandwidth: 2.000046 Mbits
00:23:49 Jitter (Latency Max - Min): 0.089997 second
00:23:49 Packet loss: 0.000000
```

Figure 3.3 Example of the summary of QoS metrics

3.3 Network traffic control

In real 5G-supported MEC systems, the mobile network delivers the packets sent by the user equipments to the application components running on the edge servers. In the framework, previously introduced services managed by Kubernetes ensured the appropriate delivery of packages. During the measurements, all client applications forwarded packets to the IP address of the k8s master node and the NodePort [33] of the server-side application. After that, the system forwarded the data to the appropriate node, where the receiving application encapsulating pod was running. In this implementation, the packages traveled an unnecessarily long path, and the operation of realistic MEC systems was not correctly modeled either. It became necessary to implement a solution capable of changing the destination of sent packets at run-time, even in the case of UDP-based applications, where this function is not implemented at the application level. Therefore, a UDP traffic control relay application component was implemented in the framework's application server, which can forward UDP traffic generated by any client to the server-side application component running on a given node (Figure 3.4). The relay is controlled by the CiL Orchestrator component and is able to redirect traffic based on the mobility information of the simulated vehicles. This is important in cases where several edge servers have served a vehicle during its movement, and when entering a new zone, where it is necessary to relocate the utilized services in which case the network traffic is also redirected to the new server.



Kubernetes worker3
Kubernetes worker2
NodePort Service
<ip worker2>:<backend app port>
Packets
<ip app server>:<iperf server port>
<ip worker2>:<backend app nodeport>
Packets
<ip app server>:<UDP relay port>
Switch
Packets
<ip worker1/2>:<backend app nodeport>
<ip app server>:<iperf server port>
<ip worker1>:<backend app nodeport>
OR
Packets
<ip worker2>:<backend app nodeport>
depending on the vehicle's location
<ip app server>:<UDP relay port>
UDP-based app server
UDP relay
UDP-based app client
Application server
Packets
<ip app server>:<iperf server port>
<ip worker1>:<backend app port>
<ip app server>:<UDP relay port>
<ip worker1>:<backend app nodeport>
Packets
NodePort Service
Kubernetes worker1
Kubernetes master

Figure 3.4 Operation of the UDP traffic control relay

4 Design and implementation of automotive use cases into the Cloud-in-the-Loop simulator

It is also necessary to create the proper simulation environment to examine the MEC system integrated into the framework and the implemented use cases. Modeling this is the task of the SUMO traffic simulator, which is also capable of implementing realistic traffic maps. The simulation traffic map used for the current measurements - which provides an excellent basis for testing edge systems - has already been implemented during previous tests [4][8]. The map is based on Budapest XI. district's urban environment around Infopark. The resources (servers) of the integrated edge cloud environment are located virtually on this map. To achieve this, I defined so-called latency zones (Figure 4.1) for the two edge servers of the cluster, which determine which resource serves the vehicle moving in the given position. In reality, the shape of these zones is affected by countless factors, such as the location of the base stations or the network structure. However, the zones created in the current simulation environment implement only one possible layout among many, but it is ideal from the point of view of testing edge cloud systems. Because the measurements carried out in the framework currently focus on the tests of the application components that model the operation of V2C use cases and the operation of the distributed system.

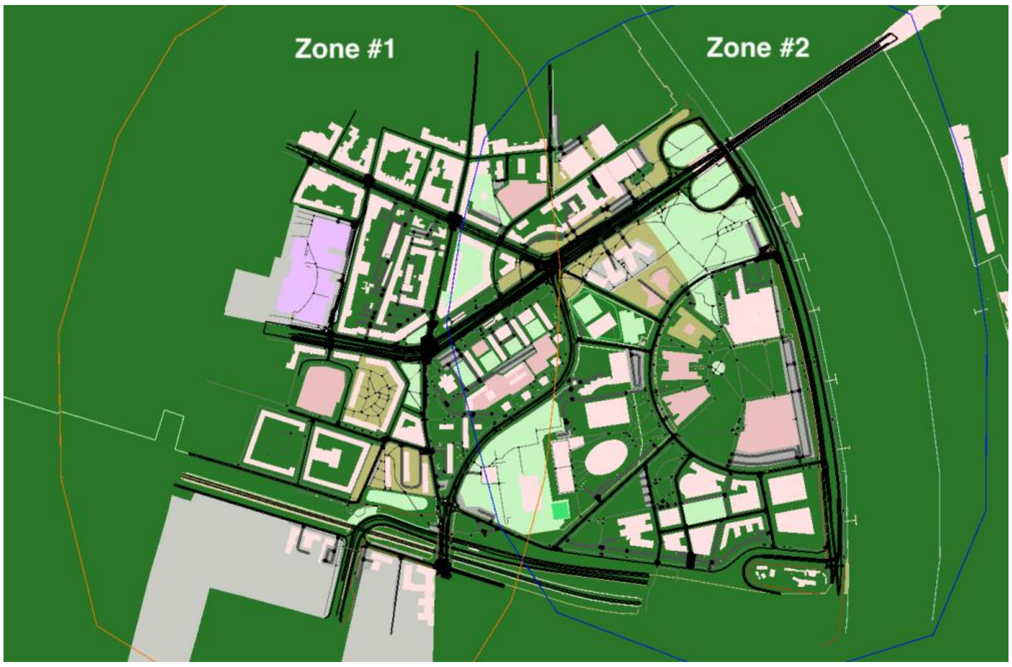


Figure 4.1 Simulation map for the implemented use cases

4.1 Deep Learning-based automotive use case implementation

One of the most promising functionalities of edge cloud systems is the possibility of outsourcing computing tasks. This can be especially beneficial in areas such as Vehicle-to-Cloud communication. Modern vehicles are equipped with many sensors, including high-resolution cameras and LIDARs. Using these devices, vehicles can collect large amounts of raw data about their environment, which are processed to support various automotive use cases. The processing of environmental information requires high-performance hardware resources, and the vehicles must also share the information extracted from the processed data. With the help of edge cloud systems supported by 5G and beyond cellular networks, it is possible for the processing of sensor data to be implemented by the resources of the distributed environment. This makes it possible to process data collected from individual sources jointly, increasing the accuracy and reliability of environment detection. One of the most efficient ways to process sensor data for object detection is to use Deep Learning-based networks. Using the CiL framework, my goal in this work was to investigate the operation of an edge cloud-compatible, cloud-native V2C application based on this technology. In the first step, it was necessary to define and implement a use case that could be used to test the functionality of these technologies.

According to the implemented use case scenario, a given V2C-capable vehicle collects environmental data using its camera sensors and then transmits it to the edge server currently serving it. On the edge server, an AI-based application processes the video stream and sends relevant feedback information to the vehicle based on the data collected from its environment. In a later phase, the scheme can also support MEC-aided sensor fusion [34] and misbehavior detection [35] purposes. Modeling and testing the use case in the framework required the design and development of two application components.

4.1.1 Client application

The GStreamer multiplatform multimedia framework implements the client-side application. The software can produce and forward a video stream recorded by a camera and based on a file. Gstreamer uses so-called pipelines to define the processing method of the resources passed to its input and also to set the output types. The pipelines offer many configuration options; the video encoding procedure the software should use can be specified, which stream protocol to use during network transmission, and to which network address the video stream should be forwarded (Snippet 4.1).

