COMMUNITY-ORIENTED EDGE COMPUTING PLATFORM

By

ABDALLA A. MOUSTAFA

A thesis submitted to the Graduate Program in Department of Electrical and Computer Engineering in conformity with the requirements for the Degree of Master of Applied Science

Queen's University

Kingston, Ontario, Canada

September, 2022

Abstract

The surging demand for Edge Computing (EC) to cope with the proliferation of latency-critical and data-intensive applications has inspired the concept of recycling ample yet underutilized computational resources of end devices, also referred to as Extreme Edge Devices (EEDs). However, maintaining data privacy and cost efficiency remain core challenges for the viability of EED-enabled computing paradigms. In this thesis, we propose the Community Edge Platform (CEP). CEP exploits business, institutional, and social relationships to build clusters and communities of requesters and EEDs to eliminate recruitment barriers and preserve privacy. Furthermore, CEP utilizes a hierarchical control paradigm to prioritize the enrollment of nearby devices as workers. CEP first seeks workers within the requester's cluster, which is composed of devices owned by the same user. In case of a lack of a suitable device, CEP resorts to devices in other clusters that belong to the requester's community. In addition to the underlying architecture and structure of CEP, we also consider the fact that community-imposed constraints on resource allocation can lead to unbalanced work distribution. To address this issue, we introduce the Community-Oriented Resource Allocation (CORA) scheme. CORA strives to resolve community restrictions, minimize the flowtime and makespan for the allocated services, while retaining a reasonable scheduler runtime for real-time resource allocation. Towards that end, we formulate the resource allocation problem as a Bipartite Graph Matching problem. Furthermore, we expose tuneable parameters that allow prioritizing flowtime or makespan, which makes CORA suitable for a wide variety of scenarios. A detailed comparative study demonstrates the leverage of CEP compared to 12 prominent edge computing platforms. Moreover, extensive simulations show that CORA outperforms six prominent heuristic-based resource allocation schemes by up to 28% and 6% in terms of average makespan and flowtime, respectively, while sustaining an adequate level of runtime.

Co-Authorship

Journal Articles

- Abdalla A. Moustafa, Sara A. Elsayed, and Hossam S. Hassanein, "Community-Oriented Edge Computing Platform," 2022, (Journal paper in preparation)

Conference Publications

- Abdalla A. Moustafa, Sara A. Elsayed, and Hossam S. Hassanein, "Community-Oriented Resource Allocation at the Extreme Edge," 2022 IEEE Global Communications Conference (GLOBECOM), Rio de Janeiro, Brazil, 2022, (pending publication)

Acknowledgment

First of all, I would like to express my gratitude to Prof. Hossam Hassanein for his unrelenting support, supervision, and guidance in teaching me the intricacies of research. Special thanks and gratitude should also be given to Dr. Sara Elsayed for the great intuitions, ideas, and for the many hours, she spent caring for every detail.

I would like to acknowledge my parents, brother, and sisters for their support and encouragement, as without them I would not have been able to have the perseverance to remain on this path or the patience to learn. My family has taught me the notion of self-reliance and self-learning throughout my childhood, and it is that notion that had led me to where I currently am.

I thank my friends and housemates Ahmed Khaled and Ahmed AbdulShakoor, for the great food they made, the late-night walks, the cooking tips, and helping me settle in Kingston.

I would also like to thank all my TRL colleagues for making my academic experience awesome. Special thanks to Sherif Azmy for providing guidance in scientific and academic matters, in addition to the students' guide to life in Kingston. I also thank the ESAQ team for the fun events they organized. Lastly, thanks are due to Basia Palmer for her patient and accurate review of this thesis.

Statement Of Originality

I hereby declare that this thesis is my own work, and to the best of my knowledge, contains no materials previously published or written by another person.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ACO	Ant Colony Optimization
API	Application Programming Interface
AR	Augmented Reality
ARM	Advanced RISC Machines
CC	Cloud Computing
CEP	Community Edge Platform
CGI	Computer-Generated Imagery
CLI	Command-Line Interface
CMU	Carnegie Mellon University
CORA	Community-Oriented Resource Allocation
CPU	Central Processing Unit
DNS	Domain Name System
EC	Edge Computing
EEDs	Extreme Edge Devices
ERD	Entity Relationship Diagram
ETC	Expected Time to Compute
FCFS	First-Come, First-Served
GPU	Graphics Processing Unit

GUI	Graphical User Interface
Iaas	Infrastructure as a Service
IIoT	Industrial Internet of Things
I/O	Input/Output
IoT	Internet of Things
ILP	Integer Linear Programming
JVM	Java Virtual Machine
KEIDS	Kubernetes-Based Energy and Interference Driven Scheduler
LAN	Local Area Network
MCMF	Minimum Cost Maximum Flow
MEC	Multi Access Edge Computing
ML	Machine Learning
MTC	Machine-Type-Communications
ORC	OctaneRender Cloud
Paas	Platform as a Service
PAT	Personal Access Token
PM	Physical Machine
PSO	Particle Swarm Optimization
QoS	Quality of Service
REST	Representational State Transfer

RISC	Reduced Instruction Set Computer
RLC	Run on Lots of Computers
RPC	Remote Procedure Call
SaaS	Software as a Service
SDK	Software Development Kit
SLA	Service Level Agreement
TOPSIS	Technique for Order of Preference by Similarity to Ideal Solution
V2X	Vehicle-to-everything
VM	Virtual Machine
VR	Virtual Reality
WOA	Whale Optimization Algorithm

List of Symbols

V	A set of all vertices in the graph representation of the resource allocation problem
S	A set of service vertices
D	A set of worker vertices
v_i	The vertex number i in set of vertices
s_i	The i^{th} service in S
d_j	The j^{th} worker in D
n	Number of services to be allocated
m	Number of workers available for allocation
ETC	A function used to estimate the execution time of a service on a worker
$\tilde{E}$	A n $\times$ m matrix where each entry represents the estimated execution time for a given service on each worker
C_i	Set of communities for vertex i
$e_{i,j}$	The weight on the edge from vertex i to vertex j
E	A set of edges resulting from estimating running all services on workers that share at least one common community
t_j	The approximate time it takes worker d_j to complete all assigned services
W_j	The prior workload running on the j^{th} worker
A_j	The set of services assigned to the j^{th} worker by the scheduler

α	A tuneable parameter that indicate how much next assignments consider the the previous ones, $0 < \alpha < 1$
β	A tuneable parameter that indicate the capacity on number of services that can be allocated to the same worker in one allocation cycle

Chapter 1

Introduction

1.1 Overview and Motivation

With the progressively adopted vision of the Internet of Things (IoT), it is foreseen that 125 billion IoT devices will be connected to the Internet by 2030 [1]. This proliferation is expected to increase the momentum of IoT applications and services that require heavy processing and stringent Quality of Service (QoS), including machine learning, augmented reality, tactile internet, and healthcare applications [2]. Cloud Computing (CC) fails to accommodate the severe QoS requirements of such applications since CC requires full transmission of an excessive amount of data to distant data centers, which can significantly increase latency and cause a huge traffic influx at backhaul links [3].

Edge Computing (EC) is a promising paradigm that can resolve the aforementioned issues by providing computing services closer to end users [4]. However, the dominant majority of existing EC platforms and models fall solely under the control of cloud service providers and/or network operators [5]. Challenging this monopoly by recycling ample yet underutilized computational resources of Extreme Edge Devices (EEDs) can democratize the edge and open a new market for more players to manufacture and administer their own edge cloud. This market can enable individuals, businesses, enterprises, and even municipalities to act as edge service providers themselves and/or monetize their computing resources. In addition, EED-enabled computing paradigms can bring the computing service closer to end users, drastically diminishing the delay [6].

Despite its advantageous impact, EED-enabled computing is less secure than infrastructure-based EC paradigms. This lack of security is due to relying on dubious machines that trigger more privacy concerns. In addition, the need to recruit many EEDs for parallel execution of a single partitioned task can trigger significant recruitment costs. Recently, EED-enabled computing platforms, such as HomeEdge [7], have addressed these problems by offloading tasks to other devices in the local network owned by the same user or entity. This approach ensures a higher level of privacy since the devices and their applications are trusted and vindicated by the user. Moreover, latency can be drastically reduced because of the proximity factor. Finally, users do not pay to use their own devices. However, restricting the scope to the local network severely limits the resource pool, reducing the utilization gains and the chance of finding a suitable device for task offloading, which can significantly reduce the quality of service (QoS). The problem of limiting the resource pool is further exacerbated by overlooking the fact that nowadays most users own more than one smart device capable of running software. The number of connected devices per person is estimated to be 9.27 by the year 2025 [7]. The resources on those devices are not always utilized to the maximum. Therefore, a significant amount of potential resources tends to be wasted.

1.2 Objectives and Contributions

Our objectives can be summarized as follows:

1. Achieving cost-efficient edge democratization by eliminating the recruitment cost of underutilized EEDs without strictly limiting the resource pool.

2. Curtailing the data privacy and device security concerns associated with task offloading in extreme edge computing.

3. Achieving a high quality of service (QoS) by minimizing the makespan and flowtime of allocated services, while maintaining a practical and adequate scheduler runtime.

4. Assessing existing edge computing platforms in terms of architecture and application domain.

The contributions of this thesis are as follows:

1. *Community Edge Platform (CEP)*: CEP addresses Objectives 1 and 2. In CEP, we interweave the notion of community with edge computing. In particular, we exploit the wide range of business, institutional, and social relationships among individuals to harvest the underutilized computational resources of user/organization-owned EEDs that form communities of trustworthy and cost-free devices. A community can be a neighborhood, a group of friends, a hospital, or devices owned by an organization in different geographic locations worldwide with different time zones or load peak times. CEP fosters the concept of service for service exchange. The goal is to create a global network where users are allowed to form separate communities of trusted users, each owning one or more devices. This significantly expands the scope of the available pool of resources while preserving privacy and eliminating any recruitment costs. In addition to communities, CEP enables the grouping of adjacent devices into clusters. This enables each device to prioritize offloading its tasks to other devices in its cluster for lower latency and even higher security while having the fallback option of offloading to devices anywhere in the world as long as they are included in one of the user's communities. CEP implementation consists of two applications, the client-side running the users' devices within the clusters and the server-side running on a server accessible by all clusters. CEP is composed of several modules including user authentication, cluster manager, community management portal, data manager, scheduler, benchmark manager, notification handler, and service execution time estimator.

   To the best of our knowledge, CEP is the first edge computing platform that leverages the notion of communities. It is based on the notion that communities are everywhere and using them can benefit all involved parties. For example, in a hospital setting, CEP can ensure that patients' data remain on-premises while providing the hospital

with the computational power needed for demanding applications. For a neighborhood or a group of friends, CEP enables users to benefit from the concept of service exchange that better utilizes the diverse EEDs capabilities and battery limitations. In the case of corporate or industry, CEP can accelerate the work cycle by allowing employees to access a much higher computational power. Additionally, they can offload jobs between branches to work around peak load time and varying working hours.

2. *Community-Oriented Resource Allocation (CORA)*: CORA addresses Objective 3. Despite its benefits, the notion of community adds another dimension to resource allocation that can cause problems for off-the-shelf schemes that ignore the restrictions imposed by communities in terms of the order of assignment. In order to achieve objective 3, we propose the Community-Oriented Resource Allocation (CORA) scheme. CORA is a new resource allocation scheme using a graph-directed approach to allocate container-based services in community-oriented EED-enabled edge computing environments. CORA also exploits the use of Minimum Cost Maximum Flow (MCMF) to achieve near-optimal results in a reasonable time.

3. *Comparative Study*: To address Objective 4, we conduct a comparative study by introducing a total of 14 features grouped into three categories, namely System architecture and deployment features, Application features, and Performance features. Then, we discuss those features in detail and use them to compare CEP with 12 other prominent extreme edge computing platforms.

1.3 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 provides a literature review of edge computing paradigms, relevant edge computing systems and tools, and resource allocation at the edge. Chapter 3 provides a description of the Community Edge Platform (CEP), an overview of the underlying system, community use cases, system architecture,

information flow, and a comparative study of CEP with respect to existing edge computing platforms. Chapter 4 introduces CORA, and provides a detailed discussion of its performance evaluation compared to six prominent heuristic-based resource allocation schemes. Finally, Chapter 5 concludes this thesis and outlines some potential future directions.

Chapter 2

Edge Computing Platforms, Systems, and Resource Allocation Schemes

2.1 Edge Computing

The tremendous growth in capable end devices is unprecedented. This paves the way for new high-demand services and applications running on end-user devices. We are witnessing a data revolution. This is reflected by the fact that almost 90% of the world population currently has smart devices, while the number of devices worldwide exceeds double the world population [8]. Around 53% of the world population are active mobile internet subscribers, and this number is expected to reach 60% by 2025 [9]. Meanwhile, services that require relativity heavy processing are gaining momentum, including video streaming, machine learning, gaming, virtual reality, augmented reality, and other interactive applications. The usage of mobile applications is expected to further contribute to this trend. Data traffic growth is predicted to reach an all-time high and a 3-fold increase in mobile network speed growth from 2018 to 2023 [10]. Furthermore, the emerging Machine-type Communications (MTC) and Internet of Things (IoT) are expected to introduce a huge number of machine connections. Due to the fact that this thriving range of diverse services is becoming an essential part of our work and entertainment, the expectations for outstanding Quality of Service (QoS) are also increasing exponentially [2].

Despite new IoT devices becoming increasingly powerful in terms of the central processing

units (CPUs), they cannot keep up with demanding applications that require massive processing in a short time. In addition, older and smaller devices that cannot compete against modern applications, and battery consumption on portable devices, still pose a major obstacle for users to run highly demanding applications on their own devices fully. This has led to the development of cloud computing (CC) technologies that enable cloud computing for mobile users. CC concept revolves around having user equipment that can exploit the computing and storage resources of powerful remote centralized clouds, accessible through the mobile network and the Internet [11]. The main advantages of CC include extending battery lifetime by offloading energy-consuming computations to cloud computers, enabling complicated applications to run on mobile devices, and providing users with higher data storage capabilities [12]. However, CC imposes excessive load at backhaul links and introduces high latency, since data is extensively sent to remote data centers [3].

Edge computing has emerged as a promising computing paradigm that enables data processing at the edge of the network. In response to the fast-growing demand and interest in high-demanding and latency-critical applications, edge computing is gaining extensive momentum [13]. Additionally, edge computing platforms and tools are blooming on the industry front [14]. Offloading the tasks to EEDs is significantly more cost-effective than CC, and it can have lower latency depending on the density of devices in a user's area. The idea behind edge computing is to bring the computing service closer to the end user [15]. Edge computing deals with critical infrastructure issues like bandwidth constraints, excessive latency, and network traffic. On the other hand, there are several important factors that can influence the adoption of edge computing, such as limited capability, connectivity, and security [16].

Edge computing is frequently compared to cloud computing. Edge computing provides low latency to mobile end users' applications compared to cloud computing. Furthermore, edge computing becomes essential as data-heavy applications increase in popularity [17]. Cloud computing is typically used when businesses need storage and computing power to run

specific applications and processes, as well as to visualize telemetry data from any location. On the other hand, edge computing is the best option when low latency, local autonomous actions, low backend traffic, and confidential data are involved. In comparison to cloud computing, several users have adopted the use of edge computing since this technology reduces the amount of data that has to travel over a network hence reducing the bandwidth costs.

Most existing EC paradigms and platforms depend on infrastructure-based edge nodes that are solely controlled by cloud service providers and/or network operators [14]. Recently, capitalizing on the pervasive proliferation of IoT devices also referred to as EEDs, and their collectively profuse computational capabilities have gained significant momentum [18]. In this extreme edge computing paradigm, the underutilized computational resources of EEDs are solicited for parallel processing [19]. This can help democratize the edge and revolutionize the tech market by enabling more players to build their own edge cloud and/or monetize their resources. In addition, it can ameliorate the offloading service by expanding the resource pool and drastically reducing latency and data traffic. However, the heterogeneity of the computation and communication resources of EEDs, their dynamic nature, cost-efficiency requirements, and privacy concerns, make it crucial to design new customized systems, tools, and solutions to overcome each of these challenges.

2.2 Edge Computing Systems and Platforms

In this section we provide a comprehensive overview of current edge computing systems and tools. Starting with Cloudlet [20], which is a good representation of utilizing micro data centers at the edge. Cloudlet was first introduced by Carnegie Mellon University (CMU) in 2009 [20]. The formal definition of a Cloudlet is a trusted, powerful computer or cluster of computers that are nearby, available to mobile devices, and well-connected to the Internet. Hence, it promotes a three-tier architecture "Mobile Device-Cloudlet-Cloud." Cloudlet can be implemented on a personal computer, a small cluster, or a low-cost server. It can be

deployed at a convenient location for end users (e.g., a school, a hospital, or a library). Cloudlet has three defining features, namely, soft state, rich resources, and proximity to users. A soft state means it does not maintain long-term state information for interactions, unlike the Cloud. Instead, it holds a temporary cache for some state information [20]. Rich resources translate to sufficient computing resources, and a stable efficient power supply, which ensures its ability to handle offloaded tasks from mobile devices. Lastly, Cloudlet must be close to users on a network and physical level. This ensures higher Qos and allows for customized services based on location context [20]. Cloudlet is limited to a specific geographical location. Therefore, it is better suited as a service provided by a public place for its users rather than a way to share resources between members of a community. The existing systems and platforms are classified into three categories depending on the owner of the worker; Organization-owned Workers, Requester-owned Workers, and User-owned Workers. We present each of these categories in the following subsections.

2.2.1 Organization-owned Workers

In this section, platforms rely on enterprise-owned machines as workers. Those machines can be owned by network providers, data centers, or cloud providers. This category involves three platforms, namely, Akraino Edge Stack, Mutable, and MobiledgeX.

2.2.1.1 Akraino Edge Stack

Akraino Edge Stack, initiated by AT&T and now hosted by Linux Foundation, is a project creating an open-source software stack that supports high-availability cloud services optimized for edge computing systems and applications [21]. For example, Akraino can be used with TARS architecture for Vehicle-to-everything (V2X) applications [21]. Figure 2.1 shows the network architecture of the Akrino stack in that case. This highlights the ability of the edge to communicate with other edges, as well as the remote data center. In addition, the policy of data offload is configurable based on different applications [21]. From the

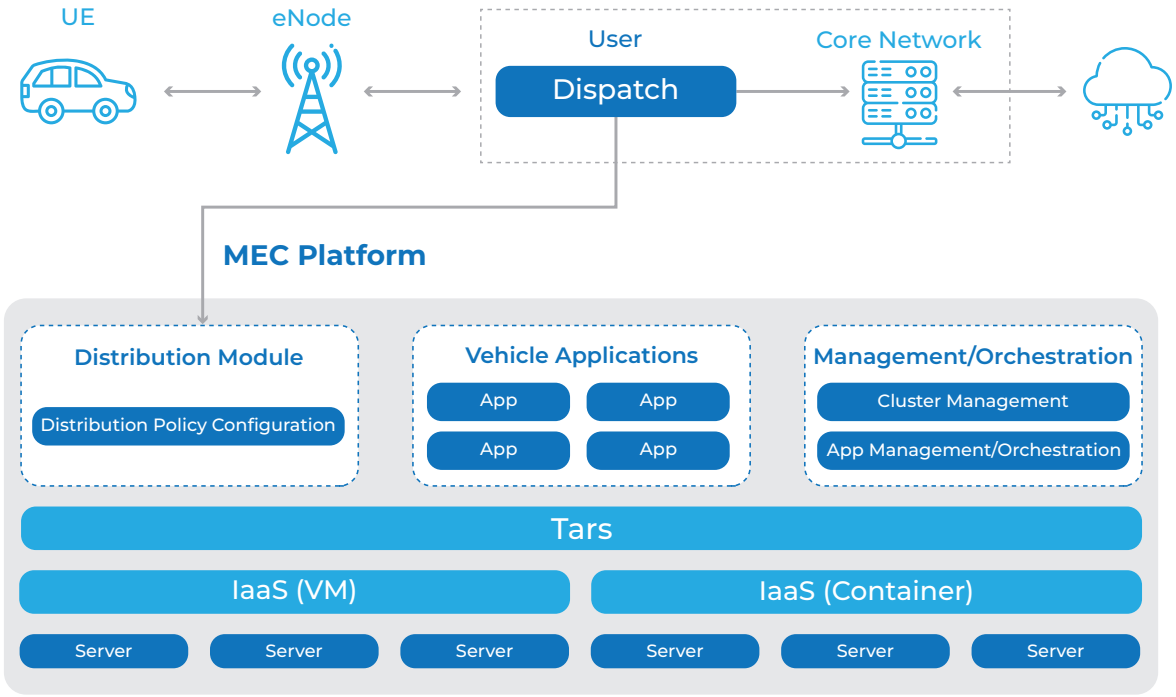

Figure 2.1: Akranio Edge Stack Architecture. Adapted from [21]

architecture viewpoint, the blueprint consists of three layers, the Infrastructure as a Service (Iaas) layer, which allows the deployment of community hardware, virtual machines, as well as containers. The platform as a Service (PaaS) layer uses the TARS framework [21]. Thus, it can support high-performance Remote Procedure Calls (RPCs), deploy services in scale-out scenarios, and provide user-friendly service management features. Lastly, the Software as a Service (SaaS) layer is on top, which contains the available applications. Akrino requires the collaboration of multiple organizations and users' data is shared with network providers. In contrast, CEP can be deployed by the user directly and it provides higher security and privacy.

2.2.1.2 Mutable

Mutable [22] is another prominent organization-owned workers platform. Mutable introduces the concept of Public Edge Cloud, which utilizes the network operators' servers by

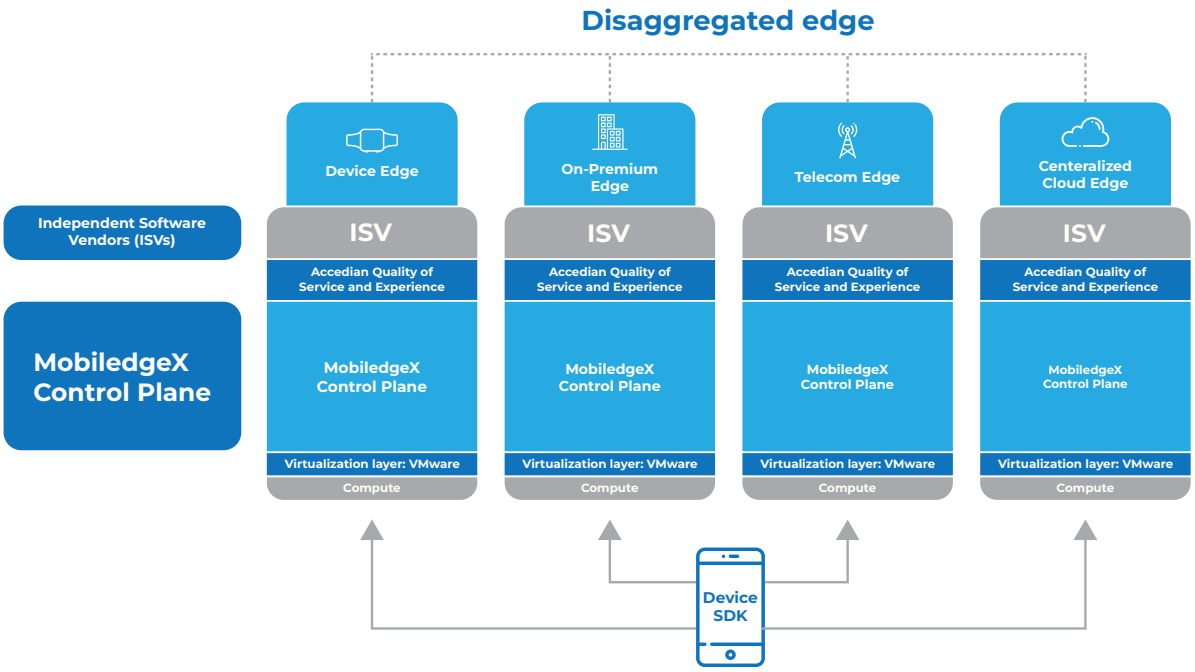

Figure 2.2: Accedian-MobiledgeX Architecture. Adapted from [23]

turning them into edge workers for next-generation applications in video and audio recognition, Virtual Reality (VR), Augmented Reality (AR), IoT, robotics, autonomous vehicle, and drones [22]. By handling the end users' requests with the network providers' servers, Mutable can achieve lower latency and higher security. Additionally, Mutable can utilize Advanced RISC Machines (ARM) and Graphics Processing Unit (GPU) architectures for a wider range of applications and offer competitive pricing compared to cloud computing [22].

