Container Orchestration for the Edge Cloud

Berat Şenel

Thèse de doctorat de Sorbonne Université

Spécialité

Informatique

École doctorale

EDITE de Paris (ED130)

Laboratoire de recherche

Laboratoire d'Informatique de Paris 6 (UMR 7606)

Container Orchestration for the Edge Cloud

Présentée par Berat Can ŞENEL

Pour obtenir le grade de Docteur de Sorbonne Université

Soutenue à Paris le 23 juin 2023

Composition du jury :

M. Olivier Fourmaux	Professeur, Sorbonne Université	Directeur de thèse
Mme. Nathalie Mitton	Directrice de recherche, Inria, Lille	Rapporteuse
M. Sébastien Monnet	Professeur, Université Savoie Mont Blanc	Rapporteur
M. Pierre Sens	Professeur, Sorbonne Université	Président du jury
Mme. Sara Ayoubi	Chercheuse, Nokia Bell Labs, Paris-Saclay	Examinatrice
M. Timur Friedman	Maître de conférences, Sorbonne Université	Co-encadrant

Acknowledgments

I would like, before all else, to express my deepest gratitude to VMware and the French Ministry of Armed Forces for generously funding my research, which allowed me to acquire invaluable professional experiences throughout my doctoral journey. I extend my special thanks to Victor Firoiu, Shanshan Song, and Sujata Banerjee from the VMware team for their support in securing the grant and in my research.

Having voiced this gratefulness, I would like to thank my thesis supervisors, Olivier Fourmaux and Timur Friedman, for their invaluable guidance and instruction that have been pivotal in the success of my research. The expertise and support they provided from day one have been crucial for my research to be successful. I am equally grateful to Rick McGeer and Justin Cappos, with whom I had the pleasure of collaborating, and whose contributions enriched my work. I would like to extend my gratitude to the members of my comité de suivi, namely Olivier Richard and Pierre Sens, for their invaluable feedback and contributions. In addition, I would like to thank Pierre Sens for kindly agreeing to serve as a member of the jury for my thesis defense. I wish to convey my heartfelt gratitude to Nathalie Mitton and Sébastien Monnet for agreeing to act as both rapporteurs and jury members for my thesis. I am also profoundly thankful to Sara Ayoubi for accepting to be a member of the jury of my thesis defense. I would also like to express my appreciation to Maxime Mouchet, for his scientific guidance and development efforts; Ufuk Bombar for his assistance with conducting measurements on the cluster-wise federation architecture; Ciro Scognamiglio for his support in running experiments on GENI, LIP6, and PlanetLab Europe resources and in maintaining the running instance of the EdgeNet software; Kevin Vermeulen for being so kind as to put a significant effort into proofreading and reviewing my works; Matthieu Gouel for being a supportive colleague; Lois DeLong for imparting her expertise in technical writing; and Prométhée Spathis, who was my thesis director at the beginning of my thesis, for accepting me as a doctoral student.

On a personal note, I owe my deepest gratitude to my wife, Selin, for her unwavering love, support, and encouragement, without which this achievement would not have been possible. She has been my foundation through all highs and lows of this journey, and I am evermore grateful to her. I would not be able to achieve this without my daughter, Defne, the light of my life. She has kept me awake day and night, providing everlasting love and joy throughout the challenging periods of writing this thesis. Finally, I am thankful to my friends and loved ones, who always bring pleasure to my life. I am profoundly beholden to all the above-mentioned individuals as well as organizations for their help and support.

RÉSUMÉ

English version

The pendulum again swings away from centralized IT infrastructure back towards decentralization, with the rise of edge computing. Besides resource-constrained devices that can only run tiny tasks, edge computing infrastructure consists of server-class compute nodes that are collocated with wireless base stations, complemented by servers in regional data centers. These compute nodes have cloud-like capabilities, and are thus able to run cloud-like workloads. Furthermore, many smart devices that support containerization and virtualization can also handle cloud-like workloads. The « containers as a service » (CaaS) service model, with its minimal overhead on compute nodes, is particularly well adapted to the less scalable cloud environment that is found at the edge, but cloud container orchestration systems have yet to catch up to the new edge cloud environment.

This thesis shows a way forward for edge cloud container orchestration. We make our contributions in two primary ways: the reasoned conception of a set of empirically tested features to simplify and improve container orchestration at the edge, and the deployment of these features to provide EdgeNet, a sustainable container-based edge cloud testbed for the internet research community.

We have built EdgeNet on Kubernetes, as it is open-source software that has become today's de facto industry standard cloud container orchestration tool. The edge cloud requires multitenancy for the sharing of limited resources. However, this is not a Kubernetes-native feature, and a specific framework must be integrated into the tool to enable this functionality. Surveying the scientific literature on cloud multitenancy and existing frameworks to extend Kubernetes to offer multitenancy, we have identified three main approaches: (1) multi-instance through multiple clusters, (2) multi-instance through multiple control planes, and (3) single-instance native. Considering the resource constraints at the edge, we argue for and provide empirical evidence in favor of a single-instance multitenancy framework.

Our design includes a lightweight mechanism for the federation of edge cloud compute clusters in which each local cluster implements our multitenancy framework, and a user gains access to federated resources through the local cluster that her local cloud operator provides. We further introduce several features and methods that adapt container orchestration for the edge cloud, such as a means to allow users to deploy workloads according to node location, and an in-cluster VPN that allows nodes to operate from behind NATs.

We put these features into production through the EdgeNet testbed, a globally distributed compute cluster that is inherently less costly to deploy and maintain, and easier to document and to program than previous such testbeds.

French version

Avec l'essor des infrastructures de type edge où les ressources informatiques sont en périphérie de réseau, la tendance est une fois de plus orientée vers la décentralisation. En plus des appareils à ressources contraintes qui peuvent effectuer des tâches limitées, le « edge cloud » se compose de nœuds de calcul de classe serveur qui sont colocalisées avec des stations de base des réseaux sans-fil et qui sont soutenus par des serveurs dans des centres informatiques régionaux. Ces nœuds de calcul ont des capacités de type cloud et ils sont capables d'exécuter des charges de travail (workloads) typiques du cloud. En outre, de nombreux appareils intelligents qui supportent la conteneurisation et la virtualisation peuvent exécuter de telles tâches. Nous pensons que le modèle de service « containers as a service », ou CaaS, avec sa surcharge minime sur des nœuds de calcul, est particulièrement bien adapté pour l'environnement edge cloud qui est moins évolutif que le cloud classique. Pourtant, les systèmes d'orchestration de conteneurs en cloud ne sont pas encore intégrés dans les nouveaux environnements edge cloud.

Dans cette thèse nous montrons une voie à suivre pour l'orchestration des conteneurs pour des edge clouds. Nous apportons nos contributions de deux manières principales : la conception raisonnée d'un ensemble de fonctionnalités testées empiriquement pour simplifier et améliorer l'orchestration des conteneurs pour des edge clouds et le déploiement de ces fonctionnalités pour fournir une plateforme edge durable, basée sur des conteneurs, pour la communauté de recherche sur Internet.

Ce logiciel et cette plateforme s'appellent EdgeNet. Elle consiste en une extension de Kubernetes, qui est l'outil de facto standard d'orchestration de conteneurs pour l'industrie cloud. L'edge cloud nécessite une architecture mutualisée, ou « multitenancy », pour le partage de ressources limitées. Cependant, cela n'est pas une fonctionnalité native de Kubernetes et alors un cadre spécifique doit être ajouté au système afin d'activer cette fonctionnalité.

En étudiant la littérature scientifique sur les cadres multitenancy dans le cloud ainsi que les cadres multitenancy déjà existants pour Kubernetes, nous avons développé une nouvelle classification de ces cadres en trois approches principles: (1) multi-instance via plusieurs clusters, (2) multi-instance via plusieurs plans de contrôle et (3) instance-unique. Compte tenu des contraintes de ressources à l'edge, nous défendons et apportons des preuves empiriques en faveur d'un cadre multitenancy qui est instance-unique.

Notre conception comprend un mécanisme léger pour la fédération des clusters de calcul de l'edge cloud dans lequel chaque cluster local implémente notre cadre multitenancy, et un utilisateur accède à des ressources fédérées par le biais du cluster local fourni par son opérateur de cloud local. Nous introduisons en outre plusieurs fonctionnalités et méthodes qui adaptent l'orchestration des conteneurs à l'edge cloud, telles qu'un moyen de permettre aux utilisateurs de déployer des charges de travail en fonction de l'emplacement du nœud, et un VPN en cluster qui permet aux nœuds de fonctionner derrière des NAT.

Nous mettons ces fonctionnalités en production avec la plateforme d'expérimentation d'EdgeNet, un cluster de calcul distribué à l'échelle mondiale qui est intrinsèquement moins coûteux à déployer et à entretenir, et plus facile à documenter et à programmer que les plateformes d'expérimentation précédents.

Contents

List of Figures

List of Tables

NOMENCLATURE

$5G$	Fifth generation mobile network
AM	Aggregate manager
API	Application programming interface
$CaaS$	Containers as a service
CAD	Content access duration
CDN	Content delivery network
CSP	Cloud service provider
$ETSI$	European Telecommunications Standards Institute
HA	High availability
$IaaS$	Infrastructure as a service
IoT	Internet of things
IT	Information technology
$K8s$	Kubernetes
LAN	Local area network
LF	Linux Foundation
MEC	Multi-access edge computing
NAT	Network address translation
OS	Operating system
$PaaS$	Platform as a service
RAN	Radio access network
RTT	Round trip time
$SaaS$	Software as a service
$TTFB$	Time to first byte
VM	Virtual machine

Chapter 1

Introduction

The open-ended debate on whether to centralize or distribute IT resources was initiated following the dawn of the technology itself [41]. Many determining factors steer centralization or decentralization decisions [119], which includes telecommunication/hardware costs, infrastructure/hardware capacities, application/service requirements, and organizational/operational capabilities. The pendulum, therefore, constantly swings back and forth between centralized and decentralized approaches as a function of changes in such factors. To date, we deem four breakthroughs that have strongly influenced the tendency: centralized mainframe computing, the advent of personal computers, cloud computing, and edge computing [54]. With edge computing, the trend is once again towards decentralizing IT resources.

It is perceived that edge computing is only about resource-constrained devices that can run tiny tasks. However, in addition to such devices, edge computing infrastructure consists of server-class compute nodes that are collocated with wireless base stations [36, 88], which are complemented by servers in regional data centers [134]. These compute nodes have cloud-like capabilities and are thus able to run cloud-like workloads [143]. Furthermore, many smart devices that support containerization and virtualization can also handle such cloud-like workloads [135].

In our anticipation of the future of edge computing infrastructure, multiple providers, such as cloud providers and operators, offer compute resources [24] in many locations on a global scale [126, 37]. Put in other words, edge clouds will be launched by multiple providers who will mainly be operators, and they will be geographically scattered worldwide. These edge clouds will be more constrained in compute resources than are in clouds. Edge cloud customers will deploy and constantly move their workloads across these resource-constrained edge clouds to meet service requirements. The reasoning behind this can be ascribed to having workloads remain deployed on each edge location would prove uneconomical for some customers, as well as inefficient use of constrained resources. One more reason can be a need for disaster recovery mechanisms regarding autonomous cars in the event of a critical failure of internally situated hardware. We may need to quickly deploy and move workloads across the closest roadside infrastructure to pull over malfunctioning self-driving cars in motion safely. The aforementioned factors necessitate a new service model

that, imposing low overhead, facilitates customers to deploy and move their workloads from one operator's edge cloud to another in different locations as well as fosters interoperability between edge clouds.

Containers are being lightweight than VMs. They introduce low overhead and also are portable, enhancing workload mobility with the ability to rapidly spin up and down with the minimal cost of resources. These characteristics of containers have contributed to the widespread adoption of them by the cloud community, especially for application and service deployments. This speedy uptake has even given rise to a new cloud service model called CaaS, which is typically built upon container orchestration tools (See Sec. 2.1.3 for details).

We think the CaaS service model is particularly well adapted to the less scalable and widely distributed cloud environment that is found at the edge. Still, cloud container orchestration systems have yet to catch up to the new edge cloud environment. In consideration of the resource-constrained and geographically dispersed essence of edge clouds that multiple providers offer, the available multitenancy frameworks for CaaS are either not efficient or not designed to allow multiple tenants to share such federated infrastructure. As for CaaS federation tools, they do not make a standard solution that is ample in scope to incentivize providers, including small-sized ones, to offer compute resources to this federated edge infrastructure.

With this thesis, we recognize the need and show a way forward for container orchestration for the edge cloud. While undertaking this task, our main emphasis is on two building blocks, *multitenancy* (Chapter 3) and *federation* (Chapter 4), and is on how to incorporate these with supplementary features to present a method of enabling a sustainable *edge cloud testbed* (Chapter 5) to function for the research community.

Within this introduction, we foremost provide the terminology and conventions that this thesis follows (Sec. 1.1), then bring forward the problem domain that is accompanied by our motivation for studying container orchestration for the edge cloud (Sec. 1.2), and conclude this chapter with our contributions to the field (Sec. 1.3).

1.1 Upcoming edge infrastructure and use cases

This section provides background information regarding edge computing, its envisaged architecture, and potential use cases. Since edge computing is a relatively new computing paradigm, there is not yet a consensus on the terms in both academia and industry. For this reason, we, in this thesis, follow edge cloud terminology from the *Sharpening The Edge* 2020 white paper [135] and from a project on edge computing glossary[1] of Linux Foundation.[2] Below is the summary of conceived infrastructure these sources introduce, along with some edge-specific use cases.

The edge computing term intrinsically is associated with an infrastructure that consists of low-capacity devices located at the edge of the network. However, the edge infrastructure is not limited to such low-capacity devices. It also incorporates geographically dispersed

[1] Open Glossary of Edge Computing https://stateoftheedge.com/projects/glossary/
[2] Linux Foundation https://www.linuxfoundation.org/

resources that support cloud-like capabilities [143]. Briefly, there are two main infrastructure tiers: *Service Provider Edge* and *User/Device Edge*, as seen in Fig. 1.1. When deploying services closer to the centralized data centers, users typically experience higher latency and jitter, whereas the infrastructure can ensure better scalability and physical security. The more go in the opposite direction reduces latency and jitter, despite driving the lack of scalability and possibly physical security. Ownership of resources, contrary to centralized data centers, belongs to many providers, enterprises, and individuals. Workloads running on each tier can work in concert with the ones on others, providing compute continuum.

The infrastructure of service provider edge is made up of IT hardware that delivers cloud-like capabilities. Providers deploy these resources in close proximity to users and to where data is produced. This tier encompasses a geographic area extending from the access networks to the nearest internet exchange points as a means of support to the metropolitan network. Service provider edge is further subdivided into two sublayers to fulfill this duty: *Regional Edge* and *Access Edge*.

As its name implies, regional edge consolidates resources in regional data centers. This server-class infrastructure is expected to ensure a latency that ranges from 30ms to 100ms. Organizations like cloud providers and CDNs can also colocate their compute resources in these sites in order to reduce latency by decreasing hops between the services and end-users. Access edge is the nearest infrastructure to the physical last-mile networks, a maximum of one hop away from RAN or cable headend to provide latency between sub-1ms to 30ms. This server-class infrastructure is deployed in sites such as base stations, roadside cabinets, and central offices.

The constrained compute resources on the user side of the last-mile network composes of the infrastructure in user/device edge. These constrained resources that hardware like mobile devices, gateways, computers, as well as servers offer are highly heterogeneous. As data is processed on-site, it prevents data from traveling through the access network, thus saving bandwidth. It also makes sure of significantly lower latencies than does service provider edge. There are three subcategories of user/device edge: *On-Premises Data Center Edge*, *Smart Device Edge*, and *Constrained Device Edge*.

On-premises data center edge refers to data centers consisting of servers that are under the control of users. Although this sounds like existing on-premise computing, edge computing integrates it with the continuum of computing. In smart device edge, compute resources can still support virtualization and containerization, as does in on-premises data center edge, which allows for handling cloud-like workloads. Smartphones, IoT gateways, and PCs are typical examples of these resources. However, constrained device edge is of low-capacity devices, including microcontroller-based devices, which do not support cloud-like workloads. Such devices can only run tiny tasks but can do that in real time and offload tasks to other tiers.

