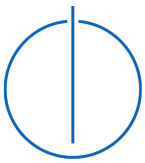
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master Thesis

A Flexible, Application-focused Benchmarking Suite for Edge Infrastructures

Simon Bäurle



DEPARTMENT OF INFORMATICS

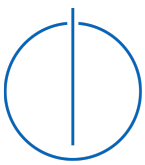
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master Thesis

A Flexible, Application-focused Benchmarking Suite for Edge Infrastructures

Entwicklung einer flexiblen, anwendungsorientierten Benchmarking Suite für Edge Infrastrukturen

Author:	Simon Bäurle
Supervisor:	Prof. Dr. Jörg Ott
Advisor:	Dr. Nitinder Mohan
Submission Date:	15.12.2021



I confirm that this master thesis is my own work and I have documented all sources and material used.

Munich, 15.12.2021 Simon Bäurle

Acknowledgments

I want to thank the Chair of Connected Mobility and my supervisor Prof. Dr. Jörg Ott, for the possibility to perform my master's thesis with them. In particular, I want to thank my advisor Dr. Nitinder Mohan for guiding me in the right direction and an open ear for all challenges during this time.

I want to thank my friends and colleagues that supported me in this journey. When unforeseen problems arose, you were there for me and calmed me down again. Thanks for believing in me, even if I couldn't do so myself.

A special thank goes to my grandparents, my mother, my father, and my sisters, who were there for me no matter what. You carried me through the stresses of my studies and shaped who I am today.

Vielen Dank!

Abstract

Edge computing is a vibrant research field, where applications previously hosted in the cloud shift parts of their workload towards the network's edge. This promises decreased network latencies, especially for rural areas, and allows our current infrastructure to handle the increasing communication demand of smart applications. Specialized edge devices integrate low-power hardware accelerators to deal with the higher computation demand in an energy-efficient way. However, this heterogeneity introduces additional complexity for developers that want to adopt edge capabilities into their application design. The dispersed development of edge components leads to a complex landscape of software libraries, each supporting only a small subset of devices. Innovative orchestration approaches abstract the complexity of the edge from the developer and allow for easy and descriptive deployments. Comparing hardware and software platforms for the edge proves to be a challenging task, where established benchmarking approaches are not suitable. This thesis proposes an edge-based, application-focused benchmarking suite that helps developers evaluate this vast space. Benchmark workloads resemble realistic applications that consist of multiple, interconnected microservices. These services get placed using a novel scheduling scheme, where an extensive exploration of possible deployment options is evaluated. The suite is designed with future edge developments in mind and allows for the integration of new, custom workloads that utilize new technologies. A descriptive configuration structure simplifies this integration process and supports custom metrics. The thesis provides a realistic video analytics pipeline as a sample workload that highlights the different aspects of edge devices.

Kurzfassung

Edge Computing ist ein dynamisches Forschungsgebiet, bei dem Anwendungen, die zuvor komplett in der Cloud gehosted wurden, gewisse Teile an den Rand des Netzwerks verlagern. Dies verspricht verringerte Netzwerklatenzen, insbesondere für ländliche Gebiete, und ermöglicht unserer aktuellen Infrastruktur, den steigenden Kommunikationsbedarf zukünftiger, intelligenter Anwendungen zu bewältigen. Spezialisierte Edge-Geräte integrieren stromsparende Hardwarebeschleuniger, um dises höheren Rechenbedarf energieeffizient zu bewältigen. Die Heterogenität führt jedoch auch zu zusätzlicher Komplexität für Entwickler, die Edge-Funktionen in ihr Anwendungsdesign integrieren möchten. Dies führt zu einer komplexen Landschaft an Softwarebibliotheken, von denen jede nur eine kleine Teilmenge von Geräten unterstützt. Innovative Orchestrierungsansätze abstrahieren diese Komplexität und ermöglichen einfache und anschauliche Bereitstellungen. Trotzdem erweist sich der Vergleich von Hard- und Softwareplattformen als eine anspruchsvolle Aufgabe, bei der etablierte Benchmarking-Ansätze nicht geeignet sind. Diese Masterarbeit umfasst eine Edge-spezifische, anwendungsorientierte Benchmarking-Suite, die Entwicklern hilft die Leistung verschiedener Ansätze zu bewerten. Benchmark-Workloads ähneln realistischen Anwendungen, die aus mehreren, miteinander verbundenen Microservices bestehen. Diese Dienste werden unter Verwendung eines neuartigen Scheduling-Schemas platziert, bei dem eine umfassende Untersuchung möglicher Bereitstellungsoptionen evaluiert wird. Die Suite wurde im Hinblick auf zukünftige Edge-Entwicklungen entwickelt und ermöglicht die Integration neuer, benutzerdefinierter Workloads, die neue Technologien verwenden. Eine beschreibende Konfigurationsstruktur vereinfacht diesen Integrationsprozess und unterstützt benutzerdefinierte Metriken. Die Arbeit bietet eine realistische Videoanalyse-Pipeline als Beispiel-Workload, die die verschiedenen Aspekte von Edge-Geräten hervorhebt.

Contents

1 Introduction

1.1 Introduction

The cloud paradigm has been established as the predominant approach in computer science. Originating in the rise of successful, data-driven companies like Amazon, Google, or Facebook, many companies adopt similar strategies to their IT infrastructure. Different vendors offer a diverse product portfolio available for usage-based rent. Applications can operate more flexibly without any upfront investment in hardware. Capital intense operations on smaller, company-owned data centers suspended and shifted into the cloud. Cloud vendors centralize resources on a much larger scale and can hugely benefit from economies of scale [94]. The operation happens on a global scale allowing data centers located in economically beneficial locations [10]. While the cloud reduces upfront costs, the centralization in limited locations raises serious concerns for the future scalability of this computing model. Infrastructure connecting the data center and the end-user has to handle growing demands. Estimations expect the total number of Internet users to grow to 5.3 billion people by 2023 [19]. In combination with recent applications utilizing heavy Artifical Intelligence (AI) and Machine Learning (ML) workloads, this led to reinforced research into more distributed paradigms.

Edge computing is such a novel approach, where application workloads are shifted from centralized providers towards the edge of the network [61]. It utilizes (existing) hardware located at cellular antennas, routers, or small data centers along the path of communication from the user to the cloud [1]. This reduces the communication distance between the user and parts of the application and enables latency-critical or bandwidth heavy applications [76], such as autonomous vehicles [67] or augmented/virtual reality [86]. Other motivations include the decentralization of current cloud environments and a more distributed computational load where pre-treatment and possible data reduction at the edge prevent overloading the network's infrastructure. For the increasing number of mobile and IoT devices [19], such an approach provides an efficient way to deal with the increasing traffic volume and sudden local load spikes [103]. New and innovative technologies (like Azure IoT Edge [53] or AWS Greengrass [52]) are developed that help to advances the adoption of edge computing in real-world scenarios. However, their novel nature makes comparisons between different approaches challenging [45].

Their heterogeneity limits comparability when employing established benchmarking procedures used for off-the-shelf computer hardware. Researchers have proposed a variety of edge-specific benchmark approaches to evaluate the performance of different workloads across different edge deployments [24, 46, 20, 36, 62, 72]. Despite these efforts, current techniques paint an incomplete picture of the specific requirements imposed by the edge [104]. Lastly, these benchmarks focus on particular aspects of edge computing and lack a realistic edge workload that utilizes the heterogeneous devices at the edge [104].

1.2 Problem Statement and Research Goals

Based on the shortcomings of current benchmarking approaches, this thesis identifies three major goals in order to refine existing research:

RG1 **Realistic and representative edge workload:** Most approaches focus on Central Processing Unit (CPU) and memory benchmarking as their dominant metric, disregarding new, energy-efficient accelerators (like Google Edge Tensor Processing Unit (TPU) [39] or the Nvidia Jetson [21] devices) designed for intense computations at the edge [104]. Thus the workload must integrate arising hardware and computing trends into its general structure. Despite its significant impact on the overall performance, networking is at most measured implicitly [104] mandating an explicit consideration in the workload design. Most benchmarks do not operate on real-world data, focusing only on specific aspects of the edge [104]. They lack significance in representing actual edge workloads employed by researchers and the industries. The workload needs to resemble a realistic mapping of actual edge applications, compiled of different services representing certain parts of the edge.

RG2 **Integration of software platforms and different virtualization techniques:** Current strategies do not integrate established or emerging software platforms like Kubernetes [99] or EdgeIO [115] into their approach. These platforms play a huge role in tackling the still existing challenges of edge computing and play a substantial role in the performance of the applications. The lack of different deployment options and virtualization techniques employed strengthen the assumption that current benchmarks cannot capture the performance in multi-tenant environments [104]. The benchmarking suite should integrate different software platforms and employ distinct virtualization techniques, to capture these aspects of edge computing. This enables a delimitation of current approaches regarding their integrated performance in the edge.

RG3 **Extensible platform with openly accessible component:** The considerable research effort in edge computing results in an evolving landscape and arising trends, resulting in rapidly changing requirements [91]. These changes could impair the expressiveness of the benchmarking suite in general, so the approach must be adjusted or extended easily. Individual components must be publicly accessible and allow other researchers to verify and improve the benchmarking application.

1.3 Contribution

This thesis proposes an extensible edge benchmarking suite specifically focused on realistic application workloads. The implemented solution allows developers to adjust the environment or integrate new benchmark services based on a comprehensive configuration. By interpreting the assignment of workload services to edge devices as a bipartite graph, the benchmarking suite integrates a novel matching scheme and generates an exhaustive list of workload schedules. This helps developers explore counter-intuitive deployment options that may perform better in certain edge environments. The platform architecture is easily expandable to integrate support for modern orchestration platforms such as Kubernetes or EdgeIO. The separate metric system allows the benchmark suite in secluded environments, where the benchmarked environment may operate in challenging network environments.

On top of the benchmarking suite, this thesis contributes a novel video analytics pipeline that decomposes the tasks for object tracking into four distinct microservices. It supports the heterogeneity of edge deployments and utilizes different processor architectures and hardware accelerators to reach the full potential of the used edge devices. The explicit communication between the services represents recent software architecture trends with small, separate services deployed on different nodes. The modular pipeline architecture enables the replacement of individual services to integrate future and more advanced approaches.

1.4 Thesis Structure

The remaining thesis consists of five chapters that describe the concepts and technical details behind the proposed benchmarking suite. Chapter 2 explores modern application architectures, details edge computing and existing benchmarking approaches, and explains the general concepts for the pipeline services. In chapter 3, we describe the architecture of the employed video analytics pipeline and highlight how this workload represents state-of-the-art technologies employed by industries and researchers. Next,

chapter 4 shows the overall architecture of the benchmarking suite and describes the configuration structure, the workload matching, and the metrics system in more detail. Chapter 5 evaluates our benchmarking suite on two distinct environments and describes our findings. The last chapter 6 discusses these results, summarizes the thesis, and gives an outlook for future work in edge benchmarking.

2 Background and Related Work

2.1 Modern Application Design

The ongoing globalization forces companies to operate in permanent competition on a global scale. This places increasing stress on traditional business models and requires companies to adjust to this changing environment [106]. Future economic success depends on the ability to efficiently operate on a large scale and adapt to quickly changing circumstances [54]. Agile strategies aim to reflect these requirements by focussing on incremental improvements in short cycles. This allows businesses to gain feedback during early product development and fine-tune the approach to the actual reality [92]. Recent years have seen the establishment of new software architecture approaches that reflect this need for small-scale changes. Monolithic applications tend to result in codebases that share functionality between different modules, which leads to complex dependencies that hinder independent adjustments without affecting other components [78]. Architectures have shifted to more flexible, service-oriented concepts where complex applications get decomposed into loosely-coupled, independent microservices [107]. Each service provides a subset of the overall functionality and is strictly separate from the other components [78]. They may be improved or replaced with little to no influence on the other services and allow different implementations to harness the potential of different programming languages and libraries to provide the best solution for each subproblem [40]. Global players like Google or Amazon have facilitated this trend and developed advanced packaging, networking and, orchestration approaches that help handle the increasing number of services [54].

Service Communication Microservice architectures result in distributed systems that coerce the developers to comprehend the used communication channels and the consequent implications on the system [57]. This is in contrast to previous implementations of distributed systems that utilized complex middleware (like Common Object Request Broker Architecture (CORBA) [80]) to abstract the actual service location and potential networking from the developer. The missing explicitness resulted in suboptimal development and obscure application behavior [33]. The strong cohesion of individual microservices, on the other hand, prevents unwanted dependencies between compo-

nents and requires the specification of well-defined interfaces for data exchange [57]. This communication employs lightweight communication protocols such as Representational State Transfer (REST) [32], gRPC [41] or message queuing systems like Apache ActiveMQ [95]. They ensure interactions between services happens in an efficient and strict way [54].

Execution Environments Deploying numerous small microservices is a complex and rather failure-prone task of the development process. Code may work locally on the developer's machine but fail upon deployment. Differences in the software environment (e.g., conflicting library versions or Hardware (HW) capabilities) can introduce a collection of problems that are notoriously hard to diagnose [73]. While Infrastructure as Code (IaC) solutions enable the creation of reproducible and consistent environments, bare-metal execution is too inflexible for today's applications. A large number of potential users may overload the system, so the used hardware must accommodate buffer capacity. For times outside peak load, this hardware tends to idle but cannot get decommissioned [110]. To counteract this over-provisioning and better utilize the existing resource, several virtualization approaches have been proposed:

- **Virtual Machines:** A hypervisor provides virtualized resources on top of bare metal hardware. Virtual Machines (VMs) execute a complete guest operating system on top of this hardware. One system can operate multiple VMs that are logically separated from each other and can integrate different applications, libraries or, even operating systems.
- **Containerization:** Software containers are common processes that get executed in an isolated environment of the operating system. They share the same underlying kernel and utilize several shielding techniques to separate different containers. This approach provides more lightweight isolation without the additional overhead of VMs.
- **Unikernels:** A novel approach that integrates high-level application code with a small kernel runtime to produce a specialized unikernel. They only integrate the required libraries and system calls which reduces their size [71].

Cloud native applications Cloud vendors use these techniques to partition their hardware into virtual blocks available for rent on a flexible basis. They operate additional hosted services such as databases or Application Programming Interface (API) gateways to integrate into a cloud-native application. Figure 2.1 shows an example for such an architecture, where different microservices and cloud offerings are

combined into a unified design. The individual services are implemented using the most efficient programming language and libraries for their problem. Communication between the components uses defined interfaces that utilize state-of-the-art protocols. Service instances can be scaled using the flexible, cost-efficient resources provided by cloud vendors and scales according to the applications load.

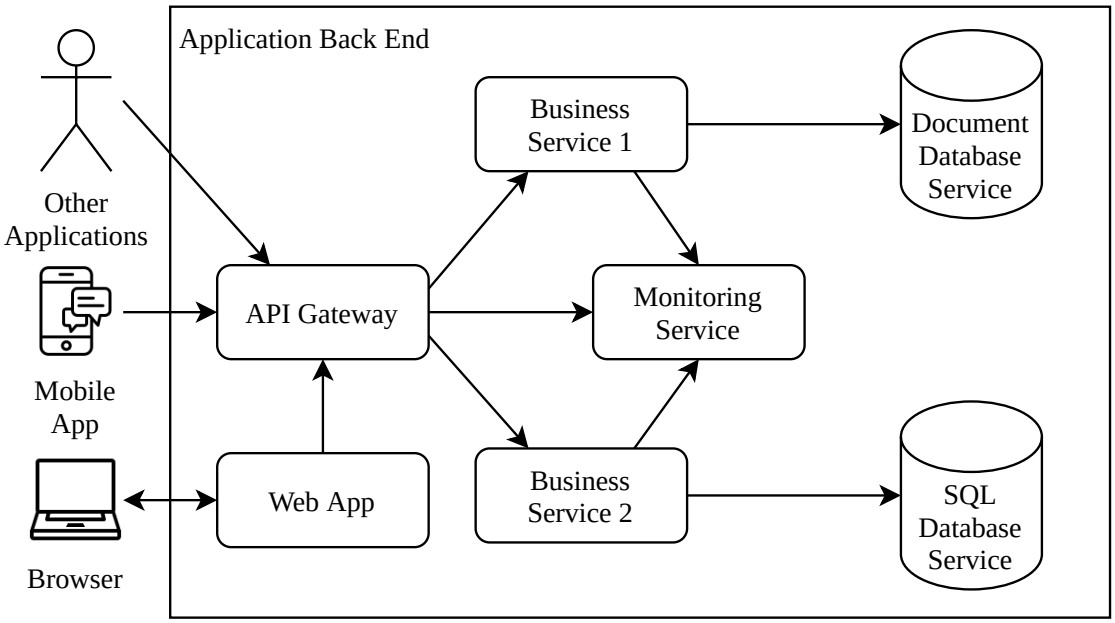


Figure 2.1: Cloud Native Application, adapted from [88]

2.2 Edge Computing

The shift towards cloud computing has changed the overall shape of the software industry [116]. Public cloud providers like Amazon Web Services (AWS), Microsoft Azure, or Google Cloud Platform (GCP) highlight the absolute economic success and ongoing adoption of cloud computing. Service-based computation reduces upfront costs and provides overall economic benefits through flexible, consumption-related billing [94, 47]. Business models evolve into service-based approaches forwarding this underlying flexibility to the customer [30]. Developers have adopted this approach as the predominant type of application deployment and operation [116]. Recent HW trends lead to a growing number of smart devices integrated into this computing landscape. They utilize progress in AI, video-processing, automation, and sensor technology and lead to growing network demands. Emerging applications are estimated to contribute

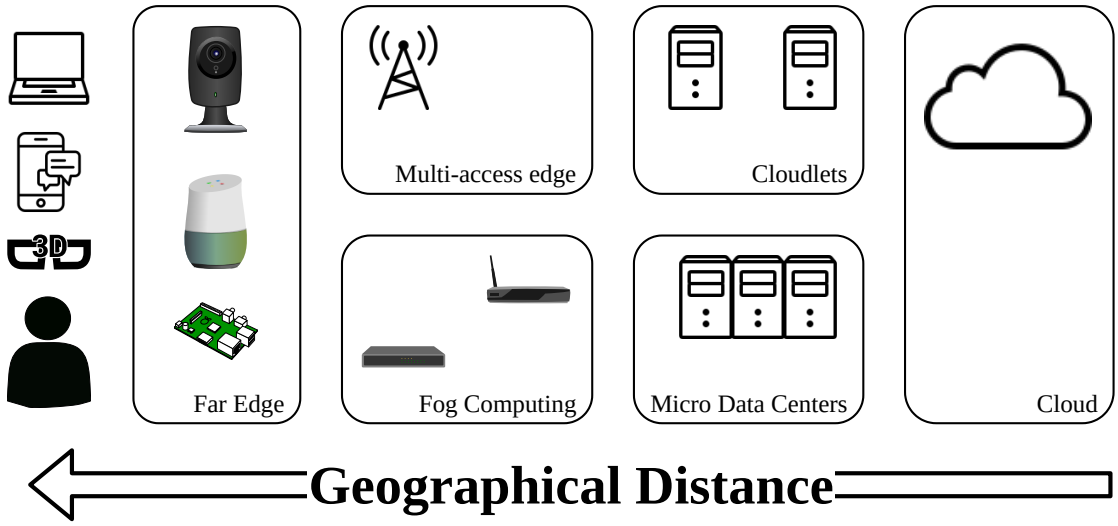


Figure 2.2: Terminologies on the edge spectrum (Courtesy: Dr. Nitinder Mohan)

a big part of future bandwidth demands, with video processing and Augmented Reality (AR)/Virtual Reality (VR) workloads as dominant contributors. Data gets consumed in a more distributed and mobile fashion, placing strong demands on the infrastructure of the future [19]. Such workloads are possible in well-connected locations with datacenters nearby, yet more remote or less connected areas pose a significant challenge [76]. Multiple approaches for these challenges have been proposed, including micro data centers [89], cloudlets [89, 61] or fog computing [113]. Their general motive is the relocation of data processing tasks along the path of communication towards the edge of the network, thus shaping the term edge computing [91]. Figure 2.2 shows different terminologies and their placement along the edge spectrum.