2.2.1.3 MobiledgeX

Accedian and MobiledgeX [23] joined forces to enable enterprises to deliver applications to the end user on a large scale with consistent performance and maintain an acceptable level of security by utilizing the edge. The MobiledgeX Edge-Cloud R2.0 platform provided by MobiledgeX acts as the application environment deployed at data centers and commodity clouds [23]. Accedian contributes to the common trust model of network and application performance monitoring. MobiledgeX maintains ubiquitous computing by granting devel-

opers access to Application Programming Interfaces (APIs) that can utilize the underlying infrastructure capabilities and orchestration across any environment. Developers can use the provided SDK to write their application once, and then it can be deployed anywhere the MobiledgeX platform is deployed. As illustrated in Figure 2.2, Accedian and MobiledgeX provide an open-source ecosystem for low latency 5G digital experiences by utilizing edge devices. The MobiledgeX control plane unifies the management and development for different vendors and bridges the gap between EEDs, On-Premise Edge, Telcom Edge, and Centralized Cloud Edge [23].

The previously mentioned platforms, Akraino Edge Stack, MobiledgeX Edge-Cloud platform, and Mutable Public Edge Cloud, rely on enterprise-owned machines as workers. Those machines can be owned by network providers, data centers, or cloud providers. This is one step away from cloud computing because worker devices are closer to the end user and the workers are at the edge of the network. This falls under the umbrella of edge computing. Nonetheless, this group still shares some cloud computing limitations, such as privacy risk from sharing data with corporate-owned devices and relatively high costs. Those approaches provide a significantly lower latency depending on the density of available servers in the geographical location of end users and how close they are compared to cloud servers [3]. This can vary depending on the region and time, which severely limits the range and scalability of such platforms. In contrast, our proposed platform, CEP, provides a flexible deployment, good scalability, and better data privacy.

2.2.2 Requester-owned Workers

The platforms in this category utilize the requester's machines to do their own work. They only provide a way to facilitate deployment, management, and monitoring of deployed services, in addition to some prebuilt modules that can be adapted to suit the requester's needs. This category involves six platforms, namely, EdgeX Foundry, Azure IoT Edge, Apache Edgent, Kubernetes, AWS IoT Greengrass, and HomeEdge.

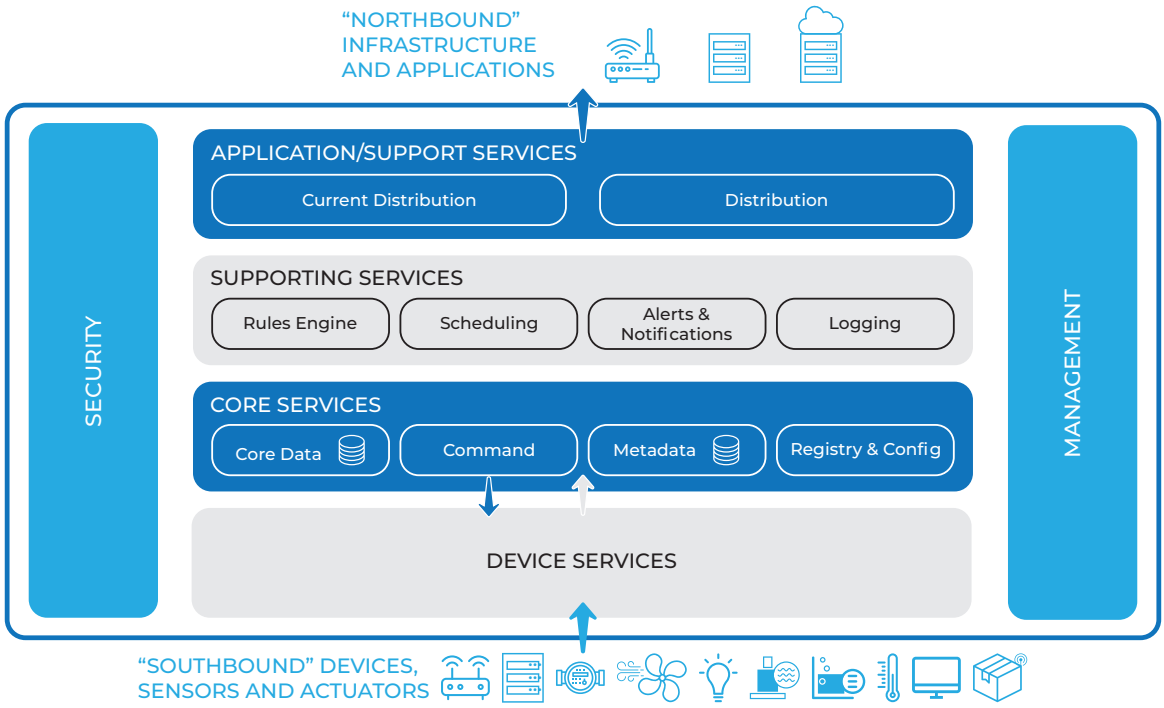

Figure 2.3: Architecture of EdgeX Foundry. Adapted from [24]

2.2.2.1 EdgeX Foundry

EdgeX Foundry is a standardized open-source interoperability framework for IoT edge computing, best suited for edge nodes, such as routers, gateways, and hubs [24]. It has the ability to connect, manage and collect data from various sensors and devices via a variety of protocols, as well as the ability to export the data to local applications at the edge of the cloud for further processing. EdgeX is agnostic to hardware, CPU architecture, operating system, and environment. Figure 2.3 illustrates the architecture of EdgeX foundry. It starts with the "South Bound" at the bottom, where all EEDs and IoT objects operate. The network connects with those devices and sensors to collect data. At the other end of the architecture, we have the "North Bound", which contains the cloud used to store, aggregate, and analyze data to turn it into useful information. EdgeX Foundry acts as a link between these two sides regardless of the differences in hardware, software, and network. EdgeX utilizes the concept of a device profile to define key information about the EED; the object

type, the collected data format, the stored data format, and the commands that can be used to manipulate this object. In addition to storage, EdgeX has an SDK that allows third-party developers to create and manage device services. The device service handles data formatting and translation of commands to operations that are executable by the devices specified by the service. As shown in Figure. 2.3, EdgeX consists of a total of six layers. Four of those are service layers, accompanied by two augmenting layers. The layers in a bottom-up order are as follows [24]:

- Device Services Layer: This layer converts the data format from collected to stored, forwards the formatted data to the core services layer, and translates the commands from the core services layer. All of those operations are done according to the profile of each device.
- Core Services Layer: This layer includes four components, core data, command, metadata, and configuration. Core data stores and manages the data collected from the EEDs. Command offers the API for command requests from the northbound to EEDs. Metadata holds and manages the metadata, such as device profiles and services. Registry and configuration allow configuration and modifications to all other microservices operating parameters.
- Supporting Services Layer: Supporting services provide edge analytics and intelligence. The rules engine handles cases where some commands need to be triggered for a specific range of collected data. Meanwhile, alerting and notifications provide the ability to notify other people or systems by email or using a Representational State Transfer (REST) callback. The scheduling component can trigger periodic operations on a specified schedule to clean the data for example. Logging stores the running information and warnings.
- Export Services Layer: This layer acts as the connecting bridge between EdgeX and the northbound. Client registration allows cloud or other applications to be included

in the data recipients list of one or more devices, While Distribution delivers the actual data to those registered clients.

- System Management: This augmenting layer handles the management operations for EdgeX, such as installation, upgrade, starting, stopping, and monitoring.
- System Security: This augmenting layer is implemented to protect the data and device information within EdgeX.

In this thesis, we use EdgeX foundry as a single module in our platform for centralized data storage. However, we build several other modules to give users the ability to run custom code inside services and not just synchronize and react to data. Additionally, we aim to enable devices to offload work to other devices in geographically remote areas if needed.

2.2.2.2 Azure IoT Edge

Azure IoT Edge is a cloud service provider managed by Microsoft Azure [25]. Its purpose is to migrate cloud analytics to edge devices. These edge devices can be gateways, routers, or any other device that can provide computing resources. The programming model of Azure IoT Edge is similar to other Azure IoT services. That is, it allows users to move their existing application hosted on Azure to the edge for lower latency. The convenience simplifies the development of edge applications. Furthermore, combined with other Azure services, it can be used to deploy advanced tasks on edge devices, such as machine learning and image recognition [25].

As depicted in Figure 2.4, Azure IoT Edge consists of three components; IoT Edge modules, IoT Edge runtime, and IoT Edge cloud interface. IoT Edge modules and IoT Edge runtime run on edge devices. Meanwhile, IoT Edge cloud-based interface runs in the cloud. IoT Edge modules are containerized instances running the user code. A module image is a docker image containing the customer code. IoT Edge runtime is the local manager on

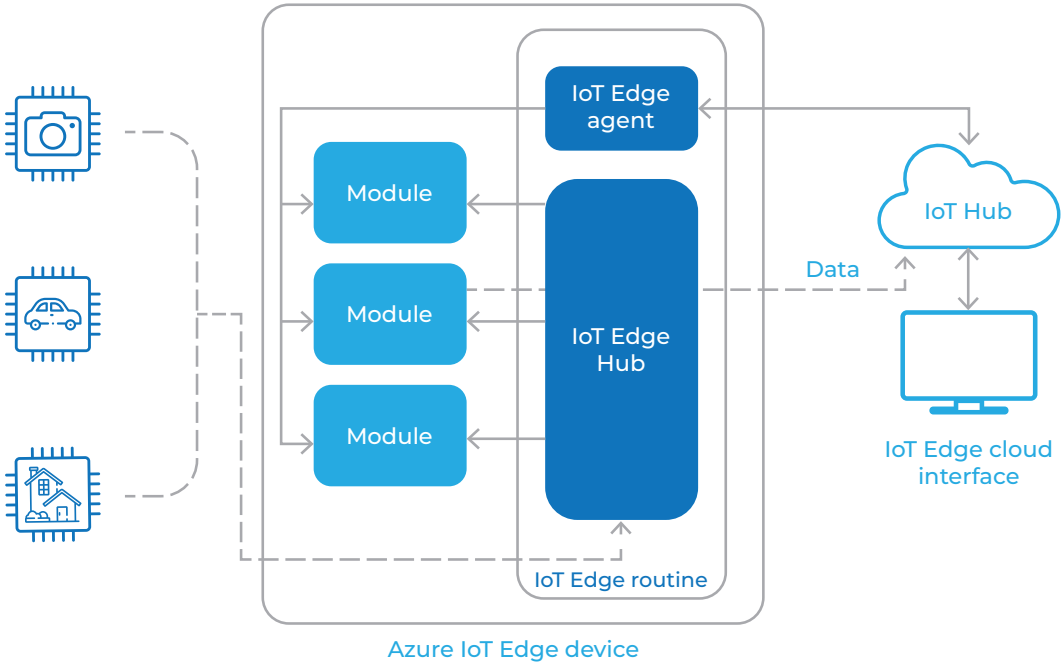

Figure 2.4: Azure IoT Edge. Adapted from [25]

EEDs. It includes two modules, IoT Edge hub, and IoT Edge agent. The IoT Edge hub is a local proxy for the hub and is a central message hub in the cloud. It allows modules to communicate with each other and send data to the IoT hub. The IoT Edge agent handles the deployment and monitoring of the IoT Edge modules. IoT Edge cloud interface acts as a portal for users to manage their applications by creating, deploying, and monitoring the applications on edge devices.

CEP differs from Azure IoT Edge, as it depends on generic containerized docker images instead of predefined templates, which opens the innovation space to include any programming language or service. Also, any public docker container can be treated as a service in CEP, whereas in IoT Edge, users are limited to a specific marketplace or their own custom code that follows a specific template available in a few programming languages. Furthermore, we do not rely on other paid services to run our system. Additionally CEP allows users to execute services on other devices not owned by the requester as long as they are part of the same community.

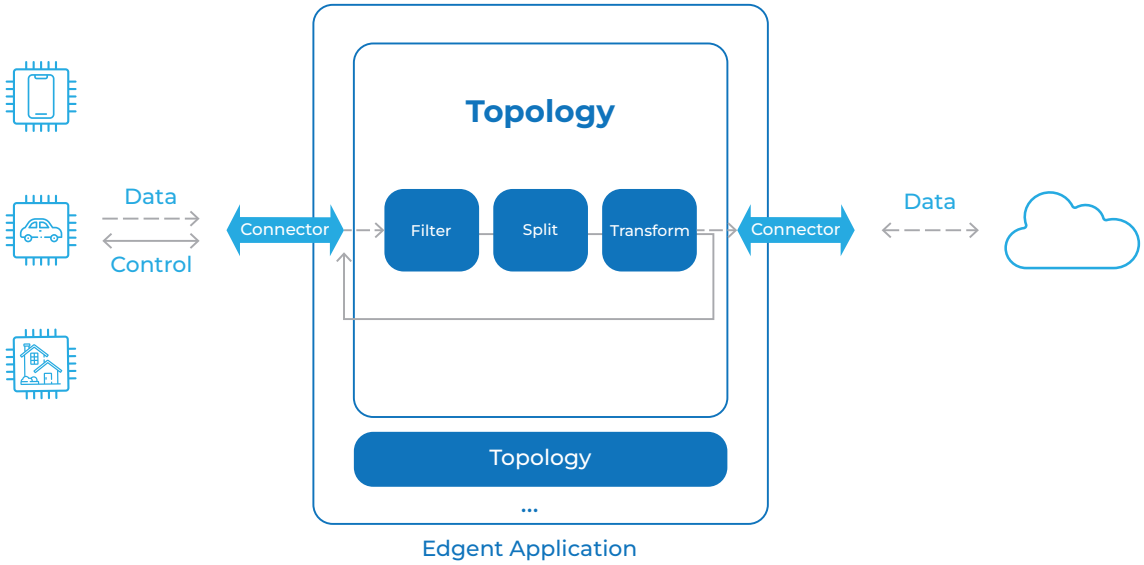

Figure 2.5: Model of Apache Edgent Applications. Adapted from [26]

2.2.2.3 Apache Edgent

Apache Edgent [26], an Apache Incubator project, is an open-source programming model and lightweight runtime for data analytics. It can be run on small devices, such as routers and gateways at the edge. Apache Edgent focuses on data analytics at the edge, thus accelerating the development of data analysis. Edgent provides API to build edge applications. Figure 2.5 demonstrates the model of such an application. Edgent uses topology as a graph to represent the processing transformation of streams of data. A connector is used to pass streams of data between the application on one side and edge devices, such as sensors and devices, or the cloud on the other side. Since Apache Edgent is focused on data analysis, the primary API allows streams to be filtered, split, transformed, or processed by any other operation in a topology. The strength of Apache Edgent as a data stream analysis tool comes at the cost of its weakness as a general platform. Edgent cannot run other types of applications and services besides stream analysis. In contrast, CEP has no restrictions on workloads or application types.

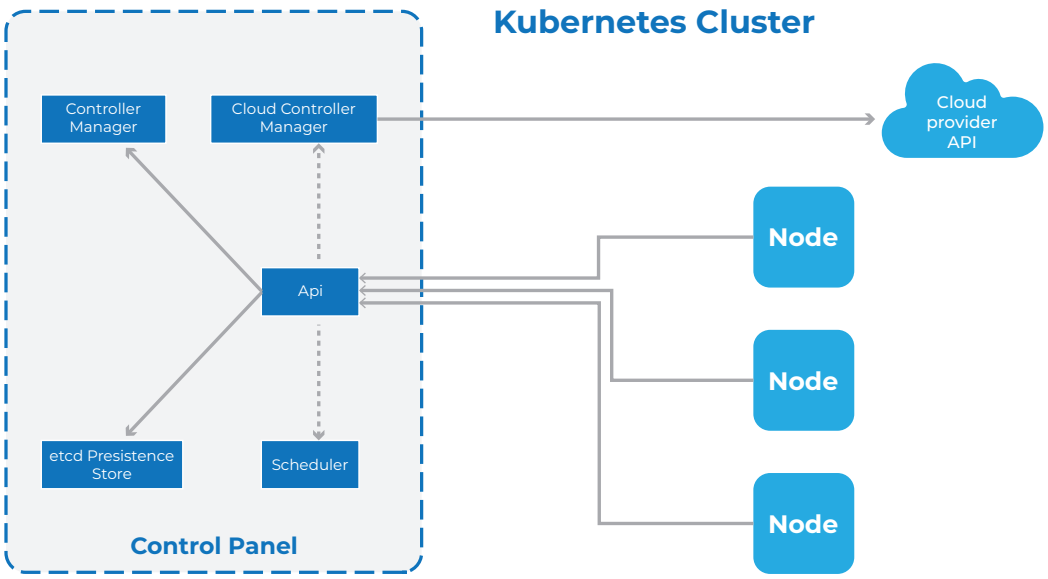

Figure 2.6: Architecture of Kubernetes Cluster. Adapted from [27]

2.2.2.4 Kubernetes

Kubernetes [27], also known as K8s, is an open-source system for automating the deployment, scaling, and management of containerized applications. It was originally designed by Google but is now maintained by the Cloud Native Computing Foundation. Figure 2.6 depicts a Kubernetes cluster that consists of a set of worker devices (i.e., nodes). These nodes run the containerized Docker applications and there is at least one node per cluster. The control plane manages the worker nodes and the Pods in the cluster. The control plane can run across multiple computers with multiple nodes per cluster, resulting in fault tolerance and high availability. The API server exposes the Kubernetes API to communicate with nodes. Etcd is a consistent key-value store used for backing up all cluster data [27]. Scheduler watches for new unassigned pods and selects a node for them to run on. The controller manager is responsible for management tasks, such as responding when nodes go down and populating the endpoints object. The Cloud controller manager links the control

plane to a cloud provider's API.

Kubernetes focuses on managing long-running services like servers or microservices, and handling reliability and scalability issues. It needs to be preconfigured on a list of servers with specific running and scaling conditions. That makes it better for software development lifecycle and hosting live applications on dedicated servers owned by the service entity, compared to our model, which focuses on short-lived services and a constantly changing network of clusters, communities, and benchmarks.

2.2.2.5 AWS IoT Greengrass

AWS IoT Greengrass [28] is an open-source edge runtime and cloud service for building, deploying, and managing device software, created and managed by Amazon. It can bring cloud processing and logic locally to edge devices and operate even with intermittent or no connection. Moreover, it can deploy and manage device software and configuration remotely and at scale without firmware updates. Greengrass relies on a core device to rally the communication between EEDs and the AWS cloud. Figure 2.7 illustrates how the data flows in the deployment architecture: first, a producer Lambda function collects the data and writes it to the local data stream. Then, an aggregator Lambda function reads the data, aggregates it, and saves the results to another local data stream. After local processing, the data can be displayed on a local display by a reader application, sent to the cloud using a stream manager, or both. In the cloud, other AWS services can be used to further analyze, combine and store the data.

Greengrass's main focus is to develop and deploy edge-ready applications to test devices of end users, it is configured to work with a list of tested device models. It requires following a structured pattern of templates and artifacts, which extends the process of developing a compatible application, by adding extra steps for developers to learn and implement. In addition, it requires paying a monthly fee per device to facilitate management and deployment.

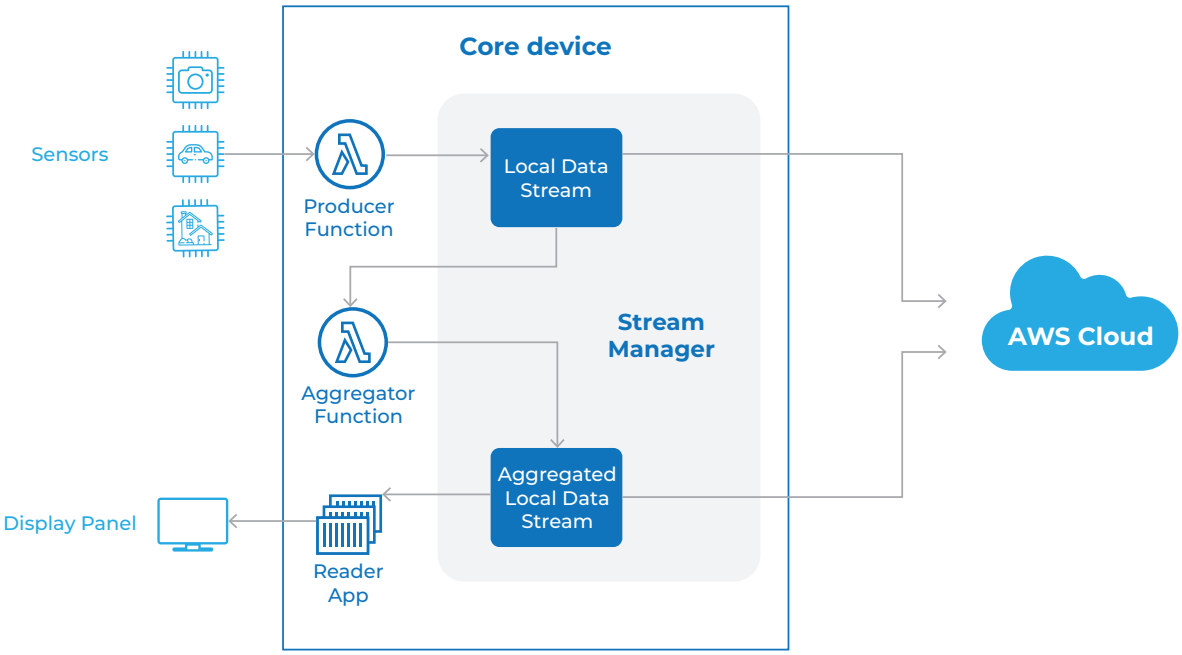

Figure 2.7: AWS IoT Greengrass Architecture. Adapted from [28]

2.2.2.6 HomeEdge

HomeEdge is an open-source edge computing framework and ecosystem that runs on a variety of devices at home or in a local network environment [29]. HomeEdge relies on Docker containers as distributed applications that can be offloaded to other machines on the same network. All the devices connected to the HomeEdge Network are considered HomeEdge devices. However, only some of the devices are considered HomeEdge, which are the devices that are capable of running containerized applications due to having enough computational power. Some devices run the orchestrator module continuously, scan the network for new devices, and assign them to nodes so other devices can communicate and offload tasks. The decentralized nature of HomeEdge allows the ecosystem to be robust and resilient against sudden device disconnects.

The decentralized control paradigm of HomeEdge limits its scalability since, for any container to be offloaded, the requester needs to query all devices benchmarks, calculate

scores and select the worker by itself. This process severely limits the pool of resources to select from, which can significantly affect the QoS. In contrast, our proposed platform, CEP, utilizes a hierarchical control paradigm to provide better scalability without sacrificing the QoS, since the resource pool is not limited to local devices.