The edge infrastructure is reasoned by the fact that some emerging use cases require lower latency and jitter, better control of data security, privacy, and sovereignty to comply with regional data regulations, and broader bandwidth than clouds can offer. To better understand these novel requirements, the research community endeavors to characterize edge-specific use cases. Video analytics, smart homes, and smart cities are some of these use

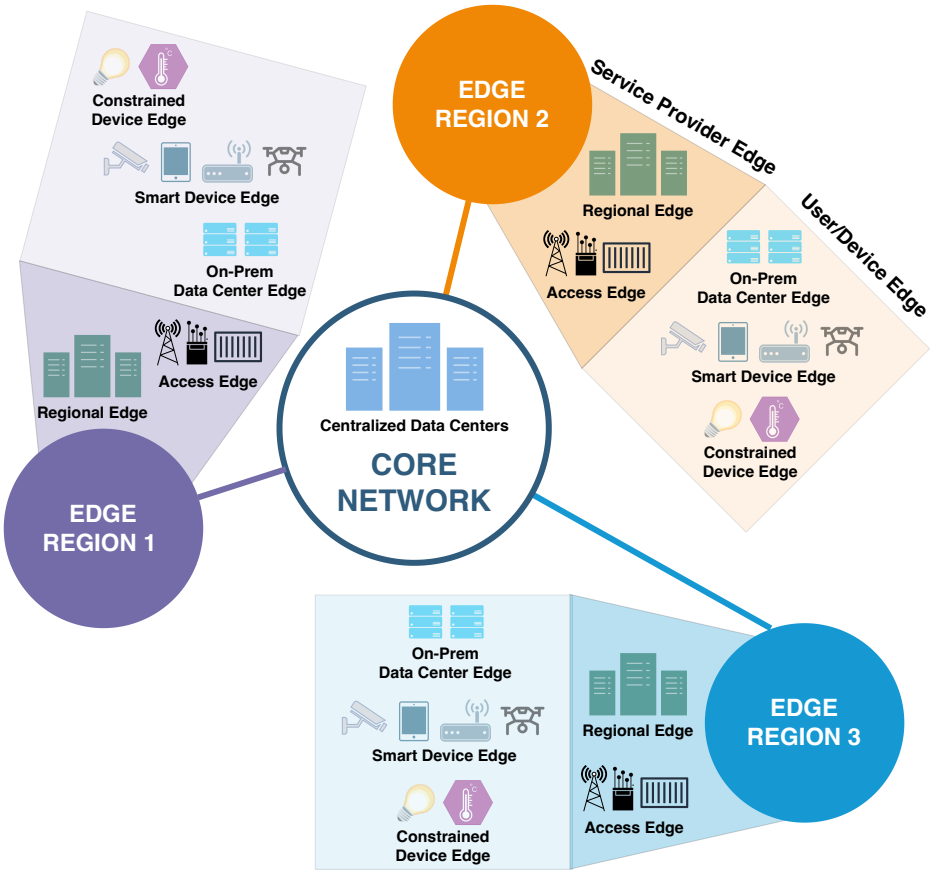


Figure 1.1: Edge continuum, derived from LF Edge's white paper [135].

cases studied by researchers [138], as are intelligent transportation, industry automation, and tactile internet, which are latency-critical [108].

On the one hand, use cases like autonomous vehicles, drone delivery, and factory floor, where communication between devices takes place without user intervention, compel lower latencies and jitters. On the other hand, many applications nowadays allow users to share and stream videos and make video calls, which drives them to produce data. Considering data generated both through these applications and by IoT devices, data transmission toward clouds is inflated, saturating bandwidth. Both these scenarios can be addressed through processing as well as distributing data on edge nodes in proximity.

1.2 Problem domain and motivation

This section introduces the problem domain that provides the context in which one of the problems of container orchestration for the edge cloud lives and our motivation to address this problem.

Cloud infrastructure consists of centralized data centers on which customers run their services. A strong point of this infrastructure is that these services can be scaled out as end-user demand increases. Cloud providers can also launch these centralized data centers in

sites at a number of different geolocations in order to lower latency between services and end-users. This allows cloud customers who offer services over the Internet to make them available from one or a few of these centralized data centers if needed. Overall, this is a solution that meets the requirements of many applications.

In contrast, edge computing infrastructure will consist of numerous data centers that are geographically dispersed worldwide. We will see that services are offered from the edge of the network, closer to the users [122], or to the location where data is generated. Such services will run in combination with services offered from centralized data centers as well as from the user/device edge. With 5G, edge cloud customers will offer their services from servers that are co-located with wireless base stations [88], maybe in roadside cabinets as well. Server locations will be many, near users and devices, across cities and countries worldwide.

We assert that multiple operators will potentially cover these geographical locations rather than a single one spanning across all those regions. In such a scenario, some workloads will need to be moved from one tier to another regarding edge continuum, from one location to another, and from one operator's infrastructure to another. As compute resources in edge clouds are more constrained than are in clouds, multitenancy along with such workload mobility needs to be handled with minimal resource consumption. In certain edge use cases, moving workloads swiftly between edge clouds may also be necessary. But there is not yet a standard orchestration system that allows multiple providers to serve many cloud/edge customers simultaneously, which also imposes such a low overhead.

1.3 Contributions

In this thesis, we make our contributions in two primary ways: the reasoned conception of a set of empirically tested features to simplify and improve container orchestration at the edge and the deployment of these features to provide a sustainable container-based edge cloud testbed for the internet research community.[3] We have implemented these features to extend Kubernetes,[4] made them available as free, open-source software,[5] and enabled the EdgeNet testbed, a running instance of the software code, that is open to researchers worldwide.[6] As of April 2023, the size of the codebase, regarding the main components, is *16,548* lines of code written in Go, which excludes both blank lines and comment lines.[7]

We present a unique interpretation of the CaaS service model to address the challenges

[3]Contributions have been communicated with the scientific community through five peer-reviewed publications: an extended abstract published in NSF workshop, WOMBIR 2021 [133]; an extended abstract, demo session, published in IEEE INFOCOM International Workshop, CNERT 2021 [130]; a short paper, best paper award, published in EdgeSys workshop held in conjunction with the ACM EuroSys 2021 [129]; a short paper published in the SLICES workshop held in conjunction with IFIP Networking 2022 [131]; an extended abstract, demo session, published in IEEE ICDCS 2022 [128]. In addition to these, a preprint has been made available on arXiv [132].

[4]The EdgeNet contributors https://github.com/EdgeNet-project/edgenet/graphs/contributors

[5]The EdgeNet software https://github.com/EdgeNet-project/edgenet

[6]The EdgeNet testbed https://edge-net.org/

[7]We used the SCC tool to calculate the size of the codebase (https://github.com/boyter/scc)

that stem from the edge infrastructure and use cases that we anticipate. With our multitenancy framework, we disengage CaaS multitenancy from the multi-instance model that imposes high overhead to shift it to the single-instance model that introduces low overhead and is also lightweight. This change allows a single cluster to support up to *10,000 tenants*, whereas it is *40 tenants* for the closest multi-instance approach. The marked contrast in tenant count that is supported in a single cluster becomes crucial for the resource-constrained edge cloud. Likewise, for four simultaneous tenant creation requests, our framework provides a median creation time of under a *second* per tenant, which is more than a *minute* for the nearest multi-instance approach. Regarding pod creation time, our framework also imposes a negligible overhead. These findings show that our multitenancy framework can be adopted to achieve a CaaS offering that enables multitenancy and workload mobility with minimal overhead.

A lingering issue still lacks a definitive solution: how can multiple providers offer their compute resources within edge clouds in many locations, and how can these edge clouds interoperate? We develop a federation strategy for CaaS in which a federation can be established node-wise, cluster-wise, system-wise, or a combination of these three. Node-wise architecture, as nodes are deployed to the same cluster, does not hinder interoperability, but the other two architectures do so. These three federation architectures, regardless, work together with our multitenancy framework to take advantage of its lightweight build. Especially for clusters operating with the same container orchestration tool, this also helps accomplish *interoperability* between edge clouds, as our multitenancy framework is designed to accept federation deployments that spring from tenants of other clusters. The measurements we have conducted on our prototype of cluster-wise architecture show that it takes under *two seconds* for a tenant to deploy a pod on a remote cluster and have it scheduled there.

Incorporating our multitenancy framework and node-wise federation with additional features, we have enabled an edge cloud testbed. At the time of writing, *51 tenants*, who are research organizations, research groups, and individual researchers, have registered with the testbed. Since its launch, over *10 experiments*, up to *7 parallel experiments*, have been conducted on EdgeNet, besides several class exercises.

We organize this thesis in the sense that the two building blocks, multitenancy and federation, and the testbed that leverages these two and exploits other supplementary features are covered through three separate chapters. The basis of our reasoning for devoting two chapters to multitenancy and federation is that each constitutes a component of our overarching vision for CaaS to run for clouds and edge clouds at scale, and as such, they are characterized by profound expounding. Below is the synopsis of the scope and content of each of these chapters:

- *Lightweight and Federation Oriented Multitenancy Framework* in Chapter 3, which improves the CaaS multitenancy frameworks such that it ensures that each edge cloud provider can accept incoming workloads that originate from tenants of other providers without collisions while imposing low overhead to support a large number of tenants in a single cluster to promote a future where CaaS can thrive, particularly at the network edge [129, 128, 132]. A list of contributions of this study:
  - We provide a novel classification of the multitenancy frameworks into three main approaches (Sec. 2.3.1.1).

  - We identify four features that bring CaaS to run for edge clouds as well as for clouds: consumer and vendor modes (Sec. 2.3.1.3), tenant resource quota for hierarchical namespaces (Sec. 2.3.1.4), variable slice granularity (Sec. 2.3.1.5), and federation support (Sec. 2.3.1.6).
  - We have developed the multitenancy framework that covers these four features as a free and open-source extension to Kubernetes (Sec. 3.2).
  - We benchmarked the three main multitenancy approaches using a representative implementation for each of them to explore their pros and cons from a tenancy-centered edge computing viewpoint (Sec. 3.3).
- *Integrated Federation Strategy*, comprising three levels as node, cluster, and system in Chapter 4, which expands the current CaaS federation approaches in a way that multiple providers can offer compute resources in many locations to a federated edge infrastructure while preserving the efficiency of our multitenancy framework [129, 131]. A list of contributions of this study:
  - We have devised an integrated federation strategy for CaaS to help establish a federated edge infrastructure at scale: node-wise federation (Sec. 4.1), cluster-wise federation (Sec. 4.2), and system-wise federation (Sec. 4.3).
  - We have developed a node deployment procedure (Sec. 4.1.2).
  - We have crafted a functioning prototype for cluster-wise federation, which removes the need for a centralized federation control plane while preserving the lightweight nature of our multitenancy framework (Sec. 4.2.1).
- *Edge Cloud Testbed* for researchers in Chapter 5, which improves software and hardware sustainability to provide and maintain a geographically distributed testbed as well as enhances their use to deploy experiments on such infrastructure [133, 130, 129, 131, 128, 132]. A list of contributions of this study:
  - We put our multitenancy framework into production as the EdgeNet testbed, which allows multiple researchers to conduct measurements in parallel (Sec. 5.1).
  - We have developed a node selection feature through which users deploy containers onto nodes based on their locations (Sec. 5.2).
  - We have made our node deployment procedure operational within the EdgeNet testbed, which allows contributors to provide the cluster with nodes (Sec. 5.3).
  - We have implemented an aggregate manager (AM) to federate EdgeNet with Fed4FIRE+ so that researchers can use EdgeNet as they do for the other federated testbeds (Sec. 5.4).
  - The EdgeNet testbed has supported more than 10 experiments (Sec. 5.5).

The following chapter provides the state-of-the-art passage, then Sec. 2.4 caters to this section with a concise description of the problem this thesis tackles.

Chapter 2

State of the art and problem statement

This chapter gives background information, discusses the rationale, provides the state-of-the-art in the domains in which our contributions exist, and concisely describes the problem statement. Background information is in Sec. 2.1 fundamentally related to cloud computing and edge computing, including containers and their orchestration, acknowledging the advancements in virtualization technologies. Sec. 2.2 discusses the reasoning behind our proposed architecture. Then a specific related work for each particular chapter is given in Sec. 2.3, which also presents how our work differs from others. We then introduce a concise description of the problem statement with challenges in Sec. 2.4. These four sections elucidate the problem domain, our motivation, and the problem that our contributions address.

2.1 Background

In this section, we first present a brief chronicle of centralization and decentralization history. We then introduce cloud computing and edge computing, discuss the advantages and disadvantages of each, argue the differences between the two, and look at their projected market sizes. Container orchestration is then explored.

2.1.1 Pendulum of centralization and decentralization

In the early age of the technology, centralization was more feasible due to costs and limited capacity of hardware [41, 109]. Mainframe systems running batch processes [8] were a noticeable centralization movement. Primarily, large organizations adopted these mainframe computers [168], which are then also used through the timesharing model [21, 8], but they were unavailable to the broad public access. With the advent of personal computers, centralized mainframe computing offloaded its tasks to a decentralized model [21]. This moment was another breakthrough in the computation domain.

Over the course of time, determining factors changed once again. Personal computers being constrained in resources and being required to be physically accessible to make use

of the compute resources and data became a limitation. The spread of the Internet created a possibility to consolidate IT resources at a centralized location and make them accessible through the Internet, with which the cloud computing paradigm emerged. Cloud computing enables broad on-demand access to compute resources at scale, with the pay-as-you-go pricing model protecting clients from up-front costs [16]. Over the past decades, virtualization advancements in concert with cloud computing, from virtual machines to containers, have changed how we develop and deliver applications. However, use cases such as autonomous vehicles and drone delivery, along with the rise of IoT, impose stringent latency and jitter requirements that clouds cannot meet. The more devices at the network's edge, the more likely bandwidth saturation clouds confront as well. Hence, the pendulum now swings toward decentralizing IT resources to meet these requirements, so toward edge computing that brings compute resources near locations where data is produced [137], as well as closer to end-users [76].

2.1.2 From clouds to edge clouds

As discussed in the previous subsection, the trend between centralization and distribution of IT hardware constantly shifts back and forth. Over the last few years, we have witnessed another change in interest toward decentralization via edge computing.

2.1.2.1 Cloud computing

The centralization trend via cloud computing[1] has changed how we develop and deploy applications as well as services over the last two decades. Clouds provide their customers with on-demand access to IT resources such as storage, computing, and network resources through the Internet, ensuring broad network access to these resources, along with the pay-as-you-go model that prevents the customers from undergoing upfront investments for needed infrastructure. What makes the cloud distinctive is also the way of delivering on-demand access to resources. In simple terms, a cloud provider manages infrastructure on behalf of customers.

There are many reasons a customer chooses cloud computing over purchasing and maintaining on-premise infrastructure. It can be, for example, financial such as upfront costs, or operational such as the need for employing on-site engineers for infrastructure management. Then a question arises: to what extent is it feasible for a customer to delegate the responsibility of managing resources to a cloud provider?

This question is addressed by cloud computing, which offers diverse service models that provide different levels of abstraction for infrastructure management. For example, such abstraction delivered by a service model can span different layers, such as hardware, virtualization, operating system, and application. Cloud operators enable various abstraction alternatives through many service models [117, 116] among which IaaS, PaaS, and SaaS

[1] The oldest use of the cloud computing term with respect to its current context dates to the mid-1990s, and the term itself has been broadly adopted in the mid-2000s [114].