```
filesrc location=~/ai_testvid.mp4 ! decodebin ! autovideoconvert ! x264enc
tune=zerolatency ! queue2 ! rtph264pay ! udpsink host=10.96.20.3 port=32705
```

Snippet 4.1 Example of a gstreamer pipeline

I defined two clients that model two vehicles when implementing the use case. During the measurements, these clients transfer video streams for processing to the two edges implemented in the simulation environment. The basis of the sent video stream is a 30-second video recording of a traffic scenario from a car perspective, modeling a real-time stream recorded by a video sensor by an actual V2C-capable vehicle (Figure 4.2). The CiL Orchestrator component manages clients' operation, deployment, and scheduling.

Figure 4.2 Frame of the AI test video [36]

4.1.2 AI-based edge application component

From the point of view of the implemented use case, the edge application must perform three critical tasks. 1) it must receive and process the video stream transmitted by the clients. 2) it must perform object detection on the received video frames, and 3) provide feedback to the client. The source code implementing the neural network was supplied by a pre-developed python code [37]. The pre-trained object detection model is based on the Google SSD [38] method, which is well suited for real-time processing videos containing multiple objects simultaneously on one frame. The utilized MobileNetSSD model was implemented with the framework called Caffee [39], the use of which in the application is realized by the OpenCV [40] function library, which is compatible with CaffeModels. The pre-trained neural model can recognize 20 different object types, including cars, buses, and motorcycles, which provides a perfect basis for testing the functionality of the use case I am investigating. I also used the GStreamer framework in the edge application to receive and process the transmitted video stream because, in addition to being a versatile and efficient software, it is also supported by OpenCV. OpenCV had to be recompiled with appropriate parameters because the compiled libraries available in the repositories do not support GStreamer by default. I implemented the feedback methodology with the help of UDP packets, which provide feedback to the clients about the recognition confidence of the cars appearing on the frames in accordance with the tests detailed in Chapter 5 (Snippet 4.2). On the client side, I designed the reception and logging of the feedback signals with a script utilizing the netcat software [41].