The systems and tools mentioned above utilize the client's (i.e., requester's) machines to do their own work. They provide the users with platforms, frameworks, and services to facilitate the communication, scheduling, and monitoring of varying devices in different environments. However, the user with the work to be done still needs to rent or purchase, maintain all of the workers, and have administrative privileges over those machines in order to set up and maintain the work environment. This results in significantly better privacy and security since the data can be kept on the user's own machines in most cases. Some of the platforms eventually offload the data to external clouds, which can be maintained by the user as well for an extra fee. The systems in this category are expensive since all of the workers need to be owned or rented. Moreover, limiting the resources to devices on the same network or owned by the same user hinders the system's reliability and resource utilization potential, especially when involving inconsistent workloads. Thus, the next category of edge systems can provide a better fit for users with occasional or varying workloads searching for more affordable alternatives to cloud computing.

2.2.3 User-owned Workers

In this section, platforms execute workloads on user-owned worker machines. They allow any user to exchange their machine computational power for profit, usually, in the form of a cryptocurrency, and offer those resources to the requesters. This category involves three platforms, namely, Golem Network, iExec, and OTOY.

2.2.3.1 Golem Network

Golem Network is a global, open-source, decentralized supercomputer available to the

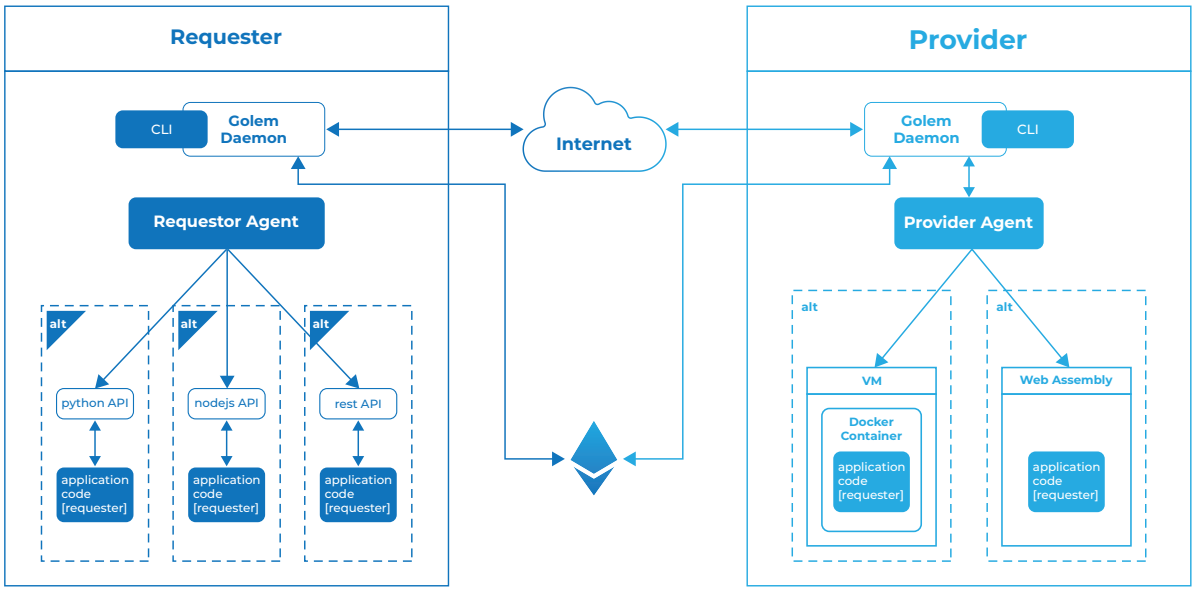

Figure 2.8: Golem Network Requesters and Providers. Adapted from [30]

public [30]. This supercomputer is a combination of the computing power of the user's devices, which can vary from personal PCs to data centers. Users in Golem are classified into two roles; requesters and providers. Figure 2.8 depicts the different possibilities for each role and how the communication between roles can be done regardless of machine type. The payment is handled using Polygon, a decentralized Ethereum scaling platform. All of the work requests and available machines are listed in a decentralized market, where users can exchange services by offering or taking other users on their offers. This model allows users to run their workloads on other people's machines for less cost compared to the cloud. Moreover, the market can scale up to a very large number of users.

Despite the clear benefits, Golem has technical barriers for new users as they have to re-implement their workload to use the provided Golem SDK. Moreover, the privacy of requesters' data and the security of providers' machines could be at risk due to running unverified workloads on untrusted machines. In contrast, CEP supports unrestricted workloads in the form of containers and ensures a higher level of security and privacy.

2.2.3.2 iExec

The iExec platform [31] acts as a bridge between cloud resource sellers with cloud resource buyers, encouraging an ecosystem of decentralized and autonomous, privacy-preserving applications. iExec's technology relies on Ethereum smart contracts and builds a virtual cloud infrastructure with high-performance computing services on-demand. iExec cloud resource providers are classified into three types; application providers, computing providers, and data providers. Application providers can monetize their applications by setting a fixed fee per usage of their software. Currently, iExec only supports tasks that run once and return a single result compared to long-running services. Dataset providers offer a fixed dataset or AI-trained model that can be used to generate data. Computing providers, also known as workers, provide their machine's computing powers to execute computational tasks in exchange for a reward in Run on Lots of Computers (RLC) tokens. Worker machines are grouped into worker pools led by Pool managers. A pool manager is a lead entity that organizes the workload, signals how many tasks it can process, and determines the price for each task. The pool manager receives a fee for managing the worker pool despite not executing the actual workload. Pool managers compete to attract workers to their worker pool by providing efficient management and guaranteeing earnings for workers.

The iExec pool managers approach lifts the burden of scheduling from the system to users acting as pool managers. However, it reduces the system's reliability even more since the pool manager acts as a single point of failure for all connected workers. iExec is also more prone to privacy and security risks due to the nature of the system's openness to new requesters and providers without a real way of verifying workloads and workers. In contrast, CEP utilizes a more robust hierarchical control paradigm, eliminates recruitment costs, and ensures a higher level of security and privacy.

2.2.3.3 OTOY

OTOY [33] was founded in 2009 with the vision of providing GPU-based software solu-

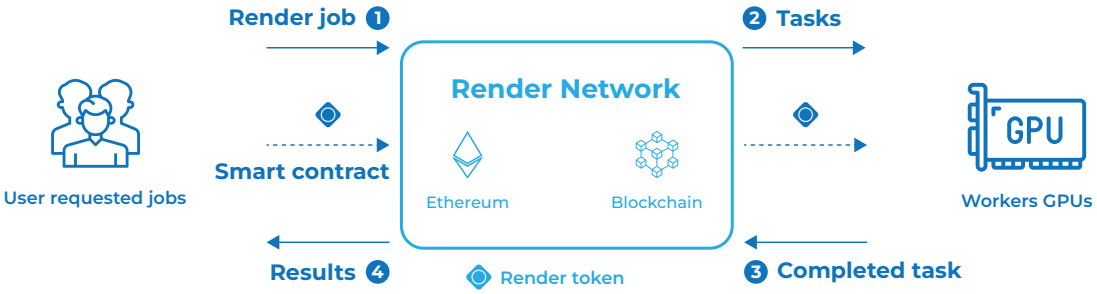

Figure 2.9: OTOY System Flow. Adapted from [32]

tions that enable the delivery of cutting-edge digital content, such as Computer-Generated Imagery (CGI) in movies and video games. Rendering is a task that takes a long time to process and can be easily distributed across different threads or machines. Thus, it is logical to make use of underutilized EEDs with ideal GPUs for rendering. OTOY, currently the leading company in cloud-based rendering, is planning on expanding to edge computing using Render Token as the primary unit to exchange rendering and streaming services. OTOY utilizes blockchain with Ethereum smart contracts technology with proof-of-render work on the OctaneRender Cloud (ORC) as illustrated in Figure 2.9. Limiting the workload to rendering reduces the risk of malicious workloads. Additionally, splitting the work into render tokens reduces the possibility of data leaks. OTOY is limited to rendering tasks and cannot be used for general tasks or services and it still requires a payment to execute the workload. In contrast, CEP does not impose restrictions on workload format or application types.

2.3 Resource Allocation at the Edge

In recent years, Docker container technology has been widely popular [34]. Commercial container orchestrators, such as Docker Swarm [35] and Google Kubernetes [27], guarantee users the freedom to execute a wide range of jobs and services. However, these orchestrators rely on simple and generic resource allocation algorithms.

Google Kubernetes [27] uses a scheduler and a work queue to assign resources. The scheduler selects a random task and assigns it to the device that has the minimum workload and maximum available resources. Another common resource allocation approach is binpack [36]. This strategy places containers on the host that has the most assigned load and that still has enough resources to run the given workload. Binpack drives resource usage up and maintains a spare capacity for running containers with significant resource requirements.

Docker Swarm [35] has a more complicated scheduling strategy, referred to as spread. The spread strategy is a heuristic approach that attempts to schedule a service based on resources available on cluster nodes; This means that services are evenly distributed across the cluster nodes as much as possible. For example, if a service is created with multiple replicas, each replicated task will be scheduled on a separate node. Scaling an existing service is the only case where the spread scheduler deviates from the normal spread procedure. The scheduler will look for a node, if one exists, that is not currently running any task for the same service being scheduled, regardless of the number of other tasks it is running overall. If no node matches these criteria, then the node with fewer tasks from the same service is selected. This approach is focused on the replicas case, where the same service belonging to the same user is run on multiple machines. This case differs from resource allocation of diverse services, each belonging to a different user.

In literature, resource allocation is typically modeled as an optimization problem [37], which finds the solution by defining the optimization parameter and choosing the optimization technique. Proposed approaches to solve the optimization problem are classified into four categories [37]; Mathematical modeling [38, 39], Heuristic [40–45], Meta-Heuristic [46–49], and Machine Learning (ML) [50, 51].

2.3.1 Mathematical Modeling Techniques

One of the most prominent mathematical modeling techniques is Integer Linear Programming (ILP) [37]. ILP is an iterative method that models the offloading problem as a linear function

with a set of linear constraints. By utilizing branch-and-bound algorithms [52], the global optimal solution could be found.

Kaur et al. [38] introduce a Kubernetes-based Energy and Interference Driven Scheduler (KEIDS), which is a multi-objective ILP formulation for Industrial Internet of Things (IIoT) applications in edge computing ecosystems. KEIDS aims to minimize the emission of carbon footprints, interference, and energy consumption; This is achieved by minimizing the energy utilization of edge nodes in IIoT for optimal utilization of green energy. Thus, KEIDS takes less time to schedule applications on the nodes maintaining minimum interference with other applications, which in turn results in optimal performance for the end users. KEIDS guarantees a significant carbon footprint and energy consumption reduction compared to the classic Kubernetes scheduler.

Overhead-aware resource allocation is proposed and designed by Lu et al. [39] for cloud container services. This resource allocation approach can be used efficiently in both offline and online settings. A job is considered to be a combination of interruptible and uninterruptible tasks limited by a specific deadline and resource requirements. Lu et al. formulated an ILP for offline resource allocation optimization. Results show that given sufficient cloud resources for job execution, the proposed approach can guarantee predefined deadlines for all defined jobs while minimizing the total interruption overhead.

The main drawback of this approach is its computational expense. The problem of scheduling resources for containers is NP-complete in complexity; hence, ILP is only suitable for small-scale systems. However, edge computing systems deal with big data with hundreds of gigabytes of information all of which are processed for time-sensitive tasks [37]. Consequently, the computation cost renders overhead-aware resource allocation impractical for large-scale systems.

2.3.2 Heuristic Techniques

Multiple heuristics have been proposed to allocate containers [40–45]. These approaches are generally faster and more scalable than the approaches mentioned above. Most of these heuristics reuse, combine, and enhance existing techniques, such as bin packing, Min-min [40], work queue [27], binpack [36], and spread [35]. However, the solutions are not guaranteed to be optimal.

Min-min [40] uses the minimum completion time as a metric, meaning that the task that can be completed the earliest is given a higher priority. Max-min [40] starts in the same way as Min-min by calculating the minimum completion time for every job but then proceeds to select the one rendering the maximum-minimum completion time.

LJFR SJFR [42] is a combination of both the Min-min and Max-min heuristics, as it alternates between them by assigning the longest job to the fastest available device, then the shortest job to the fastest available device. This sequence repeats until all jobs have been assigned.

In the Sufferage heuristic [42], priority is given to the jobs that suffer the most from not assigning them at the current step. This is done by calculating the difference between the minimum and second minimum completion times for every job and choosing the one with the maximum sufferage.

Mendes et al. [53] propose an extension to the Docker Swarm resource allocation algorithm [35]. The oversubscribing resource allocation algorithm optimizes energy efficiency, thus addressing the scarcity of resources in the edge computing environment. This optimization is achieved by improving resource utilization to levels at which energy efficiency is maximized. Oversubscribing has been shown to improve CPU and memory utilization over Spread [35] and Binpack [36]. However, achieving better resource allocation results comes at the expense of a complex decision-making algorithm that takes more time to generate a solution.

The scheme proposed in [41] is also based on Min-min. However, the target function is

custom designed to consider a physical machine's energy consumption. The authors empirically prove their approach is preferable in heterogeneous edge data centers if each container is placed on the physical machine with the least increase in energy consumption. They propose a hypothesis stating that resource utilization is balanced between two nodes when the estimated minimum energy consumption is the same.

The authors in [43, 44] propose optimization using the Technique for Order of Prioritisation by Similarity to Ideal Solution (TOPSIS) algorithm. In [43], the spread algorithm by Docker, combined with bin packing strategies and TOPSIS, are used to compromise between node parameters, the number of containers in each node, the number of available CPUs, and available memory. In [44], the authors extend their work on Kubernetes and add aggregated single rank parameters into the optimization criteria. This parameter is calculated by averaging the utilization rates of CPU, memory, disk space, and power consumption on each node.

Fu et al. [45] a progress-based scheme, called ProCon, for short-lived applications. Prior to offloading, the proposed scheme relies on instant resource utilization, as well as the current contention rate, to estimate a future completion date, which is used to minimize the future contention rate. By monitoring the progress of running jobs, ProCon can balance the resource utilization on different nodes and minimize the overall completion time, as well as the makespan.

2.3.3 Meta-heuristic Techniques

Meta-heuristic evaluation is a higher-level procedure designed to use incomplete information or limited computing power to perform a partial search that may provide a sub-optimal solution to a problem. This method has the advantage of making relatively few assumptions about the optimization being solved. The most common approaches in this category can be classified under three major classes [37]; Ant Colony Optimization (ACO) [54], genetic algorithms [55], and Particle Swarm Optimization (PSO) [56].

ACO is a population-based search algorithm. It is inspired by ants' pheromone trail laying behavior while searching for food [54]. One approach introduced by Lin et al. [46] is a multi-objective ant colony resource allocation technique. The optimization objectives include CPU and memory utilization, rate of failure, and network transmission. The experimental results show considerable improvement in the optimization of cluster service reliability, cluster load balancing, and network transmission overhead compared to existing techniques, such as Spread and GA-based approaches [46]. Nonetheless, this approach does not consider energy consumption and does not handle any matching constraints that can be mapped to community-imposed constraints.

Genetic algorithms are evolutionary search techniques based on the natural selection hypothesis. In the evolution process, the most suitable variations have a higher chance of survival and procreation. These variations are selected, and thus, their features are multiplied from generation to generation [55]. Researchers in [47] address the problems of contention and migration. They argue that deploying multiple containers on a single node leads to an overall decrease in QoS. They proceed by introducing a container balancer that periodically profiles containers and decides the optimal placement. It is also responsible for migrating containers to their appropriate nodes by the end of each profiling cycle. The container balancer utilizes genetic algorithms and runtime metrics as heuristics to optimize a custom-made objective function. Runtime metrics include CPU, memory usage, and access count for I/O blocks.

PSO is a stochastic optimization technique inspired by the movement and intelligence of swarms of birds while traveling long distances. It uses a swarm of agents acting as particles that move around the search space, searching for the solution [56]. Liu et al. [48] introduced an improved PSO algorithm. The improved algorithm utilizes inertia weight parameters and regularization techniques to increase the convergence speed of classic PSO. More importantly, it optimizes CPU and memory usage while making minimal assumptions drawn from the history of user behavior and an affinity factor, both of which are implemented in Kubernetes

native dispatcher.

Al-Moalmi et al. [49] adopt a different approach to the problem by introducing an additional layer of placing VMs on Physical Machines (PMs). This layer, combined with placing containers on VMs, forms a new optimization problem, in which, in which the Whale Optimization Algorithm (WOA) is used. WOA targets the minimization of power consumption while maximizing resource utilization. Experimental results comparing this approach against existing methods illustrate the superiority of the WOA method. Nonetheless, this method can not be extended to the case of communities in extreme edge computing because it requires full access to the workers' PMs.

2.3.4 Machine Learning-based Techniques

Machine learning is a group of algorithms that recognize patterns in data to learn and make informed decisions on newly introduced cases. Deep learning is one subfield of machine learning that has recently seen a surge in popularity due to its effectiveness in providing enough computing resources for massive data. Major applications for deep learning are computing vision and natural language processing [57].

Several attempts have been made to utilize machine learning techniques to solve the offloading optimization problem. Authors in [50] consider energy consumption as a target function to be minimized. Based on Bayesian optimization, the authors proposed a statistical online learning technique to achieve energy-aware offloading in a cloud data center. This method requires fewer data points to recognize the patterns and solve the container consolidation problem. The experimental results show that the proposed method improves the total energy consumption at the expense of service response time compared to various existing approaches.

In [51], Liu et al. propose a scheme to consolidate containers without violating the Service Level Agreement (SLA) at the Virtual Machine level (VM-level). The proposed scheme relies on linear regression to predict CPU usage in the corresponding physical machines; this, in

turn, is exploited to re-balance over-utilized and under-utilized machines. The solution is demonstrated through simulations on real workloads. The experimental results show that the container consolidation scheme, combined with usage prediction, reduces energy consumption and the number of required container migrations while complying with SLA.

To the best of our knowledge, none of the existing resource allocation schemes is optimized for communities. Some schemes address the concept of matching constraints for allocation. For example, in [58], the authors recommend an online resource allocation scheme with matching constraints that optimize for time-changing costs. However, the scheme assumes that the cost function is known at any point in time. In addition, it is based on two linear programming algorithms that have poor scalability. Such approaches still fail to address the restrictions associated with communities. In contrast to existing schemes, we propose a resource allocation scheme optimized for communities.

Chapter 3

Community Edge Platform

In this chapter, we present the Community Edge Platform (CEP). We provide an overview of the system, its use cases, and underlying architecture. Then, we conduct a comparative study to compare CEP to 12 prominent edge computing platforms.

3.1 System Overview

In CEP, an edge device can be any machine, stationary or portable. Any device running the software can act as a requester, a worker, or both. Note that we interchangeably use the terms EEDs and workers throughout this thesis. Each cluster is composed of a group of devices that are within the local network owned by the same user, with one device acting as the local scheduler for this network, called the cluster head. A cluster head acts as a communication gate between the cluster and the server that connects it to other clusters. A community is composed of one or more clusters. A community refers to a group of users that are open to exchanging services and executing offloaded tasks among each other. Note that a user can be a member of several communities.

To further illustrate the concept of clusters and communities in CEP, Figure 3.1 depicts a system of four clusters owned by a total of three users (user A, user B, and user C), which form two communities (community X and community Y). Any device that is a member of a cluster owned by user B can offload services to any device owned by users A or B. This is since user A and user B are both members of community X. In contrast, the devices of user

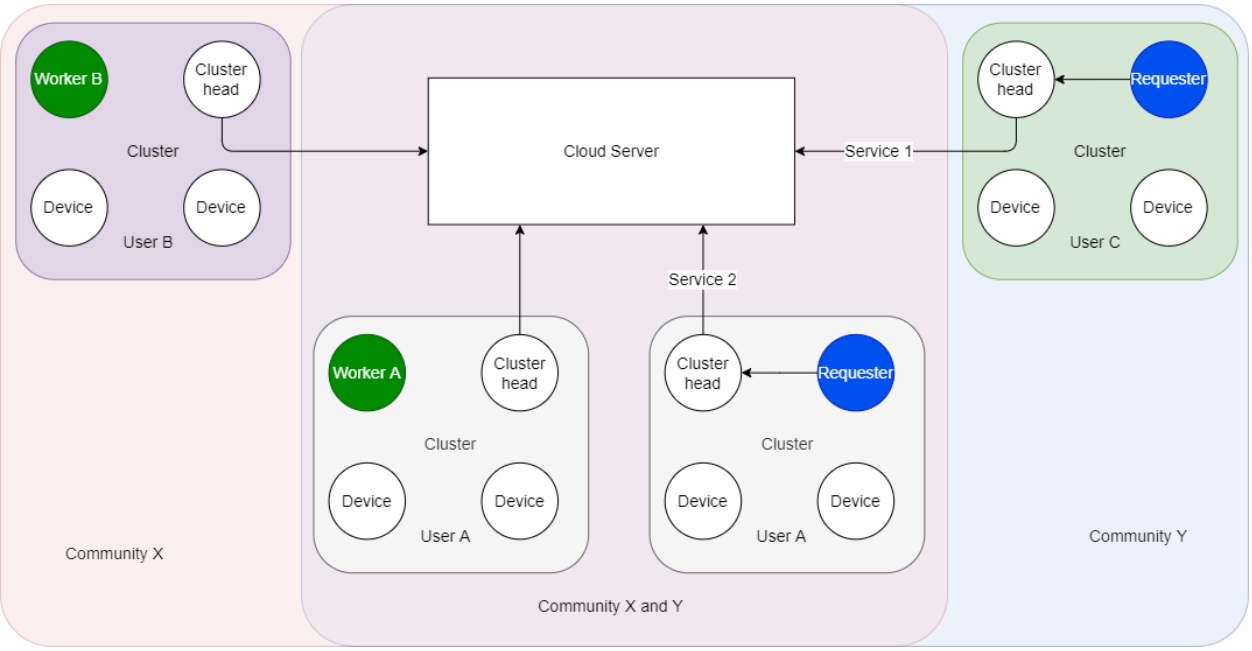

Figure 3.1: Clusters, Users, and Communities in CEP

A can exchange services with users B and C.

CEP enables requesters to offload their custom tasks, with as few limitations as possible, to trusted devices that are members of their community. This offloading is done regardless of whether the devices are within the same cluster or in a remote location, or whether all clusters in this community are owned by a single user, a single organization, or by several entities, as long as all cluster owners trust their community members. In CEP, all connected devices can act as requesters, workers, or cluster heads, depending on the configuration and the running scenario. Beginning at the cluster level, where every group of devices on a LAN can be handled by a single device known as the cluster head. This cluster head in turn has access to the server (i.e., scheduler) that can receive and schedule services between different clusters within the same community. Note that the notion of community can enable fostering a wide range of user custom services and applications.

Fostering user custom applications in CEP requires overcoming some new challenges, especially when dealing with a wide range of devices and operating systems. Therefore, we opt to use the containers ecosystem. This ecosystem allows the requester to provide almost

any source code as a task that can be offloaded in a containerized form that can run on a large, diverse group of smart devices, with no regard to the programming language, required libraries, or application domain. In addition to flexibility on the software side, this enables the services to run on any Docker Container-enabled devices. Docker containers are selected due to their quick deployment, easy management, safety, and hardware independence [37].

In CEP, each worker is a member of a cluster. Each cluster belongs to a specific user, and each user is a member of one or more communities. Let $S = \{s_1, s_2, \ldots, s_n\}$ denote the set of containerized services in the server queue that need to be offloaded to workers within communities. We assume independent services (with no inter-service data dependencies), and preemption is not allowed since the container state cannot be relocated to a new device without wasting additional resources. Each service originates from a requester within a cluster that belongs to a user with a set of valid communities, each of which has candidate workers to which the service can be offloaded safely. The set of workers available for service s_j assignment is denoted $D_j = \{d_j1, d_j2, \ldots, d_jm\}$. Each worker has its updated benchmarks, which indicate the available resources, as well as the cluster identifier. The latter can be traced back to the corresponding requester and thus to the associated set of communities. On the machine level, the assigned services are executed in a First-Come, First-Served (FCFS) order.