The reduced geographical distance and more distributed execution promise several benefits addressing upcoming challenges in cloud scenarios:

- Better scalability for the increasing number of smart devices contributing to large data amounts [116].
- Low latency for latency-critical applications such as autonomous vehicles or emergency services [91].
- Reduced bandwidth requirements for applications like content delivery networks or video processing [19].
- Overall improved Quality of Service by eliminating single points of failure in form of cloud data centers [116].

- Counteracting privacy and security concerns with centralized data silos [90].
- Higher fault tolerance by further geographical distribution [90]

While this shift offers improvements in latency and bandwidth optimization, the heterogeneity and heavily distributed nature of the edge introduces new technical challenges and overhead. Current research focuses on improved latency and bandwidth utilization as the incentive behind edge computing. Yet research has shown that many applications, initially targeted at the edge, can be operated using improved and simpler services in the cloud [76].

2.3 Edge Devices and Accelerators

Ongoing research in computation-heavy AI technology (ranging from natural language processing to large-scale video processing) will influence future applications. They employ novel techniques that utilize heavy machine learning to achieve a market advantage [105]. These new applications require substantial computing capacity on top of existing infrastructure, resulting in the development of new, specialized hardware. While recent years have seen a shift datacenter workloads CPU workloads to General Purpose Graphic Processing Unit (GPGPU) processing, their high energy consumption is not suitable for restricted edge deployments [13]. More constrained edge devices require original approaches that deliver sufficient computing power while being far more energy-efficient. Different edge accelerators are developed that speed up the execution of ML workloads through specialized components [64]. For example, many modern System on a Chips (SoCs) integrate an extra co-processor specifically targeted at AI workloads [83, 4]. Other companies, including Google, Intel and Nvidia, develop optimized, standalone hardware accelerators targeted at edge scenarios [21, 39, 51]. Figure 2.3 gives an overview of such accelerators and groups them by the underlying technology. The development of new accelerators evolves rapidly and results in a dispersed field mostly comprised of specific island solutions. Standardized interfaces or even similar physical properties (e.g., USB pluggable accelerators and integrated devices) are missing, complicating the integration into current applications. Their hardware architecture and the underlying software platforms (Operating System (OS) and Software Development Kits (SDKs)) differ from traditional hardware, rendering common benchmarking approaches incompatible [102, 64].

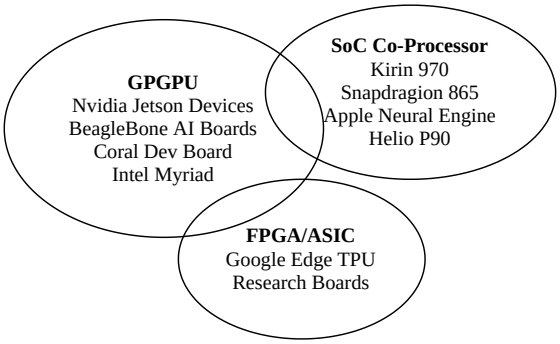


Figure 2.3: Overview of Edge Accelerators

2.4 Edge Orchestration and Software Platforms

Envisioned edge environments consist of many constrained devices in geographically distributed locations. The heterogeneous hardware capabilities and configuration techniques prove difficult for state-of-the-art cloud orchestration platforms such as Kubernetes [99]. Their implementation assumes strong assumptions present in data center environments, complicating the integration of loosely coupled edge infrastructures [14]. Devices are more likely to fail or change connections resulting in constant rearrangement and service migrations [15, 7]. Federated environments may get operated by different legal entities, which causes different execution environments, network policies, or node capacities [77]. Novel work on edge-focused orchestration frameworks integrates scheduling across ownership boundaries and supports the different hardware capabilities through more advanced scheduling approaches. KubeEdge adds RPC-based communication channels and edge multi-tenancy to the Kubernetes framework [108]. Kubefed proposes a similar approach, where Kubernetes multi-cluster environments get integrated into a common architecture that allows distributed scheduling [59]. ioFog is the first independent approach that introduces a prediction-based model to place tasks at sufficiently fitted nodes at the edge-cloud continuum [2]. EdgeIO proposes a hierarchical multi-cluster architecture and service-level based scheduling that takes user-defined restrictions into account [115]. It integrates a novel networking component designed to overcome the limitations of edge devices deployed in restrictive network conditions such as Network Address Translations (NATs) or firewalls [9].

2.5 Edge Benchmarking

The rapidly evolving edge landscape has sparked interest in the development of benchmarking systems targeted at the specific characteristics of the edge. These systems aim to compare the performance of different edge platforms and applications. They evaluate the benefits and implications of new approaches and can guide future research directions [104]. This section will give a brief overview of benchmarking techniques, potential metrics and highlight identified improvements for available solutions.

2.5.1 General Benchmarking

Before moving into edge-specific benchmarks, it is important to understand the general term "benchmarking". It is defined as the act of measuring the quality or performance of a new solution by comparing it to some accepted standard. This reference is called the benchmark or benchmarking problem and can range from business competitors to predefined goals [16]. In computer science, benchmarks use a standardized problem that is used to compare the performance of different environments or devices [75]. There are two major benchmarking categories:

- **Synthetic benchmarks:** These are artificial programs designed to measure the raw performance of a computer system or its components. They seldom resemble real applications and introduce rather arbitrary loads (e.g., copying random files between disks) [49]. Examples for such benchmarks are 3DMark [101] for GPUs or CrystalDiskMark [22] used for hard drives. Synthetic benchmarks allow for easy comparison between systems, though the real-world performance may vary for actual applications.
- **Application benchmarks:** Application-focused benchmarks, on the other hand, integrate realistic workloads that resemble characteristics of typical applications executed in the benchmark environment [63]. Their results paint a more diverse and complete picture of the actual performance of the computer system as perceived by the end-user. Good examples for such benchmarks are in-game sequences provided by some 3D engines [56] or the later discussed edge benchmarking approaches.

Another subdivision of benchmarks decomposes approaches into micro-benchmarks that target individual system components and macro-benchmarks that focus on the overall performance of the system. We enumerate and discuss the capabilities of existing edge benchmarks in section 2.7.

2.5.2 Evaluation Metrics

To calculate performance scores for the evaluated systems, the benchmarking solution must collect metrics from the employed workload. Aggregated results give developers a quick overview, while deeper analysis of the raw data helps to identify strengths and weaknesses of new approaches and guide the direction of further development [5]. Table 2.1 list typical performance metrics that can be aggregated in edge scenarios [8, 31, 5].

Metric Type	**Metrics**
Performance	Throughput Latency Bandwidth Packet Loss Processing Power (FLOPS/IPC) Memory Speed I/O Speed Speedup
Application	Rate of failure Number of concurrent clients Quality of Service Computation/Communication ratio Failure handling/mitigation
Environment	Temperature Energy Consumption Power Efficiency
Platform	Utilization Overhead Failed scheduling decisions Number of resource conflicts Adaption time

Table 2.1: Performance indicative metrics for edge environments

Performance metrics show the operation speed of the edge devices. They do not need any complex setups and can get measured directly. Application-level metrics provide insights into the performance of the benchmark workload but require minor adjustments to measure them. Environment and platform metrics, on the other hand, need more sophisticated approaches to allow consistent and reliable measurements. Benchmarks must be conducted in controlled and well-fitted environments, further

complicating measuring them.

2.5.3 Challenges and Improvements in Edge Benchmarking

In section 2.7 we discuss the individual approaches in more detail, summarized in table 2.2. While existing benchmarks provide a comprehensive mixture of possible workloads, they lack the integration of edge-specific accelerators or software platforms. The majority of existing benchmarks focus on well-established CPU and memory metrics combined with application-specific metrics. The diverse workloads capture many possible edge scenarios comprehensively but fail to grasp the heterogeneity of edge environments. Based on our findings, we identified four major areas of improvement that should be addressed by future benchmarking approaches [104]:

I-1 Most edge benchmarks only use CPU and memory performance as their primary evaluation metric for node performance. The influence of edge accelerators, storage devices, and network connections requires further research.

I-2 Current approaches disregard the performance impact of software platforms employed at the edge. However, their service orchestration plays a vital role in the resulting application performance Future benchmarks should support different orchestrators, service models, and scheduling implementations and compare their impact on the benchmark performance.

I-3 Application-specific Quality of Service (QoS) metrics (e.g., throughput, failure ratio, concurrent users) are the most common metric type in current benchmarks. Other criteria like energy consumption or hardware utilization may provide further insights into edge performance.

I-4 Most benchmarks do not compare the influence of different virtualization techniques or offloading strategies. Their workload may not be representative of large-scale, distributed workloads. The focus lies on single application performance, while multi-tenant environments could capture a more comprehensive picture of performance at the edge.

2.6 Object Detection and Tracking

The vision for edge computing includes different computation-heavy applications made possible through a closer execution near the user. Examples include video surveillance, smart home, smart city, AR/VR or autonomous vehicles [76]. Most of these applications utilize heavy ML techniques to capture the user's environment. For video surveillance

or autonomous vehicles, this includes the processing of (multiple) video inputs to identify the location of visible objects (car, people, etc.). Both object detection and object tracking are crucial aspects of such tasks and are actively researched. In recent years there has been remarkable progress in the accuracy of these approaches as well as significant performance speed-ups. This section will give a short overview of current object detection and object tracking techniques.

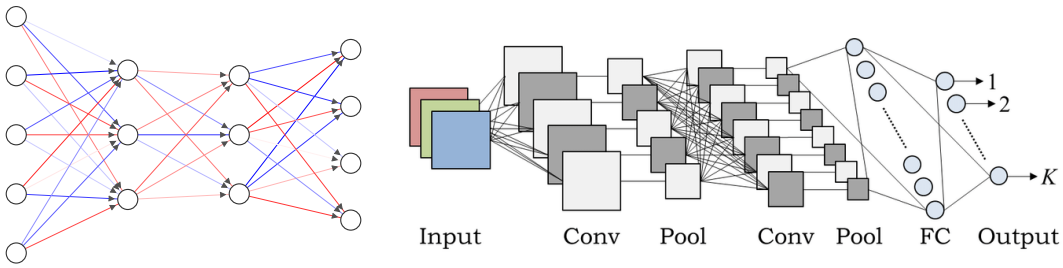
2.6.1 Object Detection

Extracting the location and type of objects in a continuous stream of video frames is a big challenge in computer vision [11]. Research has made significant progress, from traditional detectors that use few handcrafted image features to approaches that utilize extensive ML using Convolutional Neural Networks (CNNs) [118]. This subsection describes the theoretical foundation for neural networks and explores current object detection approaches based on it.

Neural Networks Neural networks are inspired by the biological structure of the human brain. They consist of artificial neurons that mimic the behavior of biological neurons. Each neuron processes an input $x \in \mathbb{R}^d$ by defining a function $f(x) = \wp(wx^T + b)$ with $w \in \mathbb{R}^d$ being the weight vector and $b \in \mathbb{R}$ as the bias. The neural network (see Figure 2.4a[1]) consists of directed connections between multiple neurons that take the initial input I and pass it through further (hidden) layers until the output layer is reached. Each of these links bears an individual weight that influences its importance for later layers [38]. In a later training stage of the network, they get optimized using a loss function that models the performance of the neural net based on the training input [69]. Two common network types are Recurrent Neural Networks (RNNs) that allow directed circles in the network structure, and Feedforward Neural Networks (FNNs) that only connect with layers ahead [44].

Convolutional Neural Networks (CNNs) In FNNs, each neuron in a hidden or output layer connects to all neurons in the previous layer. The number of weights is the multiplication of the number of neurons in each layer. For large input sizes (such as images), this results in a drastic increase of machine learning parameters, making meaningful model training impossible Reducing the number of neurons in later layers reduces this issue, yet the drastic effects on classification performance make it a non-viable option [44]. CNNs address this problem by introducing two layer types:

[1] Created with NN-SVG



(a) Feedforward Neural Network (b) Convolutional Neural Network [48]

Figure 2.4: Architectural Basics of Neural Networks

- **Convolutional Layer:** Neurons get rearranged into n separate blocks that cover the whole input still. Correlation between distant inputs (e.g., opposite corners of an image) is low, the receptive field (input) of each neuron can be reduced to a smaller $m \times m$ part. Neurons arranged in the same block share their weights, which further reduces the number of parameters. Calculating the output of each block corresponds to the convolution using a $m \times m$ filter and results in a significantly smaller feature map as output. CNN architectures typically learn from multiple such filters in parallel [44].
- **Pooling Layer:** Generating multi-dimensional feature maps using convolutional layers only partially solves the problem of large input sizes. Pooling layers address this by replacing an area of inputs from the previous layer with only one output. Common CNN architectures use the maximum values or averages as the output of the pooling operation. Additionally pooling layer help to make the neural net more invariant to small input changes (e.g., small variances in pixel brightness) [38].

Figure 2.4b shows a typical CNN built from multiple convolutional and pooling layers followed by one fully-connected layer before the output. Such architectures allow for much deeper neural networks for large or unknown input sizes and are the basis for recent progress in object detection [44].

You Only Look Once (YOLO) Initial deep learning approaches for object detection utilized two separate stages (proposal detection and verification) in their architecture. While they produced accurate results, their processing speed was fairly low and not suitable for real-time applications [118]. In 2016 You Only Look Once (YOLO) [84] abandond this principle, by formulating a regression problem directly from the image input to bounding boxes [66]. Figure 2.5 show the network architecture consisting

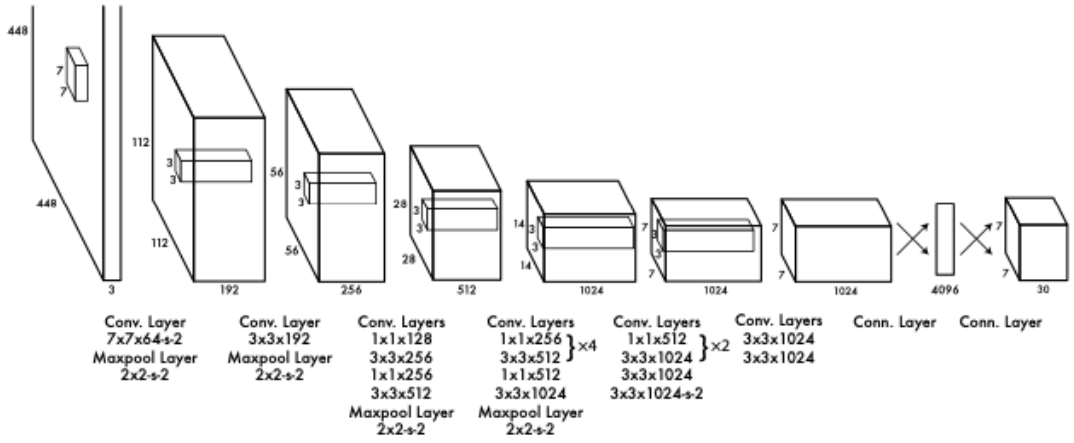


Figure 2.5: YOLO architecture [84]

of 24 convolutional and pooling layers followed by two fully connected layers. The approach divides input images into an $S \times S$ grid, where a cell detects objects if its center falls into them. Each cell predicts multiple bounding boxes, confidence, and class probabilities which get passed through non-maximum suppression to only output the most probable detection [84].

The network's performance is significantly faster than previous approaches, reaching up to 45 fps on a Titan X GPU. A reduced version of the network, called TinyYOLO, uses only nine convolutional layers and achieves 145 fps throughput [84]. Further iterations of the initial architecture have improved the detection performance while keeping a similar throughput [85].

Conclusion The adoption of deep learning for object detection has greatly improved tracking accuracy and performance. Two-stage approaches like RCNN [37] generate very accurate results but lack the necessary throughput for real-time applications. Unified approaches such as YOLO [84] or SSD [68] trade a better throughput for small accuracy losses which makes them capable of detection in real-time.

2.6.2 Object Tracking

While object detection helps understand single video frames and detect present objects, applications such as autonomous vehicles require more input than the object's location. Tracking approaches address this challenge by extracting the location and the trajectory of objects in a video stream [6]. Objects that are in motion can introduce a variety of

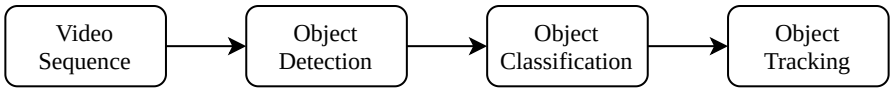


Figure 2.6: Basic steps for object tracking [6]

new problems, where light levels suddenly change, objects get occluded, or motion blurring occurs in the video stream [65]. Additionally, the projection of 3D objects onto 2D video frames causes a loss of information where rotations or object deformations challenge the tracking approach [114]. Figure 2.6 shows the general steps performed by current tracking approaches.

Detection + Classification Before the actual tracking algorithm, the initial location of tracked objects needs to be initialized. This can either be done manually by the user or automatically using CNNs and ML approaches described in the previous subsection 2.6.1 [18]. How often new detections get integrated depends on the tracking algorithm and the available computing power. For moving objects, this step employs other detection techniques based on frame differences to disregard non-moving objects [6, 26]. Many approaches include an additional step, either integrated with the detection algorithm or as a separate procedure, to classify the detected objects into different classes [6].

Tracking The following tracking algorithm takes the classified detections as input and updates their position in future video frames. Current approaches are grouped into three major categories [6]:

- **Point Tracking:** These algorithms focus on single feature points of each object. Typically they first update the object's positions based on new frame inputs and correct wrong assumptions in a second step. Examples of these approaches are Kalman or particle filters.
- **Kernel Tracking:** Moving objects are calculated with the usage of non-linear regression. The solution is expressed as a linear combination of samples and attained using a kernel function. Such approaches (like Kernelize Correlation Filters (KCF) or mean shift tracking) use the object geometry as the input.
- **Silhouette Tracking:** Such approaches generate an object silhouette based on the initial detection. The tracking tries to adapt to the specific object shapes (e.g., hands) instead of geometric shapes. Current approaches are based on either object contours or object shapes.

Multiple Object Tracking (MOT) The described methods are effective in tracking the trajectories of single objects. However, most applications need information about multiple objects, so the tracking approaches must calculate multiple object trajectories. This Multiple Object Tracking (MOT) has its unique set of challenges, where tracked objects can temporarily occlude each other and need to be re-identified when they appear again. Objects can interact with each other and mutually affect each other's state (e.g., a person that drives a bike) [18].

2.7 Related Work

This section reviews and classifies current edge benchmarking approaches. Table 2.2 summarizes the most relevant edge benchmarks and highlights the components benchmarked by their approach. We describe the benchmark workload and highlight missing aspects for each approach.

Benchmark	CPU	Accelerators	Memory	Storage	Network	Orchestration	Sevice Model	Schedulers	AI platforms
CoAP benchmark [58]	+	-	+	-	-	-	-	-	-
RIoTBench [93]	+	-	+	-	-	-	-	-	-
EdgeBench [24]	+	-	+	-	-	-	+	-	-
Edge AIBench (concept only) [46]	-	-	-	-	-	-	-	-	+
DeFog [72]	+	-	-	-	+	-	-	-	-
Edge accelerator benchmarking [87, 27]	+	+	-	-	-	-	-	-	-
EdgeBench (2) [111]	+	-	+	-	+	-	+	-	-
OpenRTiST [36]	+	+/-	-	-	+	-	-	-	+/-
Scission [70]	+	+/-	-	-	-	-	-	-	-

Table 2.2: Comparison of benchmarks and tested components, modified from [104]

Benchmarking of IoT devices The benchmark can be split into two separate methods to evaluate constrained off-the-shelf hardware (Raspberry PI, BeagleBone, and BeagleBone Black). First, Imbench [74] is used to measure the performance of each device.

It computes operation speeds and latencies to determine the overall machine performance. The second phase deploys a gateway for the constrained application protocol (CoAP) on the devices and measures the response latency for repeated requests [58]. The benchmark gives a good overview of the CPU performance of edge devices and evaluates their gateway performance. It does not consider the distributed execution of more complex workloads and leave out edge accelerators.