```
if recog_class == "car":
     print("Car detected")
     sock.sendto(bytes(str(count) + " " + label_glob + "\n", "utf-8"),
(ip_addr, port))
```

Snippet 4.2 Part of the object detection edge application's source code

After developing the source code of the object detection application, I tested the operation of all functionalities (video stream reception, object detection, and feedback transmission to clients) (Figure 4.3).

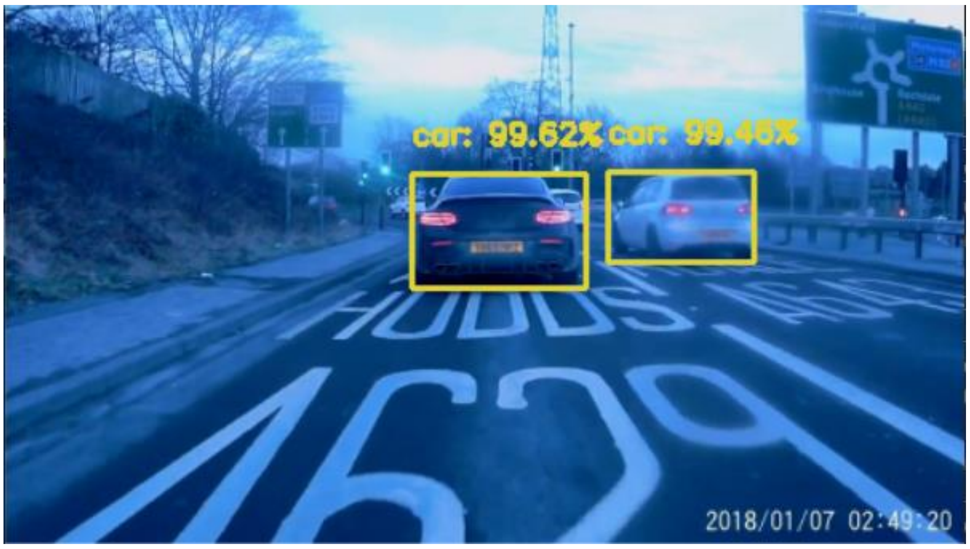


Figure 4.3 Visualizing the object detection on the AI test video

After that, creating a cloud-native, Kubernetes-compatible design for the application became necessary. I implemented the encapsulation of the source code, the used function libraries, and dependencies with Docker [42] containerization. Then I uploaded the docker image to a private Docker repository, which allows quick and easy redeployment in the test executions. In a real environment, individual client-side applications run on vehicle user equipments. In the framework, the running environment of these applications is implemented by the dedicated application server. Edge-side applications run on individual nodes (worker1/edge1, worker2/edge2) of the Kubernetes-based distributed system integrated into the framework (Figure 4.4).

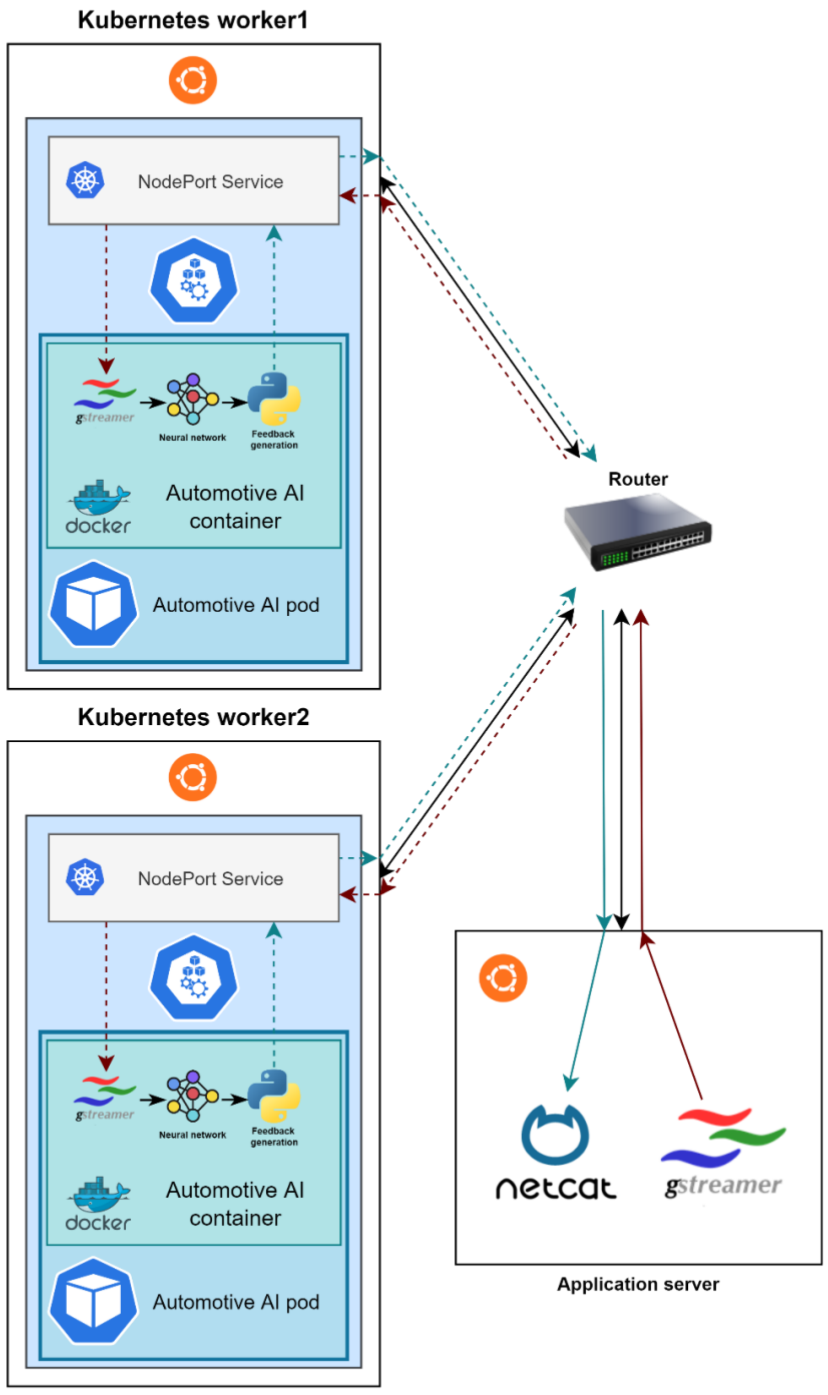


Figure 4.4 The integration of the AI-based automotive use case

5 Measurements and evaluation

During the measurements, I focused on examining the application implementing the automotive use case detailed in Chapter 4.1 and on the performance examination of the integrated k8s-based distributed environment. For this, it was first necessary to create a suitable test case. The most critical aspect of offloading computing tasks offered by edge cloud systems is that the systems must ensure high Quality of Service (QoS) and the proper Quality of Experience (QoE) at the application level. In the case of the AI-based object detection application, I examined the QoE based on a pre-defined KPI, which characterizes the application's performance and the distributed cloud-based environment during the measurements. During the tests, with the help of the client-side applications representing V2C vehicles (4.1.1), I transmitted a 30-second reference video material implementing raw data from a camera sensor to the edge-side applications. I also deployed two copies of the application components implementing the AI engine to both k8s worker nodes implementing edge servers integrated into the simulation environment. I calculated the performance of the object detection application and the QoE provided by it from detecting a car-type object on the reference video. The application detects and classifies objects frame by frame. It determines the confidence value for each recognized object, which shows how accurately it identifies an object type based on the trained model. At every frame evaluation, the application sends feedback to the client about the recognition confidence of car-type objects. During the measurements, the average of the (frame-by-frame) recognition confidences of the 30-second reference video provides the KPI based on which the system's operation can be examined under different traffic and network load scenarios:

$$ARC = \frac{\sum_{i=1}^{N} \frac{\sum_{j=1}^{M_i} RC_{i,j}}{M_i}}{N} \tag{1}$$

Where ARC stands for the average recognition confidence, N is the number of measurements, M_i is the number of object detections performed in the i-th measurement, and $RC_{i,j}$ is the j-th recognition confidence of the i-th measurement. During the tests, an AI engine ran on both edges of the simulation environment. Accordingly, I ran two clients in the two latency zones (Chapter 4.), which forwarded the video stream to the edge servers corresponding to the zones. I performed the measurements according to several scenarios. To test the performance of the system and the application, I generated background load UDP packet traffic using the UDP benchmarking tool (3.2.2) and the simulator, according to different vehicle numbers and data speeds. In order to achieve this, I placed vehicles performing movements causing application relocation operations (zone switching) into the simulation environment and such generated a background load corresponding to the number of vehicles.

5.1 Evaluating the QoS of the integrated distributed environment

Before measuring the performance of the AI application, I also performed Quality of Service performance tests on the integrated telco-grade edge cloud system. For this, I used the UDP benchmarking tool presented in chapter 3.2.1. I used UDP relay applications running on the edges already used in my previous measurements to carry out the tests. In the simulator, I created a vehicle for each client application, which causes the (live migration type) relocation of the relay applications on the edge servers due to the zone changes resulting from their mobility. Each vehicle is served by a relay application

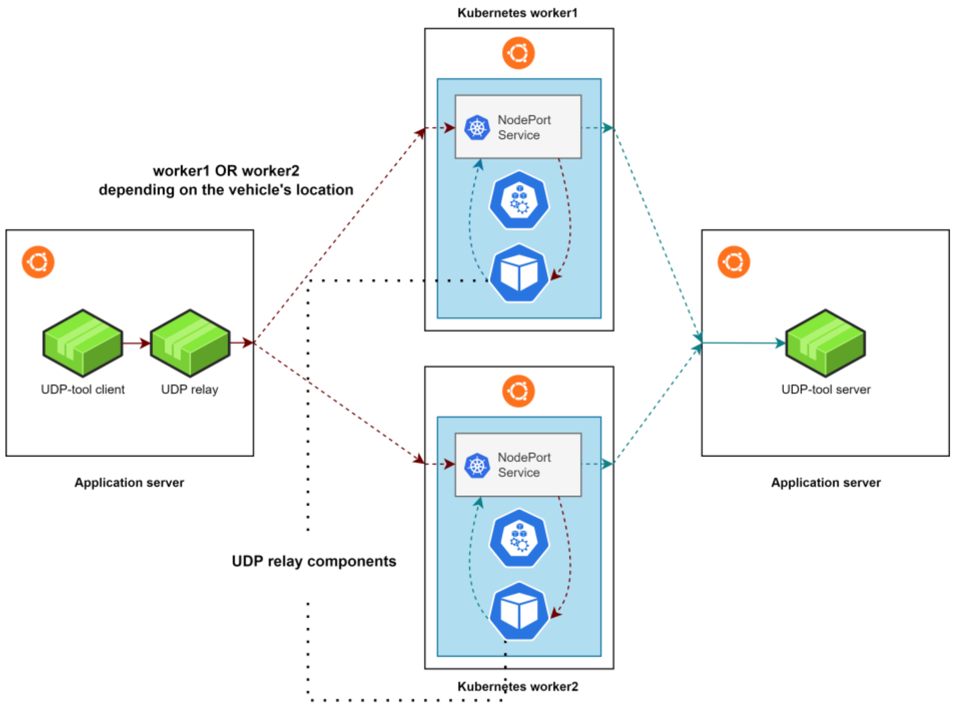


Figure 5.1 Application components of the QoS evaluation