CEP consists of different components, including user authentication, cluster management, community management, data management, scheduling, benchmarking, notification handling, and service execution time estimation. The architecture of CEP and its underlying components and modules, as well as its use cases, are explained in the next sections.

3.2 Use Cases

In this section, we discuss three different use cases of CEP, where the notion of community can be leveraged in real-life scenarios. In each use case, we present the way devices are labeled as part of a community, how the community is created and managed, and the benefits that

CEP brings to the participating members. The three use cases are presented as follows:

1. Hospital: In a hospital setting, a community should encompass hospital-owned servers and devices with relatively high capabilities. Sensitive devices that are designed for a specific medical critical purpose can be excluded from the worker list. Certain sections or departments of the hospital can be isolated into more secure communities if they process more sensitive data or run critical jobs. The hospital IT can create and manage the hospital's communities by adding all valid devices manually. Adopting the use of CEP inside the hospital can solve the need for computational power for demanding applications, such as patients' data analysis, medical diagnosis, and treatment evaluation. At the same time, CEP ensures that the patients' data remain on-premises, which makes it less vulnerable to data breaches and data tampering.

2. Social Associations: A group of neighbors, friends, or relatives often have mutual trust in one another. Furthermore, each user can have one or more capable devices. In this environment, a community includes all EEDs, such as personal computers, laptops, phones, routers, smart cars, and all IoT devices, that belong to users with trusted social associations. Any person can create a community, and invite friends and neighbors. The invitees can see the members list and decide whether to opt into this community or not. Users can be members of several communities at the same time, and they can create several communities as well. Thus, users can benefit from service exchange that better utilizes the diverse EEDs capabilities and battery limitations.

3. Corporation: Many corporations utilize the cloud to provide computational resources to employees or clients. A community in this setting would encompass all devices owned by the corporation. This includes EEDs at separate branches and corporate-owned laptops in employees' homes. A community can be started by the IT department at the head office, where they can add the corporate mainframes. Then, they can invite other branches of ITs to do the same and invite employees to add the devices owned by

the organization in their possession. The corporation can accelerate the work cycles by granting employees access to a much higher computational power. Additionally, they can offload jobs between branches to work around peak load time and varying working hours.

Overall, members of a community are more willing to dedicate resources to members of the same community compared to complete strangers. Communities eliminate the recruitment cost since users are participating in a service-for-service exchange or helping trusted members. Lastly, this preserves data privacy and maintains a high level of security, since the workload is originating from a source (i.e., requester) trusted by the provider and running on a machine (i.e., worker) trusted by the requester.

3.3 System Architecture

CEP implementation is split into two different applications; the client-side, which runs on the user's devices within clusters, and the server-side, which runs on a server accessible by all clusters. The client-side has two modes, one for all devices to allow the machine to act as a worker or requester. The other mode is used to enable a device to run as a cluster head. The cluster head collects benchmarks and service requests from cluster members and forwards them to the server. In addition, the cluster head forwards notifications and results from the server to the original requester and has a separate scheduler module to act as the first and preferred level of offloading handling system. The server-side of the system is hosted on a public server that can be accessed by potential clusters. The server manages user authentication, keeps track of active clusters, stores and provides service data, allocates services to workers, estimates execution run time, and handles service notifications. Additionally, there are a few components that facilitate the communication between the two application sides (i.e., client and server), including a message queue, and some external service providers, such as Docker repositories.

3.3.1 Client-side (Devices and Cluster Head)

In CEP, the client-side, implemented at the devices, leverages some of the modules used in HomeEdge [29]. This is since such modules can be deployed as needed, and they provide a reliable and tested implementation for basic functionalities, such as device discovery, data storage, basic benchmarking, docker container pulling and execution, and network communication between devices. However, in its implementation, CEP ensures a centralized rather than a decentralized approach. This is because although decentralized systems require fewer network messages to communicate and are more resilient against one point of failure issues, they are more vulnerable to malicious actions. This is since any device can send request services directly to any other device. Thus, if anyone gets access to the network of the devices, they can run any code they want on all devices Furthermore, in a decentralized approach, each device needs to constantly keep track of all the other devices in the network and ask for their benchmarks whenever there is a request. This traffic increases the load on the network, which exponentially increases with the number of devices, making it unsuitable for large-scale networks.

In CEP, the cluster needs to be centralized around the cluster head to facilitate communication with the server. Hence, we design a centralized paradigm. Additionally, we implement several new modules at the cluster head to collect workers' information and requests, authenticate users, register each cluster to a given server, forward benchmarks, as well as requests, and results to the server, and forward assigned services to workers, and allocate resources within the cluster using a community-aware scheduler. The scheduler is implemented at the server and is discussed in detail later in chapter 4.

3.3.2 Server-side (Scheduler and Community Manager)

The server side acts as a communication gateway between clusters, similar to the cluster head but on a larger scale. Moreover, it manages communities, centralized databases, and execution time estimation. We opted for a stateless approach, in which each session is carried

out as if it was the first time and responses are not dependent on data from a previous session. In contrast to a stateful approach, a stateless approach provides high scalability, scheduling in high availability environments, easier caching, less need for active storage, and better failure recovery. The modules implemented at the server-side in CEP are defined as follows:

- User Authentication and Authorization: This component handles users' login and registration to the platform. Additionally, it manages the user's sessions and requests from clusters to ensure the separation between clusters in different communities and prevent malicious activities from unknown devices. A username and password are used to authenticate users to access server APIs, such as managing accounts, generating tokens, and creating and managing communities. For the clusters to automatically authenticate, a Personal Access Token (PAT) can be generated and encapsulated in the cluster head, allowing faster and automated connections between the cluster and server. Furthermore, keeping the PAT on the cluster head only increases security since it would usually be the most secured device in the cluster. PATs are less prone to brute force attacks and guessing, due to their long and random nature. Additionally, they can be reused across multiple clusters.
- Cluster Manager: This module maintains clusters, keeps track of active clusters, and identifies new connecting clusters upon requests. Thus, it can provide the scheduler with potential cluster candidates to offload tasks to.

  Figure 3.2 shows a state machine diagram highlighting the different states a cluster can be in and the possible transitions between them. A cluster is in the "active" state when its workers are ready to be assigned new services along with the services that they have already been assigned. After missing one scheduling cycle (discussed further in 3.4), the cluster changes in state to "busy", indicating that it cannot be assigned new services. However, the cluster still retains any service previously assigned to it. If the cluster connects to the server with any request, it will be moved back to the active state. Otherwise, it remains busy for x cycles, which is a configurable number, then

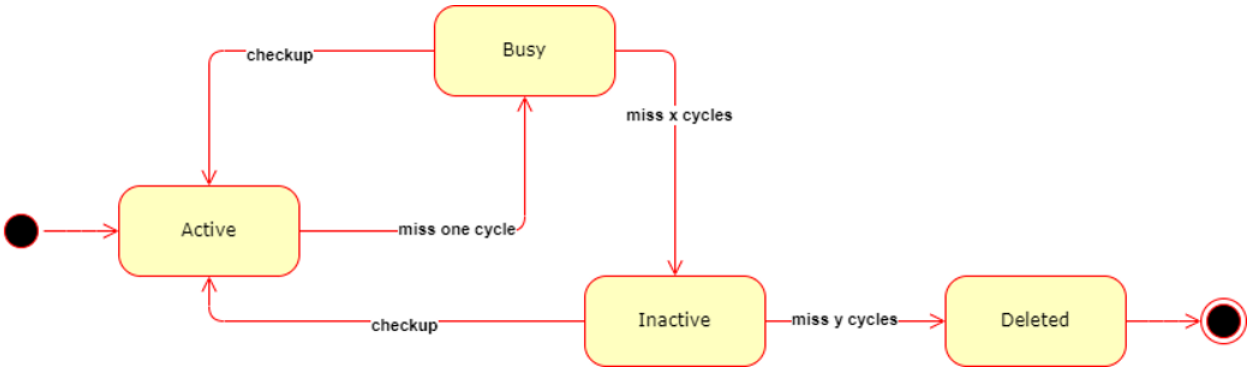

Figure 3.2: Clusters State Machine.

it is moved to an inactive state. In the inactive state, the scheduler assumes failure on the client-side and reassigns all services to a new cluster. New services cannot be assigned to an inactive cluster. Any call from the cluster will move it back to the active state directly, where it can be assigned a new service. If the cluster does not contact the server in y cycles, which is another configurable number, it will be deleted from the list of clusters to save storage space and when it connects back, it will be treated as a new cluster and assigned a new id.

- Community Management Portal: CEP provides a fully-fledged API endpoint that allows community owners to manage their communities while giving members the option to view and leave communities when desired. A more detailed explanation is given in Appendix A via the Entity Relationship Diagram in Figure A.1. Communities management is done using the invitations paradigm and it has a total of eleven different endpoints, as illustrated below:

  1. Create community: Any user can create a new community by providing certain information, including the desired community name and description.
  2. Edit community: Community owners are authorized to edit the community information.
  3. Delete community: Community owners are authorized to delete the community if it is no longer needed.

4. Leave community: Community members, except for the owner, do not have the authority to delete a community. However, they can decide to opt out of this community at any time.
5. List community: Provides a list of all communities the user is a member of. It highlights the user's relationship with each community as an owner or a member.
6. List member: Provides a list of members in a specific community, any member of a community can view this list, in addition to any user with an active invitation to the community.
7. Send invitation: Community owners can invite any member to their community. Users can be identified using ids, usernames, or emails. Any user who is not a current member of the community is eligible to be invited.
8. List user invitations: Any user can view a list of all invitations received from communities.
9. List community invitations: Community owners can view a list of actively extended invitations to users.
10. Revoke invitation: Community owners can revoke the invitation to any invited users as long as it is not responded to yet by the target user, or remove them from the community if they already accepted the invitation.
11. Respond to an invitation: Users can accept or decline any invitation after viewing the community.

- Data Manager: Data synchronization and delivery is not a simple task. That is why this module, alongside the message queue that receives messages from all local storage databases, are responsible for directing data to the correct place and retrieving it upon request. Moreover, the data need to be protected and should be provided to authorized users only. Finally, if a service has some results that need to be passed to the original requester, it can be done through this module. Sending data and receiving

results directly when handling service requests is better from a latency and privacy point of view. However, this approach introduces several issues and challenges. For example, the data needs to be sent through different levels of controllers to reach its final destination, passing through the cluster controller and the global controller. Another concern is resending all this data to a new worker in the unfortunate case of failure or disconnection. In addition to establishing and maintaining a peer-to-peer connection between devices for a long time to enable data synchronization. For all the reasons mentioned above and the fact that in some ecosystems, such as IoT, most of the data is already stored in a centralized database, we implement another approach that users can benefit from. We do that by providing a centralized data approach with two-level database architecture. The first level is the cluster level which enables low latency offloading when available. The second level is the global level, which fosters a large, centralized data storage that local cluster databases are synchronized to, on a fixed schedule using message queues. This schedule can be extended to allow on-premises data, thus increasing the level of data privacy for critical applications.

- Scheduler: One of the core modules that map services to devices within clusters. After acquiring the device benchmarks from the score manager, the scheduler needs to prioritize running on clusters owned by the same entity, followed by clusters owned by members of the same community as the requester. The implemented scheduler is required to be online to allocate resources in real time. We use a time-driven approach with time cycles rather than an event-driven approach. This is because a time-driven approach has more potential in terms of optimization, and can thus help address the constraints imposed by communities more efficiently. This optimization potential can be attributed to the larger number of services and devices available for the scheduler at the time of allocation. The scheduler needs to achieve the best possible allocation of resources. In particular, the scheduler strives to minimize makespan and flowtime, while being fast enough to run within the cycle time frame. The scheduler module, re-

ferred to as Community-Oriented Resource Allocation (CORA), is discussed in detail in chapter 4. The time of each cycle can be configured depending on the application type and the number of concurrent users. A longer cycle allows for more optimization in the allocation but it can negatively impact the latency hence it is more suited for non-real time services. For real time services, a shorter cycle or an event-driven approach can be used to minimize latency. CEP can utilize the hierarchical control paradigm by implementing different approaches at different levels. For example, CEP can use an event-driven at the cluster level to minimize latency for quick services, while utilizing CORA on the server level to benefit from the better allocation for longer services.

- Benchmark manager: This module is tasked with collecting benchmarks from active clusters regarding available devices in a group of communities. Later, it combines all the gathered results in a single list that can be passed to the service scheduler to calculate scores and make decisions. The benchmarks used in our system are as follows:
  - CPU Usage: which measures the current utilization percentage of a CPU.
  - CPU Count: number of available CPU cores.
  - CPU Frequency: the frequency of each CPU core.
  - Free Memory: the amount of memory that is currently not used.
  - Available Memory: the amount of memory that is available for allocation to a new process or to existing processes.
  - Net Bandwidth: the device network bandwidth.
  - Network MBps: a measurement of the device read and write speeds over the networks.
  - RTT: the time it takes for a message to travel back and forth between two devices (worker and scheduler).
- Notification Handler: Some services last for a long time. Thus, relying on requests' responses for the results can result in timeout exceptions. This justifies the need for

a notifications module. The service requesters need to be notified when their service is terminated, whether it failed or succeeded (discussed further in Appendix A). This module passes the final notification from the worker to the cluster head with the required information to deliver to the requester. In addition to notification passing, the handler receives periodic notifications from running clusters to ensure the services are still in progress, notify the scheduler of any network failure, and take the necessary actions.

- Estimator: Utilizing a combination of analytic benchmarking and statistical prediction, we can make use of historical knowledge from prior executions of the same services to estimate the execution time on a machine with different benchmarks, either due to machine variation or device usage during offloading [59]. Furthermore, we can extend this approach by using a two-stage machine learning approach with a model better suited for this application, such as random forest, thus significantly increasing estimation accuracy [60]. A few adaptations are needed to fit this model into our case. For example, because we have no control over the containers offloaded to embed a tracking code, we have to replace the resource tracking built into services with docker monitoring technologies. Other required adaptations are creating models for new services once they are introduced to the system and automatically selecting training variables by analyzing the service parameters. The last key adaptation is switching to an online machine learning model or retraining models with new data to retain accuracy and relevance.

3.4 System Flow

In CEP, the resource allocation is processed using a time-driven scheduler that executes once every cycle of a specified amount of time. As shown in Figure 3.3, the cycle has four events that run in order; update benchmarks, allocate jobs, assignment check, and update cluster states. During any time in the cycle, the requester can send service requests, which

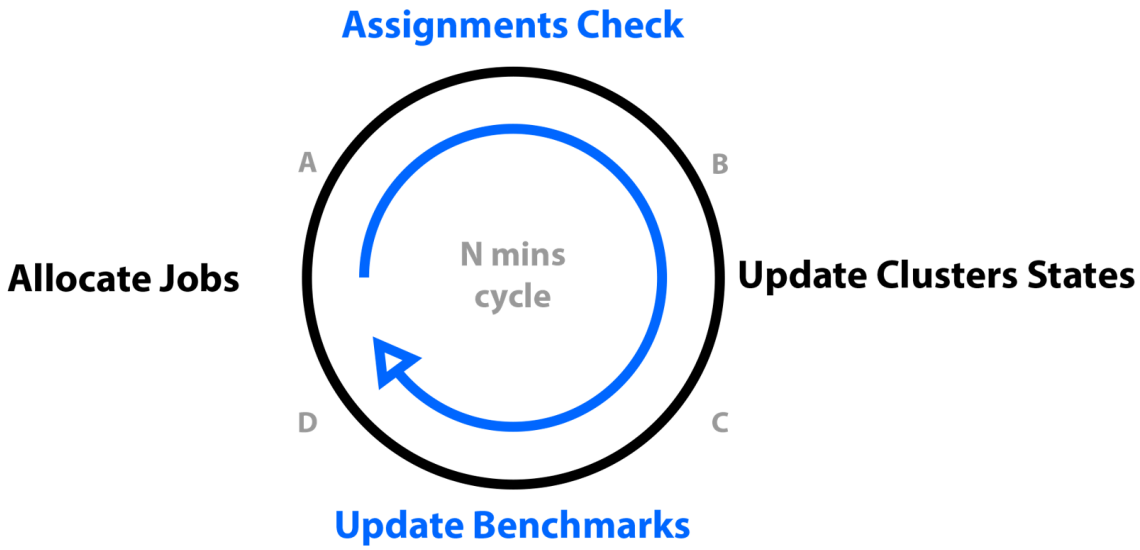

Figure 3.3: CEP Scheduler Cycle. Black events are triggered by the server, while blue events are triggered by the client

are queued in a service queue waiting to be scheduled. Starting at the update benchmarks event, all clusters are required to send the updated benchmarks of all available workers to the benchmark manager at the server. This update is managed by a cron expression that is passed to the cluster head when the cluster registers with the server. After a configurable amount of time marked by D in the diagram, the server triggers the scheduler to do the allocation. The services are pulled from the queue, and allocated to the available workers. Following this step with another time frame indicated by A, the cluster heads must check for any service assigned to their workers. Once again, this is timed using a cron expression passed by the server at cluster initialization. Workers are expected to start executing services as soon as the cluster head distributes them to workers. The following step in the cycle is triggered at the server within the cluster manager. In this step, the cluster states are updated as mentioned in section 3.3. Finally, the cycle goes back to the update benchmarks event after time C. Overall, the sum of times A, B, C, and D define the total cycle time, and all of those times are configurable and are not necessarily equal.

Figure 3.4 illustrates the flow throughout the system. The components in the figure are color coded as follows: orange highlights components related to data storage and manage-

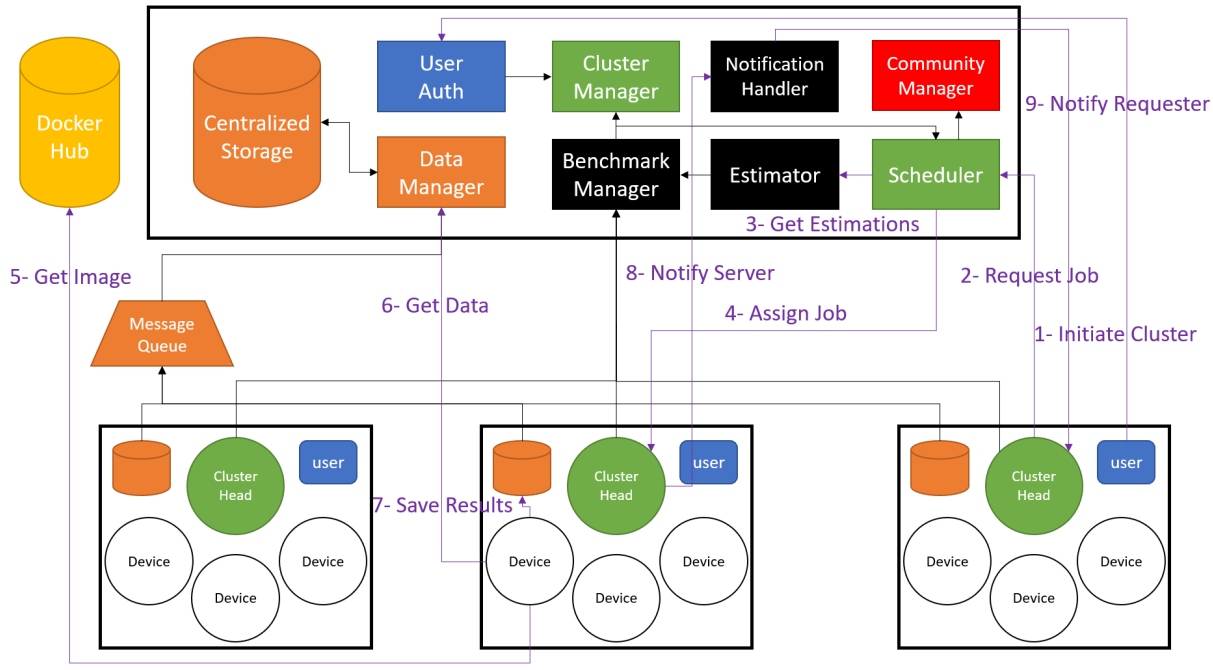

Figure 3.4: CEP Architecture and Flow

ment, green indicates components related to management, yellow maps components outside the system, Blue is for user authentication, red is the community management module, and black is for estimation and notifications.

The normal flow of a service request, cannot start unless the cluster in which the requester device exists, is initialized. After this, any service request will be directed to the cluster controller, which can either handle the request on a cluster level if possible or offload it to the global system, where the user authentication module confirms the cluster owner's identity to get a list of the owner's communities. Following this at the scheduling cycle, the service scheduler queries the benchmark manager, which keeps track of benchmarks for all active cluster workers, accessed via the cluster manager, and accumulates all the results in a single comprehensive list. This list, alongside variables of services queued for allocation, is sent to the estimator module, which calculates the execution time estimations and forwards them to the scheduler to make a decision. Finally, the scheduler passes the service information to the selected device cluster head.

When a cluster head receives the service request, it passes it to the selected worker device, which in turn gets the service image from the Docker repository, downloads any required data, and starts processing. After processing is completed, the results, if any, are uploaded to the centralized data storage, and a notification is sent to the cluster head with the state, in which the service finished processing. The cluster head forwards this notification to the notification handler, where it gets passed to the original requester.

3.5 Comparative Study

We conduct a comparative study to highlight and illustrate the differences between CEP and 12 other prominent edge computing systems (introduced in section 2.2). In total, we compare the systems based on 14 features grouped into three categories; System architecture and deployment features (Table 3.1), application features (Table 3.2) , and performance features (Table 3.3). The system architecture and deployment features include deployment location, workers ownership, control paradigm, worker OS support, and network coverage. The application features include a user access interface, workload format, application area, and cost. The performance features include data privacy, worker security, deployment, scalability, and mobility. The 14 features of the aforementioned three categories are analyzed below under their corresponding category:

I. System Architecture and Deployment Features (Table 3.1):

1. Deployed at: The deployment location shows the type of devices suitable for such a system. This feature clarifies where the controller is deployed. Deployment location is a key factor that determines the suitability of a system for certain applications, depending on the user's access to such devices. Akraino edge [21] could be deployed at cell towers, central offices, and other service providers' locations, such as wire centers. Edgex Foundry [24], Apache Edgent [26], and Azure IoT Edge [25] must be deployed at the edge nodes, such as gateways, hubs, and routers. HomeEdge [29] is

Table 3.1: Comparison of Edge Computing Platforms - Architecture Features

System	Deployed At	Workers Ownership	Control	OS support	Network Coverage
Akraino [21]	Cell towers	Network Providers	Hierarchical	Linux	Unrestricted
EdgeX Foundry [24]	Gateway	Requester	Centralized	Various OS	Local Network
HomeEdge [29]	EEDs	Requester	Decentralized	Various OS	Local Network
Azure Edge [25]	Gateway	Requester	Centralized	Various OS	Unrestricted
Apache Edgent [26]	Gateway	Requester	Centralized	Various OS	Local Network
Kubernetes [27]	EEDs and Server	Requester	Centralized	Various OS	Unrestricted
AWS Greengrass [28]	EEDs and Cloud	Requester	Hierarchical	Various OS	Unrestricted
Mobiledgex [23]	Data centers	Cloud Providers	Centralized	Various OS	Unrestricted
Mutable [22]	Network operators' servers	Network Providers	Centralized	Mutable OS	Unrestricted
Golem [30]	Cloud	Users	Decentralized	Ubuntu	Unrestricted
OTOY [33]	Cloud	Users	Centralized	Various OS	Unrestricted
iExec [31]	EEDs and Cloud	Users	Decentralized	Various OS	Unrestricted
CEP	**EEDs and Server**	**Community**	**Hierarchical**	**Various OS**	**Unrestricted**

deployed on all EEDs that need to run the system on the local network. Kubernetes [27] has a kubelet on each node, combined with a Kube controller manager on one device that acts as an entry point to distribute workload, and give the user access to monitor and manage nodes. This controller can be on-site or at a remote location, such as a cloud. AWS IoT Greengrass [28] requires a combination of a controller on one device acting as the core device and AWS services running in the cloud to monitor, manage, and, deploy new applications.