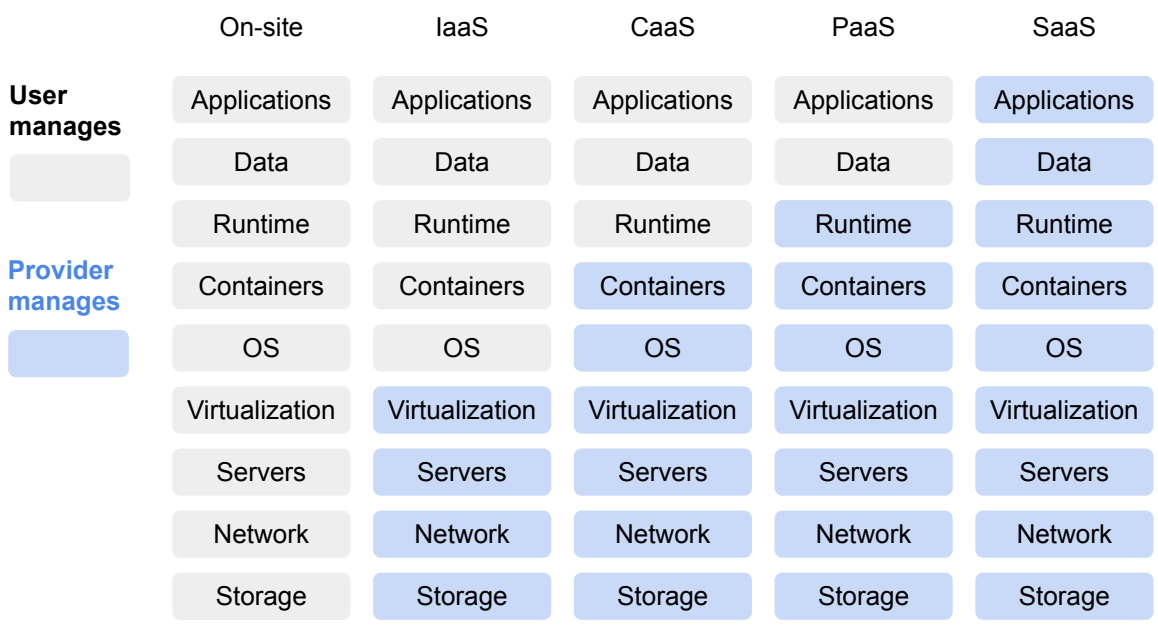


Figure 2.1: Levels of abstraction in cloud computing for on-premise (on-site) infrastructure, IaaS, CaaS, PaaS, and SaaS. This figure is derived from a Red Hat article [61].

are the three primary ones [95]. Cloud computing, including these service models, typically uses hypervisors for hardware virtualization, allowing multiple customers dynamically share infrastructure in isolation through multitenancy [13].

On the other hand, advancements in OS-level virtualization make containers an appealing option to isolate workloads. Although containers cannot provide the same isolation as hardware virtualization offers [139], they are more lightweight and faster, impose lower overhead, and perform better in many aspects than VMs [44, 111]. This more lightweight and portable nature of containers compared to VMs has led to the rapid uptake of CaaS in recent years, which falls between IaaS and PaaS regarding control abstraction. Fig. 2.1 depicts a high-level comparison of the abstraction level in resource management for self-managed, on-premise infrastructure, IaaS, CaaS, PaaS, and SaaS.

Regarding the abstraction levels, an organization that has and maintains self-managed, on-premise infrastructure is responsible for entirely managing it from the hardware to applications, as well as for security risks. Such an organization can offload some of its hardware and virtualization management responsibility, including risks, to a cloud provider through IaaS, becoming a customer. Meaning that, in IaaS, customers rent compute resources on-demand with the help of hardware virtualization.

Customers, who want more abstraction in management control, can take advantage of the flexibility of CaaS while disengaging from OS-related tasks. CaaS is based on OS-level virtualization, so containers, where customers benefit from underlying compute infrastructure, typically via container orchestration tools [139]. A user manages the lifecycle of containers through a container orchestration tool, which we explore in greater detail in Sec. 2.1.3: some operations are their deployments, scale, and networking.

PaaS abstracts control further by managing runtime on behalf of customers, in broad terms, providing them with the tools for application development and deployment. These are such tools with which customers can develop, build, and deploy their applications while committing minimal effort. As the last one, SaaS distinguishes itself as a software delivery method that uses multitenancy, where users belonging to different tenants commonly access an instance of the software via a web browser. This reduces the burden on software vendors: no longer delivering software to each customer, reduced maintenance workload, and accelerated software release cycle.

Cost-effectiveness, scalability, latency reduction, broadband network access, and robustness of clouds are the pillars behind the constantly increasing adoption of cloud services. With its above advantages, cloud computing allows consumers to access scalable compute resources instantly and at an affordable price, giving the power to consumers [20]. Cloud computing combines certain characteristics with a set of enabler techniques, through which it makes renting out compute resources that are consolidated at a central location via the Internet *economically viable*. This economically feasible model for both providers and consumers contributes to the continuous growth of cloud computing. It is forecasted that worldwide end-user spending on cloud services, including IaaS, PaaS, and SaaS, will reach $494.7 billion in 2022, a 20.4% enlargement from the previous year [52], and such an upward trend is estimated to persist to 2030 at least [29].

2.1.2.2 Edge Computing

The cloud computing paradigm converges compute resources through centralized data centers that are located relatively distant from the end-users and devices than are for envisaged edge computing infrastructure; each hop between these data centers with the end-users and devices causes more latency, bandwidth consumption, and more jitter. Although cloud providers establish sites in different regions to shorten this physical distance, such an infrastructure is of limited use for bandwidth-intensive applications as well as for latency-critical applications, even including latency-sensitive ones. Network congestion in the cloud may also lead to delays and jitter, impeding real-time communications that are required by some applications. Given that connected devices are expected to proliferate in the following years [30, 53], cloud bandwidth will likely become considerably more saturated.

Contemporary domains such as IoT, video analytics, and industry 4.0 introduce novel application requirements, due to which cloud architecture's capabilities have been put in question. With edge computing, compute resources, appearing in numerous locations compared to clouds, are brought closer to the end users and devices in order to achieve three critical enablers: low latency, low jitter, and high bandwidth, all provided by local or near-local processing. A high-level comparison of cloud computing and edge computing is presented in Table 2.1.

Given that edge computing extends CDN's concept,[2] which emerged in the late 1990s, to running computer code, similar to how cloud computing runs code but near end-users and devices, CDNs can be considered the initial footprints of edge computing [125]. In CDNs,

[2] CDNs cache content at edge nodes close to consumers in order to save bandwidth and provide low latency.

Table 2.1: A concise summary of cloud computing and edge computing, derived from a paper that reviews edge computing reference architectures [141].

	Cloud computing	Edge computing	
		Service Provider Edge	User/Device Edge
Strengths	Scalable, big data processing, broad network access, secured facilities, easier maintenance.	Data security, lower latency and jitter, saved bandwidth, cloud-like capabilities.	Real time responses, ultra low latency, low jitter, high bandwidth, autonomous.
Weaknesses	High latency and jitter, bandwidth bottleneck, no offline mode, little control over data locations.	Limited scalability of compute resources, operational costs due to numerous locations.	Constrained compute resources, limitations of user-owned private networks.

Edge computing does not replace cloud computing but rather supports it.
Many edge workloads will run in conjunction with those running on clouds [24].

saved bandwidth and low latency are ensured through caching, but in edge computing, these are provided through processing at the edge, which also presents low jitter and involves devices in addition to end-users. 1997 is the first time edge computing's potential value has been demonstrated by offloading computational tasks from a resource-constrained mobile device to a nearby server for speech recognition and advancing this approach to improve battery life; two breakthrough improvements were made in the following years [125]: the foundational concept discussing two-level architecture, cloud infrastructure as it is, and distributed resources called cloudlets [126], along with the term fog computing, which refers to scattered cloud infrastructure [18].

In short, the foremost characteristic of edge computing is its end-user and device-centric approach, deploying compute resources where real-time processing needs to happen. This way, it ensures low latency, low jitter, and reduced bandwidth consumption, which some edge use cases require. The second one is its ubiquitous infrastructure, maintaining sites in widely dispersed locations that host compute resources. With this decentralization movement, real-time processing near devices will be achievable at scale extensively. Likewise, location-aware systems will put services on one or more of the feasible edge nodes that are in the closest proximity to end users. Through such infrastructure, edge computing, therefore, will guarantee not just low latency, low jitter, and reduced bandwidth consumption but also broad access to these benefits.

We contend that multiple operators will establish the edge sites occurring at widely dispersed locations [126], making interoperability more challenging than is in the cloud. It is not only the multi-provider aspect that hinders interoperability but also the heterogeneity that comes along with this decentralized infrastructure. By the heterogeneity term, we refer to nodes with varying environments such as locations, network settings, hardware in terms of amount and type, operating systems, device connection protocols, and API types. Pulling all together, the lack of interoperability becomes a hindrance to edge computing's adoption.

With 5G, the MEC technology enables cloud-like capabilities at the edge, more precisely at the edge of the wireless access networks [88]. This technology takes advantage of the wireless access technologies provided by 5G [36], allowing edge services to be offered at the network edge closer to end users [122], thus ensuring low latency and high bandwidth performance. ETSI addresses the above-mentioned interoperability issue through *Inter-MEC system communication* while standardizing MEC paradigm [99].

Since the trend is now toward decentralization, hyperscaler cloud service providers extend their architecture with the goal of reaching all regions rather than remaining with their core locations [144]. Similar to cloud computing, the edge computing market has constantly been growing in recent years. By 2025, it can achieve an economical size between $175 billion and $215 billion in hardware [30]. In terms of overall spending on hardware, software, and services, it is forecast to reach $274 billion through 2025 [31].

2.1.3 Container orchestration

Containers provide virtualization at the OS level, harnessing *Linux namespaces* for isolation and *Linux control groups (cgroups)* for resource consumption, thus being lightweight and portable compared to VMs [96]. Namespaces, such as user, process ID, network, and mount, are used to confine a container by removing both its ability to see and access the environment outside the container, and cgroups is responsible for controlling the resource consumption of a group of processes, including setting limits for them [44]. This lightweight OS-level virtualization approach, taking advantage of these two key features, makes containers an alternative to VMs. Containers are lightweight, faster, and more performant and even introduce less overhead compared to VMs, but they cannot ensure isolation at the same level that VMs offer.

Above mentioned benefits contribute to the rapid adoption of containers by the industry, thus stimulating container orchestration tools. A container orchestration tool, in broad terms, manages the entire lifecycle of containers from their deployment to networking, including scaling up and down according to demand. Someone who wishes to deploy containerized services to the cloud has a choice of open source container orchestration systems with which to do so, four of the most prominent being [69]: Apache Mesos,[3] Docker Swarm,[4] Kubernetes,[5] and Rancher's Cattle.[6] We focus on Kubernetes, as it has in recent years become the de facto industry standard. All of the major cloud providers offer Kubernetes-based CaaS to their customers (See Table 2.2). And Datadog, a company that provides cloud monitoring and security services, reports [32] that nearly 50% of their customers that deploy containers use Kubernetes to do so, this having increased about 10 percentage points over the past three years. We have, therefore, implemented our contributions in this thesis in a way that natively extends Kubernetes.

Container orchestration has a scheduling aspect for which each tool offers its own solution. Constraint satisfaction problems and scheduling are two domains of research that the researchers have intensely been studying, so research on the scheduling algorithms for container orchestration is also conducted in regard to cloud computing and edge computing. This thesis does not tackle this aspect, so the scope of this thesis does not include scheduling algorithms. However, we propose a federation architecture in Sec. 4.2 where each cluster can employ its own scheduler, thanks to Kubernetes which natively supports replacing the

[3] Apache Mesos https://github.com/apache/mesos
[4] Docker Swarm https://github.com/docker/swarmkit
[5] Kubernetes https://kubernetes.io/
[6] Rancher's Cattle https://github.com/rancher/cattle

Table 2.2: Major cloud providers' Kubernetes-based containers-as-a-service (CaaS) offerings.

Cloud provider	Kubernetes-based CaaS offering	URL
Amazon Web Services	Elastic Kubernetes Service	https://aws.amazon.com/eks/
Microsoft Azure	Azure Kubernetes Service	https://azure.microsoft.com/en-us/products/kubernetes-service
Google Cloud Platform	Google Kubernetes Engine	https://cloud.google.com/kubernetes-engine
Alibaba Cloud	Alibaba Cloud Container Service for K8s	https://www.alibabacloud.com/product/kubernetes
Oracle Cloud	Oracle Container Engine for K8s	https://www.oracle.com/cloud/cloud-native/container-engine-kubernetes/
IBM Cloud	IBM Cloud Kubernetes Service	https://www.ibm.com/cloud/kubernetes-service
Tencent Cloud	Tencent Kubernetes Engine	https://www.tencentcloud.com/products/tke
OVHcloud	Free Managed Kubernetes	https://us.ovhcloud.com/public-cloud/kubernetes/
DigitalOcean	DigitalOcean Kubernetes	https://try.digitalocean.com/kubernetes-in-minutes/
Linode	Linode Kubernetes Engine	https://www.linode.com/lp/kubernetes/

default scheduler with a custom one.[7] For reasoning a need for such architecture, we present a quick overview of scheduling below without delving into them for simplicity.

Due to the sheer amount of resources available in public clouds, scheduling persists as a research interest [70, 136, 26] to improve cost-effectiveness through achieving high resource utilization of the infrastructure. Many studies have already explored the effectiveness as well as scheduling architecture of container orchestration tools, along with their other aspects [63, 127, 159, 156]. We think that scheduling solutions depend on the context: a scheduler can consider image transmission cost, CPU and memory usage, hardware requirements for containers, and container grouping as main factors [86], and another may prioritize eco-friendly objectives to ensure lower carbon emissions and energy consumption [75]. Similarly, the approach can differ for optimization; one can frame the scheduling problem as minimum cost flow problem [66, 65], whereas another may make use of ant colony optimization [73]. The heterogeneous nature of edge computing introduces diverse factors, which depend on the context, regarding scheduling, which studies address in various ways [87, 45, 7, 85, 62]. As there is no one-size fits all approach to solving scheduling problems, there are also efforts that empower users to set their constraints on the cluster configuration along with their optimization objectives [146].

2.2 Rationale

We envisage a future in which tenants deploy services on a continuum of computing resources from cloud to edge cloud, about which we make the following assumptions:

- Edge clouds are ubiquitous, scattered across the world [37].
- Compute and storage resources are constrained in the edge cloud, making it harder to scale tenant workloads there than in the cloud.
- Tenants value the ability to easily move their workloads from one edge cloud cluster to another and between the edge cloud and the cloud.

[7] Kubernetes documentation: *Configure Multiple Schedulers* https://kubernetes.io/docs/tasks/extend-kubernetes/configure-multiple-schedulers/

- Each tenant's user database is maintained by that tenant. User management is not a functionality provided by the compute clusters.
- Tenants and their users are unreliable. They may purposely or accidentally harm each other, or the compute cluster, or themselves.

These assumptions instruct and support us in conceiving our architecture, and we describe the rationale for our design choices in the following subsections: the necessity of a novel Kubernetes CaaS multitenancy framework (Sec. 2.2.1) that takes container-specific security and performance considerations into account (Sec. 2.2.2), and that enables federation across edge clouds and control over slice granularity at the edge (Sec. 2.2.3).

2.2.1 Multitenancy

It is an often-repeated commonplace that cloud computing is not just "using someone else's computer", as the cloud goes beyond this to promise more flexible, convenient, and cost-effective access to computing resources. Multitenancy is required to realize this promise. The *NIST Definition of Cloud Computing* [95] mentions resource pooling as one of the "five essential characteristics" of cloud computing, saying that:

> *The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand.*

And in their *Defining Multi-Tenancy* paper from 2014 [72], Kabbedijk et al. state:

> *Multi-tenancy is a property of a system where multiple customers, so-called tenants, transparently share the system's resources, such as services, applications, databases, or hardware, with the aim of lowering costs, while still being able to exclusively configure the system to the needs of the tenant.*

Multitenancy is a standard feature of the three established cloud service models, Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS) [117, 14]. If Containers as a Service (CaaS) is to provide the promised benefits of the cloud and the edge cloud at scale, then it requires an efficient multitenancy model as well. We further discuss why such efficiency is required for CaaS to run for clouds and edge clouds in Sec. 2.3.1, and the results of our experiments in Sec. 3.3 support our contention.

Multitenancy has a broad meaning and can be enabled at different cloud abstraction layers using different techniques to share resources among multiple customers. This thesis discusses multitenancy in the context of CaaS and methods for accomplishing it. CaaS offerings are mostly based upon Kubernetes [170], so we focus on the ways in which it can serve multiple customers using multitenancy. To be clear, with respect to the discussion of multitenancy in the Kubernetes documentation, which describes how a tenant can deploy an application in a Kubernetes cluster to serve its multiple customers using a multi-tenant model: that is also multitenancy, but at the application layer, and more precisely at the SaaS layer, however it is not multi-tenant CaaS, which is what this thesis considers.