running on the edge server belonging to its zone. Relay applications forward UDP packets received from clients to server-side applications (Chapter 3.2.2) running on the application server. In this way, detailed QoS metrics describing the performance of the system can be generated based on the evaluations carried out on the server side (Figure 5.1). The results obtained in this way describe the effect of background load (vehicle number, generated data traffic) and the effects of application relocation events.

I performed measurements with 100, 150, 200, 250, and 300 simulated vehicles within the 1,5 km^2 map area and data traffic initiated from the client side with data rates of 1, 2, 3, 4, and 5 Mbit/s. The average of 1,000 pcs 30-second sessions in each scenario gives the results. I divided the results into those measurement results where application relocation occurred during the 30-second sessions and those measurement results where no migration occurred for the cars under test. The results are based on the packet loss rates from the QoS metrics (Tables 5.1, 5.2, and 5.3; Figures 5.2, 5.3, and 5.4). Based on this, the performance of the system under a given load and the impacts of relocation events can be evaluated.

	1 Mbits/s	2 Mbits/s	3 Mbits/s	4 Mbits/s	5 Mbits/s
100 vehicles	3.032%	3.730%	4.072%	3.155%	3.264%
150 vehicles	3.524%	3.499%	4.049%	4.457%	3.639%
200 vehicles	3.334%	3.463%	3.593%	3.806%	6.678%
250 vehicles	2.439%	3.439%	3.869%	12.015%	55.287%
300 vehicles	2.214%	4.037%	13.269%	45.986%	72.099%

Table 5.1 Packet loss results aggregated

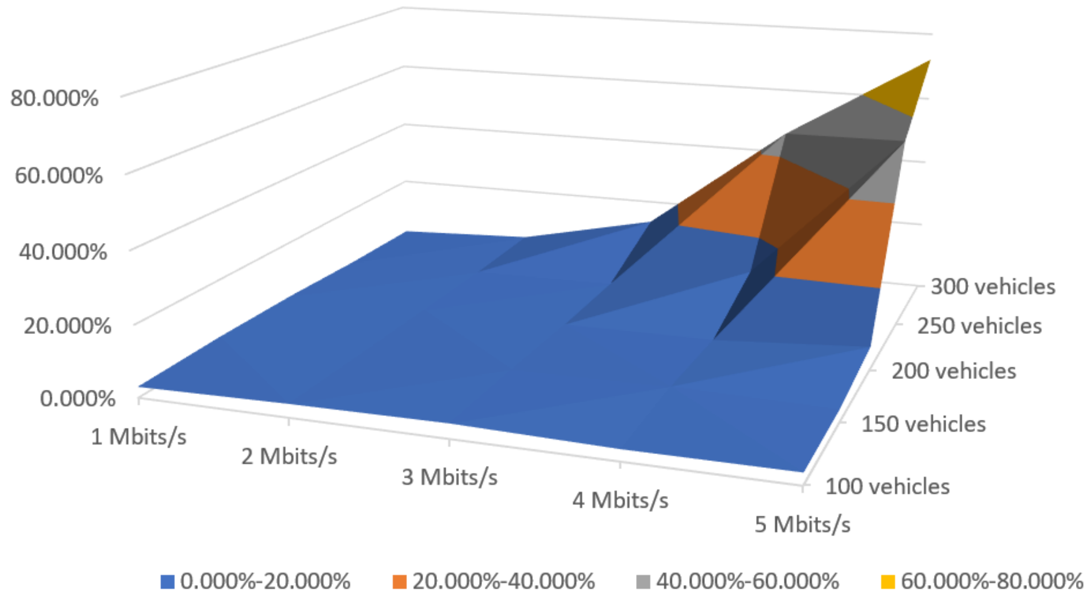


Figure 5.2 Visualizing packet loss results aggregated

	1 Mbits/s	2 Mbits/s	3 Mbits/s	4 Mbits/s	5 Mbits/s
100 vehicles	0.890%	1.296%	1.522%	0.940%	1.030%
150 vehicles	0.982%	1.600%	1.699%	1.744%	1.549%
200 vehicles	1.161%	1.528%	1.792%	2.045%	5.300%
250 vehicles	0.825%	1.857%	2.990%	10.846%	54.090%
300 vehicles	0.716%	2.847%	12.495%	45.570%	72.768%

Table 5.2 Packet loss results without relocation events

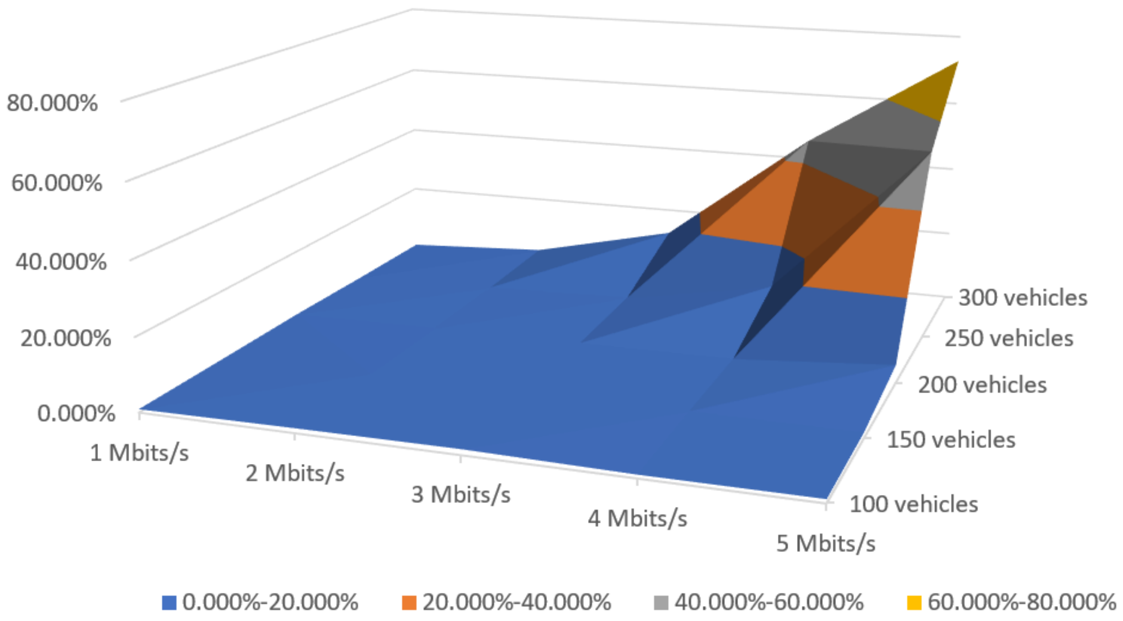


Figure 5.3 Visualizing packet loss results without relocation events

	1 Mbits/s	2 Mbits/s	3 Mbits/s	4 Mbits/s	5 Mbits/s
100 vehicles	11.721%	14.557%	14.867%	12.541%	12.845%
150 vehicles	18.442%	14.486%	18.679%	19.489%	15.865%
200 vehicles	19.461%	17.977%	18.225%	19.691%	16.986%
250 vehicles	15.940%	18.615%	14.463%	23.813%	68.859%
300 vehicles	17.456%	15.566%	24.395%	53.149%	62.625%

Table 5.3 Packet loss results due to relocation events

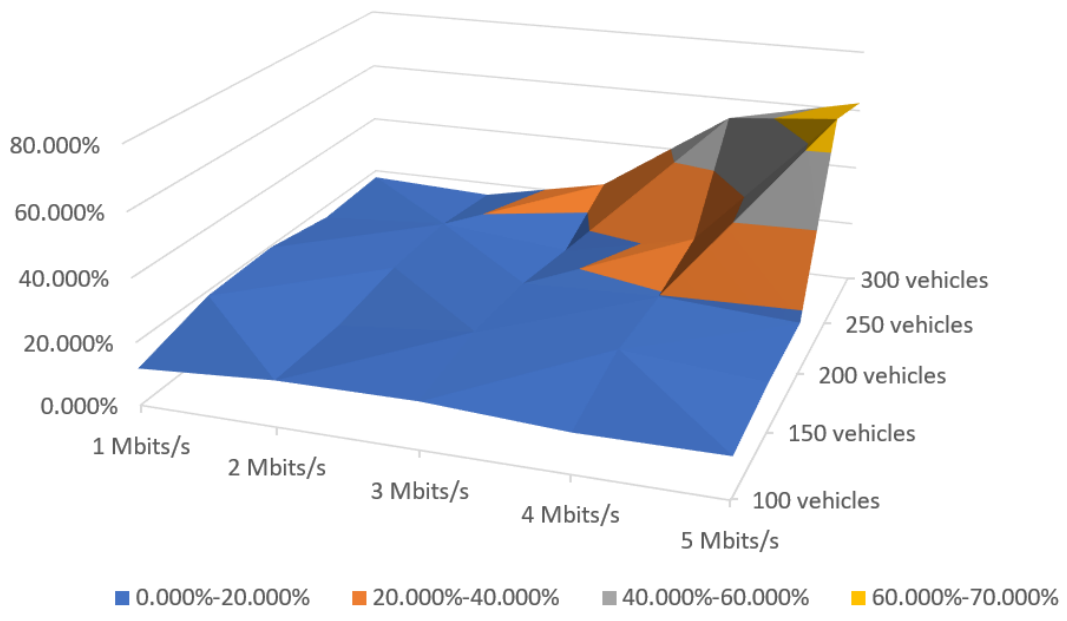


Figure 5.4 Visualizing packet loss results due to relocation events

The measurement results clearly show that a lower load (fewer vehicles and lower data speeds) results in a low packet loss rate, as expected. In some cases, a lower packet loss rate occurred due to a higher load, but these differences are usually minimal. However, there were also results when significantly lower packet loss rates were generated under a higher load than in the case of a lower load. The packet loss rates increase significantly above 250 vehicles and 4 Mbits/s data speed. In the results affected by relocations (Figure/Table 5.3), for example, in the case of 250 veh/5 Mbps and 300 veh/5 Mbps, a lower packet loss rate due to higher load can be observed. I evaluate these contrary-to-expected results in detail in Chapter 5.2.

5.2 Evaluation of the Deep Learning-based automotive use case implementation

As discussed in the introduction of Chapter 5, I examined the results generated from the average recognition confidences during the measurements with the application components implementing the Deep Learning-based automotive use case. I reviewed the operation of the application using methods that implement the background load, which is the basis of the measurement results presented in Chapter 5.1. For this, I created various measurement scenarios with 100, 200, and 300 simulated vehicles and data traffic initiated from the client side with data rates of 1, 2, 3, 4, and 5 Mbits/s. In these, I investigated how the average recognition confidence of car-type objects in the reference videos changes due to a given background load (Table 5.4 and Figure 5.5). The results are provided by the total average of the confidences generated by the application instances running on the two worker nodes, with around 100 measurements for each scenario (1 measurement is given by the confidences calculated for each frame of the 30-second reference video) per node. In order to be able to compare the results, I first measured the KPI without load, which shows what QoE the AI engine can provide if the distributed system does not serve any other clients. The average recognition confidence, in this case, was 52.80% based on 718 measurements (per node).

	1 Mbits/s	2 Mbits/s	3 Mbits/s	4 Mbits/s	5 Mbits/s
100 vehicles	52.92%	52.91%	52.71%	52.86%	52.88%
200 vehicles	52.87%	52.86%	52.73%	52.71%	52.78%
300 vehicles	52.84%	52.77%	52.84%	52.64%	52.80%

Table 5.4 The averages of the object detection confidences in different scenarios

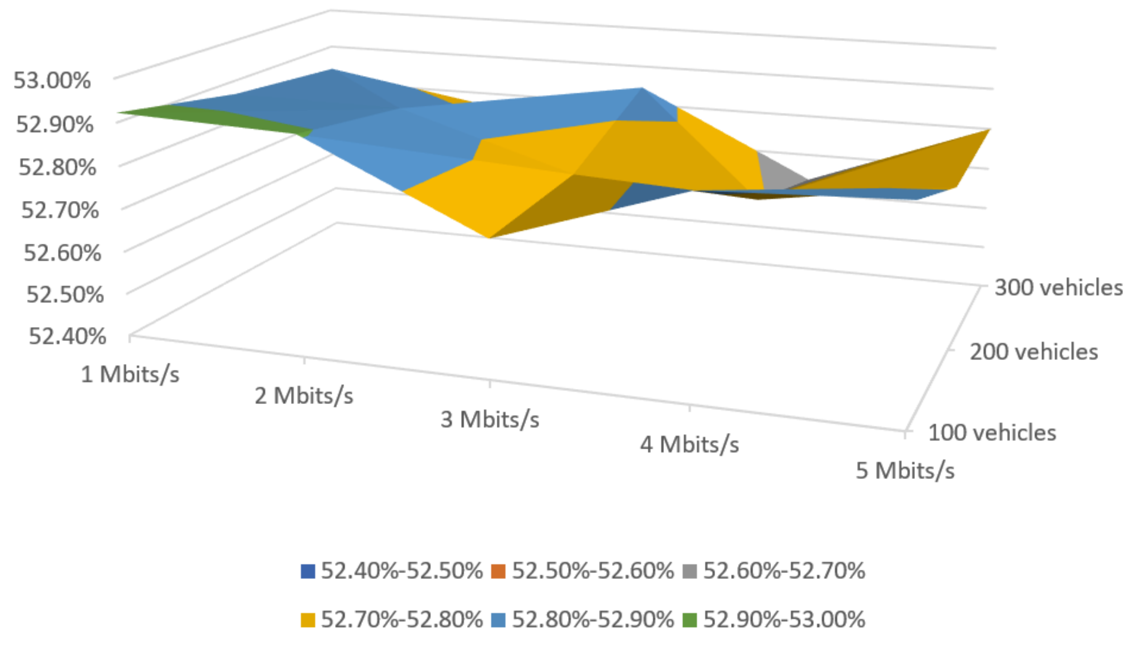


Figure 5.5 Visualizing the averages of the object detection confidences in different scenarios