RIoTBench focuses on distributed stream processing system (DSPS) that provides an intuitive dataflow model for scalable, low-latency streaming applications. The suite provides 27 distinct micro-benchmarks that address different types of streaming tasks. They are categorized into data parsing and filtering, statistical and predictive analysis, pattern detection, visual analytics, and I/O operations. On top of the artificial micro-benchmarks, the authors have identified four real-world applications representative of IoT-stream workloads [93]. The actual benchmark focuses on processing performance in data center environments, where only the data input stems from datasets generated by edge devices.

EdgeBench evaluates the performance of the commercial edge offering AWS IoT Greengrass [52] and Azure IoT Edge [53]. The platforms are based on the serverless computing paradigm and extend current cloud offerings to the edge. Device management happens in the unified cloud portal and enables easy integration of edge capabilities. The benchmark provides a speech-to-text decoder, an image recognition model, and a scalar value generator as workload. The workloads are executed in the cloud and on constrained edge devices, respectively [24]. Benchmarking focuses on CPU and memory configurations and disregards potential hardware accelerators available at the edge.

Edge AIBench proposes a conceptual benchmarking suite primarily focused on AI tasks. The authors propose four workload types with prospective potential for extensive edge computing:

- Predicting heart failures of monitored patients in an ICU.
- Identifying people on virtual camera devices emulating surveillance cameras.
- Speech/Face Recognition on smart home devices.
- Road sign recognition for autonomous vehicles.

The benchmark is not fully implemented and lacks qualitative results on edge devices [46].

IoTBench implements benchmark tasks primarily focused on the edge processing part of IoT applications. They provide workloads in the area of computer vision (video summarization, depth estimation, image recognition, localization, and mapping), speech recognition (Mel-Frequency Cipstal Coefficient, Beamforming), and signal processing (compressive sensing). While the benchmark measures many low-level metrics like Million Instruction per Second (MIPS) and L1/L2 cache miss rates, it does not evaluate the influence of multiple services communicating over the network [62].

DeFog focuses on different deployment options on the edge-cloud spectrum. It can operate services in cloud-only, edge-only, and mixed environments. The benchmark provides a diverse mixture of workloads: object classification using ML, speech-to-text conversion, text-audio synchronization, geo-location-based gaming, an Internet of Things (IoT) gateway application, and face detection from video streams. The assets for these tasks are hosted in the cloud and transferred to the service destination upon scheduling. The IoT gateway is the only workload that executes multiple services distributed on different devices [72] It misses the integration of edge accelerators and explicit integration of network connections into more workloads.

Edge accelerator benchmarking Reuther et. al. [87], and Dinelli et. al. [27] propose the only comprehensive study of edge accelerator performance. They benchmark different accelerators by executing CNN workloads and measuring the device throughput. Specifically, they compare different pluggable accelerators (Google Edge TPU, Intel Movidius Compute Stick) with general CPU execution. Their research primarily focuses on the performance of different accelerators and not on application-based generic edge benchmarking.

EdgeBench (2) implements an extendable benchmarking suite for functional edge workflows. The benchmark uses OpenFaaS on top of Kubernetes to execute either a video analytics pipeline or an IoT hub as predefined workloads. They integrate cloud database applications such as InfluxDB or Minio as storage backends for the defined pipelines. It is the only benchmark so far that allows users to integrate custom workloads based on custom pipeline configurations [111]. The benchmark integrates distributed execution of multiple services with explicit communication between them but lacks integration of edge accelerators.

OpenRTiST implements an end-to-end benchmarking approach that utilizes Neural Style Transfer (NST) to transform a live video stream in the style of a reference painting. The benchmark takes the camera input from a mobile phone and processes the image

at different hardware on the cloud-edge spectrum. It focuses on end-to-end metrics for processing the image and includes network communication only between the phone and the device executing the neural net [36]. While the benchmark integrates cloud and cloudlet resources with different GPUs, it lacks additional low-power edge accelerators and devices for computation.

Scission The benchmark proposes context-aware distribution of deep neural networks across multiple devices, so the individual layers are scattered across the edge-cloud continuum. The network layers get split at specified partitioning points determined by the architecture of the neural net (linear or branching). Eight-teen common networks get evaluated, determining their optimal distribution on the cloud-edge spectrum [70]. While the approach provides insights into the distributed execution of common CNNs, the network aspect is only simulated and never explicitly benchmarked.

3 Workload Specifics

This chapter describes the architecture and specifics of the proposed object tracking application that serves as the base workload for the benchmarking suite. Section 3.1 covers the workload-specific requirements stemming from the general research goals, possible performance metrics and, identified improvements of current edge benchmarking platforms. The chapter continues with the general architecture of the pipeline (section 3.2) followed by a detailed explanation of each service. The last section 3.7 details the containerization of the workload and its specific needs for the benchmarking system.

3.1 Requirements

The quality of the workload determines how representative and thorough benchmark results are. Achieving the first research goal (**RG1**) highly depends on the concept for the real-time video analytics pipeline. Based on the research goal of this thesis as well as the identified shortcomings of current benchmark approaches, the following list of requirements serves as a precise specification for the workload design:

R1 *Representativeness:* The workload must represent common applications considered for edge computing. It should stress edge devices through heavy computation and derive the maximum capacity of the benchmarked environment. The workload should be based on a realistic scenario and utilize realistic data that reflects relevant edge use cases and challenges.

R2 *Modularity and Extensibility:* Edge computing is a dynamic field in current research with unforeseeable developments to be expected in the future. This results in fast-changing environments that necessitate adjustments of the workload. Therefore, the workload must use a modular architecture with components that can be replaced or extended without affecting other parts. Integration of new modules has to be uncomplicated and should require only a few or no changes at other components.

R3 *Distributed execution:* Modern application design involves smaller microservices that interact with each other over well-defined interfaces. Services are distributed

across separate machines, and communication facilitates state-of-the-art technology. The workload design must mirror similar behavior and force communication between different modules.

R4 *Utilize edge heterogeneity:* Edge environments consist of a huge variety of different devices (e.g. Raspberry PI, BeagleBone or Nvidia Jetson devices). Combined with pluggable acceleration hardware, this results in a very heterogeneous environment for the benchmark. Contrary to available benchmarks, the workload must exploit this computational potential and utilize different accelerators and instruction set capabilities in its components.

R5 *Technological independence:* Current benchmarks lack support for emerging software platforms at the edge. The workload architecture must support the different virtualization techniques or service models employed by them. Any assumptions regarding networking, scheduling, or other technological requirements may not impair the possibility of executing the workload.

3.2 Workload Architecture

The proposed workload portrays a real-time video analytics pipeline that detects and tracks objects in a video stream. Figure 3.1 shows the proposed architecture composed of four interacting microservices (**R2** and **R3**), each responsible for a subset of the problem.

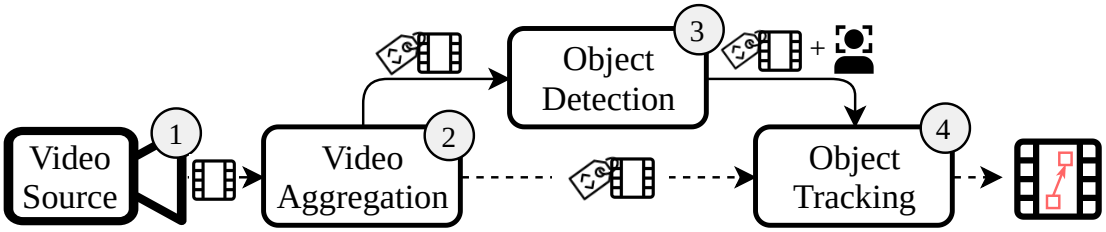


Figure 3.1: Architecture of the Tracking Pipeline

The **video source** ① takes a video input (e.g. video file or webcam) and ingests them into the application. Multiple video inputs are fed to the **video aggregation service** ② that performs necessary pre-processing steps on the video frames and combines them into a synchronized stream for later stages. The **objects detection service** ③ utilizes a CNN to detect objects in the video frames and attaches the detections to the frame metadata. The detections are forwarded to the **object tracking service** ④ that

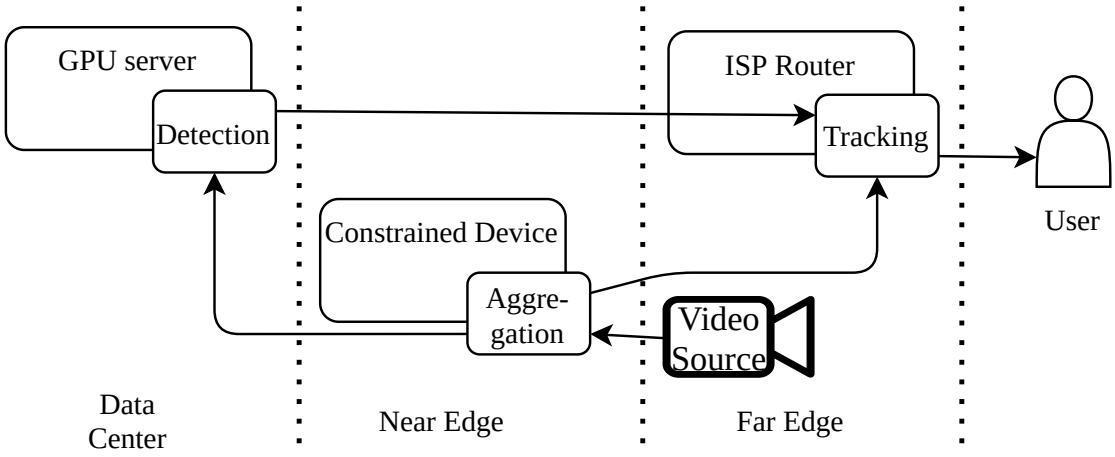


Figure 3.2: Example deployment of the video analytics pipeline

tracks the movement of detected objects and outputs the resulting bounding boxes to an external service.

General Procedure Current approaches to multiple object tracking perform object detection for each video frame. This results in good tracking accuracy but requires significant computing power, which is not available in edge environments. The employed tracking algorithms, on the other hand, are much lighter, making them more appropriate for such constrained devices. The pipeline procedure reduces calls through the detection service to periodic updates to identify new or reappearing objects. All remaining frames are forwarded directly to the tracking stage and only get processed there. The architecture balances accuracy losses from skipping the detection stage with the throughput gains of only the tracking stage. Depending on the configurable detection frequency, this speeds up the pipeline considerably but can cause significant inaccuracies for fast-moving video scenes.

This approach enables diverse deployments that are more representative of actual edge applications (**R1**). Figure 3.2 shows an example distribution of the workloads. The video source(s) is placed near the edge of the environment and closely resembles an actual IP cam. The aggregation service is deployed near its inputs to decrease network latencies and decrease the bandwidth for the later stages by performing the necessary pre-processing. The detection is deployed on a powerful server that offers GPGPU capabilities for the CNN execution, while the tracking service is executed on a more constrained device nearer to the user (e.g., edge device operated by the ISP). With such a configuration, the pipeline can benefit from both the higher computing power of the

GPU server as well as the lower network latencies of the nearer tracking service.

Communication Technology Formulating the multiple object tracking as a set of separate microservices requires service-to-service communication. For this, many applications use REST based HTTP interfaces as the defacto standard. However, recent years have seen the rise of alternative technologies such as GraphQL [96], Falcor [79] or gRPC [41] for APIs. To reflect these developments and integrate state-of-the-art communication approaches (**R3**), the workload utilizes gRPC. We chose gRPC for several advantages to more traditional REST based communication schemes:

- Service APIs are specified explicitly and upfront based on Google's Protobuf [81] interface definition language (IDL). Extensive tooling enables automatic generation of client and server code simplifying development.
- API messages use more efficient binary serialization compared to text-based JSON. In addition, gRPC utilizes the more modern HTTP/2 by default which results in overall performance benefits when compared to REST.
- gRPC supports many common programming languages, so current services can be replaced by better fitting implementations in the future (**R2**). New data fields or interface methods can be added in a backwards compatible way, further decoupling the individual workload services.
- More advanced aspects such as message streaming, integrated load-balancing, or service discovery prepare the communication design for prospective improvements.

3.3 Video Source

The design of the video source reflects common networked cameras used for video surveillance [112]. This allows for replacing the source with an actual camera device and perform benchmarks with real inputs.

This camera emulation is based on a Real-time Streaming Protocol (RTSP) [97] server, hosting a prerecorded video file. RTSP provides commands for the video sink to start and control the video transmission. It uses the Real-time Transport Protocol (RTP) [98] to actually send the encoded video stream over the network. RTP allows both UDP and TCP as underlying protocol and supports common video codecs like MPEG or H264. The video sink uses the RTSP `DESCRIBE` command to receive a current configuration for the requested video stream and can adjust its implementation to it. This enables the

emulation of different stream characteristics such as bit rate, codecs, or communication protocols by simply changing the configuration of the RTSP server.

Implementation The RTSP server is implemented in C and based on the GStreamer library [42] and the associated RTSP plugin [43]. Input files are read using a descriptive GStreamer pipeline that can be adjusted by the developer (**R2**). Figure 3.3 shows the standard pipeline configuration for de-multiplexing an H264 encoded video file, queueing the individual frames, and encoding them into a RTP stream with custom payload type. Any auxiliary processing steps or video codec changes are supported by changing the dynamic pipeline configuration[1].

```
filesrc location=\"%s\" ! qtdemux name=demux demux.video_0! queue !
rtph264pay pt=96 name=pay0
```

Figure 3.3: GStreamer Pipeline for H264 video

Dataset The video source uses the WILDTRACK multi-camera dataset [17] as pre-recorded video input. It features seven distinct, overlapping perspectives in front of the main building of ETH Zurich. The dataset was chosen for its object quantity, the versatile camera arrangement, and potential for future benchmark workloads (**R1** and **R2**). Figure 3.4a shows the camera arrangement and visualizes their overlapping areas. The videos are of high resolution (1920x1080) and high framerate (60 fps) and were recorded in good weather conditions. Data points for the object location are provided both per camera as bounding boxes and as a location in the three-dimensional space for the overlapping area. This makes a versatile dataset that can be used for two distinct problems. The individual camera perspectives can be used to perform traditional multiple object tracking, while their complete set serves as input for multi-camera detectors based on the location. Figure 3.4b shows some screenshots of the dataset.

3.4 Video Aggregation Service

The video aggregation service connects to multiple video sources and extracts static frames from video streams. In the next step, the frames are resized to a configurable format needed for the CNN of the later detection stage. Such pre-processing is envisioned for edge applications, where services near the data source reduce the transmission size by performing already performing some calculations (**R1**). The service

[1] https://gstreamer.freedesktop.org/documentation/tutorials/basic/dynamic-pipelines.html

(a) Camera Arrangement

(b) Dataset Screenshots

Figure 3.4: WILDTRACK Dataset, taken from [17]

attaches an incremental frame number and the current timestamp as metadata before forwarding the frames using the real-time scheme explained in the next paragraph.

Real-time emulation The pipeline concept requires fast and reliable processing of the video stream. Decisions for video surveillance or autonomous driving systems need up-to-date locations of their surrounding. Current MOT benchmark approaches process the video input in batch format and measure the throughput in from of Frames Per Second (FPS) [25]. The tracking accuracy only depends on the underlying algorithm and does not fully reflect the real-time requirements of actual applications. We propose a real-time inspired emulation scheme that processes the video frames at a fixed framerate. Limited buffering is applied to handle small variances in processing speed and smoothen the pipeline sequence. If the static framerate exceeds the pipeline capacity, the buffer will fill up, and older frames must be discarded and make room for newer ones. While this affects the quality of service, it allows the pipeline to retain a reduced form of operation. This is especially useful for constrained edge devices with low processing capabilities. Even if the available resources do not suffice for full operation, the pipeline can still operate and generate benchmark results (**R1** and **R5**).

Figure 3.5 visualizes the current implementation of this concept. Two separate worker threads emulate the real-time behavior for each video source. The first thread connects to the video source and reads arriving extracts arriving frames from the stream. It performs the necessary pre-processing and adds them to a shared FIFO queue. The second thread reads frames from the queue at the specified framerate and generates a new send task for it. It chooses the following service based on the frame number and the specified detection frequency. The thread tracks the progress of the previous task and only starts the new transmission if it is finished. Otherwise, the new task is skipped, and the next frame is read from the queue. This approach ensures that frames are sent in order which makes them easier to handle in the later stages. The successful frames are an ordered subset of all frames that were received from the video source.

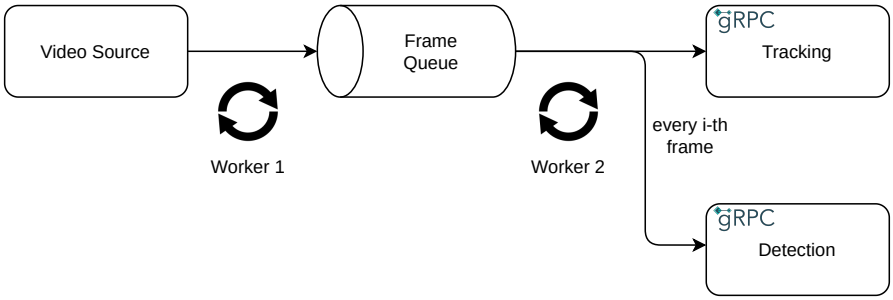


Figure 3.5: Real Time Emulation of the Video Aggregation Service

Implementation and Metrics The implementation uses Python and the OpenCV library to read video inputs and perform the frame resizing. Custom binaries of OpenCV integrate FFMPEG and OpenCL support to support common video stream formats and allow for GPU accelerated pre-processing of frames (**R5**). The service measures the overall processing time for resizing and the number of skipped frames as metrics for benchmarking suite. Early implementations measure network latencies for gRPC calls, yet the results differed only marginally from the other services processing time due to the blocking API approach.

3.5 Object Detection Service

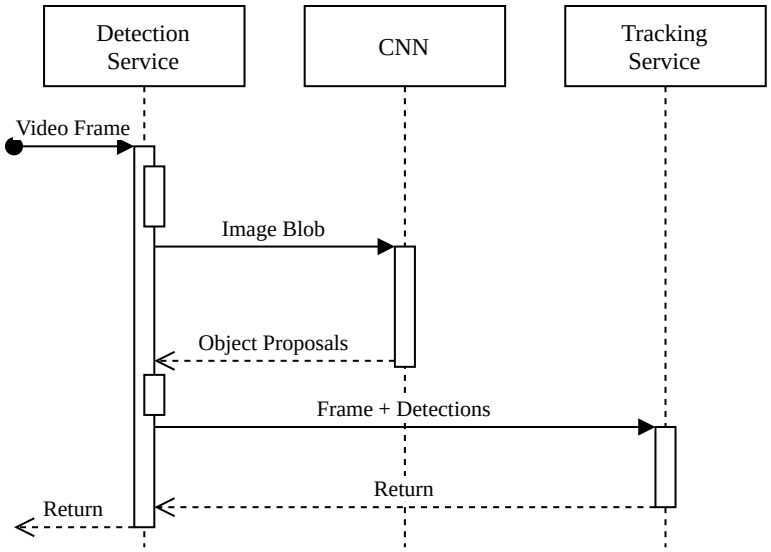


Figure 3.6: Object Detection Service

The object detection receives the resized video frames as input from the aggregation service. In the first step, the service ensures the correct image size and generates an image blob by performing mean subtraction, further image scaling, and channel swapping specific to the used CNN. The next step inputs the image blob into a pre-trained object detection CNN and calculates a list of potential detections. The last step uses non-maximum suppression to determine the best subset of proposals and filter any overlapping bounding boxes. This also rejects weak detections by removing any detections below the specified confidence threshold. The generated detections, inclusive object class, and detection confidence, are then attached to the metadata of the video frame and forwarded to the tracking service.