2.2.2 Security and performance

While multitenancy is an essential cloud feature, it raises security issues that researchers have been considering for over a decade [20], notably with respect to the IaaS service model [33]. For example, potential users are concerned about the security of their data when multiple tenants share the same infrastructure [12], and the resulting lack of trust can hamper cloud adoption [117].

Virtualization is used to isolate tenants from one another, but containers tend to offer weaker isolation [13], which introduces new concerns for multitenant container platforms [120], such as information leakage between colocated containers [51]. In general, Sultan et al. [145] have identified four categories of threat in containerized environments: malicious applications within containers, one container harming another, a container harming its host, and a container within an untrustworthy host.

In Kubernetes, container security must be considered in the context of the *pod*, which is that system's smallest deployable unit, consisting of a set of one or more containers. The Kubernetes pod security standards define three profiles, Privileged, Baseline, and Restricted.[8] However, these standards address a single-tenant environment, and so overlook some of the multitenant security issues mentioned above.

We therefore see the need for a solution that diminishes the security risks of running colocated containerized workloads. In order to be of interest for CaaS, such a solution needs to maintain the performance advantage of containers over virtual machines.

2.2.3 Edge computing, federation, and slicing

As described in the Linux Foundation's 2021 *State of the Edge* report [143], cloud-like infrastructure is being developed at the network edge in order to serve edge devices that produce bandwidth-intensive and/or latency-sensitive workloads. ETSI's multi-access edge computing (MEC) architecture [99] provides a standard structure for making servers at cellular operators' radio access networks available for the deployment of such workloads by third parties. That is, the emerging edge cloud will be a multitenant cloud [5].

Since the MEC architecture anticipates that workloads may be containerized, we argue that there is a need for a multitenant CaaS framework that meets the specific requirements of the network edge. The prime edge requirements that we identify are federation and variable slice granularity.

MEC facilities will be provided by multiple operators. Just as a mobile phone user is able to roam from one regional operator to another today, a mobile edge device will need to be able to connect to different operators and find its containerized edge services spun up near each base station to which it connects. And ETSI describes a requirement for edge devices to be able to engage in low-latency interactions with each other when they are near each other,

[8] Kubernetes documentation: *Cloud Security Standards* https://kubernetes.io/docs/concepts/security/pod-security-standards/

even if they are connected to different operators' base stations. ETSI uses the term *federation* to describe such interoperability scenarios.

To enable federation, we argue, a CaaS framework must support the deployment of third parties' containers across multiple operators' edge clouds. That is, the framework will not just be multitenant, it will also be multi-provider, with providers furnishing geographically dispersed heterogeneous resources. Those who deploy CaaS services to a multi-provider environment will be in need of a unified interface that simplifies the task of moving workloads between remote clusters that are owned by different providers [166].

In addition, as anticipated by the Next Generation Mobile Networks Alliance in 2016 [35], operators will have to support third party services that put a much more heterogeneous set of requirements on their networks than is currently the case. Extreme requirements are incompatible with a one-size-fits-all approach. The way that MEC handles this is through *slicing* [88, 100, 169], which allows network and compute resources to be allocated and custom-configured to meet the specific needs of individual services. In the CaaS context, we argue that no single slice granularity will meet the full range of needs. The standard CaaS sub-node-level slicing, in which containers are provided from a shared resource pool on individual node, while no doubt appropriate for many services, will not be appropriate for those that are the most sensitive to performance variation. For those services, node-level slice granularity will be needed.

2.3 Related work

We subdivide this section into three in the sense that each subsection corresponds to a chapter in the thesis. This decision is based on the opinion that each major contribution should be accompanied by its corresponding related work for convenience.

- *Multitenancy* (Sec. 2.3.1) corresponds to *a native multitenancy* (Chapter 3).
- *Federation* (Sec. 2.3.2) corresponds to *a federation that spreads by local action* (Chapter 4).
- *Platforms and tools* (Sec. 2.3.3) corresponds to *a real-world instance as a distributed testbed* (Chapter 5).

In each chapter, we further examine the corresponding state of the art. We believe this choice makes the related work section more precise for readers while providing them with more context in the chapters.

2.3.1 Multitenancy

In the commercial cloud offerings, each customer gets their own Kubernetes cluster, which is a straightforward form of multitenancy. Some providers add on more advanced features. For example, an Amazon EKS customer can use a service called Fargate[9] to manage the

[9] Amazon Web Services (AWS) Elastic Kubernetes Cloud (EKS) documentation: *Fargate* https://docs.aws.amazon.com/eks/latest/userguide/fargate.html

capacity of their Kubernetes cluster, adding and removing nodes as they need to. Similarly, a Google Cloud customer can hand over control of their cluster capacity management to a service called Autopilot,[10] to do the same thing for them automatically.

While Kubernetes multitenancy in this form might be fine for large centralized data center clouds, there are drawbacks when looking to an edge cloud future. Setting up a separate cluster for each tenant is far from the most efficient approach, as we will show in Sec. 3.3. Resources are liable to be underused, which will be of particular concern in the smaller data centers that we can anticipate at the edge. And when tenants need to be repeatedly instantiated as their workloads migrate, for instance at one roadside cabinet after another to serve vehicles that are moving along a highway, spinning up an entire cluster for each arrival of a tenant risks taking too much time. We anticipate that lighter forms of multitenancy will be needed: ones that allow more efficient resource sharing, even at some cost in workload isolation, and that allow more rapid creation and deletion of tenants. Furthermore, proprietary systems for enabling multitenancy risk being a hindrance in a federated environment, in which a single customer might deploy their workloads to many edge clouds, each owned by a different operator. If all of the operators use a common open-source multitenancy framework, it will promote interoperability.

Starting in 2019, as Table 2.3 shows, a fair number of open-source Kubernetes multitenancy frameworks have been developed. Some, such as Virtual Kubelet [154] and frameworks that are derived from that code, take the same starting point as the commercial services, which is each tenant having its own cluster. But others offer worker nodes to tenants out of a shared cluster, which is more resource efficient. And some of these serve multiple tenants out of a shared control plane, which is yet more efficient.

The Kubernetes community has recognized the importance of developing such frameworks, as evidenced by the fact that one of the Kubernetes working groups, of which there are just eight,[11] is devoted to multitenancy.[12] Both of the frameworks that this working group supports take the shared cluster approach. VirtualCluster (VC) [155] offers a separate control plane to each tenant while the control plane is shared among tenants by the Hierarchical Namespace Controller (HNC) [151]. These frameworks, along with the others shown in Table 2.3, comprise the essential related work for our own EdgeNet framework.

We look at six aspects of Kubernetes multitenancy frameworks when comparing EdgeNet to the related work: the multitenancy approach (Sec. 2.3.1.1), the customization approach (Sec. 2.3.1.2), support for consumer and vendor modes (Sec. 2.3.1.3), management of tenant resource quotas (Sec. 2.3.1.4), support for variable slice granularities (Sec. 2.3.1.5), and support for federation (Sec. 2.3.1.6).

[10] Google Cloud documentation: *Create an Autopilot cluster* https://cloud.google.com/kubernetes-engine/docs/how-to/creating-an-autopilot-cluster

[11] Kubernetes working groups https://github.com/kubernetes/community/blob/master/sig-list.md

[12] Kubernetes Multi-tenancy Working Group https://github.com/kubernetes-sigs/multi-tenancy

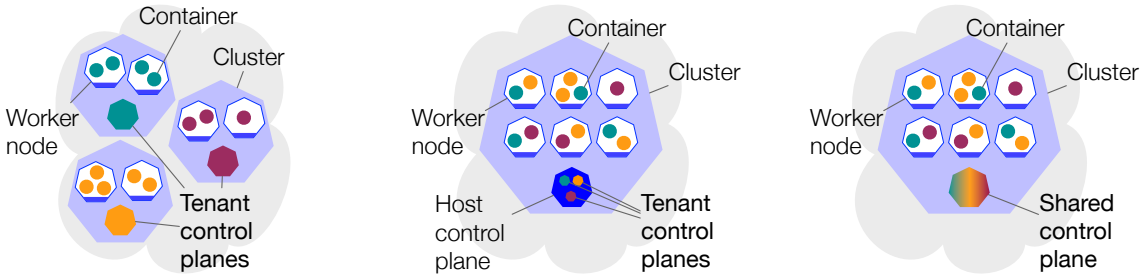
Table 2.3: Comparison table of related work (open-source Kubernetes multitenancy frameworks).

	EdgeNet	**HNC**	**Capsule**	**kiosk**	**Arktos**	**VC**	**k3v**	**vcluster**	**Kamaji**	**VK+**
	v1.0.0-alpha.5	v1.0.0	v0.3.1	v0.2.11	v1.0	v0.1.0	v0.0.1	v0.15.0-alpha.1	v0.2.1	VK v1.8.0
	2023-04	2022-04	2023-03	2021-11	2022-03	2021-06	2019-07	2023-03	2023-02	2023-03
Multitenancy Approach										
Multi-instance										
- Through Multiple Clusters										●
- Through Multiple Control Planes						●	●	●	●	
Single-instance										
- Single-instance Native	●	●	●	●	●					
Customization Approach										
Control Plane										
- Full Control Plane View						●	●	●	●	●
- Tenant-wise Abstraction					●					
- Flat Namespaces			●	●						
- Hierarchical Namespaces	●	●								
Data Plane										
- SSH Access to Worker Nodes									●	Partial
Consumer & Vendor Modes										
- Consumer Mode	●	●	●	●	●	●	●	●	●	●
- Vendor Mode	●									●
Tenant Resource Quota	●	Incomplete	●	●	●	●		●	●	●
Variable Slice Granularity										
- Node-level Slicing	●	●	●	●	●	●	●	●	●	●
- Sub-node-level Slicing	●	●	●	●	●	●	●	●		●
- Automated Selection	●									
Federation Support	●				Unknown					●

Short name	**Name**	**First release**	**Source code**
EdgeNet	EdgeNet	2019-10	https://github.com/EdgeNet-project/edgenet
HNC	Hierarchical Namespace Controller	2019-11	https://github.com/kubernetes-sigs/hierarchical-namespaces
Capsule	Clastix Labs' Capsule	2020-09	https://github.com/clastix/capsule
kiosk	Loft's kiosk	2020-02	https://github.com/loft-sh/kiosk
Arktos	Centaurus's Arktos	2020-04	https://github.com/CentaurusInfra/arktos
VC	VirtualCluster	2021-06	https://github.com/kubernetes-sigs/cluster-api-provider-nested/tree/main/virtualcluster
k3v	Rancher's k3v	2019-07	https://github.com/ibuildthecloud/k3v
vcluster	Loft's vcluster	2021-04	https://github.com/loft-sh/vcluster
Kamaji	Clastix Labs' Kamaji	2022-05	https://github.com/clastix/kamaji
VK+	**Virtual Kubelet based frameworks**		
	Virtual Kubelet	2018-02	https://github.com/virtual-kubelet/virtual-kubelet
	Liqo	2020-10	https://github.com/liqotech/liqo
	Admiralty	2018-12	https://github.com/admiraltyio/admiralty
	Tencent's tensile-kube	2021-02	https://github.com/virtual-kubelet/tensile-kube

2.3.1.1 Multitenancy Approach

The scientific literature describes two approaches to enabling CaaS multitenancy: multi-instance [60], and single-instance native [71]. We ourselves further distinguish between multi-instance through multiple clusters and multi-instance through multiple control planes, making three approaches altogether, as shown in Table 2.3. The approaches are illustrated in Fig. 2.2 and we describe them as follows:



(a) **Multi-instance through multiple clusters.** Each tenant receives a separate cluster, including both the control plane and worker nodes. This imposes considerable overhead.

(b) **Multi-instance through multiple control planes.** A physical cluster is divided into logical ones, each offered to a different tenant. It can reuse worker nodes and the networking of the host cluster.

(c) **Single-instance native.** This low overhead approach has all tenants share a single cluster and a single control plane. Isolation between tenants is ensured by logical entities such as K8s namespaces.

Figure 2.2: **Multitenancy Approaches.** The multi-instance approaches provide each tenant with its own instance of the control plane (or, at the least, of certain control plane components) and, optionally, its own set of worker nodes, ensuring better isolation between tenants. The single-instance native approach caters to multiple tenants through a single control plane, while having them share the resources of a single set of worker nodes, thereby providing improved performance.

Multi-instance through multiple clusters. Fig. 2.2a illustrates the multi-instance through multiple clusters approach, in which each tenant receives its own cluster. The proprietary commercial CaaS offerings (see Table 2.2) are structured in this way, but there is no open-source framework to enable precisely this form of multitenancy, spinning up and spinning down full Kubernetes clusters on demand for different tenants. Existing open-source tools for deploying Kubernetes clusters, such as RKE[13] and Kubespray,[14] do not address multitenancy.

There is, however, a set of open-source Kubernetes frameworks that do address multitenancy for the case in which there are already multiple tenants, each of which possesses one or more of their own clusters, even if these frameworks do not spin up or spin down the clusters on demand. These frameworks, based on the code of **Virtual Kubelet** [154], a sandbox project of the Cloud Native Computing Foundation, are designed to allow workloads from one cluster to be deployed to another cluster. Their primary focus is on cross-cluster deployment in general, and multitenancy arises only in the specific case of clusters belonging to different tenants, but since they do enable this sort of multitenancy, we examine the advantages and disadvantages of doing so.

[13] RKE https://rke.docs.rancher.com/
[14] Kubespray https://kubespray.io

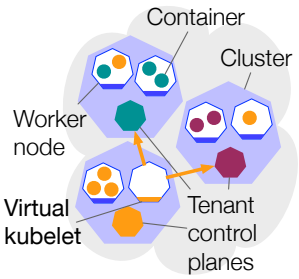


Figure 2.3: **Multitenancy through Virtual Kubelet.** The virtual kubelet masquerades as the kubelet of a node in a cluster, but in reality deploys workloads from that cluster to other clusters via those clusters' APIs. When a local cluster belongs to one tenant and a remote cluster belongs to another tenant, this results in a distinctive form of multitenancy, with clusters that, while otherwise belonging to one tenant, host pods from other tenants.

As illustrated in Fig. 2.3, Virtual Kubelet establishes a connection from one cluster to another by leveraging Kubernetes' *kubelet*[15] API. A kubelet is the agent that runs on each node of a Kubernetes cluster in order to manage the life cycles of pods, which are groups of containers associated with a workload. By implementing the kubelet API, a virtual kubelet masquerades as the kubelet of an individual node, but is in reality a stand-in for the remote cluster. It, in turn, uses the remote cluster's control plane API to deploy and manage workloads on that cluster.

Although we might think of this as a small scale form of federation, the Virtual Kubelet authors expressly say that "VK is not intended to be an alternative to Kubernetes federation", by which we understand a full-featured and scalable federation. Similarly, as we have mentioned, Virtual Kubelet is not primarily designed for multitenancy. By contrast, EdgeNet is designed precisely for federation and multitenancy. While similar to Virtual Kubelet in the sense that EdgeNet introduces agents to transfer workloads from one cluster to another, EdgeNet avoids the overhead associated with each tenant having its own cluster. This is because, in EdgeNet, it is the cloud and edge cloud providers that possess the clusters. Provider ownership of the clusters also means that an EdgeNet tenant can rely upon a provider to ensure the privacy of its workloads, rather than relying upon another tenant to do so.

Liqo [153], **Admiralty** [150] and **tensile-kube** [149] are all based on the Virtual Kubelet code. Liqo is one of the few frameworks to date to be the subject of a peer-reviewed scientific paper [67]. The authors are careful to state that some of the issues that arise from multitenancy, such as the manner in which the workloads of different tenants in the same cluster are isolated from each other, remain to be addressed.[16] Sec. 3.1.2 describes our proposed resolution for this problem.