Based on the results, it can be concluded that the impact of the background load occurring in these measurement scenarios essentially does not affect the efficiency of the application (Table 5.4, Figure 5.5). It can also be observed that the average of the recognition confidences calculated based on measurements in the scenario without background load is lower than the result of certain load measurements (e.g., 100 vehicles, 1 Mbps). This clearly shows that changes in the background load do not necessarily cause these differences in the measurement results. In addition to the averages, when examining the individual measurement points, it can also be seen that the distribution of the recognition confidence measurements for the scenario without background load and the scenario with maximum generated load (300 vehicles, 5 Mbps) is almost identical (Figure 5.6, Figure 5.7).

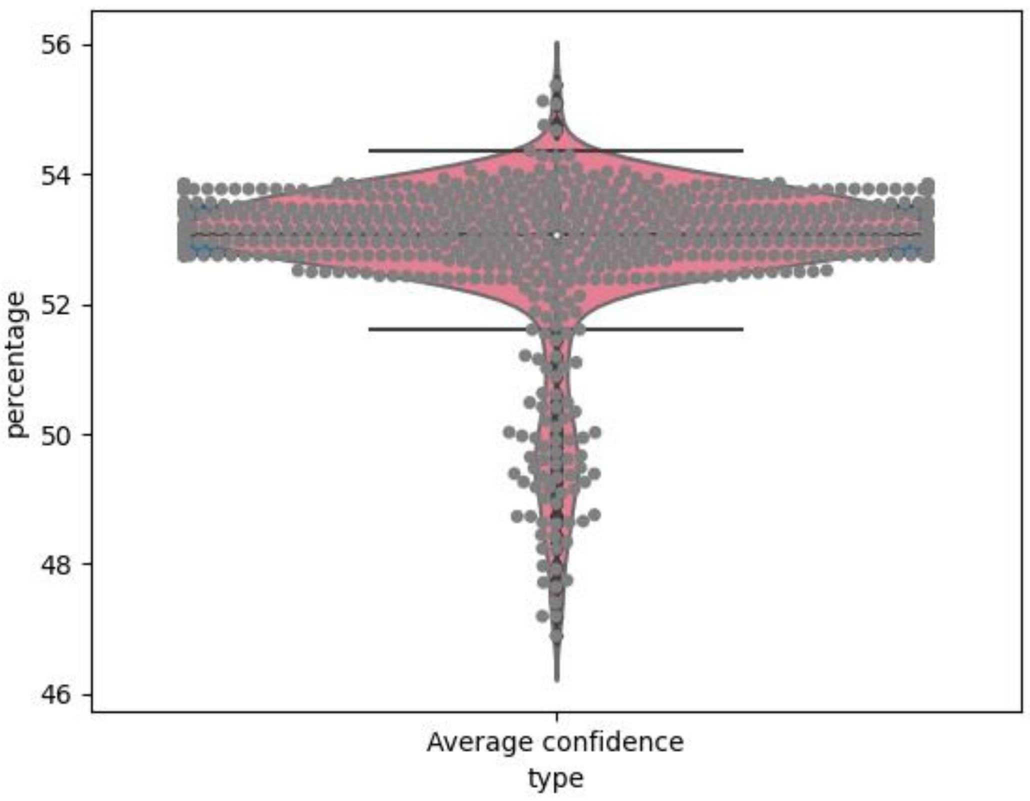


Figure 5.6 Distribution of individual recognition confidence measurement results in a scenario without background load (on worker1 node)

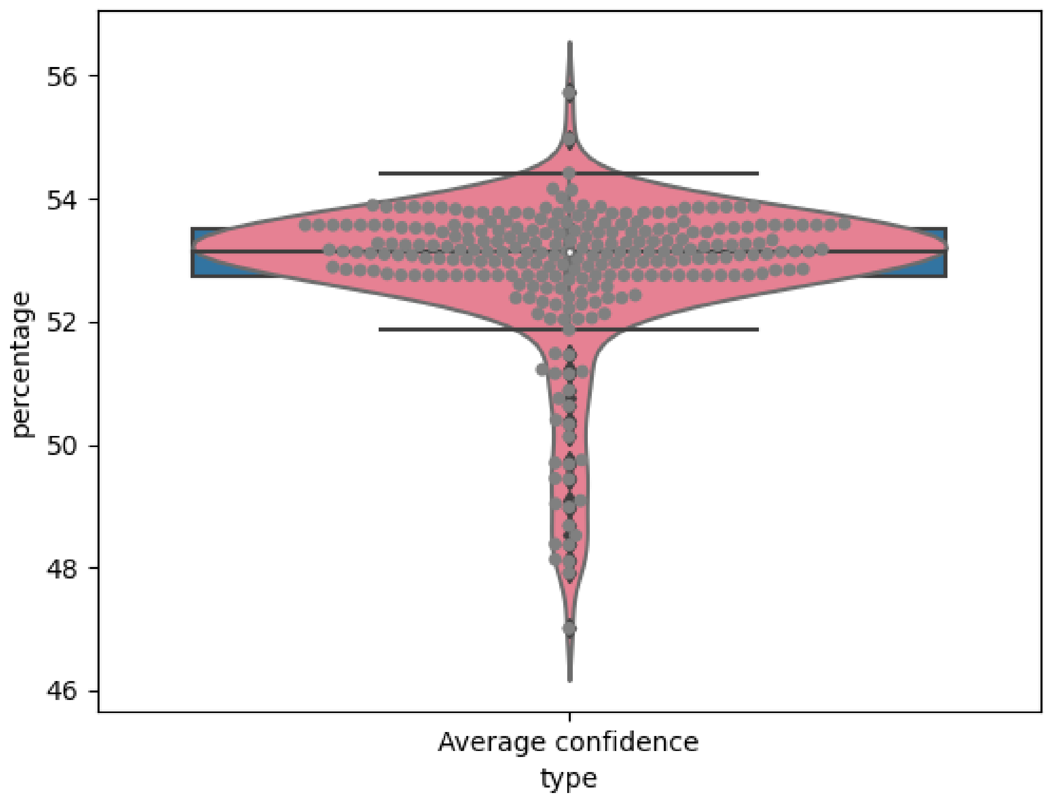


Figure 5.7 The distribution of individual recognition confidence measurement results with 300 vehicles and 5 mbps (on worker1 node)

The measurement results examining the QoE aspects of the AI application raise the question of why the effects of the background loads are not reflected in the recognition confidences. The main reason for this may be that the background load does not reach the critical level that would exert its effect. At the same time, components such as GStreamer is also able to prevent errors caused by packet loss to some extent with the help of the RTP-based H.264 encoded video [39]. However, based on the results of the QoE measurements, it can be stated that the implemented AI application works efficiently even with the effects of the background loads realized by the predefined scenarios. During the later measurements, it will be necessary to increase the background load so that I can determine when its QoE starts to degrade and how it correlates exactly with the given effects. Regarding the QoS measurements, it is important to highlight that the results shown in Chapter 5.1 represent the current level of development of the tools, which still have many opportunities for further enhancement. I concluded that the extremely high packet loss rates are caused by a bottleneck (or more) in the system. Since this is a complex system with many components, solving the problem is not a trivial task. The source of the problem can arise from the network configuration, the cause of which can be improper router configuration, k8s networking, or OS-level package management. At the same time, it is also possible that a large amount of lost packets occurs during transmission from the edge to the server. In contrast, during QoE measurements, a large amount of data transfer is only carried out between clients and edge servers, and only low-bandwidth feedback data transmission is realized from the edge to the client. The next task in this regard is to find the source of the problem and localize the bottleneck. This requires a detailed examination and possible modification of the components. After that, it will be necessary to carry out measurements with each new composition and deduce the problem based on the results. Furthermore, in order to make the QoS and QoE measurements comparable in later tests, it will be necessary to design a new model and transform the data path of the packets managed by the application components implementing the UDP measurements. Then I will be able to examine the connection between the results of the two types of measurements, and I will also be able to study the exact correlation between packet losses and recognition confidences in the AI application.

6 Summary

In my TDK paper, I presented Multi-Access Edge Computing, one of the promising new directions of cloud-based technologies. I covered the integration of MEC systems with 5G systems in detail. I gave an overview of the systems' network functions and API-based communication. I also presented the advantages that the joint use of the two systems can provide in the future. I gave an overview of the MEC-based automotive use cases supported by 5G, focusing on Deep learning-based applications and use cases based on collective perception. Furthermore, I summarized the standardization efforts that will facilitate the real-life application of these innovative technologies. In the following chapter, I gave a comprehensive overview of the Cloud-in-the-Loop framework, focusing on recent developments. In this chapter, among other things, the integration of the telco-grade distributed environment and the UDP benchmarking tool developed for QoS tests were presented. In the second half of the paper, I showed the simulation environment that forms the measurements' basis. I also introduced the implemented deep learning-based automotive use case and the application components that model and implement them in my CiL environment.