Implementation and Metrics As well as the aggregation service, the implementation uses Python and the OpenCV library. More specifically, it uses the `dnn` module to support different CNNs in common formats, including Caffe, DarkNet, PyTorch, and TensorFlow. Integrating novel CNN architectures requires only minor changes in the blob generation and the output handling, which results in an easily extensible service (**R2**). As of now, only the DarkNet based YOLOv3 network (see subsection 2.6.1) has been integrated. It was chosen for two major reasons:

- The CNN achieves over 45 fps when deployed on a single Nvidia Titan GPU, making it fast enough for real-time detection [84, 85].
- There are different variants of the network that share similar input and outputs but range in computational complexity. Benchmarking with these different versions allows for insights into the influence of the performance and accuracy of the detection stage.

OpenCV can utilize several accelerators for its `dnn` module, including OpenCL and CUDA for GPUs and other edge accelerators such as the Intel Movidius Myriad. The backend for the neural net execution can be changed during runtime based on the available hardware capabilities. The service measures the combined processing time for blob generation, CNN calculation, and non-maximum suppression and counts the number of detected objects before sending them to the tracking.

3.6 Object Tracking Service

As shown in the architectural overview in figure 3.1 the tracking service receives inputs from both the aggregation and the tracking service. Both inputs include the current input frame, the frame number, and the frame timestamp. Additionally, frames from the detection include bounding boxes of newly detected objects used to initialize the tracking algorithm. This results in dedicated execution paths for both input types:

- **Detections:** When receiving new objects from the detection, the tracking must perform two consecutive steps. First, it compares the set of new objects against the set of tracked objects to find conflicting hypotheses present in both sets. The tracking service then performs non-maximum suppression to choose the best candidate for each of these conflicts. This ensures that tracked objects are only overridden with a new location when the confidence of the detection algorithm surpasses the tracking confidence. New tracker instances are generated for the remaining new objects, and the remaining tracked objects are updated based on the new frame. After the instances were generated or updated, the union of both sets corresponds to the internal in the tracker.
- **Aggregation:** For frames from the aggregation no filtering or duplicate detection is needed. The new frame is passed through all tracker instances and used to estimate the new location of the object. Each instance returns the estimation confidence as well as a boolean value that tells if the tracking algorithm can identify the object in the present frame.

Object Disassociation These returns help identify objects that are currently not visible in the frame and must be removed from the set of tracked objects. On the other hand, objects may only leave the frame or get occluded for a limited time period and reappear after a reasonably short time. To balance the time between object disappearance and their disassociation from the internal introduces the possibility of missing objects when they return quickly. The tracking service counts the number of frames an object is not tracked successfully. Only if this count passes a specified threshold, the object is de-registered from the tracker and lost for good.

Implementation Similar to the previous two services, the object tracking service is based on Python and the OpenCV library. OpenCV provides performant implementations of common tracking algorithms such as KCF or MedianFlow. The different approaches can be switched using a simple configuration option which makes the service more flexible for different scenarios.

The tracking service uses the KCF algorithm by default. It provides comparable accuracy paired with fast processing speeds, which is well-suited for edge devices. The service measures the processing time for each frame and the number of tracked objects in its internal state. The object estimations (bounding boxes) get forwarded to the benchmarking suite as separate metrics primarily intended for QoS calculations.

3.7 Workload Packaging

The heterogeneous requirements of the edge complicate the building and packaging of the pipeline services. Devices require distinct instruction set architectures and explicit support for additional capabilities like SIMD vector extensions or hardware accelerators. The generated packages must reflect these characteristics and utilize each device to its full potential (**R4**). They must include custom compilations of the OpenCV library where non-free, accelerator-specific code is integrated.

Edge deployments often provide multi-tenant environments that require strong isolation between services. For this, the workload focuses on the use of docker containers [28] based on pre-generated images. Other techniques, like VMs, unikernels, or mixed approaches are generally supported by the pipeline architecture (**R6**) however, their integration is out-of-scope for this thesis and left for future iterations. Docker containers are chosen for their small overhead, making them a good fit for constrained devices, as well as the existence of good tooling to generate pre-compiled images for the services. They run directly on the host OS only isolated using advanced (Linux) kernel features like `cgroups` [73].

Multi-arch Images Such docker images bundle needed libraries and service binaries or scripts needed for their execution. They are based on Dockerfiles that provide a descriptive format to control the build procedure. The image structure is based on build layers, where each new layer corresponds to some command in the Dockerfile. They are coupled to the specifics of their build environment, which is described in the image manifest. It includes the operating system, the processor architecture, and a list of image layers.

Executing images built for other processor architectures is generally not possible, which mandates different images for each architecture. Alternatively, one could use the experimental support of cross execution through the QEMU hypervisor but incur a huge performance penalty on the performance.

The better approach is multi-arch images [29], where multiple operating systems and processor architectures are grouped under one shared image tag. The manifests of the individual images are stitched together and form a combined image descriptor. During execution, the docker runtime chooses the correct image, transparently simplifying the deployment procedure of the pipeline (**R4**).

Multi-stage Builds Another problem of the image generation is bulk of development dependencies required for the custom compilation of OpenCV. Those dependencies are not required when the images are executed and should be deleted after the compilation finishes. However, through docker's layered image structure, they still remain in the image layers. Such large image sizes are suboptimal for edge devices with little storage space, so the dependencies must not be included in the resulting image.

The build process utilizes multi-stage builds to overcome this problem. Development dependencies are installed in a separate container, where the needed libraries are compiled. After compilation has finished, solely the build artifacts are copied to the real image. While OpenCV still requires its runtime dependencies, this reduces the image size significantly.

Base Containers While multi-arch images handle different processor architecture, they cannot integrate transparent support for different hardware accelerators. Generating "super" images that integrate multiple hardware platforms is possible but causes larger images even for lightweight devices without specialized hardware. Multiple accelerators in a device may result in obscure behavior, where the developer cannot determine which one is actually utilized. An explicit docker image for each supported accelerator platform provides such insights and enables smaller docker images. Images are tagged according to their capabilities, so the developer can easily spot the utilized accelerator. The compilation is handled automatically using a CI/CD pipeline and

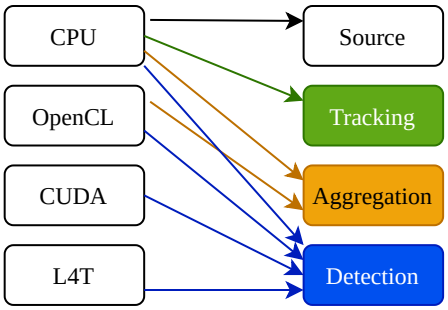


Figure 3.7: Docker Images Technology Support

results in the following four base images:

- **CPU-only** (Tag: `cpu`): Minimal compilation of OpenCV that only supports CPU execution. Architecture-specific capabilities (ARM NEON or x86 AVX) are enabled, while all other accelerators are deliberately disabled.
- **OpenCL** (Tag: `opencl`): Enables the usage of OpenCL using the transparent API (T-API) of OpenCV. Support for Intel, Nvidia, and AMD GPUs is included, though each platform requires slightly different approaches for running the docker container.
- **Nvidia CUDA** (Tag: `cuda`): Integrates Nvidia CUDA and CuDNN support which promises faster execution than OpenCL. The image utilizes Nvidia's container runtime `nvidia-docker2` which handles necessary device bindings for the GPU.
- **Nvidia L4T** (Tag: `l4t`): A more lightweight integration of CUDA and CuDNN targeted for Nvidia Jetson devices. The image only includes library shims that are replaced with the device libraries upon execution.

Workload Images On top of the base images, the actual workload images are generated. Each service supports different technologies, which is shown in figure 3.7. For completeness, the video source service is included in the figure, even though its image is based on a separate multi-stage image specific to GStreamer and does not support any execution environments other than the CPU.

4 Benchmarking Suite

This chapter details the overall benchmarking suite built around the workload described in the previous chapter. First, section 4.1 starts with a list of requirements specific for the tooling of the benchmarking suite. Section 4.2 describes the overall system architecture and details the flow for a full benchmark execution. The next section 4.3 describes the specifics of the orchestration service employed to execute the benchmark workload on different devices and in different configurations. In the last section 4.4, the separate metric and calculation system is outlined.

4.1 Requirements

The tooling around the proposed workload plays a vital role in the adaption of the whole benchmarking suite. Integrating different software platforms (**RG2**) as well as future adjustments for emerging edge technologies (**RG3**) heavily depends on the initial design decisions. Following list captures the requirements that are needed to achieve these goals:

R1 *Edge heterogeneity:* By nature, most edge environments are heterogenous with lots of specialized accelerators. Benchmark workloads should fully utilize the edge hardware by providing device-specific artifacts. The tooling must consider the specific requirements of the individual services when scheduling the benchmark. Additionally, the tooling should consider the different capacities of devices and prevent unreasonable configurations from being scheduled (e.g., running the full video analytics pipeline on one Raspberry PI).

R2 *Exhaustiveness:* The formation and capabilities of edge devices vary greatly, making benchmarking based on established or predetermined assumptions unfitting. Large scale edge deployments must constantly balance computing and networking costs and may result in unintuitive benchmark results (e.g., heavy machine-learning on a near constrained device could be faster than in the far cloud with high networking costs). The benchmarking suite must evaluate all reasonable configurations and compare them against each to discover such anomalies.

R3 *Metric integration:* To evaluate the performance of the workload, the benchmarking suite needs to collect and aggregate metrics during the execution. This collection must happen as resource-saving as possible and may not impact the overall workload performance significantly. Processing and aggregation need to be as decoupled as possible and should happen outside the benchmarked environment. Diverse networking restrictions imposed by different edge deployments (e.g., behind NATs or firewalls) should be supported.

R4 *Software platform integration:* As identified in section 2.5.3 current benchmarking approaches lack support for emerging software platforms employed at the edge. The benchmarking suite should support different deployment techniques or service models present in these platforms as workloads may target multiple platforms.

R5 *Extensibility:* The use cases of edge computing and the capabilities of edge hardware evolve rapidly. The benchmarking suite must support the integration of new workloads or software platforms in its design. Different metrics should be covered by an expressive configuration where the integration of novel edge platforms should only require minimal code changes.

R6 *Ease of use:* The benchmarking suite should use articulate configuration structures and well-defined procedures to simplify integration for inexperienced users. Interactive tooling helps with configuration generation and further reduces the entry threshold. While all raw data must be recorded and made available, the benchmarking suite should calculate easy-to-understand figures and aggregations to provide a quick overview of the results.

4.2 System Design

This section describes the general vision behind the benchmarking suite, gives an overview of the overall architecture, and shows the interaction between the individual components. Figure 4.1 represents the architectural overview and the communication paths between different parts of the system.

Benchmark Vision: Benchmark environments for edge platforms consist of a diverse mixture of devices. Developers may combine emerging hardware with existing devices to allow for comparison between them. The benchmark workload described in chapter 3 serves as the initial benchmark workload and points further research in the correct direction. Custom workloads can be integrated to fathom aspects not targeted by the

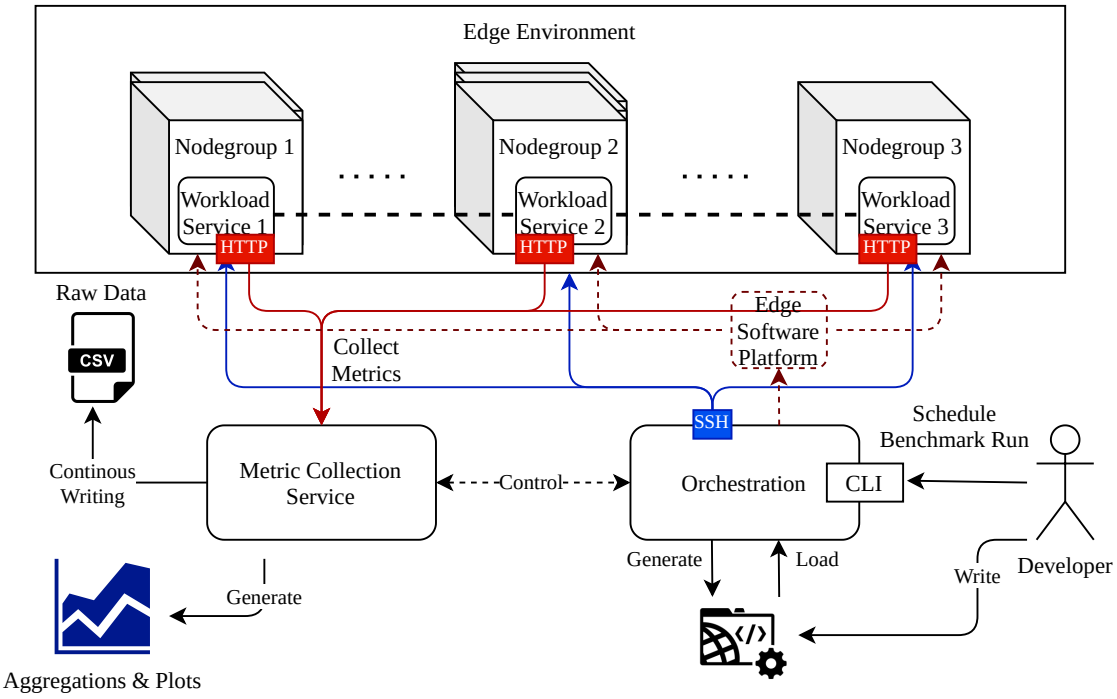


Figure 4.1: Benchmarking Suite Overview

video pipeline. The individual workload services produce custom metrics that get processed and visualized by the benchmark suite. Metric types are configurable and allow the developer to integrate custom post-processing procedures specific to the benchmark workload.

Architecture Description The devices in the benchmark environment have different characteristics that influence the later scheduling. Multiple devices of similar type (e.g., several Raspberry PIs) get grouped into node groups that share common properties like ISA, computing power, or RAM capacity. Currently, these node groups are based on the configuration structure described in subsection 4.3.1. Future iterations of the benchmark suite will include an interactive CLI that helps the developer by probing the provided devices and generating the node configuration automatically. However, for complex environments where this interactive process is insufficient, the developer may revert to editing the configuration file directly.

The configuration includes details about the benchmark workload and generates multiple runs that cover all meaningful deployment scenarios (**R1**, **R2**). Different scenarios are scheduled automatically and generate metrics specific to the configured

benchmark workload. An external metric collection service collects them using simple HTTP requests and logs the raw data for later analysis. Upon completion of a scenario run, the evaluation calculates aggregations and generates plots of the raw data. They provide a condensed view of the benchmark run and help the developer explore the raw data.

The benchmark suite provides a built-in executor that directly connects to the devices and schedules the services using an SSH connection. Preparations for other edge software platforms such as EdgeIO or k3s allow for easy integration of them in the future (**R4**). The separate metric collection enables more diverse network configurations where the benchmark workload cannot reach the developer but works well with an interposed machine.

4.2.1 Benchmark Sequence

The general procedure of scheduling a new benchmark run is shown in figure 4.2. In the first step, the developer generates the benchmark configuration by manually editing the configuration file or using the future interactive CLI (**R6**). Using the environment preparation described in subsection 4.3.2 he may verify and set up the devices to prevent any errors in the actual benchmark run. After the initialization finishes, the actual benchmark procedure is started. The orchestration service generates several possible deployments (mapping of workload services to node groups) and schedules sequential runs of each configuration.

For each run, the orchestration controls the separate metric calculation service using the procedure described in subsection 4.4.3. The benchmark workload produces custom metrics that are logged by metric calculation and aggregated upon completion (**R3**). After each run of a workload mapping, the orchestration demolishes all workload components and restarts the benchmark procedure from the beginning. When all possible mappings are benchmarked, the service initiates the plotting procedure at

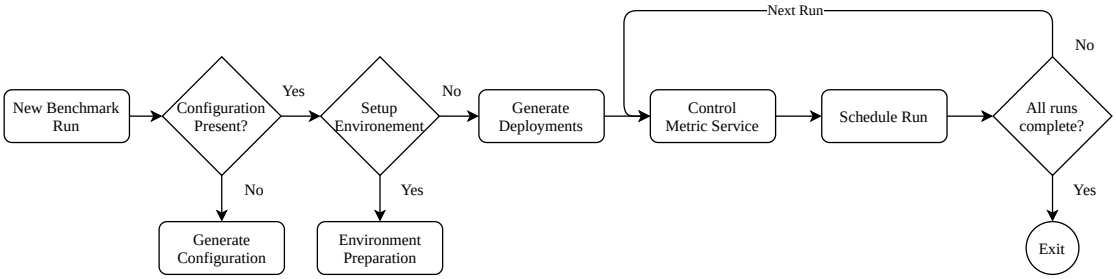


Figure 4.2: Benchmarking Flow Chart

the metric service and exits the application. This eases the developer from much of the manual work usually involved in comprehensive benchmarking of computing environments (**R6**).

4.3 Orchestration

This section describes the benchmark orchestration in more detail. It gives an overview of the orchestration structure and shows the adaption potential for the developer. Subsection 4.3.2 will describe the separate environment preparation, while subsection 4.3.3 describes the generation of the possible deployments. The last subsection details the built-in SSH executor and describes the concept for the integration of Kubernetes-based platforms such as k3s.

4.3.1 Configuration Structure

Figure 4.3 shows the relevant parts of the configuration structure in a descriptive format. The configuration can be divided into three distinct areas, each describing a different part of the benchmark:

- **Software Platform Configuration:** The different platforms integrated into the benchmark suite require different configuration options. While the built-in ssh executor may only require the username and the path to a key file, other platforms may require route configuration, deployment template, or secret files to authenticate against their APIs. Their structure cannot be generalized, so each one provides its own model (e.g., SSH config in figure 4.5).
- **Environment Configuration:** Though edge environments are very heterogenous, most benchmark environments typically include multiple devices of the same or similar type (e.g., multiple Raspberry PIs). Scheduling the same workload service on each of them is not necessary, as it would only increase the benchmark runtime without providing any significant new insight. Thus, these devices are grouped into `NodeGroups` with similar characteristics (processor architecture, capabilities, service capacity). However, the developer may configure multiple groups for similar nodes that differ in other, not listed regards (e.g., different network connection).
- **Workload Configuration:** The workload configuration (`Workload`) lists the specific configuration for each workload service. Each item includes the used docker image and a list of available image tags. The tags correspond to the specialized images that support certain hardware capabilities (e.g., `cuda` or `opencl` for the

object detection service) so the orchestration knows which services support which accelerators (`Tags`). For the same reason, a list of supported processor architectures is included however, this information is solely used for the workload matching since the benchmark workload is expected to support multi-arch images similar to the proposed workload (`Arch`). Each service supports data mounts for large files (e.g. the dataset video) provided outside of the container (`Mounts` and `LocalData`). Standard docker port forwarding (`Ports`) and a custom container entry point with support for variable templates (`Command`) can be provided as well.

The address of the evaluation service is included as a separate option since it cannot be assigned to any of the described configuration settings.

```
Executor-Config: ...
Evaluation: evaluation-address
NodeGroups:
  - Name: human-readable-name
    Arch: ISA
    Capabilities: [list-of-capabilities]
    Nodes: [list-of-node-addresses]
    NodeCapacity: capacitiy-per-node
Workload:
  - Name: human-readable-name
    Image: docker-image-name
    Ports:
      - port-binding
    Tags: [image-tags]
    Arch: [image-architectures]
    Command: executable --args {{Evaluation}}
    Mounts:
      - tmpfile:containerpath
    LocalData:
      - file-to-distribute
```

Figure 4.3: YAML based configuration file for orchestration

4.3.2 Environment Preparation

A fresh benchmark environment typically consists of several devices with varying preconditions. Some devices may have a fresh OS installation, while others use an older, already configured one. While this heterogeneity is representative of edge scenarios, the benchmarking suite needs to establish some common ground on all devices. In a manual, upstream step, the orchestration ensures the configured requirements are met and hints the developer towards missing packages or execution environments. Alternatively, a basic preparation strategy is implemented that installs missing packages and starts needed services. However, the implementation is still early and cannot handle complex scenarios like missing root permissions or different package managers.