Mobiledgex [23] aims to deploy its system in data centers, commodity clouds, and multi-access edge computing (MEC) locations. Mutable [22] deploys its worker manager on network operators' servers. Golem network [30] and OTOY Rendertoken [33] maintain their core controller in the cloud where all workers have to communicate with it directly. iExec [31] allows users to operate their machines as workers' pool

managers where they schedule work for other workers, while the main server is used as a manager for pool managers, applications, and data providers. CEP has two controllers, The first one, which is deployed at the cluster head, manages resource allocation within the cluster and handles communication with the second controller. The latter is deployed on a separate server that is accessible by all clusters. This deployment location gives users control over the system instead of relying on a provider.

2. Worker Ownership: The workers that are responsible for executing the actual computation task belong to different beneficiaries in each system. Most systems in our comparison limit the workers to devices owned or managed by the requester. Such systems are EdgeX Foundry, HomeEdge, Azure IoT Edge, Apache Edgent, Kubernetes, and AWS Greengrass. A few systems deviate from this pattern; Akriano Edge Stack and Mutable utilize network provider-owned servers to perform the computation. Mobiledgex allows specific kinds of entities to provide workers, namely, the owners of data centers, clouds, and edge servers. Meanwhile, Golem, OTOY, and iExec opt for an open public approach. The workers in those systems are EEDs owned by any user willing to share their computation resources. In CEP, we limit the workers available for each requester to any device owned by a member of one of the user's communities.

3. Control: Three main control paradigms are used across the compared systems; centralized, decentralized, and hierarchical (i.e., multi-level control). In centralized systems, one server acts as the sole source of resource allocation decisions assigning workloads to workers. EdgeX Foundry, Azure IoT Edge, Apache Edgent, Kubernetes, Mobiledgex, Mutable, and OTOY all follow this approach as it is a simple yet effective way of managing a large number of devices. Decentralized systems allow each requester to choose the workers that are suitable for their needs. In HomeEdge, each device can request benchmarks from all devices on the network and decide which

device is the most suitable for the workload. This process has high reliability since no machine acts as a central point of failure for the system. However, this process cannot be scaled to a large number of machines. Golem and iExec follow an open market model where requesters and workers are free to present their offers or accept offers already extended. This approach allows the market price to adjust depending on supply and demand removing the burden of handling resource allocation from the system maintainers. Nonetheless, this approach tends to require more actions from the users, which is not optimal for new users, and user intervention is needed in case of workers' failure or migration. In contrast, Hierarchical control allows the system to fully leverage the advantages of edge computing by minimizing the amount of data transmitted over the network, allowing for faster decision-making and computation in latency-critical applications. Systems that follow this approach are Akraino, AWS IoT Greengrass, and CEP. Akraino has one controller on each edge site, and all controllers at one level are connected to a higher-level controller. AWS IoT Greengrass relies on the core device as a management point before the cloud. CEP operates a full controller and scheduler within each cluster, which can fall back to the central scheduler in cases where no suitable worker is available.

4. OS Support: EdgeX, HomeEdge, AzureEdge, Edgent, Kubernetes, AWS Greengrass, Mobiledgex, OTOY, iExec, and CEP support various OS on edge workers, including Linux, Windows, and Mac OS. Akraino is limited to Linux, which is reasonable since most servers are Linux-based. Hence, there is no need to support other OS. Golem Network only supports Ubuntu. However, in the future, they plan on extending their support. Mutable depends on its own OS, called Mutable OS, which is built on the NixOS open-source Linux-based operating system with built-in infrastructure-specific advantages.

5. Network Coverage: Network coverage is a key factor when deciding which system is suitable for a specific use case. A few systems are limited to a local network where

Table 3.2: Comparison of Edge Computing Platforms - Application Features

System	UI	Workload	Application Area	Cost
Akraino [21]	N/A	VM, Container, Bare Metal	Unrestricted	N/A
EdgeX Foundry [24]	API, GUI	Predefined Commands	IoT	Cloud Storage
HomeEdge [29]	API	Container	Unrestricted	Maintain
Azure Edge [25]	CLI, GUI	Restricted Container	Unrestricted	Azure Services
Apache Edgent [26]	API	Java Application	IoT	Cloud Rent
Kubernetes [27]	API, GUI	Container	Unrestricted	Rent/Maintan
AWS Greengrass [28]	API, GUI	AWS Lambda Function	Unrestricted	AWS Services
Mobiledgex [23]	SDK, API	Ubiquitous computing	Unrestricted	Rent
Mutable [22]	API	Container	Various	Rent
Golem [30]	SDK, CLI	Restricted Container	Unrestricted	Rent
OTOY [33]	GUI	Render Token	Rendering	Usage
iExec [31]	GUI	Restricted Container	Tasks	Usage
CEP	**API**	**Container**	**Unrestricted**	**N/A**

all the communication and logic reside, namely, EdgeX Foundry, HomeEdge, and Apache Edgent. All other systems can be extended and linked to different requesters and workers outside the local network. The three systems limited to one network can be extended as well but they require some programming or workarounds to reach this level, which is already provided in other systems by default.

II. Application Features (Table 3.2):

1. User Access Interface: How the user interacts with the system is a good indicator of its appeal to new users since the availability of one interface type can influence the success or failure of a use case. In our comparison, there are four prominent categories of user access interfaces; Application Programming Interface (API), Graphical User Interface (GUI), Command-Line Interface (CLI), and Software Development

Kit (SDK). API is the most common way for computing systems to communicate with users as it provides the functionality of the system in a human-readable and interactable format for anyone with a technical background while retaining the possibility of being integrated into other applications and user-made extensions or interfaces for the system. GUI is the best option for people with no technical background since it is easy to interact with. However, it can be limiting for the communication options available, since building any application or extension that utilizes its information requires far more effort. CLI was the standard way for people lacking programming knowledge to interact with applications by typing words or commands to a shell or a terminal. This type of interaction offers a compromise between ease of use and flexibility depending on how it is implemented. SDK is a collection of software development tools in one installable package that can be imported and extended to build applications that are able to communicate with another system easily.

Akraino Edge Stack is meant for network providers and is still under development. Hence, no user access interface has been made public yet. EdgeX Foundry, Kubernetes, and AWS IoT Greengrass support both a RESTful API alongside a graphical user dashboard interface for users with less technical training. Currently, HomeEdge, Mutable, CEP, and Apache Edgent only provide a RESTful API. In contrast, OTOY and iExec limit their interface to GUI. Azure IoT Edge accompanies its GUI with a console-like interface that allows users to monitor and deploy new applications and services. Golem and Mobiledgex require requesters to write the applications using their SDK to access the system. While Mobiledgex adds an API for workers to interact with, Golem is taking the command line direction for the same uses.

2. Workload Format: Depending on the workload requesters want to offload and whether they are willing to rewrite the workload in a different format or not, the workload can be a barrier for systems to gain new users. Akraino Edge Stack provides all

three major common methods of task offloading by providing bare metal machines, virtual machines (VMs), and containers. Edgex Foundry is limited to a number of commands defined by each device and it lacks the ability to execute anything custom provided by the requester. OTOY is limited to render tokens that are used for rendering images or videos. Apache Edgent can only run applications written on JAVA that can run on a Java Virtual Machine (JVM). AWS Greengrass allows any AWS lambda function to be sent as a workload. Hence, the user code needs to be rewritten as an AWS lambda function first. Currently, Mobiledgex does not define a clear workload method. However, they highlight the need for a ubiquitous computing method, so developers can write code once then it can run on different machines and environments. Golem, iExec, and Azure IoT Edge allow the use of containers. Even so, they force some restrictions on those containers like a specific SDK or package to use, or a template to follow so that the workload can interact with the system correctly. HomeEdge, Kubernetes, CEP, and Mutable allow normal containers without required modifications, which allows users to pack their code inside a container easily, using one of the millions of premade containers online as is, or extending to perform a new task.

3. Application Area: Apache Edgent and EdgeX Foundry both prioritize IoT edge. EdgeX Foundry is tailored for communication with various sensors and devices, while Edgent is optimized for data analysis. These systems are suitable for intelligent transportation, intelligent manufacturing, and smart city applications, where various sensors and devices generate an enormous amount of data, and not all of this data is required to be sent to the cloud. Azure IoT Edge and AWS IoT Greengrass can be considered an extension of Azure Cloud and AWS Cloud, respectively. They share an extensive application area but are limited by the computation resources of edge devices. Furthermore, they make it significantly more convenient to deploy edge applications, such as ML and image recognition to edge devices with the help of

Azure and AWS services. OTOY is limited to rendering due to its workload format. Mutable lists a large number of application areas on their site, such as cloud gaming, autonomous vehicles, IoT, smart cities, drones, data processing, video conferencing, video and audio processing, Domain Name System (DNS), and networking. Mutable does not clearly state that it can be used for any application; hence it is not unrestricted. iExec states that their system is currently limited to one-off tasks, not long-running services. For Akraino, HomeEdge, Kubernetes, CEP, Mobiledgex, and Golem where there is no restriction on the kind of applications that can run on them.

4. Cost: By cost, we mean the cost that the requester of a workload must pay to get their jobs done. This is another major deciding factor for users when choosing which platform to use [61]. HomeEdge depends on smart home devices owned and maintained by the requester. Although it can be argued that this system utilizes the already existing resources, it is not guaranteed that the user already has the underutilized computation resources required to execute all the workload. In that case, the user would need to purchase additional devices for computational power, in addition to the cost of maintenance for the existing devices. Kubernetes can connect devices across different sites including the cloud. Thus, those machines can be built and maintained by the requester or rented from a cloud provider, reducing the cost, for cases of small workloads. However, in the latter case, the system suffers from the drawbacks of cloud computing. Azure IoT Edge and AWS IoT Greengrass require requesters to pay for Azure services and AWS services, respectively. EdgeX Foundry and Apache Edgent perform some processing at the edge but later perform further analysis or store the data in the cloud, which requires continuously running costs.

   Currently, Mutable and Mobiledgex do not provide a clear pricing model. However, both promise revenue for the workers' owners, data center owners in Mobiledgex, and Network operators in Mutable. Hence, it can be safely assumed that they will charge

renting costs from the requesters. Meanwhile, Akranio Edge Stack does not mention its pricing or promise profits to the network providers, so the same assumption cannot be made. Golem Network allows requesters to rent computation power from workers to execute their code. OTOY and iExec use the pay-per-task model, where requesters pay for the number of render tokens in OTOY and per-task execution in iExec. In contrast, CEP utilizes the community concept to use the already existing computation power available within communities to execute the workload. There are definitely cases where a user's community does not have enough computation power, but the resource pool of a community is usually significantly larger compared to a single user, thus eliminating costs to purchase or rent additional machines.

III. Performance Features (Table 3.3):

1. Data Privacy: Data privacy is a major deciding factor for requesters who deal with sensitive data or source code. Data privacy indicates what parties have access to the data or source code in a raw form (without network encryption). The data shared with another party does not automatically mean it is easy to access, but it can be vulnerable to a broader range of security attacks. In contrast, systems that keep data on-premises reduce the possibility of data transmission attacks over the network, making it even more secure.

   Akraino Edge Stack and Mutable execute the workload on network providers' machines. Hence, the data and source code is available to the network provider. EdgeX Foundry usually involves offloading the data to a server afterward, and this server can be a public cloud. However, the data can be kept on-premises or offloaded to a private cloud, which increases the privacy but significantly increases the cost. Apache Edgent continues to process data at the cloud, which increases the risk of a data breach. Azure IoT Edge and AWS IoT Greengrass utilize cloud services at the edge and at the cloud, thus making the data available to cloud providers. HomeEdge

Table 3.3: Comparison of Edge Computing Platforms - Performance Features

System	Data privacy	Workers Security	Deployment	Scalability	Mobility
Akraino [21]	Shared with network provider	Run untrusted code	Static	Poor	Good
EdgeX Foundry [24]	Shared with cloud	Requester verified	Dynamic	Poor	Not Supported
HomeEdge [29]	On-premises	Requester verified	Dynamic	Not Scalable	Not Supported
Azure Edge [25]	Shared with cloud	Requester verified	Dynamic	Moderate	Poor
Apache Edgent [26]	Shared with cloud	Requester verified	Dynamic	Poor	Poor
Kubernetes [27]	Private	Requester verified	Dynamic	Good	Moderate
AWS Greengrass [28]	Shared with cloud	Requester verified	Dynamic	Moderate	Poor
Mobiledgex [23]	Shared with cloud	Verified applications	Static	Poor	Good
Mutable [22]	Shared with network provider	Verified applications	Static	Poor	Poor
Golem [30]	Shared	Run untrusted code	Dynamic	Good	Poor
OTOY [33]	Shared	Limited Functionality	Dynamic	Good	Moderate
iExec [31]	Shared	Run untrusted code	Dynamic	Good	Moderate
CEP	**Shared with Community**	**Community verified**	**Dynamic**	**Good**	**Moderate**

is a Local Area Network (LAN) system, so the data is kept on-premises unless the code specifically sends it outside.

Kubernetes machines are managed by the requester. Hence, the data is kept private from other parties unless those machines are rented from a cloud provider for example. Mobiledgex runs the code on private data centers and commodity clouds. Despite utilizing Accedian security protocols, which ensure a high level of security, the possibility of data attacks still exists due to the nature of worker devices. Golem, OTOY, and iExec. share the data and source code with devices owned by end users,

making them the least secure option. Despite CEP sharing data with workers, it ensures that the data and source code are never sent to someone outside the requester's community. Hence, CEP adds an additional level of privacy that can be controlled by the requesters themselves by defining and verifying their own communities.

2. Workers Security: Workers' security and data privacy are two sides of the same coin; on one side, we have the risks the requesters are taking by offloading the workload to another device, and on the other side there are the workers taking risks by running other people's code on their machines. Most workload offloading systems rely on virtualization technology as a baseline for workers' security by isolating the offloaded workload in a separate environment. However, there is a trade-off between allowing the requester more options in types of workload that can be offloaded and isolating the execution of the code.

   Akraino does not mention security precautions. This can be because they are building and maintaining new machines just for new edge workloads similar to cloud providers but at the edge. Hence, there is personal or organizational data or code running on those machines. EdgeX Foundry, HomeEdge, Azure IoT Edge, Apache Edgent, Kubernetes, and AWS Greengrass run the requester code on their own machines and assume that the requester has already verified the code before executing it. Thus, there is no need for extreme security measures. EdgeX Foundry is limited to predefined commands, so running custom code is not an option in the first place.

   Mobiledgex and Mutable require application verification before they can be executed on workers, which moves the responsibility to the verifying entity rather than the requesters or workers. Golem Network runs untrusted custom code. However, interacting with system resources is managed through their SDK, making it harder to work around. OTOY limits the workload to render tokens, thus significantly reducing the chance of any possible security attacks. iExec does not take any additional precautions to secure the workers besides virtualization. In contrast, CEP

depends on the mutual trust between the requester and provider of the same community. Hence, there is no need for limiting precautions, which in turn leads to more flexibility in workload options.

3. Deployment: Deployment reflects how easy and quick the system can be deployed and extended for new workers. Akraino Edge Stack is designed for network providers, who need to handle additional hardware, such as access devices and network cards, apart from computing machines. Hence, adding new workers can be slow and require much human intervention. Mobiledgex and Mutable target organization-owned devices to act as workers, which results in a complex process of adding new workers to the system. EdgeX Foundry, HomeEdge, and Kubernetes are deployed on edge devices, making them possible for end users to deploy and modify the system themselves. HomeEdge workers' extension can be done by simply running the HomeEdge application on a new machine connected to the same network. EdgeX Foundry requires communication using the API to define a new device profile. Kubernetes requires changes to the cluster manager machine in addition to the deployment of the new worker node. Azure Edge and AWS Greengrass require users to utilize the cloud-based interface to develop and deploy their applications. Golem, OTOY, iExec, and CEP allow workers to participate in the system by deploying a worker instance on their machine and login into their account, making the process relatively straightforward.

4. Scalability: This feature indicates how well and easy is it to scale the system to serve a larger number of connected users. Akraino Edge Stack theoretically can be scaled to any size. However, it requires hardware modifications and deployment of new controllers on network centers, which is a slow process. HomeEdge is limited to LANs, and it is decentralized, making it not scalable beyond a few devices. EdgeX Foundry and Apache Edgent are limited to LANs as well, but they can be extended to a larger number of users due to their central nature of control. It is worth

mentioning that EdgeX Foundry deployment is scalable since it consists of multiple microservices, which the user can dynamically add or remove to adapt to their needs. Mobiledgex and Mutable require negotiating with new providers before scaling the number of workers available, hence the poor scalability. Azure Edge and AWS Greengrass are connected to the cloud so they are highly scalable; however, offloading tasks between different groups of devices is somewhat challenging. Kubernetes, Golem, OTOY, iExec, and CEP, can easily scale to accommodate new requesters and workers, and communication between users anywhere on the network is built into those systems by default.

5. Mobility: Mobility measures how well a workload can be migrated or restarted on a new device. Akranio and Mobiledgex support migration of workload from one worker to another easily, due to the residence of workers on the network provider's infrastructure and being built with mobility in mind. Kubernetes, OTOY, iExec, and CEP, cannot migrate a workload once it starts. However, they can restart the workload on a different machine without user intervention in case of failure. Azure Edge, Apache Edgent, AWS Greengrass, Mutable, and Golem have some recovery mechanisms and failure handling that require user intervention. EdgeX Foundry and HomeEdge do not provide a process for failure recovery; instead, they require user intervention to handle failures and redeploy the workload by hand.

Based on the aforementioned comparative study, it is clear that CEP demonstrates superior architecture and deployment characteristics in terms of deployment location and worker ownership flexibility. Additionally, CEP supports various OS and can cover wide area networks. CEP control paradigm allows it to fully leverage the advantages of edge computing. For application features, CEP can eliminate additional costs, and impose no restrictions on application area or workload format. Furthermore, CEP ensures a high level of privacy, security, scalability, and deployment flexibility.

Chapter 4

Community-Oriented Resource Allocation

The notion of communities in CEP adds another dimension to resource allocation that can be challenging for schemes that overlook the restrictions imposed by communities in terms of the order of assignment. The main challenge that stems from such imposed restrictions is unbalanced work Distribution. This unbalance can happen due to a scheduler's inability to look ahead or reschedule if necessary. Figure 4.1 depicts an illustrative scenario that demonstrates this problem. In the depicted scenario, consider the case where the requester of job 1 belongs to community X and the requester of job 2 belongs to community Y. Now worker A is limited to community X. Meanwhile, worker B is included in both communities X and Y. The time it takes to execute job 2 on worker B is 65 seconds, and the time to execute job 1 on workers A and B is 60 seconds and 50 seconds, respectively.f we use one of the existing event-driven or heuristic resource allocation schemes, such as Min-min [40], job 1 will be assigned to worker B because it is 10 seconds faster to execute. The scheduler will then be forced to allocate job 2 to worker B as well. This is due to the imposed constraint prohibiting job 2 from being allocated to worker A since worker A and the requester of job 2 do not share any communities. Allocating both jobs to worker B achieves a better sum of all execution times of 115 seconds, which is desirable. However, it results in a significantly higher total time for all jobs to finish executing of 115 seconds because all jobs are allocated to the same worker. On the other hand, a better allocation would be job 1 to worker A and job 2 to worker B. This results in a slightly higher execution time sum of 125 seconds.

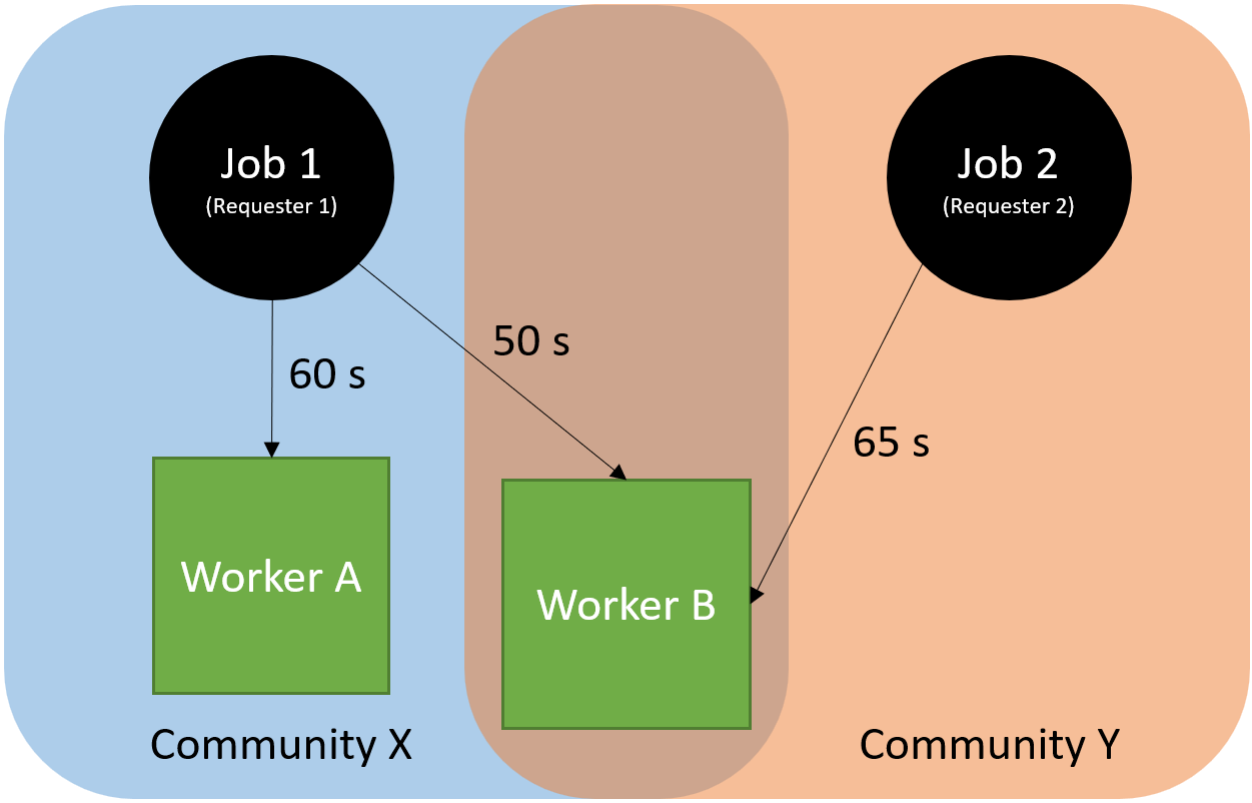

Figure 4.1: An Illustrative Scenario of Resource Allocation in Communities

Nonetheless, it achieves a significant improvement in the total time it takes for all jobs to finish executing, reducing it to only 65 seconds.

In order to address the aforementioned problem, it is imperative to have a resource allocation scheme that is tailored for community-driven scenarios. To the best of our knowledge, none of the existing resource allocation schemes is optimized for communities. As discussed in section 2.3, some approaches address matching constraints for allocation. However, Such approaches cannot be adapted to address the restrictions associated with communities. In this context, we propose the Community-Oriented Resource Allocation (CORA) scheme. CORA is based on the Min Cost Max Flow (MCMF) algorithm. In this chapter, we provide a detailed description of CORA and the underlying method used to address the challenging issue of imposed restrictions, as well as a performance evaluation of CORA compared to other prominent heuristic resource allocation schemes.