After that, in the last chapter, I discussed the process and results of the measurements in detail. In the first part of the measurements, I examined the performance of the implemented distributed environment with UDP packet traffic implementing various loads. Here, I investigated what kind of QoS the distributed system can provide based on packet loss ratios. After that, using the framework, I evaluated the implemented AI application from a QoE point of view. I defined various background load measurement scenarios and then investigated how the object detection ability of the application degrades as a result of the load. The results indicated that the application is resistant to the background loads I examined, and they do not affect its functionality. At the same time, the QoS results showed that a large amount of packet loss occurs in the system due to the investigated background loads. Based on these, I concluded that a bottleneck in one of the components implementing the QoS measurements is the cause of the significant packet losses. Therefore, the localization and correction of this problem will be highly prioritized among the subsequent tasks. Furthermore, the following study for the QoE tests performed with the AI application is to execute measurements under higher background loads. With this, my goal is to identify the level at which the load already affects the operation of the AI-based application and to determine how the load correlates with the QoE results. Future goals also include further development of the framework; one of the priority targets is currently the integration of the CiL Orchestrator into the 5G Core Network, the preparations for which have already begun.

References

[1] ETSI, ‘Multi-access Edge Computing (MEC); Framework and Reference Architecture (ETSI GS MEC 003)’. Mar. 2022.

[2] Sami Kekki *et al.*, ‘MEC in 5G networks (ETSI White Paper No. 2)’, Jun. 2018.

[3] H. Ma, S. Li, E. Zhang, Z. Lv, J. Hu, and X. Wei, ‘Cooperative Autonomous Driving Oriented MEC-aided 5G-V2X: Prototype System Design, Field Tests and AI-based Optimization Tools’, *IEEE Access*, vol. PP, pp. 1–1, Mar. 2020, doi: 10.1109/ACCESS.2020.2981463.

[4] L. Maller, P. Suskovics, and L. Bokor, ‘Cloud-in-the-Loop simulation of C-V2X application relocation distortions in Kubernetes based Edge Cloud environment’, in *2022 26th International Conference on Information Technology (IT)*, 2022, pp. 1–4. doi: 10.1109/IT54280.2022.9743520.

[5] ‘Kubernetes’. https://kubernetes.io/, Accessed 30 Oct. 2022.

[6] Ericsson, ‘5G Core (5GC)’. https://www.ericsson.com/en/core-network/5g-core, Accessed 30 Oct. 2022.

[7] ‘Simulation of Urban MObility (SUMO)’. https://www.eclipse.org/sumo/, Accessed 30 Oct. 2022.

[8] L. Maller, ‘Járműkommunikációs felhőtechnológiák modellezése és vizsgálata Cloud-in-the-Loop szimulációs környezetben’, presented at the Villamosmérnöki és Informatikai Kar 2021. évi TDK, 2021.

[9] M. Liyanage, P. Porambage, A. Y. Ding, and A. Kalla, ‘Driving forces for Multi-Access Edge Computing (MEC) IoT integration in 5G’, *ICT Express*, vol. 7, no. 2, pp. 127–137, 2021, doi: https://doi.org/10.1016/j.icte.2021.05.007.

[10] ‘European Telecommunications Standards Institute (ETSI)’, *https://www.etsi.org/*, Accessed 30 Oct. 2022.

[11] *(3GPP TS 23.501 V15.1.0) 3rd Generation Partnership Project; Technical Specification Group Services and System Aspects; System Architecture for the 5G System; Stage 2 (Release 15)*. 2018.

[12] D. Jiang and L. Delgrossi, ‘IEEE 802.11p: Towards an International Standard for Wireless Access in Vehicular Environments’, in *VTC Spring 2008 - IEEE Vehicular Technology Conference*, 2008, pp. 2036–2040. doi: 10.1109/VETECS.2008.458.

[13] Y. Wang, X. Duan, D. Tian, G. Lu, and H. Yu, ‘Throughput and Delay Limits of 802.11p and its Influence on Highway Capacity’, *Procedia - Soc. Behav. Sci.*, vol. 96, pp. 2096–2104, 2013, doi: https://doi.org/10.1016/j.sbspro.2013.08.236.

[14] J. Nilsson, C. Bergenhem, J. Jacobson, R. Johansson, and J. Vinter, ‘Functional Safety for Cooperative Systems’, in *SAE Technical Papers*, Apr. 2013, vol. 2. doi: 10.4271/2013-01-0197.

[15] Jan Jacobson and Christian Grante, ‘Functional Safety in Systems of Road Vehicles’. [Online]. Available: https://www.diva-portal.org/smash/get/diva2:962529/FULLTEXT01.pdf

[16] M. Tahir and M. Katz, ‘Performance evaluation of IEEE 802.11p, LTE and 5G in

connected vehicles for cooperative awareness', *Eng. Rep.*, vol. 4, Apr. 2022, doi: 10.1002/eng2.12467.

[17] ETSI, 'Multi-access Edge Computing (MEC); V2X Information Service API (ETSI GS MEC 030)'. May 2022.

[18] '3GPP', *https://www.3gpp.org/*, Accessed 31 Oct. 2022.

[19] 3GPP, '3rd Generation Partnership Project; Technical Specification Group Services and System Aspects; Architecture enhancements for V2X services (Release 17) (3GPP TS 23.285)'. Mar. 2022.

[20] ETSI, 'Multi-access Edge Computing (MEC); Study on MEC Support for V2X Use Cases (ETSI GR MEC 022)', Sep. 2018.

[21] AECC, 'Operational Behavior of a High Definition Map Application (White Paper)', May 2020.

[22] 'AECC', *https://aecc.org/*, Accessed 31 Oct. 2022.

[23] AECC, 'Use-case and Requirement Document (URD) Version 3.0.0'.

[24] A. Chtourou, P. Merdrignac, and O. Shagdar, 'Collective Perception service for Connected Vehicles and Roadside Infrastructure', in *2021 IEEE 93rd Vehicular Technology Conference (VTC2021-Spring)*, 2021, pp. 1–5. doi: 10.1109/VTC2021-Spring51267.2021.9448753.

[25] ETSI, 'Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Analysis of the Collective Perception Service (CPS); Release 2 (ETSI TR 103 562)'. Dec. 2019.

[26] F. Schiegg, N. Brahmi, and I. Llatser, 'Analytical Performance Evaluation of the Collective Perception Service in C-V2X Mode 4 Networks', Oct. 2019, pp. 181–188. doi: 10.1109/ITSC.2019.8917214.