Future iterations of the benchmark suite will utilize a more established approach (such as ansible or terraform) to set up the environment. This helps with the integration of other software platforms as well, where automated initialization procedures are often provided by the vendor or community. The orchestration only needs to verify its access to the required services or APIs without any custom initialization procedure.

The separate initialization stage also helps populate the benchmark configuration described in the previous subsection. In the current implementation, the orchestration only probes for execution environment supports and network access, yet an interactive CLI will aid the developer in the future. Based on a simple list of devices, the pipeline probes for node characteristics and tries to autogenerate node groups. Only if this process fails or does not detect certain characteristics, the developer must manually edit the configuration and adjust it to his needs.

4.3.3 Workload Matching

Generating an exhaustive list of mappings that match workload services to node groups is an essential part of the orchestration service (**R2**). This list of mappings must adhere to both processor architecture and node capability restrictions, so only workloads that can be executed are matched.

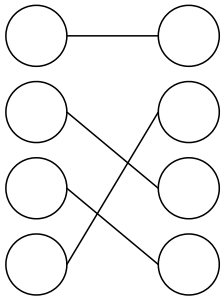
The orchestration service interprets these restrictions as edges in a bipartite graph of workloads and node groups. Workloads are connected to all node groups that support at least one of their characteristics (tags). In this graph, valid mappings correspond to a perfect (maximal) matching in the graph as depicted in figure 4.4a.

There are established algorithms such as Hopkcroft-Karp [50] and Kuhn-Munkres [60] that allow finding one perfect or maximal matching. However, generating all possible is a much more complex problem. Efficient approaches (e.g., [34]) start from one perfect matching, generated with the above algorithms, and iteratively generate all possible matchings through permutations in the graph. The computational complexity for the

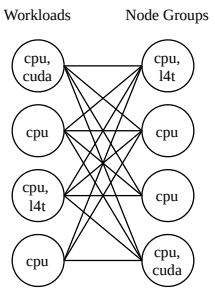
generation of perfect matchings depends on the total number of perfect matchings $O(c)$ that can be found in the graph. For fully connected graphs, this number converges towards $O(n!)$ (with n being the number of vertices on one side of the graph), so the optimization of these approaches becomes less impactful. In typical configurations, workloads and node groups share a CPU-only execution baseline that results in (almost) fully connected graphs resulting in the mentioned scenario (compare figure 4.4b). Additionally, no tested implementation of such algorithms is available for the orchestrations technology stack, resulting in significant implementation and verification effort. A much simpler, brute-force recursion achieves similar but static runtime complexity of $O(m!)$. In common use cases, the extra effort for the efficient approaches results in no real performance gain, so the orchestration uses the simpler recursion to generate all perfect matchings. Even if the graph is more sparsely populated, the small number of vertices (workloads or node groups) can be handled by modern hardware in an acceptable time. Another advantage of this approach is the ability to handle the unequal number of vertices for both sides of the graph, which would not be possible with the efficient approach. After all perfect matchings are generated, a second filtering step reduces the potential number of schedules by pruning them based on the actual number of nodes and their configured service capacity. This results in only valid matchings being output by the procedure.

4.3.4 Executors

The orchestration supports different software platforms through separate executors. They bundle the platform-specific initialization procedure and the functionality for connection and workload scheduling into a common interface. The benchmark suite forwards the generated schedules and hands over the responsibility to the designated executor. The thesis includes a full implementation of a native, ssh-based executor



(a) Perfect Matching



(b) Typical Configuration

Figure 4.4: Workload Matching as Graph Problem

as well as preparations for the future integration of Kubernetes-based platforms such as k3s or microk8s. This subsection describes the native executor and explains the Kubernetes concept, which can be adapted for other platforms (**R4**).

Native Executor (SSH) The native executor makes minimal assumptions about the edge hardware. It only requires the specific execution runtime and some type of network connection between the devices. The generated performance profile resembles the potential of the hardware and can serve as the upper standard for comparison with other platforms. The executor requires little additional configuration, as shown in figure 4.5 and is easily integrated into all sorts of environments. The current version of the SSH executor focuses on Docker containers as the only fully supported execution platform. However, small adjustments allow for the execution of native or unikernel workloads (**R4**).

```
User: username
KeyFile: keyfile-path
Commands:
  - command-template: "docker run {{.Image}}:{{.Tag}} {{.Command}}"
RequiredPackages: docker nvidia-docker2 drivers
Runtime: docker
```

Figure 4.5: SSH Executor Configuration

The initialization procedure can be split into four separate steps. First, it ensures network and ssh access to each device and checks for superuser privileges. Other nodes are pinged to ensure networking between them is set up correctly and works for the later benchmark workload. The device environment is checked for required packages with missing ones being installed. Lastly, the initialization verifies the successful execution of commands for the specified runtime (for example, by running the `hello-world` docker container). The remaining part of the executor can be divided into two submodules:

- **SSH-Client:** The module connects to the device by standard ssh connections. Their functionality is bundled into a separate module that handles authentication (via private key file or password prompt) and remote executions. To simplify device access, it expects all devices to share common login details. Remote commands can either be issued synchronously for setup tasks that require direct feedback or asynchronously with the continuous command output being logged. The client integrates the SCP protocol to copy local files to the remote device for later usage (volume binding) in the benchmark workload.

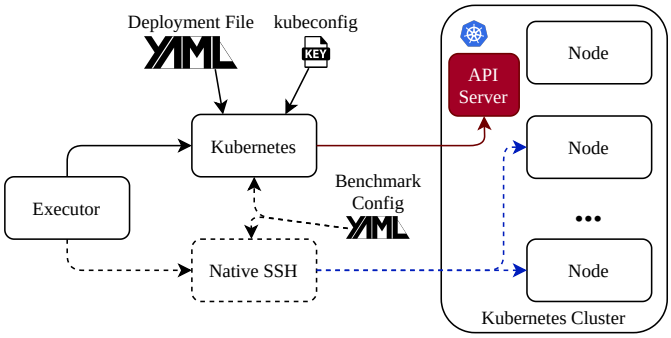


Figure 4.6: K8s Executor Concept

- **Benchmark Executor:** The actual executor receives valid schedules from the matching module and generates a list of actual benchmark runs. Each run is prepared by pre-pulling docker images, copying needed files, and generating the startup command for each workload service. The command templates are based on data-driven text templates configured by the developer and get populated using the run configuration. When at least one of the workload services returns in an orderly fashion (process exits with exit code 0) the executor ends the current benchmark run by stopping and removing the docker containers and initiating the aggregation and plot generation at the evaluation.

Kubernetes and other platforms The integration of other platforms uses their standard API-server to deploy the workload services. Environment setup is entirely managed by the platform operator, and the initialization is limited to simple API access checks. Worker nodes are labeled based on their capabilities to allow the scheduling component to execute the workload on fitting nodes. Different node accelerators are supported through custom device plugins such as the Nvidia or AMD implementation provided for their GPUs [100]. Figure 4.6 shows the envisioned extension architecture for the Kubernetes integration. It uses an existing kubeconfig file to access the Kubernetes API-server. The benchmark workload is deployed into a dedicated namespace and uses a deployment file with similar text templates as in the commands in figure 4.5. Other than for the native executor, the service communication utilizes standard Kubernetes networking components. Other platforms may adopt a similar scheme, where deployments use standard configuration files with replacements of the relevant container configuration.

4.4 Metric Collection

The metric collection is a customizable service that receives metrics from the benchmark workload and generates flexible aggregations and plots for them. As a separate executable, it can be freely placed and overcome possible performance penalties or network restrictions for the benchmark workload (**R3**). This section describes the different metric options, the concept for the fast metric collection, and the integration of the aggregation and plots.

4.4.1 Metric Calculation

The metric system expects the provided workload to provide custom metrics that it pushes to the metric service. Different workloads require flexible handlers so developers can integrate novel workloads (**R5**). The metric calculation provides configurable HTTP endpoints that utilize provided metric modules. Developers can edit or add endpoints based on the configuration file described in figure 4.7.

```
Name: service-name
Url: /http-endpoint
Module: metric-module
Fields: [list-of-metric-fields]
CSV:
  OutputFile: csv-output-file
```

Figure 4.7: Metric Configuration

Each metric endpoint takes a plain JSON input that includes the described `Fields` as keys. The input is further processed by the specific `Module` and written to the defined csv `OutputFile`. For the thesis, three metric modules have been implemented that cover the specific metrics of the video analytics pipeline:

- **Generic:** The generic module takes the JSON input and extracts the metric fields from it. Values get converted to floating-point and forwarded to the output routine.
- **Motion Object Tracking (MOT):** The MOT module receives the current frame number and an array of tracking hypotheses as bounding boxes. As the first step, it loads the ground truth annotations for the WILDTRACK dataset based on the frame number. Next, the module calculates `misses` (*m*), `mismatches` (*mme*) and

`false positives` (fp) by associating the hypotheses to the ground truth based on intersection over union (IoU) as distance metric (d). When all hypotheses have been processed, the module saves these values and forwards them file output. The $MOTA = 1 - \frac{\sum_t (m_t + fp_t + mme_t)}{\sum_t g_t}$ and $MOTP = \frac{\sum_{i,t} d_{i,t}}{\sum_t c_t}$ metrics are only calculated in the later aggregation step based on the total number of object g and the number of valid mappings (c). This subsequent calculation is proposed by the original authors and ensures more intuitive results of the pipeline [12]. The current implementation is tightly coupled to the used dataset and the pipeline implementation, so integrating other tracking approaches requires some adjustments. The WILDTRACK dataset only provides annotations for every fifth frame, so the module skips other inputs.

- **Script:** The script module allows developers to provide a custom metric script based on the tengo language [23]. Its intention is for developers to integrate their own calculation procedures without actually having to dig into the metric system code. The module loads the script from a configurable file path and creates a new script instance for it. It binds the previously decoded JSON input to the variable named `input`. The script gets executed, and the module reads its results from a variable name `output`. This output is handled similarly to the generic module and is forwarded to the output routine.

4.4.2 Calculation Decoupling

Executing the metric collection as a separate service that receives metrics in a push fashion allows it to overcome challenging network requirements (e.g., deployment behind NATs or firewalls) (**R3**). However, repeatedly calling the metric system can result in additional overhead for the benchmark workload. Clever placement of the metric service may minimize latencies and mitigate this problem yet, costly operations in the metric modules still pose a problem. To minimize this performance impact of metric pushes, the metric service decouples costly I/O operations and calculations from the actual HTTP handler. These handlers only parse the message body, add it to a queue (go channel) and return as fast as possible. A separate thread (goroutine) performs the actual metric processing defined by the metric module. It stores the intermediate results and forwards them to file output. This decoupling approach enables the benchmark suite to reduce its effect on the workload performance (**R3**).

4.4.3 Metric Aggregation and File Rotation

The described procedure results in a continuous stream of metrics that get output by the configured metric modules. This raw data is persisted in a machine-readable format

(CSV), so the developer may perform in-depth analysis on the benchmark runs (**R6**). Yet, for an easier structure, this stream needs to be broken into several independent output files for each benchmark run. Thus, the metric system logs the raw data and generates aggregated results in a separate subfolder for each run. After a complete benchmark has finished, the metric collection generates data plots that guide the developer for further evaluation of the raw data. The metric system implements a set of control routes used by the orchestration to control the current status of the service.

Figure 4.8 shows this interaction between the two services. When starting a new benchmark, the orchestration notifies the metric system, which in turn creates a new output folder based on the current timestamp. A second call follows for each actual benchmark run, which results in the creation of a new subfolder for the run and the reset of any state from previous runs. Upon receiving the next call for the end of the current run, the aggregation module generates a condensed `json` output that includes configured calculations of the raw metrics. Currently the service supports average (`AVG`), maximum (`MAX`), minimum (`MIN`) as well as singular values (`FIRST`, `LAST`) as aggregation outputs. This execution cycle continues until all generated benchmark runs have finished. Then the orchestration ends the current benchmark and triggers the integrated plotting script that generates individual graphs per run as well as aggregated plots where the best performing run is highlighted.

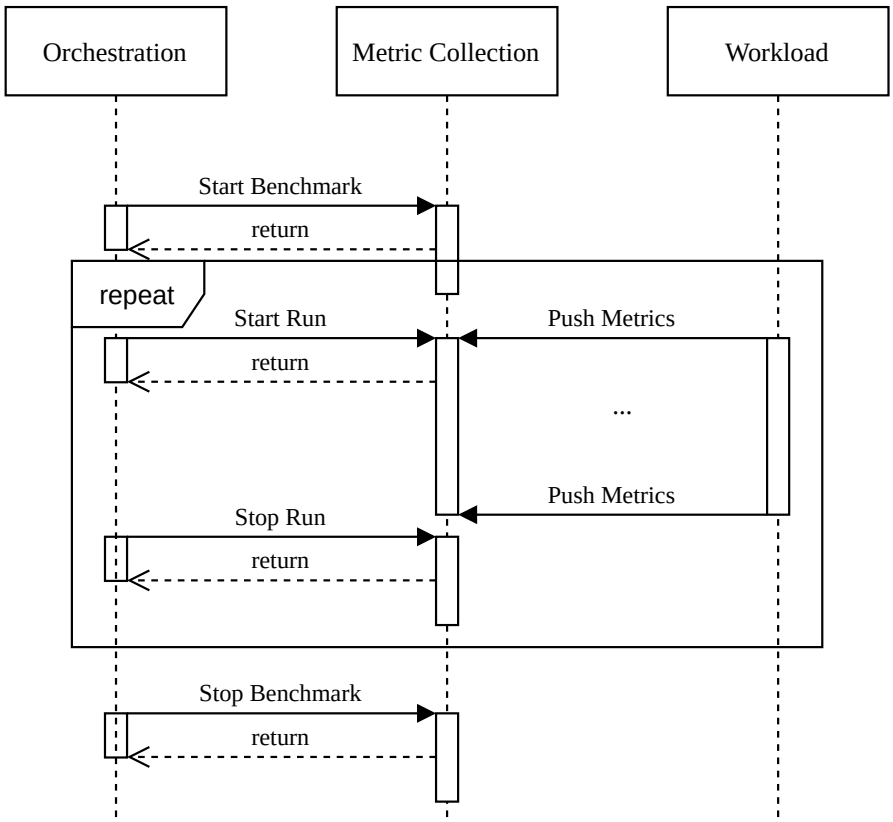


Figure 4.8: Sequence diagram for metric processing

5 Evaluation

This chapter provides deeper insights into the benchmark performance in two different edge environments. The first section 5.1 describes the used hardware and the different capabilities they provided. Section 5.2 evaluates the matching procedure for both environments by comparing the number of mappings and their distribution. The next section 5.3 highlights problems with the initial implementation of our pipeline and proposes a solution used by the following sections. In section 5.4 we compare the individual performance of different devices when operating the pipeline services. Section 5.5 utilizes the different generated mappings and highlights interesting device combinations. The penultimate section 5.6 integrates the less complex TinyYOLO network into the detection service and compares its performance. The chapter finishes with section 5.7 where we evaluate the influence of wired and wireless networking on the pipeline's performance.

5.1 Experimental Setup

We evaluated the benchmark suite and the workload performance in two separate hardware environments. Benchmark runs on each environment were performed exclusively, so any reciprocal influence is avoided. Runs were repeated multiple times, at different times, and across multiple days to balance the influence of any external factors. This section describes both environments in more detail.

5.1.1 Hasso Plattner Institut Resources

The first testbed is a homogeneous VM infrastructure provided by the Hasso Plattner Institut (HPI)[1]. The utilized resources operate on data center hardware for both computation and networking. A total of 50 virtual machines was provided, comprised of the following types:

- *Small (S):* 1GB of RAM and 1 vCore (17 machines)
- *Medium (M):* 2GB of RAM and 2 vCores (17 machines)

[1] `https://hpi.de`

- *Large (L):* 4GB of RAM and 4 vCores (12 machines)
- *Extra-Large (XL):* 8GB of RAM and 8 vCores (3 machines)

The machines were based on Ubuntu 18.04 LTS and provided x86-based vCores without any additional accelerators. For the benchmarks, we utilized four small, four medium, four large, and one extra-large machine. However, during the actual benchmark, we had to remove the small machines from the configuration since they were unable to execute some of the workload services. This homogeneous environment serves as a good reference for the more diverse environments targeted by the benchmarking suite.

5.1.2 Local Edge Deployment

The second testbed was a local deployment of several heterogenous devices representative of edge environments (see figure 5.1). It consisted of the following six devices that each provided unique processing capabilities:

- Two Raspberry PI 4s with 4GB of RAM and an ARM Cortex A72 quad-core processor. The processor supports advanced SIMD instructions using ARM NEON.
- One Nvidia Jetson AGX Xavier Developer Board, featuring an octa-core ARMv8 processor, 32GB of RAM, and a 512 core Volta GPU. The GPU and additional machine learning and vision accelerators can be utilized using the Linux for Tegra (L4T) framework.
- One Fujitsu small form factor PC with a six-core Intel 8400T x86 Accelerated Processing Unit (APU) and 8GB of RAM. The integrated Intel UHD Graphics 630 allows for OpenCL acceleration.
- A workstation PC with 16GB RAM, a six-core Ryzen 2600x x86 processor, and an Nvidia RTX 2070 GPU. The GPU features both OpenCL and Nvidia CUDA support.

The devices were connected using standard 1Gbit ethernet connections using a TP-Link switch. A second benchmark configuration utilized the built-in WiFi support by the Raspberry PIs and the Fujitsu APU. We installed a fresh installation of Ubuntu Server 20.04 LTS on all devices, with only the Workstation using the Ubuntu 18.04 derivative ZorinOS 15. The environment depicts the heterogeneous device and networking approaches present in actual edge environments.

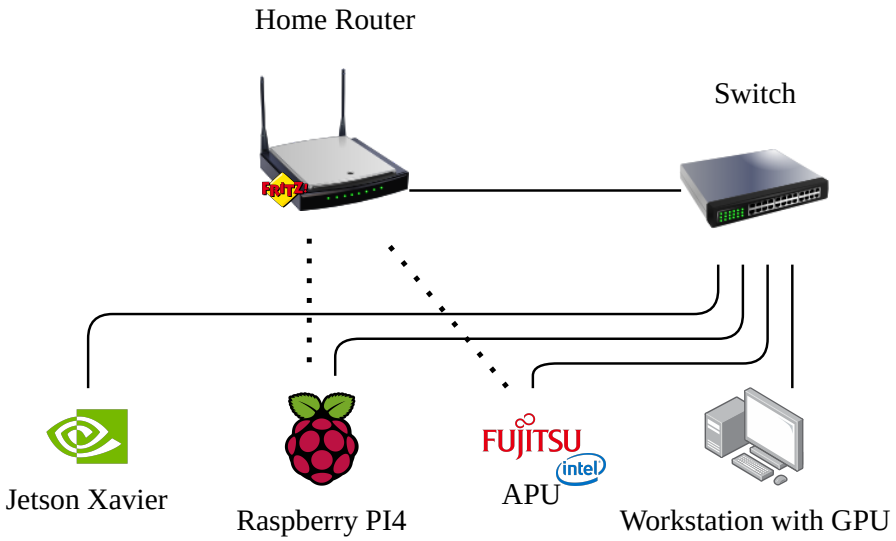
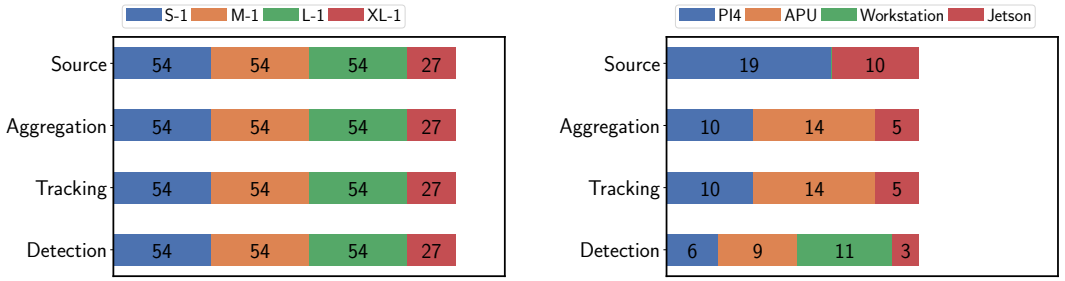


Figure 5.1: Edge Environment

5.2 Workload Generation

The benchmark suite includes a novel approach for matching workload services with node groups (see subsection 4.3.3) An exhaustive distribution of services influences the overall benchmark quality and allows the developer to find unexpected results in the scheduling. The generated mappings should be evenly distributed across the available nodes but must take the configured restrictions into account.