4.1 Graph Representation of Resource Allocation in CORA

In order to address the imposed restrictions issue caused by communities, CORA acknowledges the need to first determine the set of workers that are eligible to execute each service. This can be done by checking for union values between the data from a service set of communities and a worker set of communities. This problem can be better represented by a graph of vertices, which is the union of the workers and services sets, as given by equation (1). This graph can be divided into two groups, one for services and one for workers. The vertices from the first group (i.e., services) can only have an edge with vertices from the second group (i.e., workers). Thus, the graph can be defined as a bipartite graph.

$$\begin{aligned} V &= S \cup D \\ &= \{v_1, v_2, \ldots, v_n, v_{n+1}, v_{n+2}, \ldots, v_{n+m}\} \end{aligned} \tag{1}$$

Expected Time to Compute (ETC) is a function that can be used to calculate the estimated time to execute a service on a worker passed as parameters. A two-stage machine learning approach is used for this estimation as discussed in section 3.3. Using this function we construct the possible edges matrix $\tilde{E}$, which is a $n \times m$ matrix in which n is the number of services and m is the number of workers. Each entry in the matrix represents the estimated execution time of a given service on each worker as shown in equation (2).

$$[\tilde{E}] = \begin{bmatrix} ETC(s_1, d_1) & ETC(s_1, d_2) & ETC(s_1, d_3) & \ldots & ETC(s_1, d_m) \\ ETC(s_2, d_1) & ETC(s_2, d_2) & ETC(s_2, d_3) & \ldots & ETC(s_2, d_m) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ ETC(s_n, d_1) & ETC(s_n, d_2) & ETC(s_n, d_3) & \ldots & ETC(s_n, d_m) \end{bmatrix} \tag{2}$$

In the constructed bipartite graph, an edge between two vertices exists only if they share at least one community, as reflected by equation (3). Where E is a set of all valid edges that the scheduler can choose from during the resource allocating process. For each service

s_i and each worker d_j, $\tilde{E}_{ij}$ is the estimated execution time of s_i on d_j. We define C_i as the set of communities for service s_i and C_j as the community set for d_j. Each edge in the graph connects one service to one worker, where the intersection between C_i and C_j is not empty, and the edge weight is the corresponding value extracted from the ETC function.

$$E = \left\{ \tilde{E}_{ij} \mid C_i \cap C_j \neq \Phi, i \in \{1, 2, \ldots, n\}, j \in \{1, 2, \ldots, m\} \right\} \subset \tilde{E} \tag{3}$$

We define A_j as the set of services assigned to worker d_j by the scheduler. That is, for $j \in \{1, 2, \ldots, m\}$,

$$A_j \subset \left\{ i \in \{1, 2, \ldots, n\} \mid \tilde{E}_{ij} \in E \right\}. \tag{4}$$

The approximate time it takes worker d_j to complete all assigned services is denoted t_j, and is given by equation (5), where W_j represents the prior workload on the worker.

$$t_j = \sum_{i \in A_j} \tilde{E}_{ij} + W_j \tag{5}$$

CORA strives to minimize the average makespan and flowtime. Flowtime is the sum of the execution time of all services on their selected workers, as given by equation (6). Minimizing the flowtime should be the scheduler's goal since we aim to reduce the load on the workers to optimize resource utilization and maintain the maximum possible number of available workers.

$$\text{flowtime} = \sum_{i \in A_j} \sum_{j=1}^{m} \frac{\tilde{E}_{ij}}{n} \tag{6}$$

Makespan is the time needed for the system to finish executing the last service [42], which can be calculated as the longest time that any worker from the system takes to finish its assigned services, as given by equation (7). Thus, it is important to minimize this number to ensure that the average user service is completed in a timely manner. It is noteworthy that

makespan can be heavily impacted by communities if a generic resource allocation scheme that is not specifically designed for communities is used, since it can pile workload on a few workers, due to their available resources, before realizing that those workers are the only available option for the unallocated services.

$$\text{makespan} = \max_{j=1}^{m}\{t_j\} \tag{7}$$

4.2 Solution of the Bipartite Graph Matching Problem

Formulating the problem as a bipartite graph matching renders the Munkres algorithm [62] a suitable solution. The Munkres algorithm, also known as the Hungarian algorithm, is a combinatorial optimization algorithm capable of solving the classical bipartite graph matching, which is the assignment problem in polynomial time, specifically with time complexity of $O(N^3)$. However, in contrast to classical bipartite graph matching, where every vertex from group A is matched with a single vertex from group B, this is not always the optimal case for resource allocation. This is since multiple services can be assigned to the same work within the same cycle. On top of that, the Munkres algorithm is relatively slow, due to its multiple steps and calculations.

To address the previously mentioned problems, we need to take the problem a step back to graph matching, better known as the maximum flow algorithm [63]. More specifically, the multi-source multi-sink variation of the problem, where we add an imaginary source that connects to all the sources and an imaginary sink that connect to all sinks. To apply this to bipartite graph matching, we can set the capacity of those new edges to one, limiting the flow to one per source and one per sink (i.e., service). In our case, we want the graph to match the sinks (services) once in order to avoid assigning redundant work to workers. However, we strive to allow workers to have multiple services. Thus, we introduce our first parameter β, which allows the user to set the capacity of the number of services assigned per

worker. By default, this can be set to the number of services. Thus, any number of services can be assigned to the same worker if needed.

The maximum flow algorithm with the worker capacity set to one, can maximize the number of matches between workers and services, overriding previous matches if services assigned so far result in a dead-end, where some services are left unmatched. Hence, the maximum flow algorithm can result in the maximum possible number of matching, but they cannot be guaranteed to be the best matches. Replacing the pathfinding portion of the maximum flow algorithm with the shortest path alternative, such as the Bellman-Ford algorithm, elevates it to the minimum cost maximum flow (MCMF) algorithm, which guarantees matches that result in the lowest cost (i.e., service execution time). However, with the cap on source edges at the number of services. The MCMF can assign all services to one worker. The first approach of bipartite graph matching tends to optimize the makespan, while MCMF optimizes the flowtime. CORA bridges the gap between the two approaches. Whenever a service s_i is matched with a worker d_j, we change the cost on the edge between the added "super-source" and d_j, denoted $e_{s,j}$, from zero to the sum of edge weights for all services assigned to this worker, multiplied by α, as given by equation (8), where α is a tuneable parameter, such that $0 < \alpha < 1$.

$$e_{s,j} = \sum_{i \in A_j} \alpha \cdot \tilde{E}_{ij} \tag{8}$$

Algorithm 1 illustrates the resource allocation procedure in CORA. The number of all vertices in the extended MCMF graph is denoted as v. The set of all vertices includes services, workers, a super source, and a super sink. The source and sink vertices are assigned the index 0 (i.e., the first element in the set) and $v - 1$ (i.e., the last element in the set), respectively (lines 13-15). A capacity matrix, denoted $cap_{v \times v}$, is created for the max flow algorithm. This matrix indicates the available capacity on each edge in the graph. Hence its size is $v \times v$, where the element cap_{ij} indicates the capacity of the directed edge connecting vertex i with vertex j. The values for the capacity matrix are set to 1 for all edges between

Algorithm 1 CORA

Input:
Set of All Services S
Set of All Workers D
Set of All Valid Edges E
Execution Time Estimation Matrix $\tilde{E}_{n\times m} = \tilde{E}_{ij}$ $\quad n = |S|, m = |D|$
Service Allocation Penalty Factor α
Device Capacity β
Output:
Assignment Sets: A_j $\quad \forall j \in \{1, 2, \ldots, m\}$

$ALLOCATE\_RESOURCES(\tilde{E}, \alpha, \beta)$
Begin
  $v \leftarrow n + m + 2$
  $src \leftarrow 0$
  $sink \leftarrow v - 1$
  $cap_{ij} \leftarrow \begin{cases} 1, & for\ i = src\ and\ j \in \{1, 2, \ldots, n\}\ or\ \tilde{E}_{i(j-n)} \in E \\ \beta, & for\ j = sink\ and\ i \in \{n+1, n+2, \ldots, n+m\} \\ 0, & \text{otherwise} \end{cases} \quad \forall i, j \in \{0, 1, \ldots, v-1\}$
  $cost_{ij} \leftarrow \begin{cases} \tilde{E}_{i(j-n)}, & for \tilde{E}_{i(j-n)} \in E \\ -\tilde{E}_{i(j-n)}, & for \tilde{E}_{(j-n)i} \in E \\ 0, & \text{otherwise} \end{cases} \quad \forall i, j \in \{0, 1, \ldots, v-1\}$
  while True **do**
    $edges \leftarrow \left\{\tilde{E}_{ij} \mid cap_{(j+n)i} > 0, i \in \{1, 2, \ldots, n\}, j \in \{1, 2, \ldots, m\}\right\}$
    $dist, P \leftarrow BELLMAN\_FORD(edges, src, sink)$
    if $dist_{sink} \geq \infty$ **then**
      break //Cannot find a better path to sink
    $s \leftarrow |P| - 1$
    $f \leftarrow min(\{cap_{P_0P_1}, cap_{P_1P_2}, \ldots, cap_{P_{s-1}P_s}\})$
    for all $i \in \{0, 1, \ldots, s-1\}$ **do**
      $cap_{P_iP_{i+1}} \leftarrow cap_{P_iP_{i+1}} - f$
      $cap_{P_{i+1}P_i} \leftarrow cap_{P_{i+1}P_i} + f$ //Add reverse edge
      $cost_{P_isink} \leftarrow cost_{P_isink} + \alpha \times cost_{P_iP_{i+1}}$
  $A_j \leftarrow \left\{i \mid cap_{(j+n)i} = 1, i \in \{1, 2, \ldots, n\}\right\}$ $\quad \forall j \in \{1, 2, \ldots, m\}$
  return $\{A_1, A_2, \ldots, A_m\}$
End

the super source and all service vertices, in addition to any valid edge between a service and a worker that shares a community, based on equation (3). Other values in the matrix are set to β in the case of edges between the worker vertices and the sink or 0 if none of the previous conditions applies (line 16). Another matrix of the same size is required for MCMF, denoted $cost_{v \times v}$. Each element in this matrix is the edge cost, that the algorithm is trying to minimize. For any valid edge between a service vertex and a worker vertex, the value of the cost matrix is assigned to the estimated execution time of that service on that worker. The reverse edge between the worker and the service is given the cost of negative the estimated execution time. These reverse edges allow the algorithm to backtrack previously allocated resources, and reallocate them if a better matching is found. The remaining cost matrix values are set to zero (line 17).

As long as paths from the source to the sink exist, the following steps are iteratively repeated. First, a list of all current possible edges is calculated based on the capacity matrix (line 19). Second, the Bellman-Ford algorithm is executed, given the list of edges, the source, and the sink vertices, to find the current shortest path from the source to the sink. Note that other algorithms, such as Dijkstra with potentials, can be used to find the shortest path in a graph with negative edges. The Bellman-Ford algorithm returns an array of distances from the source to all reachable vertices, in addition to the shortest path from the source to the sink, represented as an array of vertex indices. The sink vertex is unreachable from the source vertex when the distance to the sink is ∞ or more, hence we break from the main loop and return the services allocated so far (lines 20-22). Third, the last index in the path array is set to a variable denoted s, while a variable, denoted f is set to the lowest capacity on the found path (i.e., path bottleneck) (lines 23 & 24). Fourth, we loop over the shortest path found, reducing the capacity of each edge along the path by f, and increasing the capacity of each reverse edge by the same amount. Again, this is done to allow the backtracking of this assignment later if a better allocation could be made. This backtracking can be achieved due to the combination of reverse edges in capacity, and negative costs for the reverse edges.

The cost of edges connecting the workers to the sink vertex is updated using the stacking penalty parameter α. This update works for forward and reverses edges to adjust the current cost of allocating another service to this worker since any allocation to the worker has to pass through the edge connecting the worker vertex to the sink (lines 25-28). After the loop, any reverse capacity of 1 means that this service is allocated to this worker since the reverse capacity is added after the allocation (line 29). A set of allocation sets is returned, so it can be used to pass service information to the assigned workers.

4.3 Simulation Setup

We evaluate the performance of CORA compared to six prominent heuristic-based resource allocation schemes that could be tweaked to fit into the community-oriented EED-based computing environments. These schemes are Work Queue [27], Min-min [40], Max-min [40], LJFR SJFR [42], and Sufferage [42]. In addition, we compare CORA to the Munkres algorithm [62]. We use the following performance metrics: 1) average makespan, 2) average flowtime, and 3) average scheduler runtime, which is the average time it takes the resource allocation module to allocate all services to available devices.

The data generated and used in this simulation is the $\tilde{E}$ matrix, service set of communities, and worker set of communities. This data is generated using a uniform distribution with some parameters to distinguish between different scenarios. Unless otherwise specified, the number of services and workers is set to 600 each. The simulation is repeated ten different times for each instance, and a 95% confidence interval is used to highlight the variation between runs. In total, we generate and simulate 16 classes of instances by varying the following parameters:

- Worker heterogeneity: represents the variance among the execution times of a given service across all workers. This value can range between 1 and 50 in our simulation. In extreme edge computing environments, worker heterogeneity is typically high, due to the wide range of available workers. However, in some cases of corporate-owned

devices, they can be relatively close or even identical.

- Service heterogeneity: describes the possible variation among the execution times of services on a given worker, where a low variance indicates that a given worker can run all services with a small execution time gap, while a high variance indicates a case of vastly dissimilar services. We set service heterogeneity between 1 and 50.
- $\tilde{E}$ matrix consistency: can have one of two possible values; consistent or inconsistent. In a consistent $\tilde{E}$ matrix, a worker d_j executes any service s_i faster than d_k. In this case, worker d_j executes all services faster than worker d_k. In contrast, an inconsistent matrix characterizes the case where worker d_j may be faster than worker d_k for some services and slower for others.
- Community density: defines the number of edges in the graph and the number of unique communities available in the environment at the time of simulation, which translates to the possibility of community overlapping between services and workers. This, in turn, results in matching with fewer constraints. The number of unique communities is scaled up and down with the total number of services, and it ranges between 20 and 100, while the number of edges per node can vary between 2 and 10.

4.4 Results and Analysis

Tables 4.1 and 4.2 show the makespan and flowtime results obtained from running the scheduler evaluations for every instance, respectively. The results shown for every instance are the average of ten different simulations. The mean value per approach across all instance classes within the table is shown in the last row. The first column indicates the instance name, in a format that is explained in the caption below the table, and the remaining columns indicate the values of makespan in Table 4.1 and flowtime in Table 4.2. The underlined bold value is the minimum yield value of makespan (Table 4.1) or flowtime (Table 4.2) per instance.

Table 4.1: Makespan in Seconds of CORA, Min-min, Max-min, LJFR_SJFR, WorkQueue, Sufferage, and Munkres for 600 Services

Instance	Min-min	Max-min	LJFR_SJFR	WorkQueue	Sufferage	CORA	Munkres
lo-lo-c-s	936	922	902	995	2663	**691**	**691**
lo-lo-c-d	892	884	799	965	1883	**692**	**692**
lo-lo-i-s	1092	1066	1065	1218	1471	**589**	**589**
lo-lo-i-d	1054	1034	1035	1180	1103	**563**	**563**
lo-hi-c-s	959	943	877	1047	1997	**697**	**697**
lo-hi-c-d	918	896	799	936	1519	**696**	**696**
lo-hi-i-s	1106	1073	1083	1268	1381	**616**	**616**
lo-hi-i-d	1081	1046	1053	1205	1187	**585**	**585**
hi-lo-c-s	975	955	893	1021	2128	**699**	**699**
hi-lo-c-d	906	897	801	932	2160	**691**	**691**
hi-lo-i-s	1095	1071	1067	1248	1689	**614**	**614**
hi-lo-i-d	1049	1053	1043	1186	1199	**583**	**583**
hi-hi-c-s	1736	1517	1545	8840	3384	**1289**	1380
hi-hi-c-d	1416	1259	1314	9022	2192	**997**	1013
hi-hi-i-s	1906	1640	1668	9060	2774	**1419**	1514
hi-hi-i-d	1586	1368	1403	9250	1262	1066	**1057**
Mean	1169	1102	1084	3086	1875	**780**	793

The instance format is ww-xx-y-z, where ww is service heterogeneity (high/low), xx is worker heterogeneity (high/low), y is matrix consistency (consistent/inconsistent), and z is community density(sparse/dense). The ms. stands for makespan while ft. indicates the flowtime column under each approach

As depicted in Table 4.1, CORA and Munkres share the minimum makespan across all instances. This is due to their common nature of prioritizing distributing services over different workers. In addition, they both share the possibility of reallocating to consider the allocation order that prevents service stacking that significantly increases makespan. However, it is not always desirable to have one service per worker. This idea is highlighted by the instances with high service heterogeneity and service heterogeneity alike. This can be attributed to the fact that high worker heterogeneity increases the possibility of having workers that are powerful enough to execute multiple services before other workers execute a single one. Another reason is high service heterogeneity, which can result in short services that can be stacked and completed before or close to relatively long services. In those four instances of high worker heterogeneity accompanied by high service heterogeneity, we can

Table 4.2: Flowtime in Seconds of CORA, Min-min, Max-min, LJFR_SJFR, WorkQueue, Sufferage, and Munkres for 600 Services

Instance	Min-min	Max-min	LJFR_SJFR	WorkQueue	Sufferage	CORA	Munkres
lo-lo-c-s	505	514	511	529	**501**	503	503
lo-lo-c-d	505	513	511	531	**502**	**502**	**502**
lo-lo-i-s	528	539	537	600	527	**523**	**523**
lo-lo-i-d	523	531	528	599	520	**517**	**517**
lo-hi-c-s	510	520	517	546	**504**	505	505
lo-hi-c-d	508	519	515	548	505	**503**	**503**
lo-hi-i-s	534	545	543	617	532	**529**	**529**
lo-hi-i-d	529	535	534	616	524	**522**	**522**
hi-lo-c-s	510	522	518	545	507	**505**	**505**
hi-lo-c-d	508	521	515	547	505	**503**	**503**
hi-lo-i-s	534	553	546	615	535	**530**	**530**
hi-lo-i-d	529	544	538	616	527	**522**	**522**
hi-hi-c-s	**607**	697	661	2109	619	623	639
hi-hi-c-d	**574**	646	621	2162	587	579	582
hi-hi-i-s	**679**	767	744	2224	696	690	699
hi-hi-i-d	638	710	698	2229	651	**637**	638
Mean	545	573	565	977	546	**543**	545

The instance format is ww-xx-y-z, where ww is service heterogeneity (high/low), xx is worker heterogeneity (high/low), y is matrix consistency (consistent/inconsistent), and z is community density(sparse/dense). The ms. stands for makespan while ft. indicates the flowtime column under each approach

see CORA's ability to adapt and achieve even lower makespan results. On average, CORA achieves a 28% lead compared to the second-best heuristic (LJFR_SJFR). The Munkres algorithm is not considered in this comparison because of its significantly long runtime (i.e., high complexity), which will be discussed later. This is in addition to the fact that Munkres lacks the ability to assign multiple services to a worker, which significantly reduces its potential for generalization.

In Table 4.2, the flowtime of most approaches, with the exception of WorkQueue, is observed to be relatively close to each other. However, CORA still outperforms all other approaches with a negligible difference ahead of the second-best approach (Min-min). Moreover, CORA retains the minimum flowtime for 11 out of 16 instances, which means that in those cases, it is the best option on the two fronts (i.e., makespan and flowtime). Min-min

performs better in high heterogeneity cases because CORA tends to find more optimization opportunities in those instances. Note that due to the default values of α being set to 1 and β to n, CORA prioritizes minimizing the average makespan over flowtime. Other configurations for CORA that showcase better flowtime results are discussed later in this section.

As observed in Tables 4.1 and 4.2, there is a trade-off between makespan and flowtime that the other heuristics have to make; the Sufferage and Min-min schemes for instance have a slight lead in average flowtime, while they fall behind in terms of makespan. Compared to CORA, the Sufferage and Min-min approaches have around 140% and 50% higher average makespan compared to CORA. On the other hand, LJFR_SJFR and Max-min retain a relatively low makespan but have to sacrifice some flowtime, with an average flowtime 5% higher than CORA. Breaking this pattern are Munkres and CORA.

4.4.1 Effect of Varying the Number of Services

Figures 4.2 and 4.3 depict the average makespan, and average flowtime, respectively, over a varying number of services; 200, 400, 600, 800, and 1000. We ran the same 16 instances of varying heterogeneity, matrix consistency, and density. The number of unique communities is set to $n/10$, where n is the number of services (generated by n requesters), while maintaining a varying graph density between instances since this is more representative of real-life cases.

Figure 4.2 illustrates the change in average makespan in seconds when changing the number of services. As depicted in Figure 4.2, as the number of services increases, the average makespan increases slowly in WorkQueue, Min-min, LJFR_SJFR, Max-min, CORA, and Munkres. This increase can be explained by the fact that makespan is capped at the maximum value of the minimum execution time for each service. Since it will take at least this time if this service is allocated to a worker with no other work alongside it, there is no way to go lower than this cap. Sufferage witnesses a relatively large increase in makespan as the number of services increases. This can be attributed to the effect of increasing the

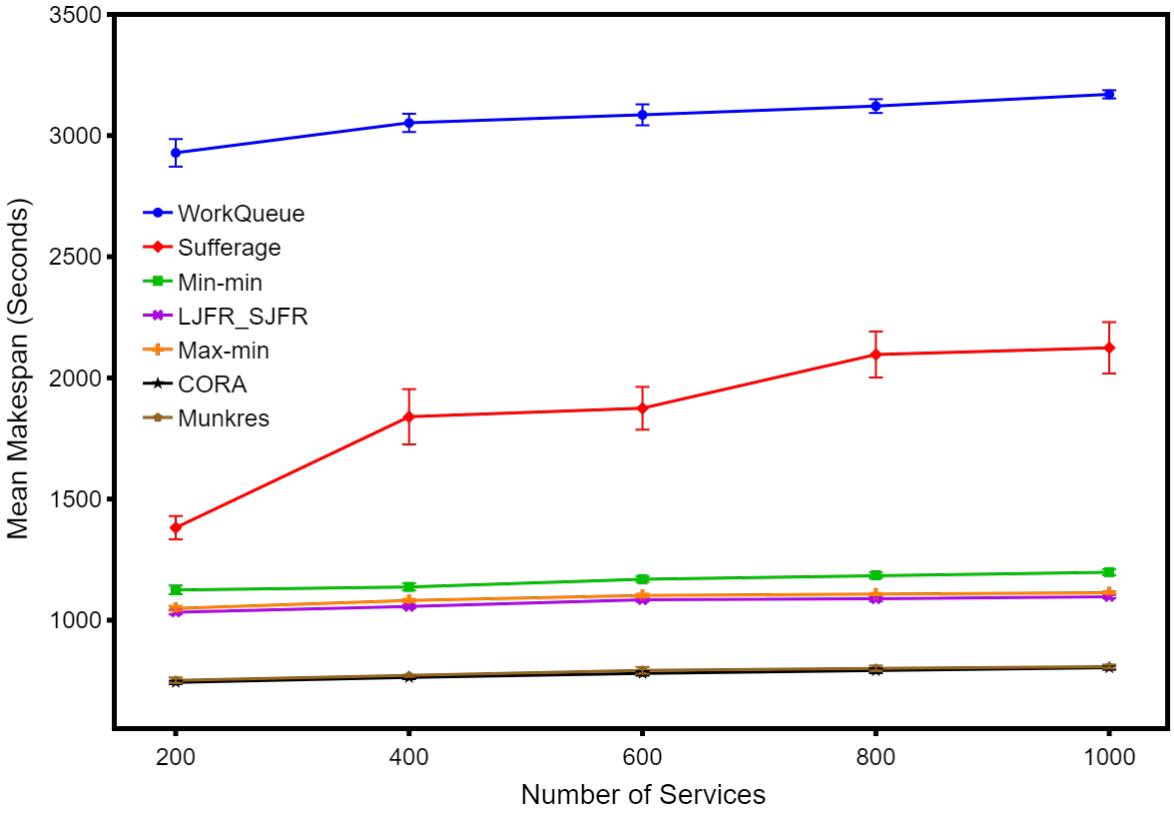

Figure 4.2: Average makespan of CORA, Min-min, Max-min, LJFR_SJFR, WorkQueue, Sufferage, and Munkres over a varying number of services

number of communities, which leads to a higher chance of isolated services with a limited number of available devices. Those isolated services tend to be a challenge for the Sufferage approach due to its process of selecting the next service to be allocated.