[27] Henrik Bäckström and Rakesh Bohra, 'Why Kubernetes over bare metal infrastructure is optimal for cloud native applications', May 03, 2022. https://www.ericsson.com/en/blog/2022/5/kubernetes-over-bare-metal-cloud-infrastructure-why-its-important-and-what-you-need-to-know, Accessed 31 Oct. 2022.

[28] 'Kubernetes - Cluster Networking'. https://kubernetes.io/docs/concepts/cluster-administration/networking/, Accessed 31 Oct. 2022.

[29] 'What is Project Calico?' https://www.tigera.io/project-calico/, Accessed 31 Oct. 2022.

[30] 'Comparing Kubernetes CNI Providers: Flannel, Calico, Canal, and Weave'. https://www.suse.com/c/rancher_blog/comparing-kubernetes-cni-providers-flannel-calico-canal-and-weave/, Accessed 31 Oct. 2022.

[31] 'iPerf - The ultimate speed test tool for TCP, UDP and SCTP'. https://iperf.fr/iperf-doc.php, Accessed 31 Oct. 2022.

[32] 'ChuanyuXue/udp-latency (tool)', *https://github.com/ChuanyuXue/udp-latency*, Accessed 31 Oct. 2022.

[33] Thibault Debatty, 'Exposing a Kubernetes application : Service, HostPort, NodePort, LoadBalancer or IngressController?' https://cylab.be/blog/154/exposing-a-kubernetes-application-service-hostport-nodeport-loadbalancer-or-ingresscontroller, Accessed 31 Oct. 2022.

[34] A. Munir, E. Blasch, J. Kwon, J. Kong, and A. Aved, ‘Artificial Intelligence and Data Fusion at the Edge’, *IEEE Aerosp. Electron. Syst. Mag.*, vol. 36, no. 7, pp. 62–78, 2021, doi: 10.1109/MAES.2020.3043072.

[35] R. W. van der Heijden, S. Dietzel, T. Leinmüller, and F. Kargl, ‘Survey on Misbehavior Detection in Cooperative Intelligent Transportation Systems’, *IEEE Commun. Surv. Tutor.*, vol. 21, no. 1, pp. 779–811, 2019, doi: 10.1109/COMST.2018.2873088.

[36] *BMW e92 hood cam- Sunday drive ft.c63s*. [Online Video]. Available: https://www.youtube.com/watch?v=u-CTsTZxRBI&t=218s, Accessed 31 Oct. 2022.

[37] ‘KingArnaiz/Object-Detection-Tutorial’, *https://github.com/KingArnaiz/Object-Detection-Tutorial*, Accessed 31 Oct. 2022.

[38] W. Liu *et al.*, ‘SSD: Single Shot MultiBox Detector’, 2016. [Online]. Available: http://arxiv.org/abs/1512.02325

[39] ‘Caffe (deep learning framework)’, *https://caffe.berkeleyvision.org/,* Accessed 31 Oct. 2022.

[40] ‘OpenCV’, *https://opencv.org/,* Accessed 31 Oct. 2022.

[41] ‘netcat’, *https://linux.die.net/man/1/nc,* Accessed 31 Oct. 2022.

[42] ‘Docker’. https://www.docker.com/, Accessed 31 Oct. 2022.

List of figures, tables and code snippets

Figure 2.1 5G and MEC integration [2] 10
Figure 2.2 Layers of HD maps [21] 13
Figure 2.3 Structure of CAM messages [26, p. 4] 14
Figure 3.1 The architecture of the Cloud-in-the-Loop simulation framework 16
Figure 3.2 The actual hardware environment (on the left) 19
Figure 3.3 Example of the summary of QoS metrics 24
Figure 3.4 Operation of the UDP traffic control relay 25
Figure 4.1 Simulation map for the implemented use cases 26
Figure 4.2 Frame of the AI test video [35] 28
Figure 4.3 Visualizing the object detection on the AI test video 29
Figure 4.4 The integration of the AI-based automotive use case 30
Figure 5.1 Application components of the QoS evaluation 32
Figure 5.2 Visualizing packet loss results aggregated 33
Figure 5.3 Visualizing packet loss results without relocation events 34
Figure 5.4 Visualizing packet loss results due to relocation events 35
Figure 5.5 Visualizing the averages of the object detection confidences in different scenarios . 36
Figure 5.6 Distribution of individual recognition confidence measurement results in a scenario without background load (on worker1 node) 37
Figure 5.7 The distribution of individual recognition confidence measurement results with 300 vehicles and 5 mbps (on worker1 node) 38

Table 3.1 The specification of the hardware environment 18
Table 5.1 Packet loss results aggregated 33
Table 5.2 Packet loss results without relocation events 34
Table 5.3 Packet loss results due to relocation events 34
Table 5.4 The averages of the object detection confidences in different scenarios 36

Snippet 3.1 Part of the Python code that implements the UDP benchmarking tool (send) 21
Snippet 3.2 Part of the Python code that implements the UDP benchmarking tool (generating packets) 22
Snippet 3.3 Part of the Python code that implements the UDP benchmarking tool (listen) 23
Snippet 3.4. Part of the Python code that implements the UDP benchmarking tool (evaluation) 24
Snippet 4.1 Example of a gstreamer pipeline 28
Snippet 4.2 Part of the object detection edge application's source code 29

List of abbreviations

3GPP	3rd Generation Partnership Project
AF	Application Function
C-V2X	Cellular-Vehicle-to-Everything
CAM	Cooperative Awareness Message
CiL	Cloud-in-the-Loop
ETSI	European Telecommunications Standards Institute
k8s	Kubernetes
KPI	Key Performance Indicator
LADN	Local Area Data Networks
MEC	Multi-Access Edge Computing
NEF	Network Exposure Function
NF	Network Function
PCF	Policy Control Function
QoE	Quality of Experience
QoS	Quality of Service
RSU	Road-Side Unit
SBA	Service-Based Architecture
SSC	Session and Service Continuity
UDP	User Datagram Protocol
UE	User Equipment
UPF	User Plane Function
V2C	Vehicle-to-Cloud
V2X	Vehicle-to-Everything

Appendix

Appendix 1. Python code implementing a deep learning-based object detection application