Figure 5.2 shows the generated workload distribution for both benchmark environ-



(a) HPI Resources

(b) Local Environment

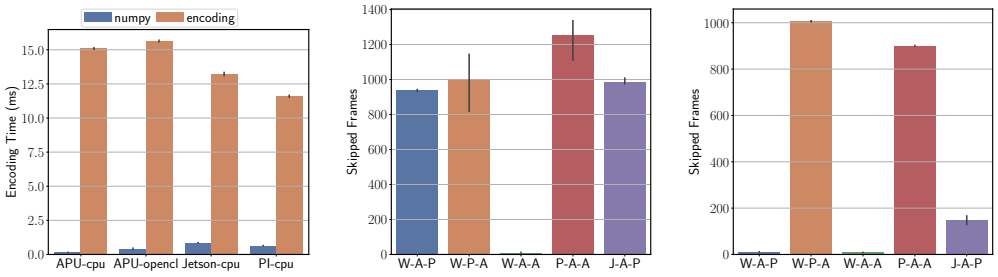
Figure 5.2: Workload Matchings

ments. The homogeneous HPI resources (similar capacity and number of nodes) result in an even distribution of workload service (see figure 5.2a). Only node groups that provide less capacity or nodes (as the XL machine) get a smaller share of workload services. It also shows the factorial growth of the number of mappings in environments that impose no restrictions on the workload scheduling. The local testbed shows a different picture in figure 5.2b. Services are placed according to their defined restrictions, which results in much less valid mappings. Utilizing the workstation only for the detection service highlights this filtering ability based on the capabilities. Processor architecture restrictions are shown by the source service, where only the Jetson and the Raspberry PIs are mapped. While the number of mappings decreases drastically, nodes with several supported technologies (e.g., APU with `cpu` and `opencl`) still result in multiple runs.

5.3 Data Compression

The initial implementation of the video pipeline used the numpy binary format (`.np`) to transfer the image between services. During the evaluation, this proved a rather poor design choice with adverse effects on the pipeline performance for distributed execution of the pipeline. Scheduling the aggregation service on the PIs or the Jetson caused a lot more skipped frames than expected (see figure 5.3b).

We improved our initial implementation by integrating an extra step that encodes the images in the JPEG image format before sending them over the network. While this introduces additional encoding overhead, as figure 5.3a shows, the new scheme improves the overall pipeline performance by quite a lot. The smaller gRPC requests allow the aggregation to operate more efficiently and support scenarios that were initially planned (see figure 5.3c). This deeper workload analysis helped us identify



(a) Encoding Time (b) Skipped Frames(Numpy) (c) Skipped Frames (JPEG)

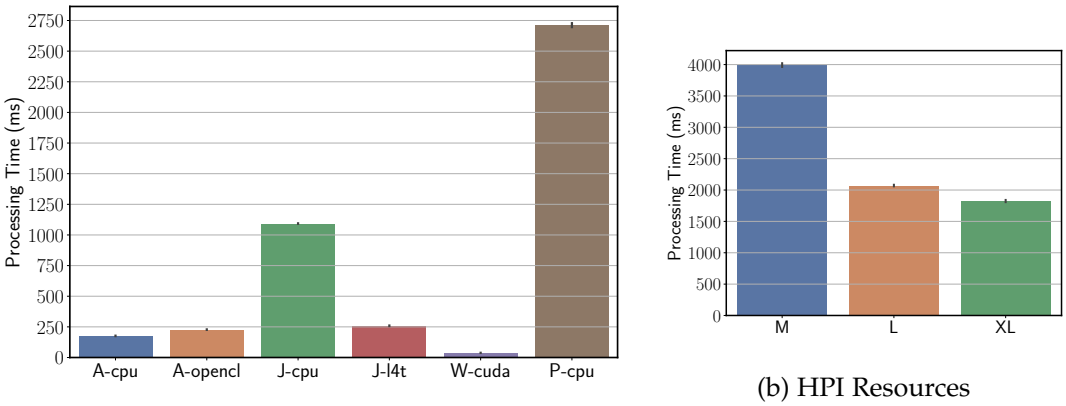
Figure 5.3: Improved Encoding Scheme (W: Workstation, A: APU, J: Jetson, P:PI)

further side effects of our current implementation. The synchronous execution of the pipeline steps requires the aggregation to wait for the potentially long execution time of the next services. Services that exceed the default timeout of two seconds result in request cancellation, which leads to unintended drops in the pipeline performance. To overcome this problem, we increased the gRPC timeout to ten seconds for our benchmark runs. We identified these problems late into the thesis and redid the evaluation only partly. However, the individual performance of the pipeline services is not affected by the new scheme, which allows us to use existing data for section 5.4 All other aspects of the evaluation, especially where the pipeline services interact over the network, are based on the new implementation.

5.4 Accelerator Performance

An essential aspect of the proposed benchmark workload is the utilization of different accelerator hardware. The workload services export the processing time needed to perform their subproblem. Figure 5.4 shows the performance of the detection service on different devices of the benchmark environment. Through the different docker containers described in section 3.7 all employed accelerators are utilized.

For the local environment (see figure 5.4a), the workstation GPU performs the fastest for the object detection with roughly 38ms. With a clear delimitation to the workstation, the APU's performance comes second, with more than four times the processing time (CPU: 176ms, iGPU: 227ms). Utilizing the Jetson's GPU acceleration performs even slower and takes 258ms per detection. The Jetson GPU and the iGPU of the APU both have significant spikes for the first frame passed through the neural net, reaching up to 20s for the Jetson. The ARM-based CPUs of the Jetson and the Raspberry PIs are far behind, with 1100ms and 2700ms average processing time. Scheduling the benchmark on the HPI resources paints a different picture for the execution times (see figure 5.4b). The provided VMs perform worse than similarly equipped machines in the local environment. The M resources take almost four seconds to process the input, while the L and XL machines take around two. With only 250ms, the speedup between the quad-core L machines and the octa-core XL machines is relatively minor.

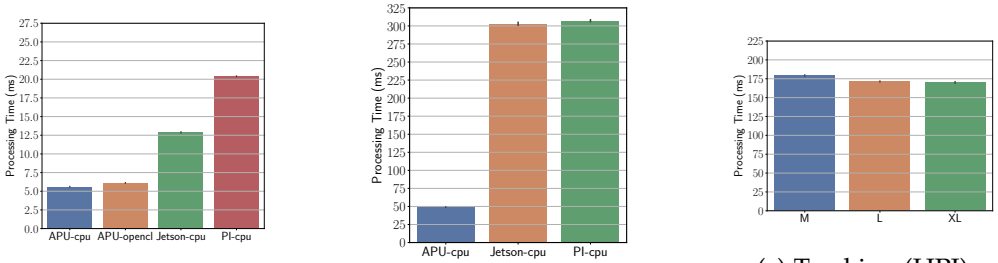


(a) Local Deployment

(b) HPI Resources

Figure 5.4: Individual Detection Performance

The aggregation and tracking service support fewer accelerators but show comparable device performances to the detection service. As seen in figure 5.5a, the Intel APU performs the fastest with 5.8ms and 6.1ms for the CPU and the OpenCL execution, respectively. The Jetson needs more than double the processing time with 12.9ms, and the PIs are further off with 20.4ms. Further investigation indicates that the aggregation service depends on single-core performance, with one thread performing most of the work. For the tracking service (see figure 5.5b), this difference between the devices becomes more apparent. While the APU still performs quite performant with 41ms tracking time, the Jetson and the PIs perform drastically slower with 220ms processing time. Similar to the aggregation service, the work is performed by only one worker thread which results in these drops for the less powerful ARM devices. Even the far slower x86-based HPI resources (see figure 5.5c) perform faster with only 175ms



(a) Aggregation (local)

(b) Tracking (local)

(c) Tracking (HPI)

Figure 5.5: Performance of other Services

processing time. The small speedup for different VM sizes reinforces our suspicions, with only one thread performing most of the work.

5.5 Pipeline performance in different scenarios

As described in section 5.2, the orchestration schedules an exhaustive list of benchmark runs combining different devices for the pipeline. The previous section shows their impact on the individual service processing time, while these scenarios show the impact of the different devices on the overall pipeline performance. Figure 5.6a[2] shows the ratio of misses for select local scenarios, while figure 5.6b shows the number of frames skipped by the real-time emulation at the aggregation. Due to problems with the dataset in combination with our object tracking implementation, we switched the envisioned MOTA and MOTP metrics for the pipeline miss ratio. The challenges and our solutions are discussed in more detail in section 6.1.

The maximum performance of the pipeline is achieved for mappings that utilize the workstation GPU or the Jetson accelerator for the detection stage. Both the tracking and the aggregation are scheduled on the APU which offers the best execution performance for them. The slower detection on the Jetson leads to circa 150 more skipped frames, which our approach is able to deflect in its architecture. They only result in minor variances of the pipeline performance, with the root causes explained in more detail in section 6.3. Other workload mappings result in a much worse pipeline performance where more frames get dropped by the real-time scheme. Scheduling one or more services on the Raspberry PIs has an adverse effect on the pipeline performance. The pipeline skips almost half of the frames in the dataset for these runs. For mappings that schedule the tracking stage on the Raspberry PIs, this behavior is expected given the service processing times. Here we also experience unwanted side-effects of the real-time scheme, where for some cases, skipped frames caused by the tracking service coincide with the detection frequency resulting in no new object initializations. Scheduling the detection stage on the PIs results in pipeline runs, where almost three-quarters of the dataset frames were dropped. This causes a significant reduction in the pipeline performance, with up to 85 percent of all objects being missed. With the improved encoding scheme, the aggregation stage no longer influences the pipeline performance as much as before.

[2] All figures that show the performance of separate mappings are enumerated in the following order: (detection-service, tracking-service, aggregation-service)

5.6 Different Object Detection Procedures

The detection service supports different neural networks through the OpenCV `dnn` module. Our current implementation allows for the DarkNet based YOLO networks to be integrated (see section 3.5). We plugged in the less complex TinyYOLO network to compare its performance impact to the YOLOv3-320 network. The network trades some of its accuracy for lower complexity and requires roughly seven times fewer FLOPS per execution.

When comparing the processing time for the TinyYOLO network in figure 5.7a with the data in figure 5.4a this results in significant speedup for all devices. The difference varies in strength, with the Raspberry PIs performing over six times faster than before. The APU performs almost four times faster on the CPU and double the speed on the iGPU. For the accelerated execution at the Jetson and the workstation, the speedup is not that drastic, with execution times of roughly 50 percent. We suspect the pipeline architecture and especially the only sporadic calls through the detection service as a possible reason for this. Optimizations for continuous operation of neural nets cannot be applied while the overhead of using an external executor (data transfer and translation) still remains. CPU-based execution benefits more from the reduced complexity, indicating a lower general overhead apart from the neural net.

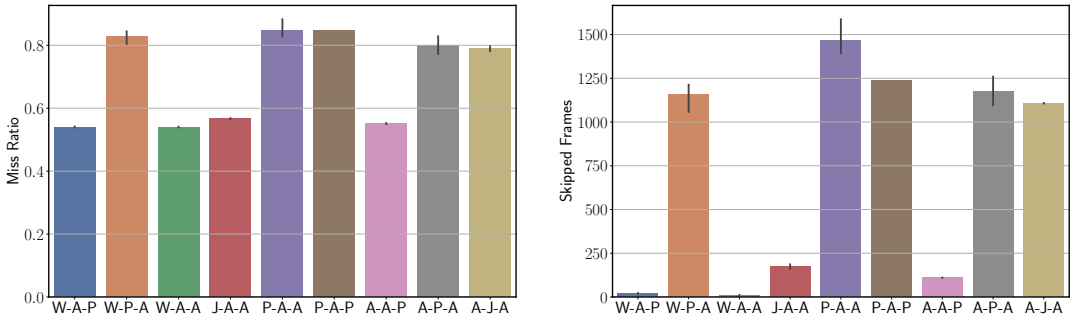
The faster neural network reduces the number of frames dropped for suboptimal deployments (see figure 5.7c) that were not able to operate with the normal YOLO network. However, the poorer detection quality results in a significant accuracy drop, as seen in figure 5.7b. For the best-performing approaches, the miss ratio rises from 55 percent to around 85 percent. The network detects fewer objects overall and produces less optimal bounding boxes for the tracking service. Additionally, inputs from the detection stage get lost relatively fast, and the missing consistency of new detection inputs results in this reduction.

5.7 Ethernet vs. wireless networks

For edge infrastructures, different network conditions play a vital role in the performance of the hosted services. Smaller devices often connect to the network using wireless connections such as Cellular, WiFi, or Bluetooth. Upcoming transmission standards like 5G or WiFi 6 promise radical improvements for latency and bandwidth, yet all wireless approaches are still slower and more unreliable than wired approaches. To evaluate the communication influence on the pipeline performance, we reconfigured the local testbed to utilize wireless connections for the APU and the Raspberry PIs.

To verify the increased communication latency, we employed a separate benchmark

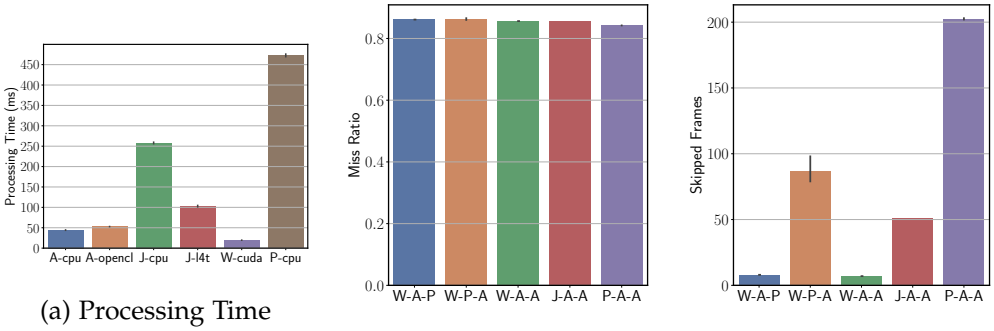
workload previously used for the evaluation of the EdgeIO platform [115, 9]. The workload is composed of a standard Nginx web server and a python script that measures the latency of 100 requests. Figure 5.8a shows this accumulated latency for the different links in the system for both network approaches. Connections that integrate the wireless connectivity of the APU or the PIs perform noticeably slower. The latency is much more unstable, with more values outside the vicinity of the mean. In the context of the pipeline performance, these unpredictable network conditions result in more frames being skipped by the aggregation. With these network conditions, the initial implementation utilizing the `numpy` encoding skips almost all frames due to the increased transfer times. With the improved encoding, this effect is softened to an increase of 200 to 300 additional skipped frames compared to the standard ethernet execution (see figure 5.8c). The majority of frames are skipped for the tracking communication, which explains the better performance for the third and fourth mapping where only the communication to the detection service partly happens over wireless LAN. The miss ratio depicted in figure 5.8b is more volatile with varying degrees of influence for the skipped frames. Overall the wireless deployment performs slightly worse than the ethernet-based approach but can maintain some form of operation.



(a) Ratio of misses

(b) Skipped Frames

Figure 5.6: Performance in select scenarios (W: Workstation, A: APU, J: Jetson, P:PI)

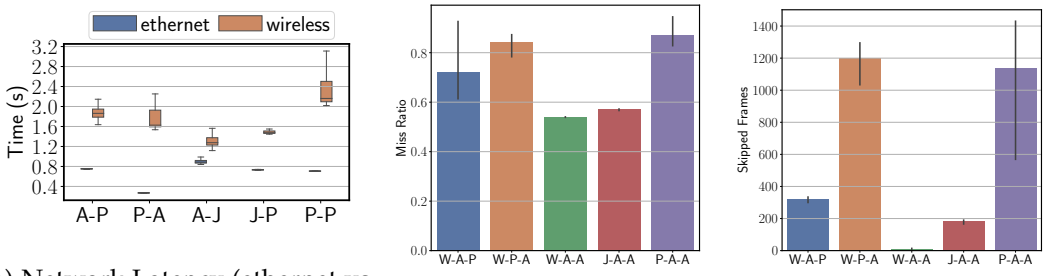


(a) Processing Time

(b) Miss Ratio

(c) Skipped Frames

Figure 5.7: Performance of TinyYOLO (W: Workstation, A: APU, J: Jetson, P:PI)



(a) Network Latency (ethernet vs. wireless)

(b) Miss Ratio

(c) Skipped Frames

Figure 5.8: Different network scenarios (W: Workstation, A: APU, J: Jetson, P:PI)

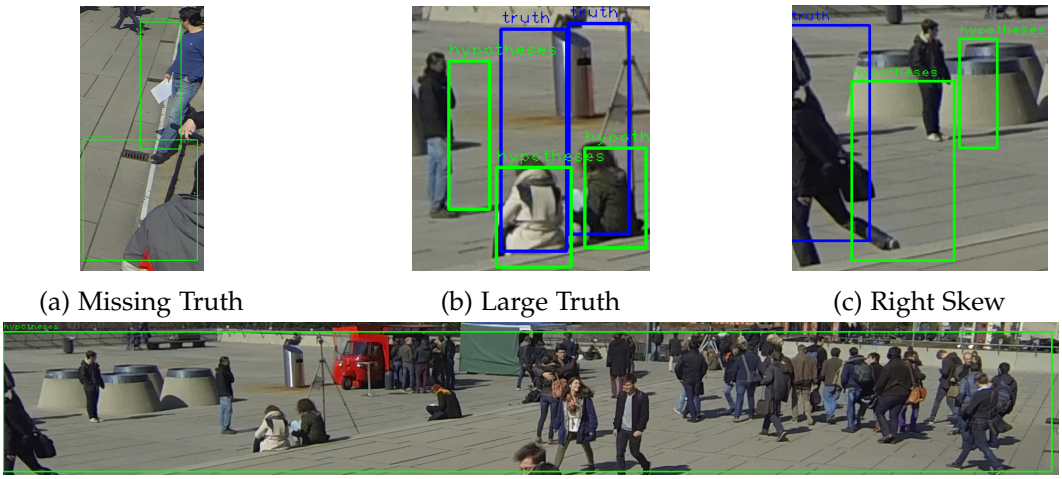
6 Discussion and Conclusion

This chapter discusses the findings from our evaluation and proposes future improvements to the benchmark suite. In the first section, we describe our challenges related to the chosen dataset and our naive tracking approach. Section 6.2 highlights our findings from the individual service performance. We discuss the influence of the real-time emulation scheme in section 6.3. Section 6.4 and section 6.5 discuss the extensibility and the scalability of our benchmark approach. The last section 6.6 revisits the initial research goals defined in section 1.2 and classifies our approach similar to the existing benchmarks.

6.1 Metric calculation and dataset problems

The MOTA and MOTP metrics are based on misses, mismatches, and false positives for the object hypotheses output by the tracking service. However, preliminary runs of the benchmark workload showed large differences between the expected values and the actual output. Visual review of the pipeline output shows a satisfactory performance of the pipeline. The resulting metrics opposed this conception, with many false positives and object misses, skewing the MOTA metric in the wrong direction. Contrary to our design goal, the better performing mappings actually resulted in a worse overall pipeline performance.