Multi-instance through multiple control planes. In the multi-instance through multiple control planes approach, all tenants are supported by a single cluster, but each tenant acquires

[15] Kubernetes documentation: *kubelet* https://kubernetes.io/docs/concepts/overview/components/#kubelet

[16] From the Liqo paper [67]: "Specifically, we foresee a *shared security responsibility model*, with the provider responsible for the creation of well-defined sandboxes and the possible provisioning of additional security mechanisms (e.g., secure storage) negotiated at peering time."

its own control plane within that cluster, as illustrated by Fig. 2.2b. In doing so, the approach gives each tenant a full control plane view for customizing its environment. One or more nodes are dedicated to supporting the tenant control planes, and containers isolate those from each other within those nodes. Isolation based upon containers, or containers grouped into pods, imposes lower overhead than isolation based on VMs. There are variants to this approach, in which some control plane components, like the scheduler, are shared among tenants, while others, such as the API server and database, are duplicated so as to provide one instance to each tenant.

Frameworks that follow this approach differ in how they isolate tenant workloads from each other. If tenants share a common set of worker nodes as they do in VirtualCluster, k3v, and vcluster, the degree of isolation will depend upon the container runtime used to run the containers. If each tenant acquires its own dedicated set of worker nodes, as happens in Kamaji, then there is better isolation.

VirtualCluster [155] is one of the two open-source frameworks incubated by the Kubernetes Multi-Tenancy Working Group. It virtualizes the control plane components per tenant, with the exception of the scheduler. For isolation between the worker nodes of different tenants, it uses Kata containers [113].

A drawback of VirtualCluster is the cost of providing separate control plane components per tenant. In a peer-reviewed scientific paper [170], the VirtualCluster authors state that this cost is a blocking point when more than a thousand tenants are in the cluster. We compare how the multiple control plane approach scales as compared to the shared control plane approach by benchmarking the relative performance of VirtualCluster and EdgeNet in Sec. 3.3 and find that the shared control plane approach allows far more tenants to be allowed into a given cluster, and allows more tenants to arrive within a short period of time. In a federated edge cloud environment, where we anticipate limited resources, large numbers of workloads, and the rapid propagation of workloads from one cluster to another, the shared control plane approach has a clear advantage. In fairness to VirtualCluster, it is designed for a different sort of environment.

In Rancher's **k3v** [112], the control plane is virtualized on a per-tenant basis, similar to VirtualCluster, but it does not provide data plane isolation, as VirtualCluster does. Exceptionally among the frameworks, k3v does not provide a mechanism for managing tenant resource quotas, as mentioned in Sec. 2.3.1.4.

vcluster [90], not to be confused with VirtualCluster, is one of two open-source frameworks developed by Loft, the other being kiosk, which takes the single-instance native approach. In the control plane, each vcluster has a separate API server and data store. Workloads created on a vcluster are copied into the namespace of the underlying cluster to be deployed by the shared scheduler.

Kamaji [28] is one of two open-source frameworks developed by Clastix Labs, the other being Capsule, which takes the single-instance native approach. Kamaji enables Kubernetes multitenancy by running tenant control planes as pods on the same host cluster, known as the admin cluster. It provides control plane isolation as well as strong workload isolation to tenants, as each tenant is also allocated to its dedicated worker nodes. As these tenant worker nodes are not shared, if they need to be run on the same hardware to reduce overhead,

isolation can be done through the use of virtual machines, which still introduces considerable overhead.

Single-instance native. In the single-instance native approach, all tenants share a single control plane and a common set of worker nodes, as illustrated in Fig. 2.2c. Control plane isolation is ensured through a logical entity, such as Kubernetes namespaces, that introduces negligible overhead but provides less control plane isolation compared to a multi-instance approach. An implementation issue, for instance, could lead to an isolation break between tenants. Workload isolation depends upon the container runtime, as it does for the multi-instance through multiple clusters that are federated approach and for the multi-instance through multiple control planes with shared worker nodes approach.

This approach demands significant coding work to give each tenant an experience akin to using their own separate cluster.

The single-instance native approach's scaling advantage is illustrated by a scenario examined by Guo et al. in which it supported thousands of tenants, as opposed to just dozens for a multi-instance approach [60]. It also has lower operational costs [25]. And it is lighter weight for workload mobility, allowing containers to be spun up and spun down with less overhead than in a multi-instance approach, as we show through benchmarking in Sec. 3.3. For these reasons, we have adopted the single-instance native approach for EdgeNet.

The Hierarchical Namespace Controller (**HNC**) is one of the two open-source frameworks incubated by the Kubernetes Multi-Tenancy Working Group, the other being VirtualCluster. HNC takes the single-instance native approach, whereas VirtualCluster takes the multi-instance through multiple control planes approach. HNC uses a hierarchical namespace structure in order to enable multitenancy.[17] Functionalities such as policy inheritance that allows replicating objects across namespaces are built upon this hierarchy.

Aspects of this work that have inspired our own multitenancy framework are its hierarchical namespace structure and the terminology that it employs. We have also designed our own framework to avoid what we perceive to be its defects:

- HNC does not enforce unique names for namespaces, opening the possibility for namespace conflicts.
- HNC's quota management system is not aligned with the hierarchical namespace structure so as to limit a child's quota based upon its parent's quota, though community documentation states[18] that work is underway to enable this.
- HNC's quota management system allows namespaces without quota to coexist alongside namespaces that have quotas, which puts those quotas at risk (See Fig. 2.5b and discussion in Sec. 2.3.1.4).

Capsule [27] is one of two open-source frameworks developed by Clastix Labs, the other

[17] Kubernetes Multi-tenancy Working Group documentation: *HNC: Concepts* https://github.com/kubernetes-sigs/hierarchical-namespaces/blob/master/docs/user-guide/concepts.md

[18] Kubernetes Multi-tenancy Working Group documentation: *HNC: Policy inheritance and object propagation* https://github.com/kubernetes-sigs/hierarchical-namespaces/blob/master/docs/user-guide/concepts.md#policy-inheritance-and-object-propagation

being Kamaji. Capsule takes the single-instance native approach, whereas Kamaji takes the multi-instance through multiple control planes approach. This is one of two frameworks that adopts flat namespaces (See Sec. 2.3.1.2) as its customization approach, along with kiosk. A tenant can create multiple namespaces, but however, this responsibility belongs to the owner of that tenant. Tenant owners can create resources across a set of namespaces of their preference, and the cluster administrator can copy resources to the namespaces of various tenants. Although this approach facilitates the management of multiple namespaces that belong to a tenant, so it eases management complexity, it may not be fully scalable for extensive tenant settings, as we further discuss in Sec. 2.3.1.2. Capsule aims at allowing an organization to share a single cluster efficiently, hence not accounting for the needs of the envisaged edge computing infrastructure.

kiosk [89] is one of two open-source frameworks developed by Loft, the other being vcluster. The kiosk framework takes the single-instance native approach, whereas vcluster takes the multi-instance through multiple control planes approach. This solution uses flat namespaces approach, as does Capsule, for customization. A tenant is represented by an abstraction called an *account*, and an account can create a namespace through an entity called a *space*. Each space is strictly tied to only one namespace. This framework permits the preparation of templates that can be employed during namespace creation, facilitating the automated provisioning of resources as defined within these templates in the designated namespaces. Despite alleviating management complexity, this approach still shares Capsule's limitations stemming from flat namespaces. Multi-cluster tenant management is listed on their roadmap, but the project does not seem to be under active development, as the latest commit in its main branch was around a year ago.

Centaurus's **Arktos** [22] takes the single-instance native approach to multitenancy. As discussed in Sec. 2.3.1.2, it is the only framework that takes a tenant-wise abstraction approach to enabling customization. Arktos achieves this through API modifications,[19] which may require a significant amount of effort to keep aligned with the upstream Kubernetes control plane code. Its architecture primarily consists of three main software entities: an *API gateway* that receives tenant requests, a *Tenant Partition (TP)* that gives the illusion of each tenant acquiring an individual cluster, and a *Resource Partition (RP)* that operates on resources like nodes [39]. Although not all of its features are precisely presented, based upon our reading of their documentation, we consider that this solution addresses some federation aspects, such as scalability and cloud-edge communication. They provide a vision of consolidating 300,000 nodes belonging to different resource partitions into a single regional control plane. However, the main branch of their project repository has not received commits for around a year, implying that it may not be currently undergoing active development.

2.3.1.2 Customization Approach

Containers-as-a-service cannot scale to a large number of tenants if the mechanism by which each tenant obtains the environments in which to deploy its workloads, and configures each environment to meet the needs of its workload, requires manual intervention at every stage

[19] Arktos documentation: *Multi-tenancy Overview* https://github.com/CentaurusInfra/arktos/blob/master/docs/design-proposals/multi-tenancy/multi-tenancy-overview.md#api-server

by the cloud administrator. Each tenant should have a degree of autonomy to: create and delete the environments in which its workloads can be deployed; obtain resource quotas and assign them to those environments; and designate users for the environments, assign roles to those users, and grant permissions based upon those roles. Some combination of automation of these processes and delegation of administrative responsibility is needed to enable that autonomy. In Table 2.3, we call the way in which a multitenancy framework does this its *Customization Approach*.

By giving each tenant its own control plane, which the tenant's administrator can use to configure its environments as they wish, the multi-instance frameworks provide the greatest flexibility. We call this approach the **Full Control Plane View**. As Table 2.3 shows, it is offered by the frameworks that follow the multi-instance through multiple clusters approach (Virtual Kubelet based frameworks), since each cluster has its own control plane, and, of course, by the multi-instance through multiple control planes approach (VirtualCluster, k3v, vcluster, and Kamaji).

Some of these frameworks (Kamaji and, partially, Virtual Kubelet based frameworks) allow additional server environment configuration to take place regarding the data plane by enabling SSH access to worker nodes, and this is noted as **Data Plane** customization in the comparison table. In Virtual Kubelet based frameworks, administrators of a tenant that owns a cluster can typically access the worker nodes in that cluster by SSH, but not the ones in other clusters, and this is classified as **Partial** in the comparison table.

In frameworks that follow the single-instance native multitenancy approach, some extensions to Kubernetes are required in order to safely enable customization. This is because in standard Kubernetes, giving a tenant's administrator the permissions necessary to configure their own environments means giving them the ability to configure other tenants' environments as well. Since there is no control plane isolation mechanism other than namespaces, an administrator who has permission to create, modify, and delete namespaces can do so freely across the board. Rather than hand out such permissions, a single-instance customization approach needs to provide one or more custom resources that a tenant's administrator can access, and the controllers of those will ensure safety while configuring the tenant environment on the administrator's behalf.

Among the single-instance frameworks, Arktos employs the most elaborate customization approach: that of introducing a new abstraction, beyond namespaces, by which to isolate tenants from one another in the control plane. As this abstraction is meant to capture the notion of a tenant, we refer to it in Table 2.3 as a **Tenant-wise Abstraction**. Our concern about this approach is the amount of development work that it might entail, both to develop this new abstraction and to maintain its compatibility with Kubernetes' upstream version of the control plane code.

Instead of introducing an entirely new abstraction, frameworks can build on Kubernetes' existing control plane isolation mechanism: namespaces. We identify two ways of doing so. The simpler one, followed by Capsule and kiosk, is to follow the standard Kubernetes approach, in which each namespace exists independently of every other namespace. This is described as **Flat Namespaces** in Table 2.3.

Another way, but one that requires more development work, is to provide controllers

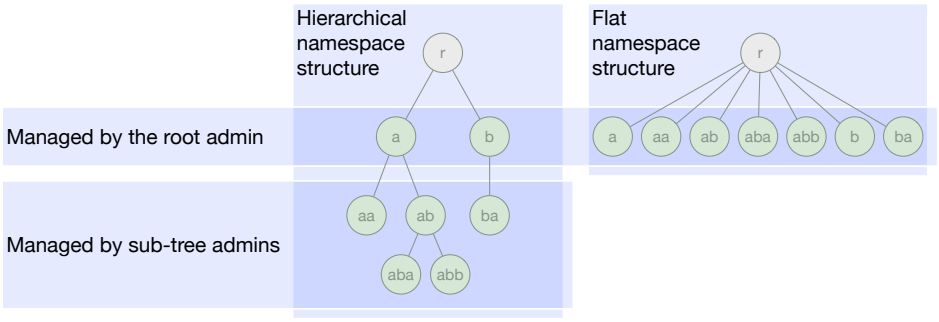


Figure 2.4: **Customization Approach: Hierarchical versus Flat Namespaces.** The same seven namespaces organized into a hierarchy (left) and without a hierarchy (right), in each case under a root environment *r*, which is not itself a namespace.

The hierarchy captures relationships between the namespaces: *a* and *b* are the core namespaces belonging to two tenants, whereas the others belong to sub-trees of those core namespaces. For example, *aa* and *ab* are subnamespaces of *a*. They belong to the same tenant as *a* and they may inherit a portion of that tenant's resource quota, user roles, and the permissions that accompany those roles. Likewise, *aba* and *abb* belongs to the same tenant as *ab* and may inherit from it. Management of tasks such as the approval of new namespaces, and the modification of quotas, users, etc., can be delegated to each tenant's administrators, and, further down the hierarchy, to sub-tree administrators.

The flat structure does not express these relationships. For example, no mechanism provides for *aa* to inherit from *a*. If they are to share configuration parameters, this needs to be expressly requested by the common administrator of the two namespaces. There are efforts to solve this issue through configuration templates to be applied to multiple namespaces. Nevertheless, as the number of namespaces that a tenant has grows, it results in management complexity for the root admin of this tenant, which makes it challenging to keep track of independent namespaces.

that keep track of the relationships between namespaces, such as several namespaces all belonging to the same tenant. Since the two frameworks that do this, EdgeNet and HNC, do so by maintaining a hierarchical structure through which to track the relationships, we identify this approach as **Hierarchical Namespaces** in Table 2.3.

Fig. 2.4 compares the two namespace structures. A hierarchical structure permits configurations to be inherited and allows for configuration tasks to be delegated, offloading tasks from administrators at the top of the hierarchy to administrators further down. The prime disadvantage of a flat namespace structure is that, even with automation, the root admins of tenants are highly solicited. EdgeNet adopts a hierarchical namespace structure, which is implemented by the architecture described in Sec. 3.2.1 and Sec. 3.2.2.

2.3.1.3 Consumer and vendor modes

Cloud services generally support two types of tenancy: **Consumer Mode**, in which the tenant is the end user of the resources; and **Vendor Mode**, in which the tenant can resell access to the resources to others.

The type of tenancy affects the visibility that the manager of a tenant has into that tenant's isolated environments. For a consumer tenant, these environments are generally termed *workspaces*, and they are created to be used by the members of that tenant's group or organization. A manager of a set of workspaces needs visibility into who the users of each workspace are, and needs fine-grained control over the rights of those users with respect to those workspaces. But a vendor tenant manages a set of subtenant environments that are destined for its own customers. A customer expects a certain level of privacy, with the users

and user rights of their subtenant environment remaining hidden from the vendor.

As shown in Table 2.3, all of the CaaS multitenancy frameworks that we have studied support consumer tenancy, but only EdgeNet and Virtual Kubelet based frameworks support vendor tenancy. We expect that the same commercial logic that has driven other cloud service models towards both forms of tenancy will lead to support for vendor tenancy being generalized for containers-as-a-service.

In order to enable any sort of tenancy, a system must support authorization and isolation mechanisms. It requires greater expressiveness to support both consumer and vendor tenancy than it does to support consumer tenancy alone. Such expressiveness, for example, allows a tenant to create a subtenant for the purpose of reselling its own allocated resources. This can be done in different ways depending upon the multitenancy approach:

- *Multi-instance through multiple clusters:* A tenant who owns a cluster can open this cluster for use by one of its subtenants. Because of the ease of doing so, we indicate Virtual Kubelet based frameworks as offering support for a vendor mode, even though their documentation does not explicitly mention this. However, since such an approach requires a cluster per tenant, this introduces high overhead, as our benchmarking shows in Sec. 3.3.
- *Multi-instance through multiple control planes:* A tenant could create a subtenant generated with its subtenant control plane instance running on top of the tenant control plane instance. None of the frameworks that we have studied currently do this.
- *Single-instance native:* A tenant can create a subtenant assigned with private namespaces that the tenant is solely authorized to remove. EdgeNet, having adopted the single-instance native approach to multitenancy, builds consumer and vendor modes on top of its hierarchical namespace structure. The implementation is described in Sec. 3.2.2.1 and illustrated in Fig. 3.5.