The markers in Figure 4.2 show the mean of all the simulations. The variations between simulations are depicted by the confidence interval on each marker using a 95% confidence interval. We can see that the confidence interval is insignificant (less than 2%) in WorkQueue, Min-min, LJFR_SJFR, Max-min, CORA, and Munkres. WorkQueue has a slightly larger interval due to the randomness involved with the order of assignment of services. In the Sufferage scheme, the confidence interval reaches 6% of the mean makespan, with an average of 4.8% across a varying number of services. This large interval can be explained by the volatile nature of the Sufferage scheme since the ordering of service assignment relies on

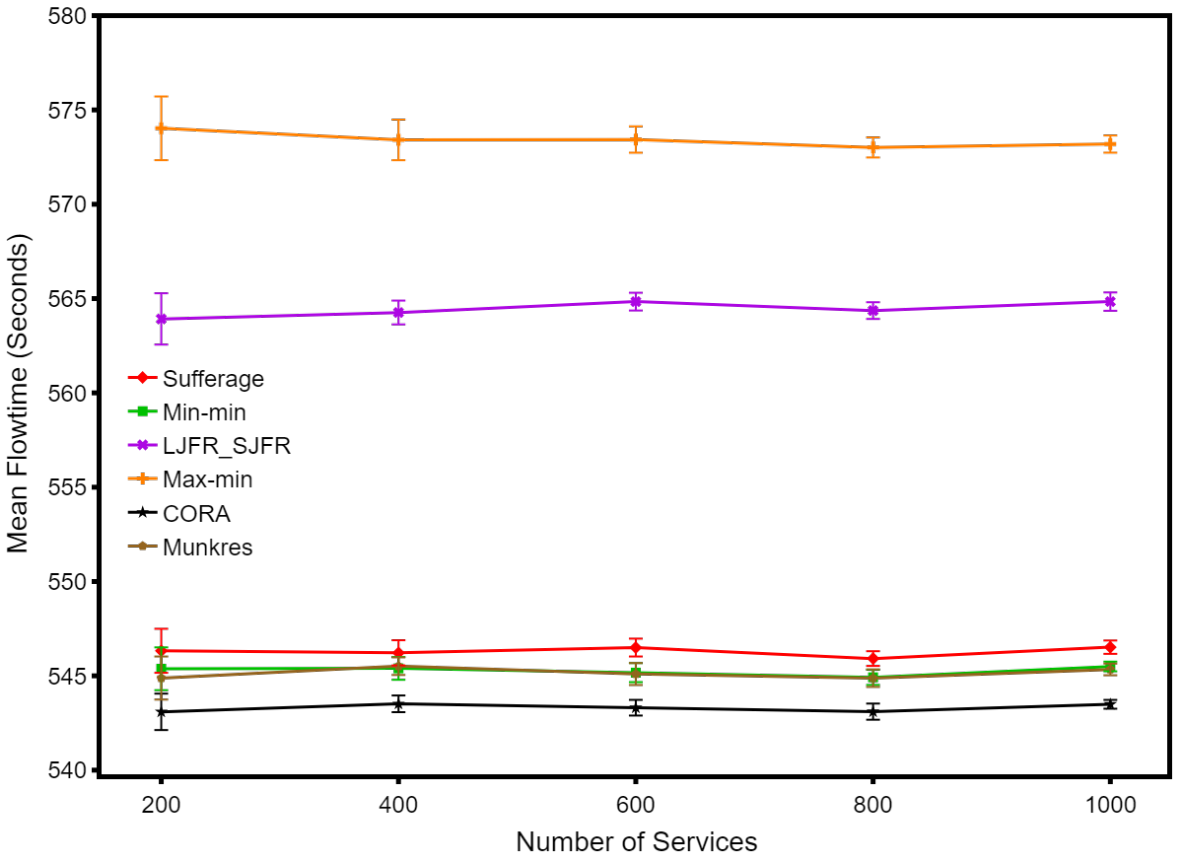

Figure 4.3: Average flowtime of CORA, Min-min, Max-min, LJFR_SJFR, Sufferage, and Munkres over a varying number of services

the difference between the best and second best possible assignment of a service. For most services, there is not that much variety across different devices, especially after most of the services are already assigned. This volatile nature leads to its performance being more influenced by the input than the other approaches.

Figure 4.3 depicts the effect of varying the number of services on the average flowtime. It shows that the flowtime remains almost the same acrosst different number of services. This is since varying the number of services does not affect the average time a service takes to execute on any suitable worker. Hence, the same ratio of services that get allocated to less optimal workers remains the same, resulting in the same average flowtime across all services within the equivalent instances.

As shown in Figure 4.3, a maximum of 1.2% confidence interval is yielded at 200 services in terms of flowtime in all approaches, slowly decreasing as the number of services increases, reaching less than 0.5% at 1000 services. This decrease is because the figure shows the mean flowtime of all services, which converges to a closer average with a higher number of services, since one service assignment, either good or bad, has less influence on the average value in a larger number of services. Since the confidence intervals are negligible, they are not explicitly depicted in the other figures for clarity of presentation.

Resource allocation that results in minimum makespan and flowtime is the main goal of the scheduler. However, calculating a near-optimal solution can take a relatively long time, sometimes, to the point where the resulting allocation is no longer relevant as the scheduler runtime exceeds the time the services could have taken in a less optimal solution. Thus, the scheduler runtime is vital to any live allocation approach. WorkQueue sacrifices optimality for speed and simplicity, hence, it has a complexity of $O(NM)$, where N is the number of services and M is the number of devices. The Min-min, Max-min, and LJFR_SJFR all share a complexity of $O(N^2M)$. Sufferage has a slightly higher complexity of $O(2N^2M)$. CORA has a time complexity of $O((N+M)^2E)$, where E is the number of edges in the formulated graph, which can vary depending on the graph density. Munkres time complexity is $O(6(N+M)^3)$. As depicted in Figure. 4.4, which shows the runtime over a varying number of services. Intuitively, the runtime increases as the number of services increases in all. The Munkres algorithm's longer runtime is due to factors resulting from the multiple steps that it has to go through before the assignment. In contrast, on average CORA does not encounter the same problem achieving a run time that is up to six times faster than the Munkres algorithm. WorkQueue is significantly faster than all other approaches due to its relatively low time complexity. CORA is around 16% faster than Sufferage, While Min-min, LJFR_SJFR, and Max-min are around 40% faster than CORA in terms of scheduler runtime.

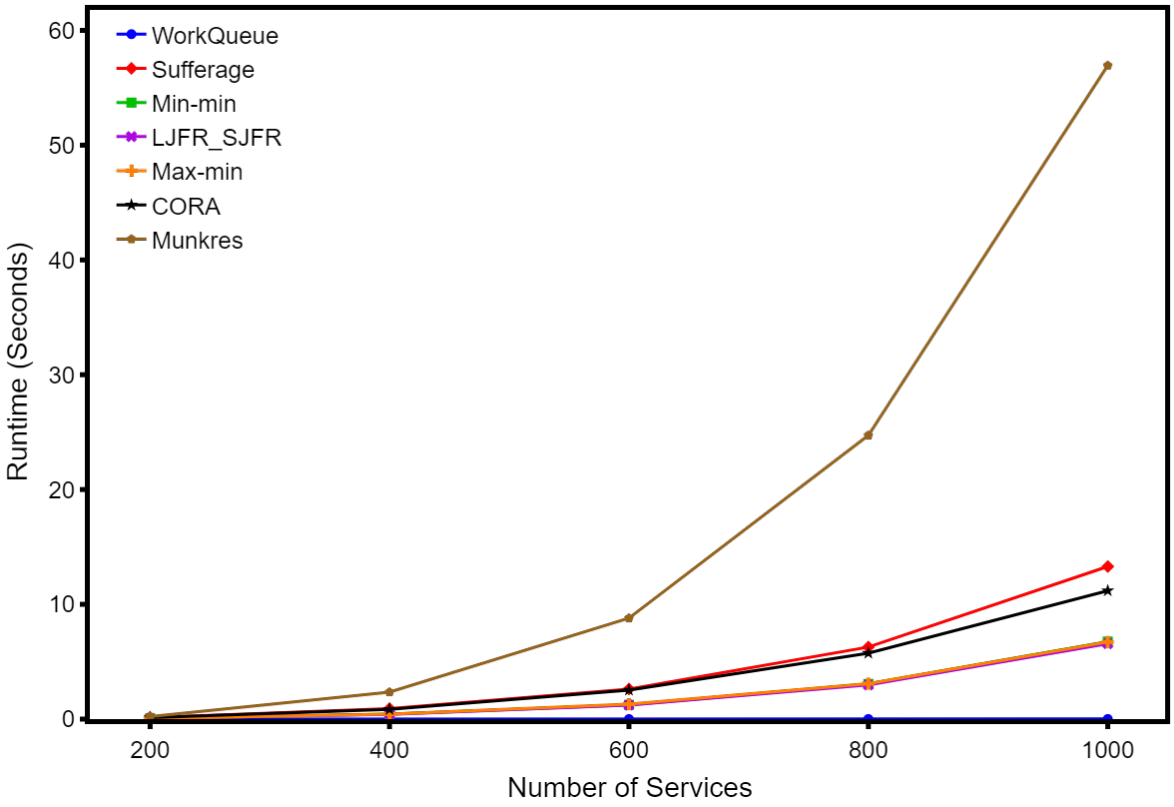

Figure 4.4: Scheduler runtime of CORA, Min-min, Max-min, LJFR_SJFR, WorkQueue, Sufferage, and Munkres over a varying number of services.

4.4.2 Effect of Omitting Communities

CORA is optimized for an environment with multiple communities. Nonetheless, it still provides acceptable performance in generic cases of one community or no communities at all, which are the same case from the scheduler's point of view since any service can be assigned to any worker. Note that, in the case of no communities the instances are reduced to eight because community density is omitted from the simulation variables. Figure 4.5 depicts the effect of having no communities on the average makespan for 600 services. As illustrated in Figure 4.5, CORA outperforms all other heuristics approaches in terms of average makespan, with an improvement reaching 78%, 6.5%, 23%, 15%, and 10% compared to WorkQueue, Sufferage, Min-min, LJFR_SJFR, and Max-min, respectively. Despite the case of no communities lacking the main challenge of community constraints that CORA

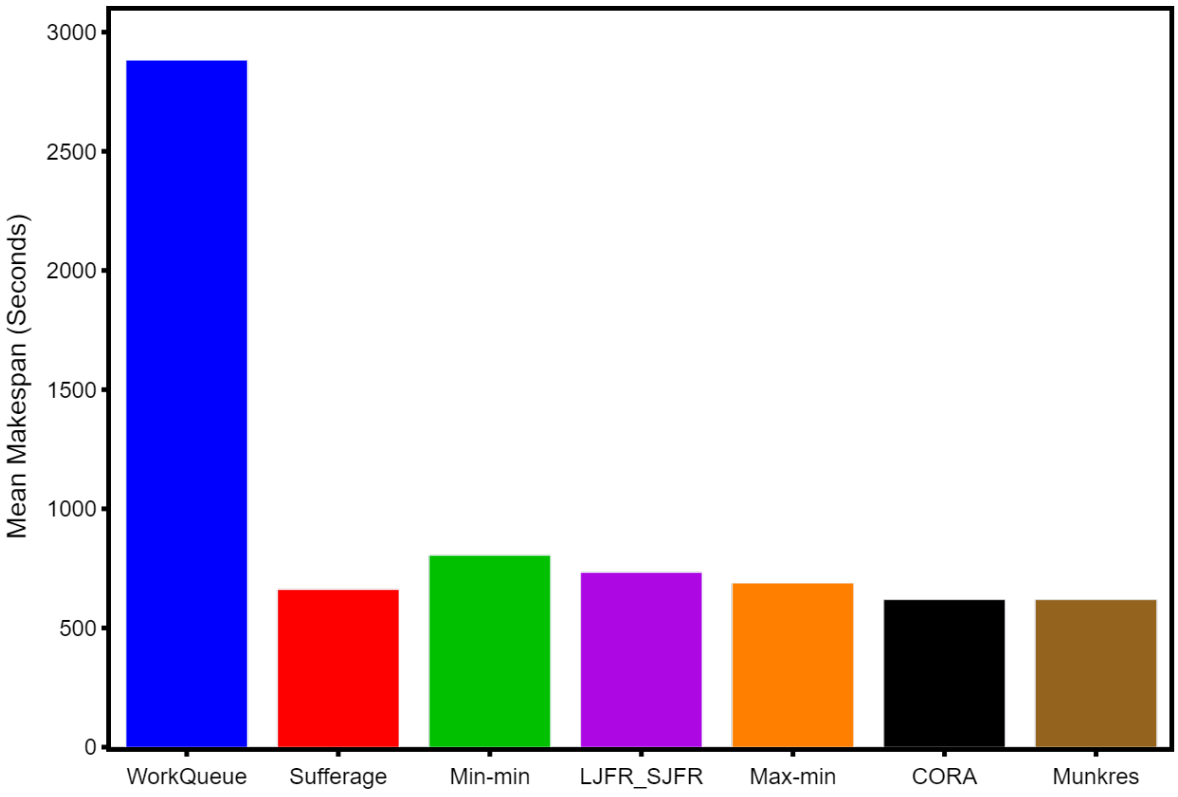

Figure 4.5: Average makespan of CORA, Min-min, Max-min, LJFR_SJFR, WorkQueue, Sufferage, and Munkres for 600 services with no communities

normally addresses, CORA manages to pull a lead on other heuristics. This lead can be explained by CORA's ability to backtrack and reassign services, Which addresses some of the cases where the other heuristic approaches fail. It is worth noting that each one of the heuristic approaches would fail in different cases.

Figure 4.6 demonstrates the effect of having no communities on the average flowtime for 600 services. As shown in the figure, CORA achieves the minimum flowtime among all approaches, with an improvement of up to 50% and 2% compared to WorkQueue and Max-min, respectively. CORA yields less than 1% lower flowtime compared to Sufferage, Min-min, LJFR_SJFR, and Munkres, which is still a negligible amount in real-life scenarios.

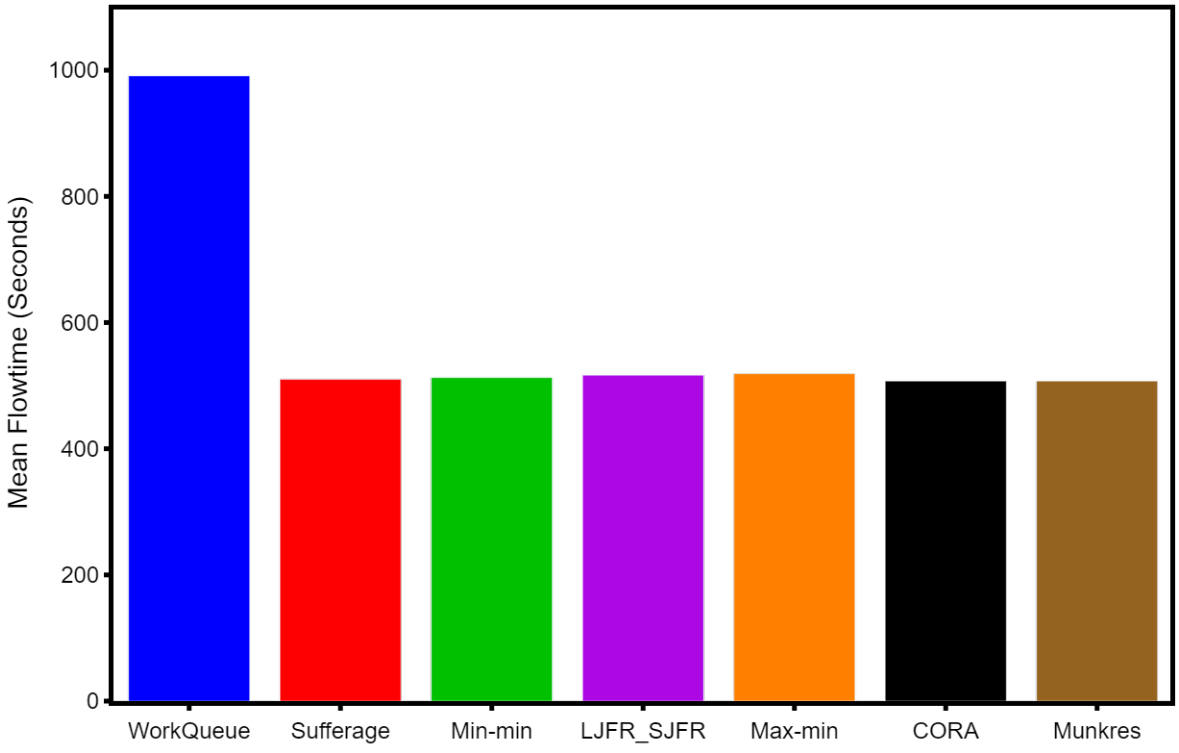

Figure 4.6: Average flowtime of CORA, Min-min, Max-min, LJFR_SJFR, WorkQueue, Sufferage, and Munkres for 600 services with no communities

4.4.3 Effect of Varying CORA Parameters

CORA has configurable parameters, that can adjust its performance based on its priorities. The one we want to highlight, as discussed before, is the α parameter. This is because changing the α parameter significantly impacts the execution of the allocated services. Figure 4.7 demonstrates the effect of varying α on the average makespan of CORA. Note that as α increases, the average makespan decreases, reaching a decrease of around 24% when $\alpha = 1$, compared to $\alpha = 0.25$. This decrease can be attributed to the scheduler increasing the penalty for allocating multiple services to the same device by increasing the weight of the edge connecting the device node to the sink. As seen in the figure, the makespan sweet spot is between 0.25 and 1; Makespan at $\alpha = 0$ is more than ten-fold the makespan at any other α. Hence it is omitted from the figure, since it is unusable in real life.

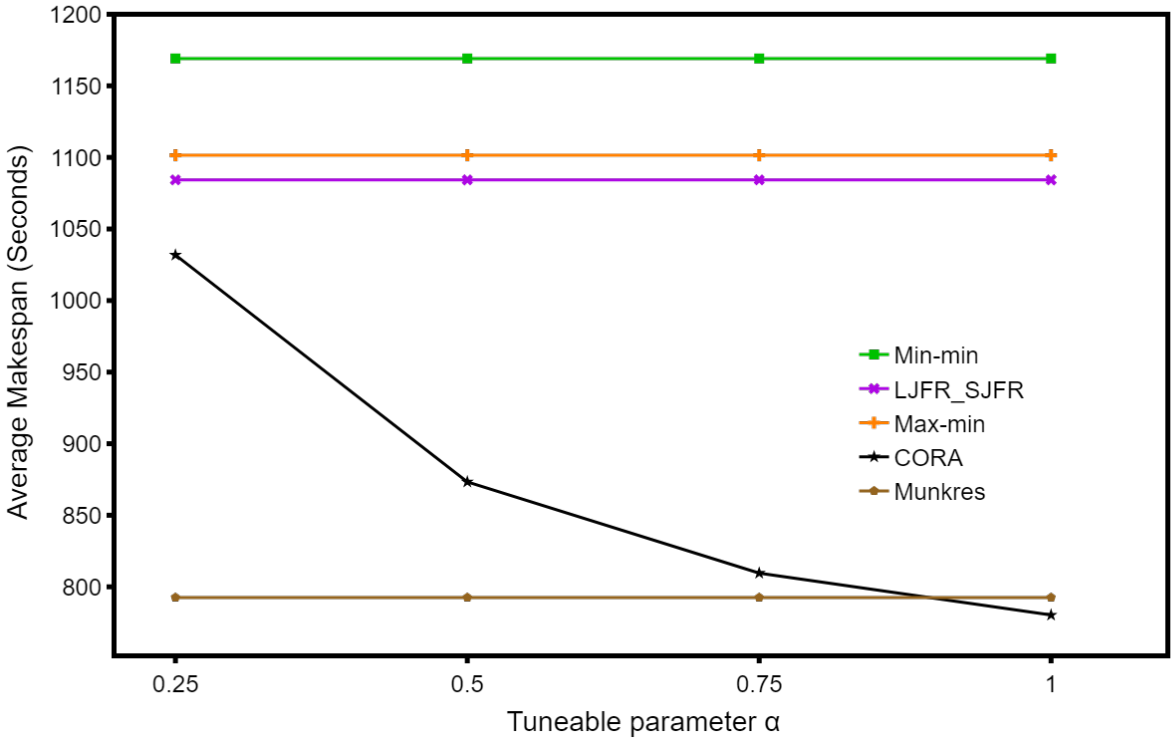

Figure 4.7: Average makespan of CORA, Min-min, LJFR_SJFR, Max-min, and Munkres over varying α for 600 services

Figure 4.8 depicts the effect of varying α on the average flowtime of CORA. At $\alpha = 0$, CORA can achieve average flowtime around 10% lower compared to $\alpha = 1$. However, as discussed in the previous paragraph, this configuration is useless in real-life scenarios due to its unreasonable makespan. The allocation resulting at $\alpha = 0$ can be achieved by assigning all services to the device yielding the minimum execution time among all the devices, regardless of the other allocated services. This minimum execution time acts as the hard cap for the minimum flowtime that can be achieved. Thus, the first usable value is at $\alpha = 0.25$. As α increases, the average flowtime increases, reaching an increase of up to 2% when $\alpha = 0.25$ compared to $\alpha = 1$. This increase can be attributed to the smaller penalty for allocating multiple service to the same device. This smaller penalty allows more service to be allocated to the device with less execution time for the given service regardless of the other allocated services. At $\alpha = 0.25$, CORA achieves around 3% lower flowtime than the second best

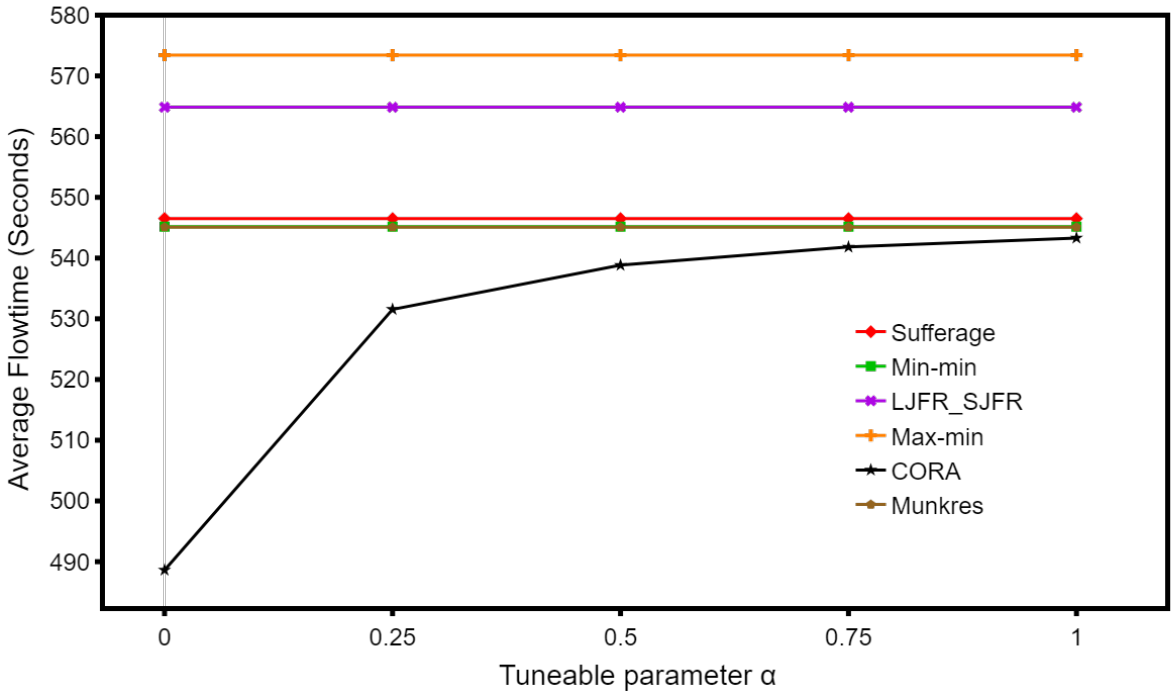

Figure 4.8: Average flowtime of CORA, Sufferage, Min-min, LJFR_SJFR, Max-min, and Munkres over varying α for 600 services

heuristic scheme (Min-min), while maintaining a 12% lower makespan. CORA can reach up to 6% lower flowtime compared to the second best heuristic scheme (Min-min) while maintaining a similar makespan. Depending on the use case, CORA can be optimized for a better makespan with flowtime similar to other approaches or a better flowtime with a makespan on par with other approaches. This flexibility is another strong advantage CORA has over competing approaches.