Further investigation into the dataset (described in section 3.3) and the ground truth provided by the dataset showed three major problems. First, the dataset states that it provides annotations for the first 2000 video frames. However, those were generated on a lower frame rate version of the video with ten fps. For the 60 fps version, this corresponds to the first 12000 frames or roughly 3:20 min. This introduces an increasing gap between our video input and the frame annotations and explains some of the false positives. Utilizing this fixed video input resulted in a significantly longer benchmark run which prohibits our proposed matching approach. We overcame this problem by resampling the input video to ten fps, so it corresponds to the source annotations. To avert other researchers from similar problems, we suggest future datasets utilize consistent frame rates for the annotation process and the video input. Secondly, we found several inconsistencies between our pipeline output and the provided ground truth. The used dataset misses people that are not fully present in the video frame



(a) Missing Truth (b) Large Truth (c) Right Skew

(d) Big False Positive

Figure 6.1: Visualization of Dataset Problems

(compare figure 6.1a). Our pipeline is able to detect and track these people, which results in false positives for objects that are correctly tracked. Bounding boxes around people do not necessarily correspond to the actual object size as seen in figure 6.1b. The big area of the ground truth results in a low intersection over union, which further increases the incorrect number of false positives. Thirdly, we resize the video input to a rectangular format corresponding to the input size of the YOLO network. Converting the rectangular bounding boxes back to the native input format introduces inaccuracies in the bounding boxes. Especially in the outer regions (see figure 6.1c) this introduces a skew towards the center of the image. Additionally, the YOLO network has its problems in handling the dataset and detects one bounding box that spans across the upper half of the image (see figure 6.1d). We have not identified any particular reason for this behavior, but we suspect the large gathering of people to be a limiting factor for the YOLO architecture. Xu et. al. [109] have identified crowded images as a weak point of the YOLO architecture as well and propose an improvement based on joint prediction. Future iterations of the tracking pipeline may integrate such an approach to evade these problems.

To overcome the above-mentioned drawbacks, we abandoned the MOTA/MOTP metrics and focused on the less expressive miss ratio $\overline{m} = \frac{misses}{totalObjectCount}$. While our tracking approach still performs worse than state-of-the-art MOT algorithms, it still allows us to compare the performance of different environments with more intuitive results. Additionally, we lowered the IoU threshold for the object association from fifty percent to thirty percent to pose a clearer picture of the pipeline performance. In the

future, we want to address these problems by integrating a better object detection and tracking approach [117, 35], a change of the dataset [25], and a complete rework of the MOT module of the metric service [55].

6.2 Execution environment

The extra layer of virtualization employed by the HPI resources introduces significant overhead and discourages the usage of virtual machines for the edge. However, the HPI and the local environment are not entirely comparable, which necessitates further research with different virtualization techniques. Other approaches, such as unikernels or AWS Firecracker MicroVMs [3], offer interesting alternatives to docker containers and should be integrated as well.

For the local environment, the service performance depends heavily on the available accelerators and the processor architecture. While the x86 based APU performs sufficiently fast when executing the pipeline on the CPU, the ARM-based devices struggle, especially for the detection stage. This gap was expected for the small-sized PIs, yet the more powerful Jetson shares similar performance characteristics. We attribute this behavior to the slower single-core speed of the ARM devices. This especially comes to fruition for the tracking service, where we identified a suboptimal implementation that utilizes only one thread. Pulli et. al. [82] have identified the integration of Intel TBB (Thread Building Blocks) as another optimization for the ARM platform. Based on these findings, we want to evaluate different threading approaches (pthreads, OpenMP, Intel TBB) as well as the influence of SIMD extensions on the tracking service performance. When utilizing hardware accelerators, the performance mostly corresponds to our initial expectations. Yet, the OpenCV `dnn` backend introduces a significant overhead for the first execution of the object detection (see section 5.4). This is particularly pronounced for the Jetson GPU and the Intel iGPU where the first forward takes up to 20 seconds. We suspect OpenCV to employ lazy, on-the-fly translation of the network weights to the specific platform, which results in this spike for the first detection. Other DNN backends like TensorFlow or Caffe may employ a different accelerator strategy and should be evaluated in the future. For commercial edge platforms based on Function-as-a-Service (FaaS) architectures, this may introduce additional problems when executing state-less, short-lived functions that utilize accelerators. Additionally, we find the slower execution on the Nvidia Jetson GPU compared to the CPU to oppose current hardware development trends. In the future, we want to integrate more machine-learning accelerators into our testbed and verify if the Jetson device is an outlier to this trend or if future accelerator development has to be rethought.

6.3 Real-Time Emulation

The employed real-time emulation scheme plays a vital part in the pipeline performance. It penalizes slow deployments where either the network or the processing speed cannot keep up with the desired frame rate by skipping frames. The network's performance is especially important for distributed applications deployed in diverse network conditions at the edge. Our findings regarding different encoding schemes highlight this influence. Future research may compare novel encoding procedures and evaluate their influence on pipeline performance. Especially for diverse edge deployments with complex network infrastructures, this provides further insights into the balance of transmission savings and encoding overhead for the edge. The current implementation for the frame skipping may have more adverse effects than intended. For some workload mappings, unfortunate circumstances may result in a substantial amount of detecting frames dropped. This significantly worsens the pipeline performance, with rare updates of the tracking service and old objects lost by the tracking algorithm. The larger computational complexity of the detection can cause a concentration of skipped frames directly after the detection stage. Valid objects may get lost quickly, caused by large jumps between the initialization data and the next video frame. For the used dataset, this is especially relevant with the changes we did to the video frame rate described in section 6.1. Future improvements of the tracking pipeline may remove the real-time emulation and rather focus on the maximum supported framerate more common in current tracking challenges [25]. Yet, adapting existing tracking algorithms to our real-time scheme may highlight their readiness for real-world edge deployments.

6.4 Extensibility

The benchmarking suite is designed with upcoming edge trends in mind. Developers can integrate custom workloads that better fit their specific use case. The latency workload utilized in subsection 5.8 was such a workload, initially used for the evaluation of EdgeIO [115, 9]. Minor adjustments were necessary to correctly containerize the workload components based on the required naming scheme. For common docker images (such as the `nginx`), this requires manually re-tagging them based on the employed technology. In the future, we want to improve this scheme by integrating the supported technologies into the image metadata instead of the image tag. This enables the orchestration to assume standard CPU capabilities for common containers and avoid unnecessary work for the developer. Integration of custom metrics requires small adjustments to the workload codebase, where accruing metrics get pushed to the metric service. Subsequent iterations of our benchmarking approach should provide good

library support that assists developers in this process. Lastly, custom workloads require the developer to adjust the workload configuration for the orchestration and the metric modules for the metric system. This split approach is a cause for inconsistencies and needs some rework in the future. We envision a unified configuration at the orchestration that configures the external metric system using additional control sequences similar to subsection 4.4.3. Additionally, the control flow between the benchmark orchestration and the metric system needs further refinement. Deploying the metric service in secluded environments with no outside HTTP communication impedes our current approach. In our tests, we deployed a CPU-based variant of our benchmark workload on Kubernetes (k8s, k3s, and microk8s) and EdgeIO to highlight the ability to deploy real-world applications. In the future, we want to concretize this integration further and compare their performance influence in similar environments.

6.5 Scalability

Our approach focuses on small edge environments and the specific performance of the employed hardware. Larger or more diverse environments result in a significant increase of generated mappings that converges to $O(n!)$ as described in subsection 4.3.3. The additional structuring into node groups and the limited node capacity soften this problem, yet homogeneous environments (like the HPI resources) are still suboptimally handled. Many benchmark runs do not offer significant insight into the environment but prolong the overall duration. We plan to address this issue by refining the matching procedure with a more restrictive approach that considers more device characteristics and significantly prunes the number of mappings. Individual device benchmarks similar to Scission [70] can serve as a characteristic that influences the scheduling. For (edge) orchestration platforms that include their own scheduler component, we want to adapt the mapping approach to the different technologies available for the workload and let the platform handle the orchestration.

Another scalability issue is the current approach for preparing nodes to run a service. We copy needed files to the nodes and pull the most recent docker image for the specific workload. For large environments, this results in significant image pulls that quickly reach the free rate limits of the docker hub repository [1]. Developers may revert to a privately hosted image repository, yet this requires the developer to set up access on each node manually. In the future, we want to integrate a local (image) cache that pulls the newest build artifacts only once and distributes them locally. This concept can be extended for non-docker artifacts such as VM templates, native executables, or unikernels and allow the execution in disconnected environments where only the

[1] `https://www.docker.com/increase-rate-limits`

developer machine can access the internet.

6.6 Classification and Conclusion

This thesis addresses missing aspects of current edge benchmarks compared in table 2.2. We extend on this in table 6.1 and classify our approach based on the same parameters. In addition, we revisit our research goals defined in section 1.2 and evaluate our proposed solution against them:

RG1 The proposed workload represents a realistic video analytics pipeline for the MOT problem. We simulate real IP-cams often used for video surveillance with the RTSP protocol. The pipeline integrates real camera data from the WILDTRACK dataset with representative input resolution. We utilize commercially available edge devices and accelerators such as the Nvidia Jetson Xavier or the Raspberry PI to capture the heterogeneity of edge devices. Explicit network communication between services utilizes gRPC as state-of-the-art technology, reflecting modern application design. The workload integrates a novel real-time emulation that penalizes deployments in suboptimal conditions and influences the pipeline accuracy.

RG2 The proposed benchmark suite paints a comprehensive picture of edge deployments and explores an extensive list of possible service schedules. We capture different (even non-intuitive) deployment options and compare their performance. The architecture of the orchestration service integrates future support for edge orchestration platforms like Kubernetes or EdgeIO. While we compare VM-based execution to lightweight docker containers in our evaluation, we currently do not support more distinct virtualization techniques used at the edge. To fully achieve this goal, more implementation effort is needed, where we further evaluate different novel orchestration platforms and integrate them into our benchmark.

RG3 Our approach provides a quickly extensible platform for future edge workloads that can be integrated into our flexible workload configuration and metric system. While our naive multi-tracking implementation performs worse than recent tracking implementation such as ByteTrack [117], the modular architecture of our pipeline allows for easy integration of such approaches. Apart from the proprietary source code used for the CUDA and L4T docker images, we build on open source components and plan to make our codebase available to the public.

Benchmark	CPU	Accelerators	Memory	Storage	Network	Orchestration	Sevice Model	Schedulers	AI platforms
CoAP benchmark [58]	+	-	+	-	-	-	-	-	-
RIoTBench [93]	+	-	+	-	-	-	-	-	-
EdgeBench [24]	+	-	+	-	-	-	+	-	-
Edge AIBench (concept only) [46]	-	-	-	-	-	-	-	-	+
DeFog [72]	+	-	-	-	+	-	-	-	-
Edge accelerator benchmarking [87, 27]	+	+	-	-	-	-	-	-	-
EdgeBench (2) [111]	+	-	+	-	+	-	+	-	-
OpenRTiST [36]	+	+/-	-	-	+	-	-	-	+/-
Scission [70]	+	+/-	-	-	-	-	-	-	-
Proposed approach	+	**+**	+/-	-	**+**	**(+)**	-	-	-

Table 6.1: Tested Components of new benchmark suite (improvements bold), extension of table 2.2

List of Figures

List of Tables

Glossary

AI	Artifical Intelligence
API	Application Programming Interface
APU	Accelerated Processing Unit
AR	Augmented Reality
AWS	Amazon Web Services
CNN	Convolutional Neural Network
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
FaaS	Function-as-a-Service
FNN	Feedforward Neural Network
FPS	Frames Per Second
GCP	Google Cloud Platform
GPGPU	General Purpose Graphic Processing Unit
HW	Hardware
IaC	Infrastructure as Code
IoT	Internet of Things
KCF	Kernelize Correlation Filters
MIPS	Million Instruction per Second
ML	Machine Learning
MOT	Multiple Object Tracking
NAT	Network Address Translation
NST	Neural Style Transfer
OS	Operating System
QoS	Quality of Service
REST	Representional State Transfer
RNN	Recurrent Neural Network
RTP	Real-time Transport Protocol
RTSP	Real-time Streaming Protocol
SDK	Software Development Kit
SoC	System on a Chip

TPU	Tensor Processing Unit
VM	Virtual Machine
VR	Virtual Reality
YOLO	You Only Look Once

Bibliography

[1] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie. "Mobile Edge Computing: A Survey." In: *IEEE Internet of Things Journal* 5.1 (Feb. 2018), pp. 450–465. ISSN: 2327-4662. DOI: 10.1109/JIOT.2017.2750180.

[2] M. Alam, N. Ahmed, R. Matam, and F. A. Barbhuiya. "ioFog: Prediction-based Fog Computing Architecture for Offline IoT." In: *2021 International Wireless Communications and Mobile Computing (IWCMC)*. June 2021, pp. 1387–1392. DOI: 10.1109/IWCMC51323.2021.9498796.

[3] Amazon Web Services, Inc. *Firecracker MicroVM*. firecracker-microvm. https://github.com/firecracker-microvm/firecracker. Dec. 2021.

[4] Apple Inc. *Apple Unleashes M1 - Apple*. https://www.apple.com/newsroom/2020/11/apple-unleashes-m1/.

[5] M. S. Aslanpour, S. S. Gill, and A. N. Toosi. "Performance Evaluation Metrics for Cloud, Fog and Edge Computing: A Review, Taxonomy, Benchmarks and Standards for Future Research." In: *Internet of Things* 12 (Dec. 2020), p. 100273. ISSN: 2542-6605. DOI: 10.1016/j.iot.2020.100273.

[6] S. R. Balaji and S. Karthikeyan. "A Survey on Moving Object Tracking Using Image Processing." In: *2017 11th International Conference on Intelligent Systems and Control (ISCO)*. Jan. 2017, pp. 469–474. DOI: 10.1109/ISCO.2017.7856037.

[7] A. Barbalace, M. L. Karaoui, W. Wang, T. Xing, P. Olivier, and B. Ravindran. "Edge Computing: The Case for Heterogeneous-ISA Container Migration." In: *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. Lausanne Switzerland: ACM, Mar. 2020, pp. 73–87. ISBN: 978-1-4503-7554-2. DOI: 10.1145/3381052.3381321.

[8] A. K. Bardsiri and S. M. Hashemi. *QoS Metrics for Cloud Computing Services Evaluation*.

[9] G. Bartolomeo. "Enabling Microservice Interactions within Heterogeneous Edge Infrastructures." MA thesis. Technical University Munich, Sept. 2021.

[10] S. Basu. *A Study on Selection of Data Center Locations*.

[11] P. Bergmann, T. Meinhardt, and L. Leal-Taixe. "Tracking without Bells and Whistles." In: *arXiv:1903.05625 [cs]* (Aug. 2019). http://arxiv.org/abs/1903.05625. arXiv: 1903.05625 [cs].

[12] K. Bernardin, A. Elbs, and R. Stiefelhagen. "Multiple Object Tracking Performance Metrics and Evaluation in a Smart Room Environment." In: (), p. 8.

[13] S. Biookaghazadeh, M. Zhao, and F. Ren. "Are FPGAs Suitable for Edge Computing?" In: (), p. 6.

[14] S. Bohm and G. Wirtz. "Profiling Lightweight Container Platforms: MicroK8s and K3s in Comparison to Kubernetes." In: (), p. 9.

[15] A. Brogi, S. Forti, C. Guerrero, and I. Lera. "How to Place Your Apps in the Fog – State of the Art and Open Challenges." In: *Software: Practice and Experience* 50.5 (May 2020), pp. 719–740. ISSN: 0038-0644, 1097-024X. DOI: 10.1002/spe.2766. arXiv: 1901.05717.

[16] Cambridge Dictionary. *Definition of Benchmarking*. https://dictionary.cambridge.org/dictionary/english/benchmarking.

[17] T. Chavdarova, P. Baque, S. Bouquet, A. Maksai, C. Jose, T. Bagautdinov, L. Lettry, P. Fua, L. Van Gool, and F. Fleuret. "WILDTRACK: A Multi-camera HD Dataset for Dense Unscripted Pedestrian Detection." In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. Salt Lake City, UT: IEEE, June 2018, pp. 5030–5039. ISBN: 978-1-5386-6420-9. DOI: 10.1109/CVPR.2018.00528.

[18] G. Ciaparrone, F. L. Sánchez, S. Tabik, L. Troiano, R. Tagliaferri, and F. Herrera. "Deep Learning in Video Multi-Object Tracking: A Survey." In: *Neurocomputing* 381 (Mar. 2020), pp. 61–88. ISSN: 09252312. DOI: 10.1016/j.neucom.2019.11.023. arXiv: 1907.12740.

[19] CISCO. *Cisco Annual Internet Report (2018–2023)*. 2020.

[20] C. Coleman, D. Narayanan, D. Kang, T. Zhao, J. Zhang, L. Nardi, P. Bailis, K. Olukotun, C. Ré, and M. Zaharia. "DAWNBench: An End-to-End Deep Learning Benchmark and Competition." In: (), p. 10.

[21] N. Corporation. *NVIDIA Embedded Systems for Next-Gen Autonomous Machines*. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/.

[22] Crystal Dew World. *CrystalDiskMark*. https://crystalmark.info/en/software/crystaldiskmark/.

[23] daniel. *The Tengo Language*. https://github.com/d5/tengo. Nov. 2021.

[24] A. Das, S. Patterson, and M. Wittie. "EdgeBench: Benchmarking Edge Computing Platforms." In: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. Dec. 2018, pp. 175–180. DOI: 10.1109/UCC-Companion.2018.00053.

[25] P. Dendorfer, H. Rezatofighi, A. Milan, J. Shi, D. Cremers, I. Reid, S. Roth, K. Schindler, and L. Leal-Taixé. "MOT20: A Benchmark for Multi Object Tracking in Crowded Scenes." In: *arXiv:2003.09003 [cs]* (Mar. 2020). http://arxiv.org/abs/2003.09003. arXiv: 2003.09003 [cs].

[26] B. Deori and D. M. Thounaojam. "A Survey on Moving Object Tracking in Video." In: *International Journal on Information Theory* 3.3 (July 2014), pp. 31–46. ISSN: 23208465, 23197609. DOI: 10.5121/ijit.2014.3304.

[27] G. Dinelli, G. Meoni, E. Rapuano, G. Benelli, and L. Fanucci. "An FPGA-Based Hardware Accelerator for CNNs Using On-Chip Memories Only: Design and Benchmarking with Intel Movidius Neural Compute Stick." In: *International Journal of Reconfigurable Computing* 2019 (Oct. 2019). Ed. by M. Margala, pp. 1–13. ISSN: 1687-7209, 1687-7195. DOI: 10.1155/2019/7218758.

[28] Docker Inc. *Empowering App Development for Developers | Docker*. https://www.docker.com/.

[29] J. Drouet. *Multi-Arch Build and Images, the Simple Way*. https://www.docker.com/blog/multi-arch-build-and-images-the-simple-way/. Apr. 2020.

[30] Y. Duan, G. Fu, N. Zhou, X. Sun, N. C. Narendra, and B. Hu. "Everything as a Service (XaaS) on the Cloud: Origins, Current and Future Trends." In: *2015 IEEE 8th International Conference on Cloud Computing*. June 2015, pp. 621–628. DOI: 10.1109/CLOUD.2015.88.

[31] S. Eyerman and L. Eeckhout. "System-Level Performance Metrics for Multiprogram Workloads." In: *IEEE Micro* 28.3 (May 2008), pp. 42–53. ISSN: 1937-4143. DOI: 10.1109/MM.2008.44.

[32] R. T. Fielding. "Architectural Styles and the Design of Network-based Software Architectures." PhD thesis. University of California, 2000.

[33] M. Fowler. *Microservices and the First Law of Distributed Objects*. https://martinfowler.com/articles/distributed-objects-microservices.html. InternetDocument. Aug. 2014.

[34] K. Fukuda and T. Matsui. "Finding All the Perfect Matchings in Bipartite Graphs." In: *Applied Mathematics Letters* 7.1 (Jan. 1994), pp. 15–18. ISSN: 08939659. DOI: 10.1016/0893-9659(94)90045-0.

[35] Z. Ge, S. Liu, F. Wang, Z. Li, and J. Sun. "YOLOX: Exceeding YOLO Series in 2021." In: *arXiv:2107.08430 [cs]* (Aug. 2021). http://arxiv.org/abs/2107.08430. arXiv: 2107.08430 [cs].