2.3.1.4 Tenant resource quota allocation

Resource quotas are popular in commercial settings, where they provide a basis for providers to bill their customers. In situations where resources are constrained, quotas are also a simple means by which to ensure an equitable allocation of those resources. Quotas are commonly used in the cloud, and Kubernetes supports them by providing a mechanism for allocating quotas to namespaces.[20] The Kubernetes mechanism is conceived for the relatively small scale scenario of a single organization using a cluster, and an administrator who manually sets resource quotas per namespace so as to share out the resources among different teams in the organization. A multitenancy framework that is built on Kubernetes needs to automate this process, to enable it to scale.

As Table 2.3 shows, all of the Kubernetes multitenancy frameworks that we have studied offer a mechanism for managing tenant resource quotas, with the exception of k3v. We classify k3v in this way as we consider its mechanism to be incomplete. In that framework,

[20] Kubernetes documentation: *Resource Quotas* https://kubernetes.io/docs/concepts/policy/resource-quotas/

which is no longer under active development, a cluster administrator can set a resource quota in the host namespace of a virtual cluster, but the tenant will not be aware of it.

In the edge cloud, we can expect resources to be more constrained than in the cloud, and so the need for a quota allocation mechanism is even stronger. Since our EdgeNet framework is designed for the edge cloud as well as the cloud, such a mechanism is a required feature of the framework.

Having made the design decision to use a hierarchical namespace structure, our quota management system needs to follow that structure. This means building in dependencies between quotas: as shown in Fig. 2.5a, at each node in the namespace tree, quota must be shared out between the parent namespace located at that node and the sub-trees that are rooted at the children of that node. EdgeNet's quota implementation is more thoroughly described in Sec. 3.2.3.

The only other framework that uses hierarchical namespaces, HNC, also allows quota to be shared out hierarchically. The mechanism employed in doing so relies on Google Cloud's Hierarchy Controller[21] as its foundation. But since it does not require that a quota be attributed to each namespace, it can end up constraining some namespaces while not constraining others, opening the possibility for a sub-tree to not enjoy the full resource quota that it has been allocated, as shown in Fig. 2.5b. In EdgeNet, quotas apply either to the entire tenant namespace hierarchy or not at all, so this problem cannot arise.

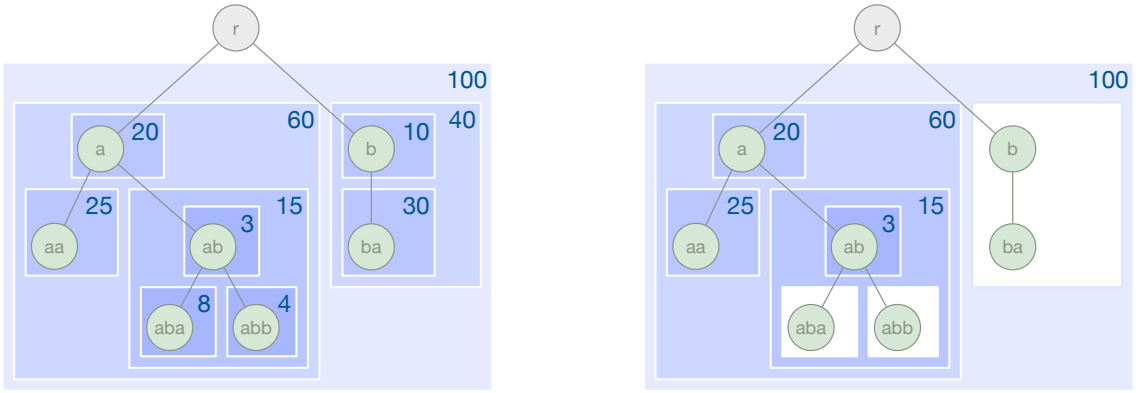
Resource quotas can be wasteful of resources if they are not used fully, while best-effort distribution of resources is more efficient without providing guarantees. None of the Kubernetes multitenancy frameworks provides an intermediate solution. Providing such a solution is on the EdgeNet development road map.

2.3.1.5 Variable slice granularity

We use the term *slicing* to refer to a mechanism that enables multitenancy by dividing a larger pool of resources into smaller portions, each portion being for the exclusive use of one of the tenants. For CaaS, the larger pool is a compute cluster that consists of nodes, which may be either physical servers or virtual machines. But what size should a smaller portion be: a full node, or a subset of the resources of a node? A subset can be acquired through the use of container, sandboxed to a greater or lesser degree, as Sec. 3.1.2 will describe. Fig. 2.6 depicts the different possible node and slice granularities. In our estimation, neither of the slicing granularities is ideal for all use cases, and a multitenancy framework should offer both, and automate the ability to switch between them.

Node-level Slicing (Figs. 2.6a and 2.6b). Slicing at this granularity, which is offered by all of the frameworks that we have studied, provides a tenant with one or more entire nodes, so that isolation of a tenant workload is ensured at the level of the node in which it runs. By this means, it offers greater freedom in choosing a container runtime to support a particular containerized workload. And it can better ensure stable access to resources. Reserving an

[21] Google Cloud Anthos: Hierarchical Resource Quotas https://cloud.google.com/anthos-config-management/docs/how-to/using-hierarchical-resource-quotas



(a) **EdgeNet example.** The quota allocated to a sub-tree must be divided among a portion reserved for the namespace at which this sub-tree is rooted and the portions allocated to each subnamespaces.

(b) **HNC example.** In the same hierarchy as sub-trees that are constrained by quotas, it is possible to have sub-trees that are not constrained in this way.

Figure 2.5: **Hierarchical allocation of resource quotas.** Examples of a quota of 100 being divided up among the sub-trees of a hierarchical namespace rooted at *r*. The tenant of the sub-tree rooted at *a* has been allocated a quota of 60, from which it reserves 20 for its core namespace and allocates 25 and 15 to the sub-trees rooted at *aa* and *ab*, respectively.

In EdgeNet, the quota of 15 must also be distributed within the sub-tree rooted at *ab*. For example, here, 3 is reserved for the namespace *ab* and 8 and 4 are allotted to the sub-trees rooted at *aba* and *abb*, respectively. Likewise, quota must be allocated to the sub-tree rooted at *b* and distributed within that sub-tree.

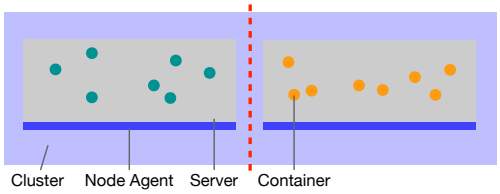
HNC, on the other hand, allows portions of the hierarchy to be free of quotas. In this example, in HNC, the administrator of namespace *ab* has, perhaps inadvertently, not set quotas for its subnamespaces, and likewise for the tenant administrator of *b*. If workloads in *aba* and *abb* were to exceed a resource consumption of 12 or the workloads at *b* were to consume resources exceeding 40, other namespaces with quotas might not be able to fully enjoy the resources quotas that had been reserved for them.

entire physical server (Fig. 2.6a) can be valuable, in particular, for a tenant that needs to meet an unusual requirement, such as guaranteed access to GPU resources. However, when entire nodes are reserved for tenants, some nodes might be under-utilized.

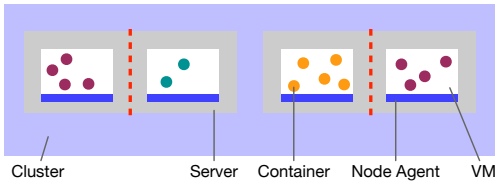
Sub-node-level Slicing (Figs. 2.6c and 2.6d). Sub-node-level slicing improves the ability of a cluster to maximize the efficiency of its resources. This is enabled through containers where each container on a node takes a portion of its resources. Isolation between multi-tenant workloads on the same host is provided at the level of containers, so it is weak. Better isolation can be ensured through container runtimes that provide sandboxes to containers. This approach restricts tenant autonomy in selecting a container runtime as there are just a few of them available.

As Table 2.3 shows, all of the CaaS multitenancy frameworks that we have studied offer node-level slicing, and all but Kamaji offer sub-node-level slicing. When it is available, sub-node-level slicing is the default. Upon the request of a tenant, a cluster administrator can manually configure node-level slicing.

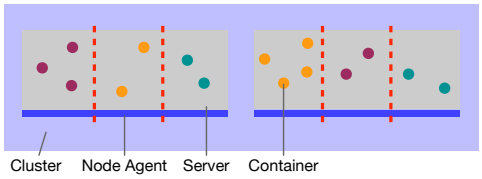
The EdgeNet framework is the only one for which the process of switching granularity is automated. Sec. 3.2.4 describes how we implement this. It might seem that the node-level slicing that we thereby enable suffers from all of the inefficiency of the multi-instance CaaS model that we critique (See Chapter 3), but this is not so, as our architecture preserves the single-instance efficiency of a single control plane.



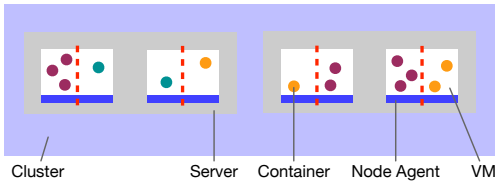
(a) **Node-level slicing of servers.** An entire node that is a physical server is made available to a tenant.



(b) **Node-level slicing of VMs.** An entire node that is a VM is made available to a tenant.



(c) **Sub-node-level slicing of servers.** A subset of resources of a node that is a physical server is made available to a tenant.



(d) **Sub-node-level slicing of VMs.** A subset of resources of a node that is a VM is made available to a tenant.

Figure 2.6: **Node and slice granularities.** Dashed vertical lines indicate how a cluster's resources are sliced so as to make those resources available to tenants. A node in a cluster can be a physical server (left illustrations) or a VM (right illustrations), presented as node granularities. Slicing can be performed so as to make an entire node available to a tenant (top illustrations) or so as to make a subset of a node's resources available to a tenant (bottom illustrations). Different node and slice granularities can coexist within a cluster (e.g., the scenarios shown in all four illustrations could appear simultaneously in a single cluster). Our EdgeNet multitenancy framework automates the process of varying the slice granularity, allowing a node to be reserved for a tenant, or returning a reserved node to the pool of nodes available to be subdivided.

2.3.1.6 Federation support

CaaS multitenancy frameworks have to date generally been aimed at the use case of a single cluster operator offering its resources to its own tenants. However, the resources of several operators from different regions or countries will generally be required by a tenant that wishes to provide its edge cloud based services to large numbers of end-users. Such a tenant might prefer to be the customer of just one operator and, through that operator, gain access to the others. We anticipate that operators will see a commercial interest in federation, which will allow them to more broadly commercialize access to their clusters. We also anticipate that operators will want to lower the barrier to entry for those who deploy services by allowing them to orchestrate their containers across multiple clusters with a single tool.

Many edge cloud services, such as cognitive services [37, 55], are expected to involve workloads that are spread across the cloud and the edge cloud [1], with workloads moving back and forth between the two, so there are voices in industry that argue [166], and we are convinced, that a unified, single interface for users is a necessity. As a first step towards this goal, the EdgeNet multitenancy architecture presents an essential first brick in such a federation architecture: the ability to generate object names that are universally unique to cluster and tenant. Such uniqueness avoids name collisions during the propagation of objects across clusters. The details of our implementation are found in Sec. 3.2.2.4.

Besides our EdgeNet framework, five of the frameworks that we study support scaling up the infrastructure that multiple tenants share, and four of them do so through federation: Virtual Kubelet based frameworks (See Table 2.3). Even for their main purpose of

enabling deployment of workloads to multiple clusters, Virtual Kubelet suffers from a significant drawback: Kubernetes' automatic scaling up and down of workloads to meet demand gets lost in remote clusters. This is because the Kubernetes objects that get deployed through a virtual kubelet are pods rather than the *Deployment* or *StatefulSet* workload resources that manage pod life cycles on a user's behalf, and the Kubernetes Horizontal Pod Autoscaling mechanism[22] in each cluster works on these sorts of objects, not on individual pods.

Like Virtual Kubelet, EdgeNet enables the deployment of workloads from local clusters on remote clusters, but EdgeNet handles this through an intermediate cluster between local and remote clusters. The intermediate cluster that does this for EdgeNet is called the *Federation Manager*. When a tenant, using its local cluster, makes a deployment in federation scope, the Federation Manager creates the deployment on the remote cluster on behalf of the tenant, as we discuss in Sec. 4.2.

Some of Liqo's extensions to Virtual Kubelet start to tackle some of the concerns that would arise in a multi-tenant federation, such as collisions between the names of namespaces generated in local clusters and in remote clusters. Liqo's solution is a naming scheme that ensures that the name of a namespace used by a workload will be unique in the remote cluster in which it is deployed.[23] However, the same workload risks running in namespaces with different names in different clusters, which can itself lead to problems. EdgeNet by contrast generates globally unique names that avoid collisions, and a workload runs in namespaces that carry the same name on all clusters to which it is deployed.

The other framework that provides for cloud-edge communication and significant scaling is Arktos, but we have been unable to determine whether federation is involved. Its stated aim is to achieve a single regional control plane to manage 300,000 nodes that multiple tenants will share.[24]

2.3.2 Federation

Kubernetes is single-tenant and single-provider by design. Neither has a multi-provider aspect for multi-ownership of nodes by default, nor it natively offers a federation solution. A lack of the former may impair scalability in the context of edge clouds. Without the latter, a cluster supports 5,000 nodes at maximum, meaning limited scalability. One can overcome this limitation by creating another cluster. However, each created cluster will be separated from others, which may lead to a problem, cluster sprawl, as cluster count increases, resulting in management difficulties. Furthermore, the applications and services will be logically confined within the boundaries of their host cluster regarding in-cluster networking. The Kubernetes community has already acknowledged the need for systems that provide inter-cluster solutions, as confirmed by the fact that a special interest group[25] is dedicated to managing multiple Kubernetes clusters, which includes a federation subproject called KubeFed.[26]

[22] Kubernetes documentation: *Horizontal Pod Autoscaling* https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

[23] Liqo documentation: *Namespace Offloading* https://docs.liqo.io/en/v0.7.0/usage/namespace-offloading.html

[24] Arktos documentation: *Large Scalability* https://github.com/CentaurusInfra/arktos#large-scalability

[25] Multicluster Special Interest Group https://github.com/kubernetes/community/blob/master/sig-multicluster/README.md

[26] KubeFed https://github.com/kubernetes-sigs/kubefed

KubeFed provides a centralized federation control plane that manages member clusters. Users communicate with this centralized federation control plane to deploy their services on the member clusters, introducing a single point of failure problem. In a split-brain scenario, it may prevent users from making use of the resources of one or more member clusters locally. Furthermore, the centralized federation control plane hampers scalability, and in terms of multitenancy, it can only support a limited number of tenants due to the namespace threshold we discussed in Chapter3.

Larsson et al. [82] introduces *Decentralized Kubernetes Federation Control Plane* to replace the centralized federation control plane of KubeFed. With the use of a shared database of conflict-free replicated data types (CRDTs), the authors state that the proposed architecture avoids the single point of failure problem that the centralized control plane for management causes as well as improves scalability. A distributed algorithm that takes advantage of CRDT storing scheduling status makes clusters iteratively collaborate to meet the desired state. This work aims to support thousands of clusters owned by a single organizational entity. We argue that thousands of clusters are not sufficient for edge cloud federation; furthermore, it does not discuss how to federate clusters provided by many different entities.

Federated Coalition Cloud with Kubernetes [6] achieves interoperability of sovereign clusters owned by different nations. It enables peer-to-peer discovery to eliminate a need for a central authority, thus avoiding central control planes so single point of failure problem. The authors contribute with main extensions; the first is Service and Resource Discovery APIs for available services and hardware capabilities, whereas the latter is Deployment Orchestrator, using data served by the mentioned discovery APIs, that does orchestration job and executes optimization algorithms. However, there are too few clusters in the federation to be able to assess its potential for a worldwide edge cloud federation.