Chapter 5

Conclusions

5.1 Summary and Conclusion

Democratizing the edge by exploiting ample yet underutilized resources of EEDs can open a new tech market for individuals, businesses, enterprises, and municipalities to create their own edge cloud and monetize underused resources. Workload offloading in EED-enabled computing environments is crucial for modern applications. Maintaining data privacy and cost efficiency remain core challenges for the viability of EED-enabled computing paradigms. In this thesis, we have proposed the Community Edge Platform (CEP) to foster cost-free and privacy-preserved democratized edge computing. Towards that end, CEP exploits the notion of communities and leverages the wide range of business, institutional, and social relationships between users. CEP utilizes clusters and communities to achieve a high level of privacy and security, eliminate additional costs, and maintain a low latency task offloading.

We have conducted a scrupulous comparative study to assess CEP compared to twelve prominent edge computing platforms. The comparison has covered 14 features, grouped into three categories. The first category encompasses system architecture and deployment features, where CEP has stood out as a flexible, easily deployed, and managed system that can cover wide area networks. The second category encompasses application features. The comparison has shown the leverage of CEP in eliminating additional costs, as well as its flexible workloads and applications. The third category encompasses performance features. The comparison has illustrated CEP's potential as a platform that provides a high level of

scalability, privacy, and security.

CEP acknowledges the need for a QoS-based community-aware resource allocation approach to retain an acceptable efficiency. Towards that end, we have introduced the Community-Orientated Resource Allocation (CORA) scheme. CORA uses a graph-directed approach to allocate container-based services in community-oriented EED-enabled edge computing environments. Extensive simulations have shown that CORA is on par with six other prominent resource allocation schemes in terms of runtime. Additionally, CORA has a 28% better makespan on average in 16 different resource allocation scenarios. Extensive evaluations have also shown that CORA can yield an improvement of up to 6%, in terms of flowtime. Accommodating potential gains from the cluster scheduler, and the ability to adjust tuneable parameters to enable CORA to prioritize makespan or flowtime, along with the leverage demonstrated in different scenarios indicate that CEP, combined with its resource allocation module CORA, is a suitable middle-ground for users prioritizing cost-efficiency, security, and privacy but also seeking high efficiency.

5.2 Recommendations

CEP enables resource allocation that leverages the use of adjacent devices owned by the same user/organization (i.e., cluster-level allocation), along with devices within the requester's communities (i.e., community-level allocation). The cluster-level allocation provides a significant reduction in latency and data transfer. Thus, it is recommended to maximize the number of capable devices within a cluster for latency-critical applications by merging nearby clusters if possible. Since the cluster in CEP is centralized, there is no theoretical threshold on the number of devices that can be included in the cluster; There is no significant communication overhead for workers, unlike in decentralized systems. If the requested services are not delay-sensitive or data-intensive, then maintaining smaller clusters will enable more services to reach the higher level scheduler, and eventually be allocated to a more suitable device in any available cluster within the user communities.

The community size is significant to the scheduler in finding suitable workers. A community composed of scarce clusters would be unable to find suitable devices for the requested services. Meanwhile, a community composed of too many clusters would slow down the scheduler. In the case of one community or no overlapping communities, CORA can be replaced with another resource allocation approach that is more optimized for that case. For example, using WorkQueue for faster scheduler cycles or event-driven online scheduling in cases where the number of services or workers is too high to allocate services in a reasonable time. However, if there are multiple overlapping communities, it is recommended to keep CORA as the default scheduler for CEP.

The container-based approach adopted in CEP is a suitable option for systems where it is required to cover a wide range of applications and run on a diverse set of worker machines. In addition, it enables ensuring data privacy and workers' security in another way, such as mutual trust between the requesters and workers. Nonetheless, other approaches can be used for better service migration, distributed workloads, or enforcing trust using a framework to govern the possible use cases and minimize security risks.

5.3 Future Directions

In the future, we plan to improve the runtime estimation approach by utilizing Docker image history. This allows us to make use of the data gathered in previous versions and share knowledge across different services' histories. Then, by generalizing the second stage of the two-stage machine learning approach currently used, we can dynamically switch between different machine learning models depending on data size and model performance. Lastly, we plan on maintaining a profile of each worker, along with periodic benchmarks to estimate workers' idle time and available computation resources during relatively long services.

Another area that can be improved is the user experience on the platform. We plan on investigating the possibility of maintaining each user's profile, including social and work relations, and using those to dynamically form communities with minimal user interference.

Additionally, implementing a GUI for the platform can facilitate the introduction of our platform to a wider range of users with a less technical background. A GUI, combined with the API we already provide, can increase the system's flexibility by covering most use cases. The GUI can be used by users with a less technical background or simple applications, while the API can be used to automate the offloading process or to integrate CEP with other existing systems.

Currently, the system is built to eliminate costs by utilizing the community relation between users. However, we can provide another mode that extends the service exchange idea beyond a single community to include service exchange between communities. Users can label their devices as open for any work, same for their service request, which can be marked as safe to run on any machine. This is provided that the service request does not contain sensitive data or private codes for example. The scheduler can then match service requests across communities with a debt system that maintains a standing balance of each community with other communities. This allows the users to run on a wider selection of workers to solve peak load problems within communities without additional costs.

References

[1] K. Gyarmathy, "Comprehensive guide to iot statistics you need to know in 2020," *VXchnge [online]. Tampa, Florida: vXchnge, 2020 (1), 3 [cit. 2020-07-10]. Dostupné z: https://www. vxchnge. com/blog/iot-statistics*, 2020.

[2] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella, "On multi-access edge computing: A survey of the emerging 5g network edge cloud architecture and orchestration," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 3, pp. 1657–1681, 2017.

[3] M. Gaur and M. Jailia, "Cloud computing data security techniques—a survey," in *Renewable Energy Towards Smart Grid*. Springer, 2022, pp. 55–65.

[4] P. Ranaweera, A. D. Jurcut, and M. Liyanage, "Survey on multi-access edge computing security and privacy," *IEEE Communications Surveys & Tutorials*, vol. 23, no. 2, pp. 1078–1124, 2021.

[5] L. Peterson, T. Anderson, S. Katti, N. McKeown, G. Parulkar, J. Rexford, M. Satyanarayanan, O. Sunay, and A. Vahdat, "Democratizing the network edge," *ACM SIGCOMM Computer Communication Review*, vol. 49, no. 2, pp. 31–36, 2019.

[6] A. Islam, A. Debnath, M. Ghose, and S. Chakraborty, "A survey on task offloading in multi-access edge computing," *Journal of Systems Architecture*, vol. 118, p. 102225, 2021.

[7] B. Safaei, A. M. H. Monazzah, M. B. Bafroei, and A. Ejlali, "Reliability side-effects in internet of things application layer protocols," in *2017 2nd International Conference on System Reliability and Safety (ICSRS)*. IEEE, 2017, pp. 207–212.

[8] Radicati Group, "Mobile statistics report 2021–2025," *Radicati Group*, 2021.

[9] GSMA, "Gsma mobile economy," 2022.

[10] U. Cisco, "Cisco annual internet report (2018–2023) white paper," *Cisco: San Jose, CA, USA*, 2020.

[11] M. Maray and J. Shuja, "Computation offloading in mobile cloud computing and mobile edge computing: survey, taxonomy, and open issues," *Mobile Information Systems*, vol. 2022, 2022.

[12] I. M. Ibrahim *et al.*, "Task scheduling algorithms in cloud computing: A review," *Turkish Journal of Computer and Mathematics Education (TURCOMAT)*, vol. 12, no. 4, pp. 1041–1053, 2021.

[13] Q. Luo, S. Hu, C. Li, G. Li, and W. Shi, "Resource scheduling in edge computing: A survey," *IEEE Communications Surveys & Tutorials*, vol. 23, no. 4, pp. 2131–2165, 2021.

[14] Y. Wu, Q. Liu, R. Chen, C. Li, and Z. Peng, "A group recommendation system of network document resource based on knowledge graph and lstm in edge computing," *Security and Communication Networks*, vol. 2020, 2020.

[15] P. Mach and Z. Becvar, "Mobile edge computing: A survey on architecture and computation offloading," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 3, pp. 1628–1656, 2017.

[16] G. Carvalho, B. Cabral, V. Pereira, and J. Bernardino, "Edge computing: current trends, research challenges and future directions," *Computing*, vol. 103, no. 5, pp. 993–1023, 2021.

[17] L. A. Haibeh, M. C. Yagoub, and A. Jarray, "A survey on mobile edge computing infrastructure: Design, resource management, and optimization approaches," *IEEE Access*, vol. 10, pp. 27 591–27 610, 2022.

[18] "Distributive homepage," https://kingsds.network/, accessed: 2022-07-01.

[19] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, and J. P. Jue, "All one needs to know about fog computing and related edge computing paradigms: A complete survey," *Journal of Systems Architecture*, vol. 98, pp. 289–330, 2019.

[20] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE pervasive Computing*, vol. 8, no. 4, pp. 14–23, 2009.

[21] "Akraino homepage," https://www.lfedge.org/projects/akraino/, accessed: 2022-06-01.

[22] "Mutable homepage," https://mutable.io/, accessed: 2022-07-01.

[23] "Accedian mobiledgex homepage," https://accedian.com/mobiledgex/, accessed: 2022-07-01.

[24] "EdgeX Foundry homepage," https://www.edgexfoundry.org, accessed: 2021-10-01.

[25] "Azure IoT homepage," https://azure.microsoft.com/en-us/overview/iot/, accessed: 2022-06-01.

[26] "Apache Edgent homepage," https://incubator.apache.org/projects/edgent.html, accessed: 2022-06-01.

[27] "Kubernetes homepage," https://kubernetes.io, accessed: 2021-10-01.

[28] "AWS IoT Greengrass homepage," https://aws.amazon.com/greengrasso, accessed: 2022-06-01.

[29] "Homeedge homepage," https://wiki.lfedge.org/display/HOME/Home+Edge+Project, accessed: 2021-10-01.

[30] "Golem network homepage," https://www.golem.network/, accessed: 2022-07-01.

[31] "iexec homepage," https://iex.ec/, accessed: 2022-07-01.

[32] OTOY, "RNDR: Distributed GPU rendering on the blockchain," White Paper, August 2017.

[33] "Otoy rendertoken homepage," https://rendertoken.com/, accessed: 2022-07-01.

[34] J. N. Acharya and A. C. Suthar, "Docker container orchestration management: A review," in *International Conference on Intelligent Vision and Computing*. Springer, 2022, pp. 140–153.

[35] "Docker swarm documentation," https://docs.docker.com/engine/swarm/, accessed: 2022-07-01.

[36] C. Munien and A. E. Ezugwu, "Metaheuristic algorithms for one-dimensional bin-packing problems: A survey of recent advances and applications," *Journal of Intelligent Systems*, vol. 30, no. 1, pp. 636–663, 2021.

[37] I. Ahmad, M. G. AlFailakawi, A. AlMutawa, and L. Alsalman, "Container scheduling techniques: A survey and assessment," *Journal of King Saud University-Computer and Information Sciences*, 2021.

[38] K. Kaur, S. Garg, G. Kaddoum, S. H. Ahmed, and M. Atiquzzaman, "Keids: Kubernetes-based energy and interference driven scheduler for industrial iot in edge-cloud ecosystem," *IEEE Internet of Things Journal*, vol. 7, no. 5, pp. 4228–4237, 2020.

[39] W. Lu, B. Li, and B. Wu, "Overhead aware task scheduling for cloud container services," in *2019 IEEE 23rd International Conference on Computer Supported Cooperative Work in Design (CSCWD)*. IEEE, 2019, pp. 380–385.

[40] S. S. Murad and R. Badeel, "Optimized min-min task scheduling algorithm for scientific workflows in a cloud environment," *J. Theor. Appl. Inf. Technol*, vol. 100, pp. 480–506, 2022.

[41] F. Chen, X. Zhou, and C. Shi, "The container scheduling method based on the min-min in edge computing," in *Proceedings of the 2019 4th International Conference on Big Data and Computing*, 2019, pp. 83–90.

[42] S. Khurana and R. K. Singh, "Survey of scheduling and meta scheduling heuristics in cloud environment," in *Computational Methods and Data Engineering*. Springer, 2021, pp. 363–374.

[43] T. Menouer and P. Darmon, "New scheduling strategy based on multi-criteria decision algorithm," in *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 2019, pp. 101–107.

[44] T. Menouer, "Kcss: Kubernetes container scheduling strategy," *The Journal of Supercomputing*, vol. 77, no. 5, pp. 4267–4293, 2021.

[45] Y. Fu, S. Zhang, J. Terrero, Y. Mao, G. Liu, S. Li, and D. Tao, "Progress-based container scheduling for short-lived applications in a kubernetes cluster," in *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 2019, pp. 278–287.

[46] M. Lin, J. Xi, W. Bai, and J. Wu, "Ant colony algorithm for multi-objective optimization of container-based microservice scheduling in cloud," *IEEE access*, vol. 7, pp. 83 088–83 100, 2019.

[47] A. Dhumal and D. Janakiram, "C-balancer: A system for container profiling and scheduling," *arXiv preprint arXiv:2009.08912*, 2020.

[48] B. Liu, J. Li, W. Lin, W. Bai, P. Li, and Q. Gao, "K-pso: An improved pso-based container scheduling algorithm for big data applications," *International Journal of Network Management*, vol. 31, no. 2, p. e2092, 2021.

[49] A. Al-Moalmi, J. Luo, A. Salah, K. Li, and L. Yin, "A whale optimization system for energy-efficient container placement in data centers," *Expert Systems with Applications*, vol. 164, p. 113719, 2021.

[50] S. B. Nath, S. K. Addya, S. Chakraborty, and S. K. Ghosh, "Green containerized service consolidation in cloud," in *ICC 2020-2020 IEEE International Conference on Communications (ICC)*. IEEE, 2020, pp. 1–6.

[51] J. Liu, S. Wang, A. Zhou, J. Xu, and F. Yang, "Sla-driven container consolidation with usage prediction for green cloud computing," *Frontiers of Computer Science*, vol. 14, no. 1, pp. 42–52, 2020.

[52] A. Chmiela, E. Khalil, A. Gleixner, A. Lodi, and S. Pokutta, "Learning to schedule heuristics in branch and bound," *Advances in Neural Information Processing Systems*, vol. 34, pp. 24 235–24 246, 2021.

[53] S. Mendes, J. Simão, and L. Veiga, "Oversubscribing micro-clouds with energy-aware containers scheduling," in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, 2019, pp. 130–137.

[54] N. Nayar, S. Gautam, P. Singh, and G. Mehta, "Ant colony optimization: A review of literature and application in feature selection," *Inventive Computation and Information Technologies*, pp. 285–297, 2021.

[55] A. Sohail, "Genetic algorithms in the fields of artificial intelligence and data sciences," *Annals of Data Science*, pp. 1–12, 2021.

[56] T. M. Shami, A. A. El-Saleh, M. Alswaitti, Q. Al-Tashi, M. A. Summakieh, and S. Mirjalili, "Particle swarm optimization: A comprehensive survey," *IEEE Access*, 2022.

[57] S. Dong, P. Wang, and K. Abbas, "A survey on deep learning and its applications," *Computer Science Review*, vol. 40, p. 100379, 2021.

[58] J. Dickerson, K. Sankararaman, K. Sarpatwar, A. Srinivasan, K.-L. Wu, and P. Xu, "Online resource allocation with matching constraints," in *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2019.

[59] M. A. Iverson, F. Ozguner, and L. C. Potter, "Statistical prediction of task execution times through analytic benchmarking for sch eduling in a heterogeneous environment," in *Proceedings. Eighth Heterogeneous Computing Workshop (HCW'99)*. IEEE, 1999, pp. 99–111.

[60] T.-P. Pham, J. J. Durillo, and T. Fahringer, "Predicting workflow task execution time in the cloud using a two-stage machine learning approach," *IEEE Transactions on Cloud Computing*, vol. 8, no. 1, pp. 256–268, 2017.

[61] T. Atobishi, M. Bahna, K. Takács-György, and C. Fogarassy, "Factors affecting the decision of adoption cloud computing technology: The case of jordanian business organizations," *Acta Polytechnica Hungarica*, vol. 18, no. 5, 2021.

[62] W. Fangyang and L. Yuming, "Extended kuhn-munkres algorithm for constrained matching search," 2021.

[63] A. Bernstein, M. P. Gutenberg, and T. Saranurak, "Deterministic decremental sssp and approximate min-cost flow in almost-linear time," in *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2022, pp. 1000–1008.

Appendix A: Community Edge Platform Implementation

The purpose of this appendix is to present a few implementation details for CEP. Starting with the Entity Relationship Diagram (ERD) of the server. As illustrated in Figure A.1, we can see the current invitation-based implementation of community management. The database stores four tables for community management: user, community, invitation, and membership. The user is one of the core tables that link the communities to the devices and services through clusters. New users can register and log in to the system using the authorization manager module. Later, they can use the community management module endpoints to create and manage communities.

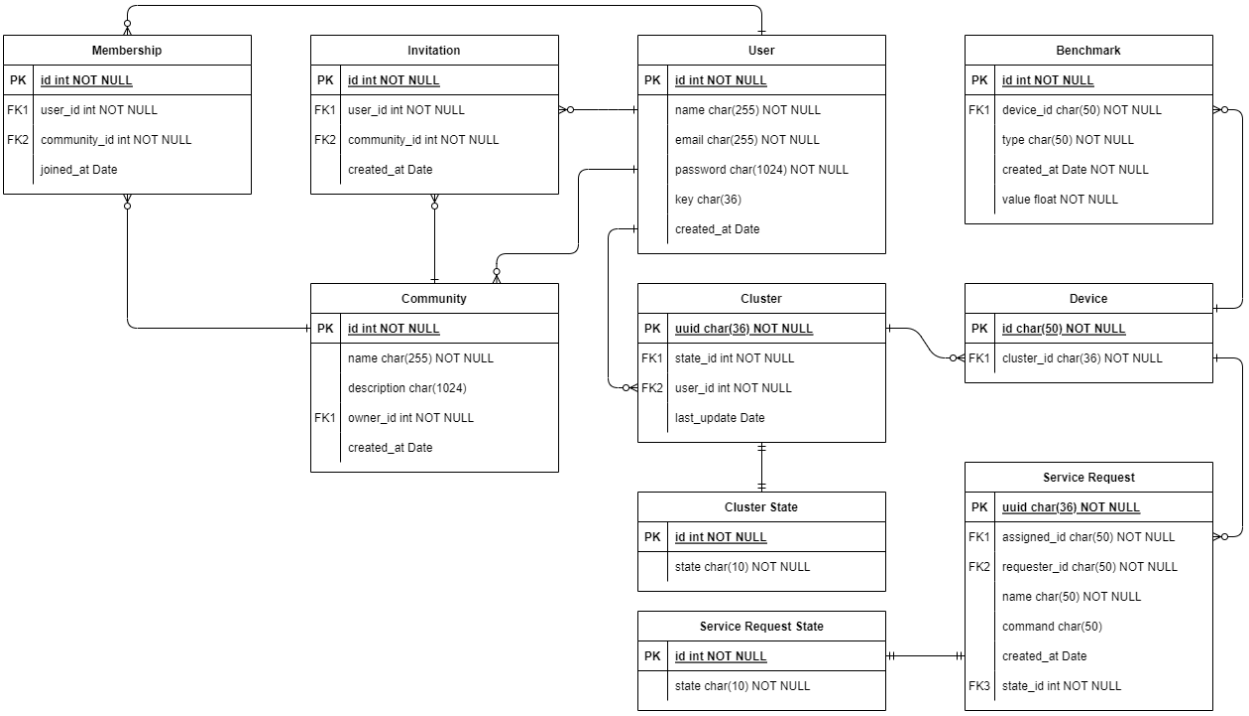

Figure A.1: CEP Entity Relationship Diagram

Users can be members of multiple communities and a community can have one or more users. Hence, forming a many-to-many relationship that needs to be implemented by an intersection table, titled membership in our case. Besides the owner who creates the com-

munity, other users need to be invited. This invitation will be stored in the invitation table, so users can be notified and respond to it. Clusters are stored under users by contacting the server using the user key (i.e., Personal Access Token (PAT)), which is generated at the user's request. Devices and benchmarks are updated during the benchmarks update stage during the scheduler cycle depicted in Figure 3.3. Meanwhile, the clusters states, are modified by the cluster manager module during the update clusters states stage. Finally, service requests are added upon the cluster head communication with the server to forward a request which can happen at any time during the cycle. The request would be queued until the service allocation stage. However, this queue is maintained in the database to ensure stateless implementation of the server.

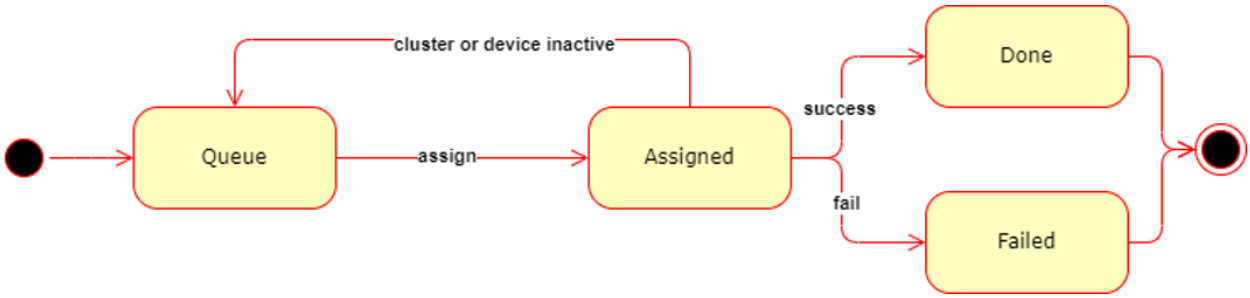

Figure A.2: Service Request State Machine

Figure A.2 shows the state machine for the requested service. It remains queued after creation until the service requests are assigned to any device during a cycle. Then, the request service state is changed to assigned. If during this time the cluster moved to an inactive state which is depicted in Figure 3.2, the service request will be added back to the queue so it can be reassigned to a new worker. After the assigned device executes the service it can be tagged as done or failed, for cases where there is a problem with the service implementation. The result is forwarded to the original requester as a notification during the assignments check stage alongside any newly assigned services. Finally, the request would be removed from the main database. However, it can be logged for future analysis of the system.