[36] S. George, T. Eiszler, R. Iyengar, H. Turki, Z. Feng, J. Wang, P. Pillai, and M. Satyanarayanan. "OpenRTiST: End-to-End Benchmarking for Edge Computing." In: *IEEE Pervasive Computing* 19.4 (Oct. 2020), pp. 10–18. ISSN: 1536-1268, 1558-2590. DOI: 10.1109/MPRV.2020.3028781.

[37] R. Girshick, J. Donahue, T. Darrell, and J. Malik. "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation." In: (), p. 8.

[38] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. Adaptive Computation and Machine Learning. Cambridge, Massachusetts: The MIT Press, 2016. ISBN: 978-0-262-03561-3.

[39] Google LLC. *Edge TPU – Inferenzen auf Edge-Geräten ausführen*. https://cloud.google.com/edge-tpu?hl=de.

[40] B. Götz, D. Schel, D. Bauer, C. Henkel, P. Einberger, and T. Bauernhansl. "Challenges of Production Microservices." In: *Procedia CIRP* 67 (Jan. 2018), pp. 167–172. ISSN: 22128271. DOI: 10.1016/j.procir.2017.12.194.

[41] gRPC Authors. *gRPC*. https://grpc.io/.

[42] GStreamer Team. *GStreamer: Open Source Multimedia Framework*. https://gstreamer.freedesktop.org/.

[43] GStreamer Team. *GStreamer/Gst-Rtsp-Server*. GStreamer GitHub mirrors. https://github.com/GStreamer/gst-rtsp-server. Nov. 2021.

[44] H. Habibi Aghdam and E. Jahani Heravi. *Guide to Convolutional Neural Networks: A Practical Application to Traffic-Sign Detection and Classification*. http://ebookcentral.proquest.com/lib/munchentech/detail.action?docID=4862504. Cham, SWITZERLAND: Springer International Publishing AG, 2017. ISBN: 978-3-319-57550-6.

[45] S. Hackel, A. Rennoch, and D. A. Willner. *Testing Challenges in Edge Computing*. https://www.edgecomputingworld.com/ecwe2021/. Mar. 2021.

[46] T. Hao, Y. Huang, X. Wen, W. Gao, F. Zhang, C. Zheng, L. Wang, H. Ye, K. Hwang, Z. Ren, and J. Zhan. "Edge AIBench: Towards Comprehensive End-to-End Edge Computing Benchmarking." In: *Benchmarking, Measuring, and Optimizing*. Ed. by C. Zheng and J. Zhan. Vol. 11459. Cham: Springer International Publishing, 2019, pp. 23–30. ISBN: 978-3-030-32813-9. DOI: 10.1007/978-3-030-32813-9_3.

[47] R. Harms and M. Yamartino. "THE ECONOMICS OF THE CLOUD." In: (), p. 22.

[48] A. Hidaka and T. Kurita. "Consecutive Dimensionality Reduction by Canonical Correlation Analysis for Visualization of Convolutional Neural Networks." In: *Proceedings of the ISCIE International Symposium on Stochastic Systems Theory and Its Applications*. Vol. 2017. Dec. 2017, pp. 160–167. DOI: 10.5687/sss.2017.160.

[49] R. W. Hockney. *The Science of Computer Benchmarking*. Software, Environments and Tools. Society for Industrial and Applied Mathematics, Jan. 1996. ISBN: 978-0-89871-363-3. DOI: 10.1137/1.9780898719666.

[50] J. E. Hopcroft and R. M. Karp. "An $n\hat{}\{5/2\}$ Algorithm for Maximum Matchings in Bipartite Graphs." In: *SIAM Journal on Computing* 2.4 (Dec. 1973), pp. 225–231. ISSN: 0097-5397, 1095-7111. DOI: 10.1137/0202019.

[51] Intel Corporation. *Intel® Movidius™ Myriad™ X Vision Processing Unit (VPU)*. https://www.intel.com/content/www/us/en/products/details/processors/movidius-vpu/movidius-myriad-x.html.

[52] *Intelligence at the IoT Edge — AWS IoT Greengrass — Amazon Web Services*. https://aws.amazon.com/greengrass/.

[53] *IoT Edge | Cloud Intelligence | Microsoft Azure*. https://azure.microsoft.com/en-us/services/iot-edge/.

[54] P. Jamshidi, C. Pahl, N. C. Mendonca, J. Lewis, and S. Tilkov. "Microservices: The Journey So Far and Challenges Ahead." In: *IEEE Software* 35.3 (Jan. 2018), pp. 24–35. ISSN: 0740-7459. DOI: 10.1109/MS.2018.2141039.

[55] A. H. Jonathon Luiten. *TrackEval*. https://github.com/JonathonLuiten/TrackEval. 2020.

[56] W. Judd. *Games with Built-in Benchmarks 2021: How to Benchmark Your PC - Review - Gaming | XSReviews*. https://xsreviews.co.uk/news/games-with-built-in-benchmarks-how-to-benchmark-your-pc/. July 2021.

[57] T. Killalea. "The Hidden Dividends of Microservices." In: *Communications of the ACM* 59.8 (Jan. 2016), pp. 42–45. ISSN: 00010782. DOI: 10.1145/2948985.

[58] C. P. Kruger and G. P. Hancke. "Benchmarking Internet of Things Devices." In: *2014 12th IEEE International Conference on Industrial Informatics (INDIN)*. July 2014, pp. 611–616. DOI: 10.1109/INDIN.2014.6945583.

[59] *Kubernetes Cluster Federation*. Kubernetes SIGs. https://github.com/kubernetes-sigs/kubefed. Nov. 2021.

[60] H. W. Kuhn. "The Hungarian Method for the Assignment Problem." In: *Naval Research Logistics Quarterly* 2.1-2 (1955), pp. 83–97. ISSN: 1931-9193. DOI: 10.1002/nav.3800020109.

[61] P. Lea. *IoT and Edge Computing for Architects - Second Edition*. https://www.packtpub.com/product/iot-and-edge-computing-for-architects-second-edition/9781839214806. Packt Publishing Ltd., Mar. 2020.

[62] C. Lee, M. Lin, C. Yang, and Y. Chen. "Iotbench: A Benchmark Suite for Intelligent Internet of Things Edge Devices." In: *2019 IEEE International Conference on Image Processing (ICIP)*. Sept. 2019, pp. 170–174. DOI: 10.1109/ICIP.2019.8802949.

[63] B. C. Lewis and A. E. Crews. "The Evolution of Benchmarking as a Computer Performance Evaluation Technique." In: *MIS Quarterly* 9.1 (1985), pp. 7–16. ISSN: 0276-7783. DOI: 10.2307/249270.

[64] W. Li and M. Liewig. "A Survey of AI Accelerators for Edge Environment." In: *Trends and Innovations in Information Systems and Technologies*. Ed. by Á. Rocha, H. Adeli, L. P. Reis, S. Costanzo, I. Orovic, and F. Moreira. Advances in Intelligent Systems and Computing. Cham: Springer International Publishing, 2020, pp. 35–44. ISBN: 978-3-030-45691-7. DOI: 10.1007/978-3-030-45691-7_4.

[65] X. Li, W. Hu, C. Shen, Z. Zhang, A. Dick, and A. van den Hengel. "A Survey of Appearance Models in Visual Object Tracking." In: *arXiv:1303.4803 [cs]* (Mar. 2013). http://arxiv.org/abs/1303.4803. arXiv: 1303.4803 [cs].

[66] L. Liu, W. Ouyang, X. Wang, P. Fieguth, J. Chen, X. Liu, and M. Pietikäinen. "Deep Learning for Generic Object Detection: A Survey." In: *International Journal of Computer Vision* 128.2 (Feb. 2020), pp. 261–318. ISSN: 0920-5691, 1573-1405. DOI: 10.1007/s11263-019-01247-4.

[67] S. Liu, L. Liu, J. Tang, B. Yu, Y. Wang, and W. Shi. "Edge Computing for Autonomous Driving: Opportunities and Challenges." In: *Proceedings of the IEEE* 107.8 (Aug. 2019), pp. 1697–1716. ISSN: 1558-2256. DOI: 10.1109/JPROC.2019.2915983.

[68] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg. "SSD: Single Shot MultiBox Detector." In: *Computer Vision – ECCV 2016*. Ed. by B. Leibe, J. Matas, N. Sebe, and M. Welling. Vol. 9905. Cham: Springer International Publishing, 2016, pp. 21–37. ISBN: 78-3-319-46448-0. DOI: 10.1007/978-3-319-46448-0_2.

[69] Y. Liu. *Getting Started with Machine Learning and Python*. https://learning.oreilly.com/library/view/python-machine-learning/9781800209718/Text/Chapter_1.xhtml. Oct. 2020. ISBN: 978-1-80020-971-8.

[70] L. Lockhart, P. Harvey, P. Imai, P. Willis, and B. Varghese. "Scission: Performance-driven and Context-aware Cloud-Edge Distribution of Deep Neural Networks." In: *arXiv:2008.03523 [cs]* (Dec. 2020). http://arxiv.org/abs/2008.03523. arXiv: 2008.03523 [cs].

[71] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. "Unikernels: Library Operating Systems for the Cloud." In: *ACM SIGARCH Computer Architecture News* 41.1 (Mar. 2013), pp. 461–472. ISSN: 0163-5964. DOI: 10.1145/2490301.2451167.

[72] J. McChesney, N. Wang, A. Tanwer, E. de Lara, and B. Varghese. "DeFog: Fog Computing Benchmarks." In: *arXiv:1907.10890 [cs]* (July 2019). http://arxiv.org/abs/1907.10890. arXiv: 1907.10890 [cs].

[73] R. McKendrick and S. Gallagher. *Mastering Docker: Second Edition*. 2nd ed. Birmingham, UK: Packt Publishing, Jan. 2017. ISBN: 978-1-78728-024-3.

[74] L. McVoy, C. Staelin, and H.-P. Laboratories. "Lmbench: Portable Tools for Performance Analysis." In: (), p. 17.

[75] Merriam Webster. *Definition of BENCHMARK*. https://www.merriam-webster.com/dictionary/benchmark.

[76] N. Mohan, L. Corneo, A. Zavodovski, S. Bayhan, W. Wong, and J. Kangasharju. "Pruning Edge Research with Latency Shears." In: *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*. Virtual Event USA: ACM, Nov. 2020, pp. 182–189. ISBN: 978-1-4503-8145-1. DOI: 10.1145/3422604.3425943.

[77] N. Mohan and J. Kangasharju. "Edge-Fog Cloud: A Distributed Cloud for Internet of Things Computations." In: *2016 Cloudification of the Internet of Things (CIoT)*. Paris, France: IEEE, Nov. 2016, pp. 1–6. ISBN: 978-1-5090-4960-8. DOI: 10.1109/CIOT.2016.7872914.

[78] D. Namiot and M. Sneps-Sneppe. "On Micro-services Architecture." In: *International Journal of Open Information Technologies* 9 (Jan. 2014), pp. 24–27.

[79] Netflix Inc. *Falcor*. Netflix, Inc. https://github.com/Netflix/falcor. Nov. 2021.

[80] Object Management Group®, Inc. *Common Object Request Broker Architecture™ (CORBA®)*. https://www.corba.org/.

[81] *Protocol Buffers - Google's Data Interchange Format*. Protocol Buffers. https://github.com/protocolbuffers/protobuf. Nov. 2021.

[82] K. Pulli, A. Baksheev, K. Kornyakov, and V. Eruhimov. "Realtime Computer Vision with OpenCV." In: (), p. 17.

[83] Qualcom Technologies Inc. *Qualcomm Artificial Intelligence | AI Machine Learning*. https://www.qualcomm.com/research/artificial-intelligence. 2021.

[84] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. "You Only Look Once: Unified, Real-Time Object Detection." In: *arXiv:1506.02640 [cs]* (May 2016). http://arxiv.org/abs/1506.02640. arXiv: 1506.02640 [cs].

[85] J. Redmon and A. Farhadi. "YOLOv3: An Incremental Improvement." In: *arXiv* (2018).

[86] J. Ren, Y. He, G. Huang, G. Yu, Y. Cai, and Z. Zhang. "An Edge-Computing Based Architecture for Mobile Augmented Reality." In: *IEEE Network* 33.4 (July 2019), pp. 162–169. ISSN: 1558-156X. DOI: 10.1109/MNET.2018.1800132.

[87] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner. "Survey and Benchmarking of Machine Learning Accelerators." In: *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. Sept. 2019, pp. 1–9. DOI: 10.1109/HPEC.2019.8916327.

[88] robvet. *Cloudbasierte Resilienz*. https://docs.microsoft.com/de-de/dotnet/architecture/cloud-native/resiliency. Sept. 2021.

[89] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. "The Case for VM-Based Cloudlets in Mobile Computing." In: *IEEE Pervasive Computing* 8.4 (Oct. 2009), pp. 14–23. ISSN: 1558-2590. DOI: 10.1109/MPRV.2009.82.

[90] M. Satyanarayanan, G. Klas, M. Silva, and S. Mangiante. "The Seminal Role of Edge-Native Applications." In: *2019 IEEE International Conference on Edge Computing (EDGE)*. July 2019, pp. 33–40. DOI: 10.1109/EDGE.2019.00022.

[91] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. "Edge Computing: Vision and Challenges." In: *IEEE Internet of Things Journal* 3.5 (Oct. 2016), pp. 637–646. ISSN: 2327-4662. DOI: 10.1109/JIOT.2016.2579198.

[92] J. Shore and S. Warden. *The Art of Agile Development*. Theory in Practice. Beijing : Sebastopol, CA: O'Reilly Media, Inc, 2008. ISBN: 978-0-596-52767-9.

[93] A. Shukla, S. Chaturvedi, and Y. Simmhan. "RIoTBench: An IoT Benchmark for Distributed Stream Processing Systems." In: *Concurrency and Computation: Practice and Experience* 29.21 (2017), e4257. ISSN: 1532-0634. DOI: 10.1002/cpe.4257.

[94] A. K. Talukder, L. Zimmerman, and P. H. A. "Cloud Economics: Principles, Costs, and Benefits." In: *Cloud Computing: Principles, Systems and Applications*. Ed. by N. Antonopoulos and L. Gillam. Computer Communications and Networks. London: Springer, 2010, pp. 343–360. ISBN: 978-1-84996-241-4. DOI: 10.1007/978-1-84996-241-4_20.

[95] The Apache Software Foundation. *ActiveMQ*. https://activemq.apache.org/.

[96] The GraphQL Foundation. *GraphQL | A Query Language for Your API*. https://graphql.org/.

[97] The Internet Society. *RFC2326: Real Time Streaming Protocol (RTSP)*. https://datatracker.ietf.org/doc/html/rfc2326.

[98] The Internet Society. *RFC3550: RTP: A Transport Protocol for Real-Time Applications*. https://datatracker.ietf.org/doc/html/rfc3550.

[99] The Kubernetes Authors. *Production-Grade Container Orchestration*. https://kubernetes.io/.

[100] The Kubernetes Authors. *Schedule GPUs*. https://kubernetes.io/docs/tasks/manage-gpus/scheduling-gpus/.

[101] UL LLC. *3DMark - The Gamer's Benchmark*. https://benchmarks.ul.com/3dmark.

[102] B. Varghese, C. Reaño, and F. Silla. "Accelerator Virtualization in Fog Computing: Moving from the Cloud to the Edge." In: *IEEE Cloud Computing* 5.6 (Nov. 2018), pp. 28–37. ISSN: 2325-6095. DOI: 10.1109/MCC.2018.064181118.

[103] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, and D. Nikolopoulos. "Challenges and Opportunities in Edge Computing." In: Nov. 2016. DOI: 10.1109/SmartCloud.2016.18.

[104] B. Varghese, N. Wang, D. Bermbach, C.-H. Hong, E. de Lara, W. Shi, and C. Stewart. "A Survey on Edge Performance Benchmarking." In: *arXiv:2004.11725 [cs]* (Dec. 2020). http://arxiv.org/abs/2004.11725. arXiv: 2004.11725 [cs].

[105] F. D. Weber and R. Schütte. "State-of-the-Art and Adoption of Artificial Intelligence in Retailing." In: *Digital Policy, Regulation and Governance* 21.3 (Jan. 2019), pp. 264–279. ISSN: 2398-5038. DOI: 10.1108/DPRG-09-2018-0050.

[106] M. F. Wiersema and H. P. Bowen. "Corporate Diversification: The Impact of Foreign Competition, Industry Globalization, and Product Diversification." In: *Strategic Management Journal* 29.2 (Feb. 2008), pp. 115–132. ISSN: 01432095, 10970266. DOI: 10.1002/smj.653.

[107] E. Wolff. *Microservices: Grundlagen Flexibler Softwarearchitekturen*. 1. Auflage. Heidelberg: dpunkt.verlag, Jan. 2016. ISBN: 978-3-86490-313-7.

[108] Y. Xiong, Y. Sun, L. Xing, and Y. Huang. "Extend Cloud to Edge with KubeEdge." In: *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. Seattle, WA, USA: IEEE, Oct. 2018, pp. 373–377. ISBN: 978-1-5386-9445-9. DOI: 10.1109/SEC.2018.00048.

[109] H.-h. Xu, X.-q. Wang, D. Wang, B.-g. Duan, and T. Rui. "Object Detection in Crowded Scenes via Joint Prediction." In: *Defence Technology* (Oct. 2021). ISSN: 2214-9147. DOI: 10.1016/j.dt.2021.10.007.

[110] M. Xu, Z. Fu, X. Ma, L. Zhang, Y. Li, F. Qian, S. Wang, K. Li, J. Yang, and X. Liu. "From Cloud to Edge: A First Look at Public Edge Platforms." In: *Proceedings of the 21st ACM Internet Measurement Conference*. IMC '21. New York, NY, USA: Association for Computing Machinery, Nov. 2021, pp. 37–53. ISBN: 978-1-4503-9129-0. DOI: 10.1145/3487552.3487815.

[111] Q. Yang, R. Jin, N. Gandhi, X. Ge, H. A. Khouzani, and M. Zhao. "EdgeBench: A Workflow-based Benchmark for Edge Computing." In: *arXiv:2010.14027 [cs]* (Oct. 2020). http://arxiv.org/abs/2010.14027. arXiv: 2010.14027 [cs].

[112] Y. Ye, S. Ci, A. K. Katsaggelos, Y. Liu, and Y. Qian. "Wireless Video Surveillance: A Survey." In: *IEEE Access* 1 (2013), pp. 646–660. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2013.2282613.

[113] S. Yi, C. Li, and Q. Li. "A Survey of Fog Computing: Concepts, Applications and Issues." In: *Proceedings of the 2015 Workshop on Mobile Big Data*. Mobidata '15. New York, NY, USA: Association for Computing Machinery, June 2015, pp. 37–42. ISBN: 978-1-4503-3524-9. DOI: 10.1145/2757384.2757397.

[114] A. Yilmaz, O. Javed, and M. Shah. "Object Tracking: A Survey." In: *ACM Computing Surveys* 38.4 (Dec. 2006), p. 13. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/1177352.1177355.

[115] M. Yosofie. "Flexible Multi-Cluster Edge Orchestration and Task Scheduling Platform." MA thesis. Munich: Technical University Munich, Feb. 2021.

[116] B. Zhang, N. Mor, J. Kolb, D. S. Chan, N. Goyal, K. Lutz, E. Allman, J. Wawrzynek, E. Lee, and J. Kubiatowicz. "The Cloud Is Not Enough: Saving IoT from the Cloud." In: (), p. 7.

[117] Y. Zhang, P. Sun, Y. Jiang, D. Yu, Z. Yuan, P. Luo, W. Liu, and X. Wang. "ByteTrack: Multi-Object Tracking by Associating Every Detection Box." In: *arXiv:2110.06864 [cs]* (Oct. 2021). http://arxiv.org/abs/2110.06864. arXiv: 2110.06864 [cs].

[118] Z. Zou, Z. Shi, Y. Guo, and J. Ye. "Object Detection in 20 Years: A Survey." In: *arXiv:1905.05055 [cs]* (May 2019). http://arxiv.org/abs/1905.05055. arXiv: 1905.05055 [cs].