Faticanti et al. [42] demonstrate Kubernetes Cluster Federation, using KubeFed, in a fog computing environment. The authors compare two setups; the former is a single control plane cluster that has worker nodes in separate regions, and the latter is a federation of multiple clusters consisting of one cloud and one edge region cluster. The results show that the cluster-wise federation approach is more resilient to network failures or a sort of service interruption as users can keep using the edge region control plane in order to deploy workloads. This approach disregards the fact that federation-level and cluster-local deployments may conflict, which is more likely to occur when the federation runs at scale. Furthermore, KubeFed offers a centralized control plane to manage member clusters, thus introducing a scalability issue considering the ubiquitous nature of edge computing infrastructure.

Although the federation provides more resiliency to network failures, as discussed above, instability in a geo-distributed Kubernetes federation based on KubeFed is further studied by Tamiru et al. [147]. This paper reveals that static configuration may lead to instability of deployments in such federation due to network conditions between host and member clusters. The authors also present a feedback controller that dynamically adjusts the configuration parameter, Cluster Health Check Timeout, that influences the stability most in order to overcome the instability problem. The experiments show that such an approach significantly increases the application deployment efficiency by decreasing the failures. Given that the experimental setup is relatively small in the context of edge computing, consisting of one host cluster and five-member clusters with distances between the range of 100 km to 850 km, it

lacks the proposed controller's evaluation for a worldwide federation. It shares the identical scalability drawback with previous work because of the centralized federation control plane.

Another work introduces an orchestration platform, mck8s [148], that offers policy-based scheduling, pod autoscaling, cloud cluster provisioning and autoscaling, and rescheduling in the context of multi-cluster management. Their scheduling policy significantly reduces the proportion of pods in a pending state from 65% in KubeFed to 6%. Furthermore, mck8s moves applications closer to where network traffic increases and scales them out as needed. They address the resource bottleneck of geo-distributed clusters by provisioning and de-provisioning cloud clusters according to the resource requirements of the multi-cluster deployment. However, centralized software entities come at the cost of scalability. The authors mention that deployment time increases as the number of clusters upsurges. This casts doubt on its suitability for an extensive federation consisting of numerous clusters.

There are also similar efforts to enable federation that uses Kubernetes [77, 68]. Karmada [152] is one of which that is further developed as a continuation of KubeFed. However, they share a similar drawback in terms of scalability that stems from having a centralized federation control plane. Since we expect to have numerous clusters federated in a typical edge computing scenario, centralized components also cause problems, such as latency between the federation control plane and member clusters. Another issue is whether multiple unreliable tenants share such federations or not. Last but not least, the number of tenants a centralized federation control plane can support directly sets bounds for the number of tenants that can access federated resources, regardless of how many member clusters there are. This also provokes a scalability deficit regarding multitenancy.

Another technique to enable Kubernetes-based cluster-wise federation relies on the Virtual Kubelet [154] abstraction that can register remote clusters as virtual nodes in a cluster. This technique allows users to make deployments as they do with upstream Kubernetes clusters without any API disruption. Another advantage is that it is possible to connect clusters to other systems through APIs. Liqo [67] expanding the functionality provided by Virtual Kubelet to enable multi-cluster topologies can establish a federation in which multiple providers can bring their own clusters. It makes propagation to remote clusters at the abstraction level of pods rather than higher-level ones such as deployments. In a split-brain scenario, as the life of a pod is ephemeral, a pod that dies can lead to weak resiliency. To overcome this issue, Liqo introduces ShadowPod, which holds the pod specification to enforce keeping the corresponding pod up and running on remote clusters. But this method requires another mechanism to manage pod autoscaling on remote clusters due to workload resources, such as deployments, not being created remotely, as we discuss in Sec. 2.3.1.6. Big virtual nodes may also become a scalability bottleneck in an edge computing scenario with many clusters, as do autoscaling the pods deployed on remote clusters through local operations. Regarding scheduling, if multiple virtual node redirections exist, the scheduling decisions must be made at each level, introducing latency and higher overhead. As discussed in Sec. 2.3.1.1, it offers a multi-instance multitenancy, leading to high overhead. Other Virtual Kubelet-based solutions, such as Admiralty [150] and tensile-kube [149], share similar drawbacks as Liqo.

Our proposed architecture offers an integrated architecture allowing providers to offer compute resources at the level of nodes, clusters, and systems level. Through node-wise

federation, besides larger providers, small-size providers can bring their compute resources in the form of nodes to edge clusters, which ensures an infrastructure at scale. Multiple providers can also offer these clusters with the aim of establishing a federation of clusters. In this case, a lightweight federation is formed, which consists of local worker clusters and regional federation managers.

Each worker cluster is shared by multiple tenants and runs workloads of federation tenants with the goal of achieving high resource utilization in geographically distributed edge clouds. Tenants gain access to federated resources through these worker clusters provided by their operators, which protects from the need for the centralized federation control plane with which users communicate. Federation managers, being responsible for resource discovery, object propagation, and federation-level scheduling, support three deployment models to federate a large number of clusters: standalone, peer-to-peer, and hierarchical.

In cluster-wise federation, scheduling occurs at two levels at most: the former at the federation manager and the latter at the worker cluster. Once a deployment is made in a worker cluster, the worker cluster scheduler can employ any required scheduling algorithm. Our federation tool works with workload resources such as deployments and statefulsets rather than pods in terms of workload placement. Given that our worker clusters manage the pod autoscaling of these with an existing mechanism, it removes a bottleneck, improving scalability. Eliminating a centralized federation control plane to manage member clusters also contributes to scalability. We further envisage system-wise federation to federate different systems, which highlights interoperability.

2.3.3 Platforms and tools

This subsection reviews similar efforts in two categories: endeavors to bring Kubernetes to the edge and the edge cloud testbeds that the networking and distributed systems research communities have provided.

2.3.3.1 Container orchestration at the edge

Much work has been done to adapt container technology to edge clouds [102, 103, 11]. A growing number of publications address various aspects, such as IoT task offloading, enabling long-running functions in containerization for IoT devices, and designing a scheduler for Kubernetes in Industrial IoT, which allows edge cloud nodes to consume less energy and to deploy applications in less time than usual. [40, 74, 75].

In Cloud4IoT [110], the authors discuss a platform using containers to deploy, orchestrate, and dynamically configure software components related to IoT and data-intensive applications while providing scalability in the cloud layer. The main difference with EdgeNet is that it concentrates on IoT solutions, meaning the device edge, whereas EdgeNet's core use case is for more somewhat more powerful nodes positioned at the network edge.

Kristiani et al. [79] provide an implementation of an edge computing architecture by taking advantage of OpenStack and Kubernetes to cover the cloud, the edge cloud, and the

device edge cloud. This implementation reduces the workload on the cloud side by assigning data processing tasks to the edge front to be performed at the edge of the network. The focus is different from the scope of this thesis, focusing on task offloading while providing communication between three layers.

KubeEdge [164], a project that is incubating within the Cloud Native Computing Foundation, offers a Kubernetes-based infrastructure that brings specific cloud capabilities to the edge. It aims to overcome edge computing challenges such as limited resources and non-connectivity. KubeEdge uses Docker as its containerization technology, Kubernetes as the orchestrator, and Mosquitto for IoT devices talking to edge nodes. Again, the focus is different from the contributions of this thesis: multitenancy, federation, easy node installation, and selective deployment.

Another project that brings Kubernetes to the edge is Rancher's Lightweight Kubernetes,[27] which focuses on lightweight Kubernetes for resource-constrained environments as does k0s[28] and MicroK8s,[29] which is a feature that a container orchestration tool should provide, but which is not the subject of the present study. Instead, we conceive our contributions to work with such certified Kubernetes distributions.

2.3.3.2 Internet-scale shared measurement platforms

The networking and distributed systems research communities have provided various edge cloud testbeds typically spanning broad geolocations such as PlanetLab [107], PlanetLab Europe, OneLab [43], Geni [94], Fed4Fire [34], Emulab [115], G-Lab [98], V-Node [101], Grid'5000 [17], Fit IoTLab [50], and Savi [84] in the past decades. All of these testbeds required dedicated hardware and delivered custom software. These two design decisions hindered efficiency and sustainability.

First, dedicated hardware has caused an increase in maintenance and scaling costs because of a need for on-site support and initial purchase investment. Thus, contributors abandoned nodes to their fate over time. Second, typically, these testbeds have been supported by researchers writing custom software. This introduces a heavy workload on coding and preparing tutorials for those who maintain that testbed. Furthermore, it commonly obliges an experimenter to learn a new control framework for each testbed. We draw lessons from decades-long experience (See Table 2.4) and provide an alternative approach to establish a production-grade, internet-scale, and general-purpose platform with extensive measurement capabilities.

The philosophy of the introduced testbed in Chapter 5 is different from these testbeds in two respects. It encourages contributors to supply virtual machines as a node instead of dedicated hardware, in order to decrease the cost of providing and maintaining the testbed. And to reduce programming and documentation workload, it adapts industry-standard open-source software for the needs of the testbed. Thus, we strive to attract potential contributors to the cluster and contribute back to the open-source community.

[27] Rancher K3s https://k3s.io/
[28] k0s https://k0sproject.io/
[29] MicroK8shttps://microk8s.io/

Table 2.4: Comparison table of testbeds.

	currently active	number of vantage points	open to run code	open to run measurements	open measurement data
Ark (prev. Skitter)	Y	over two hundred	to researchers	to researchers	Y
BisMark	N	over ten	to collaborators	to collaborators	Y
Dimes	N	hundreds	N	N	Y
Etomic	N	tens	to researchers	to researchers	Y
Geni racks	Y	tens	Y	Y	slice-dependent
iPlane	N	all PlanetLab nodes	N	N	traceroute-only
M-Lab	Y	hundreds	vetted only	Y	Y
NLanr Amp	N	one hundred	N	N	Y
Nimi	N	tens	Y	Y	slice-dependent
Ono plug-in	N	over one hundred thousand	N	N	N
perfSonar	Y	thousands	N	to NRen operators	N
Pinger	Y	tens	N	N	Y
PlanetLab {Europe}	Y	tens, formerly hundreds	Y	Y	slice-dependent
Ripe Atlas (prev. TTM)	Y	ten thousand	N	Y	Y
SamKnows	Y	thousands	N	N	N
Seattle	N	tens of thousands	Y	TCP and UDP	slice-dependent

2.4 Problem statement and challenges

We conclude this chapter with a concise description of the problem along with the challenges that this thesis addresses. The main objectives of our contributions in this thesis are then outlined again.

Presuming the occurrence of globally distributed edge cloud sites, which are not scalable as clouds, provided by multiple operators [24], nobody, even cloud providers, has a monopoly on knowing how to manage these edge clouds [106]. These edge clouds compel a service model or a set of service models that helps to achieve high resource utilization of each edge site while ensuring infrastructure cost-efficiency. An edge-adapted service model, through which numerous tenants are incentivized to deploy and move their workloads from one site to another in different locations [166], needs to be architecturally optimized to run on a ubiquitous and resource-constrained infrastructure. An edge cloud testbed, to be sustainable at scale, also needs to adopt such a service model and adapt it to its own requirements.

Regarding the challenges, it must first guarantee interoperability between edge resources that different providers deploy. Lacking such interoperability can rapidly lead to vendor lock-in problems, as is in clouds, which can lessen the cost-effectiveness of infrastructure investments due to the widely distributed nature of edge clouds. It must also be straightforward for customers to deploy workloads and move them across numerous edge clouds provided by multiple operators [135] according to demand. Otherwise, it will become a tedious task for a customer to create deployment per location and per provider and move them as needed. A relatively small edge cloud must be prepared to run workloads from potentially *hundreds of thousands* of tenants. This is also because, depending on end-user demand and device locations, such workloads will be moved across clusters at different locations.

Since the edge infrastructure is being built to support cloud-like workloads [143], it is likely that such a service model to be inherited from cloud computing but adapted to edge computing. We assert that the CaaS service model is well-adapted for this purpose. Thus,

the objectives of our contributions, introduced in Sec. 1.3, are twofold: the primary one is enabling a sustainable edge cloud testbed for researchers to leverage containers, and the second is bringing an alternative CaaS option to edge computing.

Chapter 3

A native multitenancy

Multitenancy is what makes cloud computing economical. From a single bare metal machine, a cloud provider can offer resources to multiple tenants, where each tenant is a customer that contracts for cloud services on behalf of one or more users. These resources are, for example, virtual machines in the Infrastructure as a Service (IaaS) service model, or tools for application development and deployment in the Platform as a Service (PaaS) model. Tenants that are prepared to accept less than perfect isolation from other tenants benefit from the lower prices that providers can offer thanks to more efficient use of the providers' hardware.

But, despite the greater efficiency of containers as compared to virtual machines, and despite recent improvements in ensuring isolation between containers, the cloud industry does not yet propose a multitenant Containers as a Service (CaaS) offering that takes advantage of these advances. What passes for CaaS today is in fact multiple side-by-side instances of single-tenant clusters of compute nodes, each cluster having its own container orchestration control plane and its own data plane, and isolated from other clusters through the use of virtual machines. For example, automated services such as AWS Fargate[1] and Google Autopilot[2] that manage cluster capacity on behalf of a user who is deploying containers to the cloud do not do away with virtual machine overhead and do not improve control plane efficiency.[3] In brief, although CaaS ought to offer greater efficiency than IaaS,[4] it does not yet do so.

With the emergence of the edge cloud, such efficiency will take on greater importance because resources will typically be more constrained than in the cloud. As part of the vision for 5G, it is projected that mobile network operators will become edge cloud providers, offering up compute resources from servers that are colocated with their wireless base stations [36, 88], at what is being termed the 'service provider edge' [135, 143, 144]. These operators are also expected to offer resources from their peering sites, or the 'regional edge' [134]. Such edge cloud instances will be data centers that are geographically dispersed to be closer to the

[1] Amazon Web Services' Fargate https://aws.amazon.com/fargate

[2] Google Cloud's Autopilot https://cloud.google.com/kubernetes-engine/docs/concepts/autopilot-overview

[3] Google Cloud documentation: Cluster Architecture https://cloud.google.com/kubernetes-engine/docs/concepts/cluster-architecture#nodes

[4] We make the assumption that IaaS is offered through virtual machines, which is commonly the case [44].

users of cloud services or to edge devices than are the centralized data centers that dominate the present-day cloud.[5] With fewer resources, an edge cloud will not scale as elastically as a cloud, yet it must be prepared to receive a large number of workloads that have been deployed to serve local users and devices.

The problem that we aim to resolve is how to move CaaS multitenancy away from a high-overhead multi-instance model to a more efficient one that will be suitable for the resource-constrained edge cloud. In the solution that we propose, multiple tenants share a single instance of the control plane, which is used to deploy containers that coexist within a single instance of a shared cluster, while still allowing tenants to enjoy isolation from each other as well as the opportunity to customize their resources.

Our multitenancy solution has the particularity that it is designed to work in a federated environment. Today, a cloud customer typically deploys their workloads to a single cloud provider, but if they want to extend those workloads to be close to users and edge devices, a customer will also need to obtain resources from multiple edge cloud providers [24].[6] Doing so will be easiest for a customer if those providers are federated (See Chapter 4), meaning that the customer will be able to contract with just one cloud or edge cloud provider and the customer will be able to deploy its workloads through a single interface offered by that provider [166], and the provider will manage the propagation of the workloads to the other providers. Accordingly, our multitenancy solution ensures that each cloud provider can accept tenant workloads that originate from other providers.

As we use the term, a *multitenancy framework* consists of a set of rules that govern how a cloud provider offers resources to its tenants such that each tenant can use their portion of the resources and configure those resources to meet their needs without regard for the presence of the other tenants. The rules address the creation of isolated environments, resource sharing, and user permission management. They determine which rights over resources are given to which tenants, under which conditions, and how those rights affect the relationships of other tenants with the same resources. The term equally well refers to the set of entities that are coded to enforce these rules.

In this chapter, we describe our framework, argue for it, and show how we have implemented it in EdgeNet, a production edge cloud.[7] What we henceforth refer to as the *EdgeNet multitenancy framework* is part of the larger EdgeNet code base,[8] which is free, liberally-licensed, and open source software that enables CaaS deployments to the edge cloud. It is designed as a set of extensions to the Kubernetes container orchestration system,[9] which is itself free, liberally-licensed, and open source. Our reasoning in building upon Kubernetes is that cloud customers will want to continue using this familiar system, which is today's de facto industry standard container orchestration tool.