```
# import the necessary packages
import sys
import socket

from imutils.video import VideoStream
from imutils.video import FPS
import numpy as np
import argparse
import imutils
import time
import cv2

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) # UDP
count = 1

# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-p", "--prototxt", required=True,
	help="path to Caffe 'deploy' prototxt file")
ap.add_argument("-m", "--model", required=True,
	help="path to Caffe pre-trained model")
ap.add_argument("-c", "--confidence", type=float, default=0.2,
	help="minimum probability to filter weak detections")
ap.add_argument("-p1", "--pipeline1", required=False,
	help="The gstreamer pipeline 1")
ap.add_argument("-ip", "--ipfeedback", required=False,
	help="IP address where the feedback is sent to")
ap.add_argument("-port", "--portfeedback", type=int, required=False,
	help="Port number where the feedback is sent to")
args = vars(ap.parse_args())

# initialize the list of class labels MobileNet SSD was trained to
# detect, then generate a set of bounding box colors for each class
CLASSES = ["background", "aeroplane", "bicycle", "bird", "boat",
	"bottle", "bus", "car", "cat", "chair", "cow", "diningtable",
	"dog", "horse", "motorbike", "person", "pottedplant", "sheep",
	"sofa", "train", "tvmonitor"]
COLORS = np.random.uniform(0, 255, size=(len(CLASSES), 3))

ip_addr = args["ipfeedback"]
port = args["portfeedback"]
#port = int(port)

# load our serialized model from disk
print("[INFO] loading model...")
net = cv2.dnn.readNetFromCaffe(args["prototxt"], args["model"])
net2 = cv2.dnn.readNetFromCaffe(args["prototxt"], args["model"])


# initialize the video stream, allow the cammera sensor to warmup,
# and initialize the FPS counter
print("[INFO] Capturing video stream...")
#cap_pipe = 'udpsrc port=5000 caps="application/x-rtp, media=(string)video, clock-rate=(int)90000, encoding-name=(string)H264" ! rtpjitterbuffer ! rtph264depay ! queue2 ! avdec_h264 ! autovideoconvert ! appsink'
cap_pipe = args["pipeline1"]
#cap_pipe2 = args["pipeline2"]
out_pipe = "appsrc ! decodebin ! autovideoconvert ! x264enc tune=zerolatency ! queue2 ! rtph264pay ! udpsink host=10.96.20.5 port=9999"

#vs = VideoStream(src=0).start() // ORIGINAL
vs = cv2.VideoCapture(cap_pipe, cv2.CAP_GSTREAMER)
```

```
time.sleep(2.0)
fps = FPS().start()
label_glob = []
recog_class = []

fourcc = cv2.VideoWriter_fourcc(*'H264')
print(fps)
out = cv2.VideoWriter(out_pipe, fourcc, 60, (400, 225), True)


# loop over the frames from the video stream
while True:
	# grab the frame from the threaded video stream and resize it
	# to have a maximum width of 400 pixels

	frame = vs.read()[1]
	#print(frame)
	frame = imutils.resize(frame, width=480)


	# grab the frame dimensions and convert it to a blob
	(h, w) = frame.shape[:2]
	blob = cv2.dnn.blobFromImage(cv2.resize(frame, (300, 300)),
		0.007843, (300, 300), 127.5)


	# pass the blob through the network and obtain the detections and
	# predictions
	net.setInput(blob)
	detections = net.forward()

	# loop over the detections
	for i in np.arange(0, detections.shape[2]):
		# extract the confidence (i.e., probability) associated with
		# the prediction
		confidence = detections[0, 0, i, 2]

		# filter out weak detections by ensuring the `confidence` is
		# greater than the minimum confidence
		if confidence > args["confidence"]:
			# extract the index of the class label from the
			# `detections`, then compute the (x, y)-coordinates of
			# the bounding box for the object
			idx = int(detections[0, 0, i, 1])
			box = detections[0, 0, i, 3:7] * np.array([w, h, w, h])
			(startX, startY, endX, endY) = box.astype("int")

			# draw the prediction on the frame
			label = "{}: {:.2f}%".format(CLASSES[idx],
				confidence * 100)
			cv2.rectangle(frame, (startX, startY), (endX, endY),
				COLORS[idx], 2)
			y = startY - 15 if startY - 15 > 15 else startY + 15
			cv2.putText(frame, label, (startX, y),
				cv2.FONT_HERSHEY_SIMPLEX, 0.5, COLORS[idx], 2)
			label_glob = label
			recog_class = CLASSES[idx]


	# show the output frame
	# cv2.imshow("Frame", frame)
	# Alt: send the output frames
	#out.write(frame)
	key = cv2.waitKey(1) & 0xFF
	#print(label_glob)
	if recog_class == "car":
		print("Car detected")
		sock.sendto(bytes(str(count) + " " + label_glob + "\n", "utf-8"), (ip_addr, port))
		count = count + 1

	# if the `q` key was pressed, break from the loop
	if key == ord("q"):
		break
```

```
      # update the FPS counter
      fps.update()

# stop the timer and display FPS information
fps.stop()
print("[INFO] elapsed time: {:.2f}".format(fps.elapsed()))
print("[INFO] approx. FPS: {:.2f}".format(fps.fps()))

# do a bit of cleanup
cv2.destroyAllWindows()
vs.stop()
```