As Kubernetes does not natively support multitenancy, others have identified the need for such an extension and have developed their own Kubernetes multitenancy frameworks.

[5] To be clear, we do not include low-powered IoT devices that are unable to run cloud-like workloads (the 'constrained device edge') [135] in our conception of the edge cloud that we anticipate for CaaS.

[6] In addition, a customer might bring resources to bear from its own 'user edge'.

[7] The EdgeNet testbed https://edge-net.org/

[8] The EdgeNet software https://github.com/EdgeNet-project/edgenet

[9] Kubernetes https://github.com/kubernetes/kubernetes

(See Table 2.3 for details) We show that the existing frameworks, while no doubt fine for the cloud, will not be suitable for CaaS in the edge cloud. There are a few prior studies concerning these frameworks [170, 56, 39], but this is the first study to situate them, and EdgeNet, within the existing scientific literature on cloud multitenancy.

Our contributions, and the sections of the chapters that address them, are as follows:

- We look at Kubernetes multitenancy frameworks through the lens of the scientific literature on cloud multitenancy and, in Sec. 2.3.1.1, we provide a novel classification of these frameworks into three main approaches: multi-instance through multiple clusters, multi-instance through multiple control planes, and single-instance native.
- Based upon our analysis of the literature, we distill out four features that we believe will promote a future in which CaaS can thrive, in particular at the network edge, and we describe how we have incorporated these features into the EdgeNet multitenancy framework: consumer and vendor tenancy in Sec. 2.3.1.3, tenant resource quota for hierarchical namespaces in Sec. 2.3.1.4, variable slice granularity in Sec. 2.3.1.5, and federation support in Sec. 2.3.1.6.
- We have implemented the EdgeNet multitenancy framework as a free and open-source extension to Kubernetes, and have put it into production as the EdgeNet testbed, as described in Sec. 3.2.
- Our EdgeNet multitenancy framework constitutes a prototype for the federation of clouds and edge clouds, as explained in Sec. 3.2.2.4, and we further describe this prototype in Sec. 4.2 for the future development of a full federation framework.
- We benchmark the three multitenancy framework approaches using a representative implementation for each approach, and we reveal their pros and cons from a tenancy-centered edge computing perspective in Sec. 3.3.

The chapter is structured as follows. Sec. 3.1 discusses design principles for a CaaS multitenancy framework, and Sec. 3.2 presents the architecture of the EdgeNet multitenancy framework that we have developed. In Sec. 3.3, we benchmark our framework against representative frameworks for two alternate approaches , and we conclude this chapter as well as point to our future work in Sec. 3.4.

3.1 Design decisions

Our vision for EdgeNet's multitenancy framework is to promote a future in which the CaaS service model can thrive, particularly at the network edge. We have made nine design decisions, listed below, to support this vision. The first six were discussed in relation to related work in Sec. 2.3.1, and the latter three are discussed in this section. The implementation details are provided in the Architecture section that follows (Sec. 3.2).

- **Multitenancy approach.** EdgeNet obtains the lower overhead offered by a *single instance native* approach to multitenancy, compromising on the isolation that would be offered by a *multi-instance* one (Sec. 2.3.1.1).
- **Customization approach.** We mitigate customization limitations that stem from the

single-instance approach through the use of hierarchical namespaces (Sec. 2.3.1.2).

- **Consumer and vendor tenancy.** We design EdgeNet to support both the *consumer* and *vendor* forms of tenancy (Sec. 2.3.1.3).
- **Tenant resource quota.** EdgeNet incorporates a control mechanism to manage the allocation of resource quotas in a hierarchical tenancy structure, allowing tenants to grant quotas to their subtenants and recoup those quotas from them (Sec. 2.3.1.4).
- **Variable slice granularity.** Considering that there is no ideal granularity at which to slice a compute cluster in order to deliver resources to tenants, we allow an EdgeNet cluster to be sliced into individual compute nodes or at a sub-node-level granularity (Sec. 2.3.1.5).
- **Federation support.** Our framework allows each EdgeNet cluster to receive the workloads of tenants from other EdgeNet clusters with which it is federated, while avoiding name collisions by generating object names that are unique to cluster and tenant (Sec. 2.3.1.6).
- **Kubernetes custom resources.** For ease of integration into existing systems and ease of adoption by users, we implement EdgeNet using the Kubernetes *custom resources* feature, rather than creating a wrapper around Kubernetes or forking the Kubernetes code (Sec. 3.1.1).
- **Lightweight hardware virtualization.** We compensate for the loosened isolation of workloads in the native approach through the use of lightweight hardware virtualization that is optimized for running containers (Sec. 3.1.2).
- **External authentication.** In a federated multitenancy environment, users will need to authenticate with remote clusters, and for that reason EdgeNet adopts an authentication method that is external to any individual cluster (Sec. 3.1.3).

3.1.1 Kubernetes custom resources

Kubernetes' custom resource feature[10] allows new entities to be added that, by the fact of their presence, extend the standard Kubernetes API, thereby maintaining backward compatibility with tools and interfaces that are familiar to users. By building our EdgeNet framework in this way, instead of as a wrapper around Kubernetes or as a separate system that interacts with Kubernetes, we increase the chances that the framework will be compatible with a variety of Kubernetes distributions. For example, we have successfully tested and run EdgeNet framework as an extension of k3s,[11] a lightweight certified Kubernetes distribution for IoT and edge computing.

We have containerized the EdgeNet extensions, and we provide them in the form of public Docker images and configuration files. The core Kubernetes code remains untouched, and there is no need to recompile any existing code that runs a cluster. Any cluster administrator can deploy the extensions to their cluster with a single *kubectl apply* command without the need to bring down the cluster or interrupt its work in any way.

[10] Kubernetes documentation: *Custom Resources* https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/

[11] K3s https://k3s.io/

Aside from the choice of Kubernetes and of Kubernetes custom resources, all of our other design decisions should, in principle, apply to enabling multitenancy in any other container orchestration tool.

3.1.2 Lightweight hardware virtualization

The choice of virtualization technology, in the context of edge computing, between hypervisors providing the best isolation and containers being lightweight [167], is a longstanding discussion. We prioritize virtualized environments as they are lightweight and incur low overhead; in so doing, we favor enhanced performance over delivering the best isolation [54]. A native framework with operating-system-level virtualization satisfies these requirements, but it presents security concerns having to do with containers sharing the same kernel. We want to offer each tenant the security of its own guest kernel, which hardware virtualization provides, but without going so far as to adopt a multi-instance approach that would negate the performance advantages of containers over VMs. Fortunately, this is possible through the use of lightweight virtual machines, which offer the isolation benefits of hardware virtualization while offering near-container-level performance. Our multitenancy framework therefore adopts a single-instance native approach with lightweight hardware virtualization.

We follow earlier work [49, 113] that has recommended the Kata runtime[12] for providing isolation between containers in a multitenant environment [158, 80, 2, 162]. Kata spawns a lightweight VM that is optimized to run containers, delivering near-container-level performance [158, Fig. 5] and better isolation than OS-level virtualization.

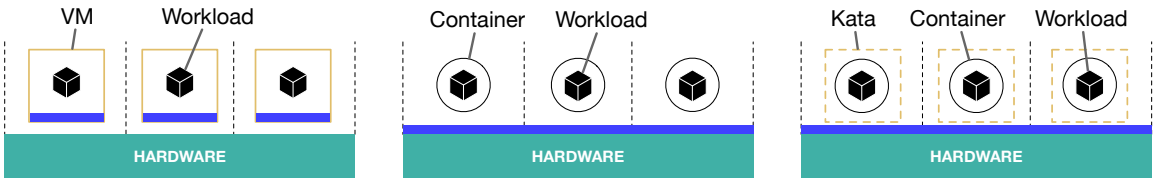
Fig. 3.1 depicts three methods for workload isolation: virtual machines, Docker containers, and Kata containers. We consider a single workload per method that can improve isolation and performance at the cost of overhead. One workload per virtual machine provides the best isolation among the three while introducing high overhead. The containerization technique can lower such overhead, having one workload per container, although it diminishes the isolation. The Kata method falls between VMs and containers in terms of isolation and overhead, as a containerized single workload runs in a lightweight virtual machine.

Tenants who require better isolation and performance at the same time, can obtain these using the *slice* software entity in our framework. As described in Sec. 3.2.4, this entity provides a tenant with the option of selecting container runtimes on an isolated subcluster so that the tenant can select one that meets its application requirements.

3.1.3 External authentication

A tenant's users must authenticate themselves in order to access the resources that they are authorized to access. For multitenant CaaS to run at scale, it is not feasible to require users to have individual accounts at every different cluster location where they will deploy their workloads [15]. Instead, authentication should be managed by an integrated identity management system. For example, an identity federation that consists of multiple identity providers, using

[12] Kata containers https://katacontainers.io/



(a) One workload per virtual machine, the best isolation among the three, introducing high overhead.

(b) One workload per container, the weakest isolation among the three, providing improved performance.

(c) One workload per container sandboxed by Kata, providing both intermediate isolation and intermediate overhead.

Figure 3.1: Methods for isolating workloads. The dashed vertical lines distinguish one tenant from another. Each thick blue horizontal line designates a node.

OpenID Connect (OIDC)[13] running on top of OAuth 2.0[14] as the authentication method, can support large-scale federations. With this in mind, EdgeNet uses this type of authentication (See Sec. 3.2.8).

3.2 Architecture

Our EdgeNet architecture has been conceived around the design decisions articulated in Sec. 3.1, with the aim of introducing as low overhead as possible while making Kubernetes ready for the edge. As a reminder, our main design decision has been to take a single-instance native approach, meaning that tenants share a cluster's control plane components and compute nodes, rather than having each tenant acquire its own control plane components and compute nodes. To compensate for the diminished isolation that comes with sharing the same cluster, EdgeNet uses lightweight VMs to isolate workloads while retaining low overhead.

The architecture of our EdgeNet multitenant CaaS framework is illustrated in Fig. 3.2. It is designed as a set of custom resources and custom controllers that extend Kubernetes from within. The framework consists of six principal new entities:[15]

- *Tenant* is the fundamental entity that isolates a tenant from other tenants (Sec. 3.2.1).
- *Subsidiary Namespace* is an isolated environment created by a tenant (Sec. 3.2.2).
- *Tenant Resource Quota* controls a tenant's use of resources (Sec. 3.2.3).
- Two entities, *Slice* and *Slice Claim*, allow dynamically reserving sub-clusters isolated from multitenant workloads, entitled node-level-slicing (Sec. 3.2.4).
- *Admission Control Webhook* enforces custom policies[16] such as employing Kata Containers for multitenant workloads (Sec. 3.2.5).

[13] OpenID Connect https://openid.net/connect/

[14] OAuth 2.0 https://oauth.net/2/

[15] EdgeNet multitenancy software entities: Principal custom controllers https://github.com/EdgeNet-project/edgenet/tree/v1.0.0-alpha.5/pkg/controller/core/v1alpha1, Assistant custom controllers https://github.com/EdgeNet-project/edgenet/tree/v1.0.0-alpha.5/pkg/controller/registration/v1alpha1, Admission control webhook https://github.com/EdgeNet-project/edgenet/tree/v1.0.0-alpha.5/pkg/admissioncontrol

[16] Kubernetes documentation: *Dynamic Admission Control* https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/#what-are-admission-webhooks

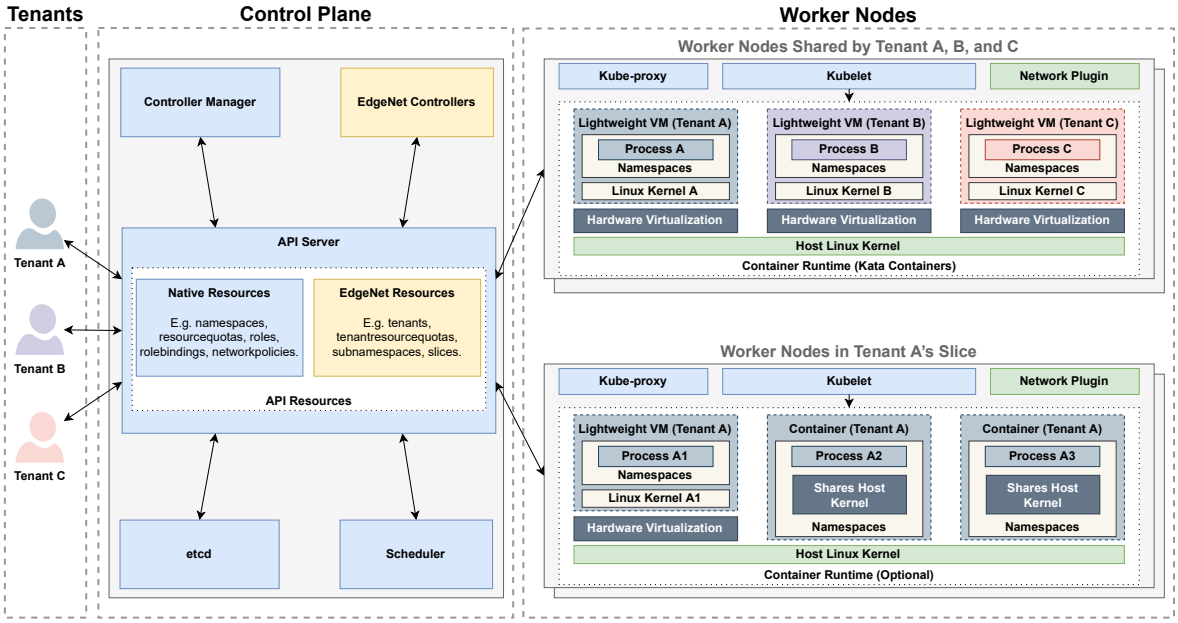


Figure 3.2: Architectural overview of EdgeNet multitenant CaaS framework implemented on Kubernetes. Much of the architecture is built upon native Kubernetes (blue) and third-party software components (green). Our innovation can be seen in the control plane, which consists of resources and controllers (yellow). This allows multiple tenants (left) to make use of the same control plane (center) via the API server to create workloads (right). What is more, our contributions enable the creation of two types of worker nodes; shared (top right) and reserved to a node-level slice (bottom right). Upon inclusion in a slice, a worker node's type switches to reserved, reverting to shared after slice termination. Multitenant workloads (top right) share the compute resources of shared worker nodes, yet, each is isolated from the others through hardware virtualization, lightweight virtual machines that are optimized for running containers, called Kata Containers. Every pod has its lightweight VM-based sandbox for isolation, and container(s) defined in a pod specification runs in the virtual machine tied to that pod. This single instance shared approach eliminates any overhead introduced due to employing conventional virtual machines for the isolation of single-tenant clusters from each other, addressing the overhead not only related to worker nodes but also the control plane. With that being said, worker nodes in a slice that is dynamically created by a tenant (bottom right) are isolated from multitenant workloads, hence providing the tenant with container runtime selection. In this way, the tenant can make the most of the advantages of containerization, such as lower overhead, and shorter creation and startup time. At the bottom right of the figure, several worker nodes are subclustered in Tenant A's slice, each hosting the workloads, for which the tenant employs Kata Containers as well as another container runtime like runC.

These are assisted by new entities that facilitate cluster and tenant management: *Role Request* (Sec. 3.2.6), and *Tenant Request* and *Cluster Role Request* (Sec. 3.2.7). Our architecture also covers user authentication via existing mechanisms (Sec. 3.2.8). Aside from these, it provides cluster operators with configuration files in YAML format that can be carefully customized, which define runtime class[17] and predefined role resources.

3.2.1 Tenant

In the context of the namespace structure maintained by the EdgeNet framework, the *Tenant* entity is a controller that acts at the top level of the hierarchy: creating, updating, and deleting the core namespaces of cluster-scoped tenants, which are the ones that are admitted into the cluster by the cluster's administrator. Here, we describe the *Tenant* entity, while Sec. 3.2.2 describes the *Subsidiary Namespace* entity, which acts lower down in the hierarchy, on the subtenants that are admitted either by top-level tenants or, recursively, by subtenants. The roles of these two controllers are shown in Fig. 3.3.

[17] Kubernetes documentation: *Runtime Class* https://kubernetes.io/docs/concepts/containers/runtime-class/

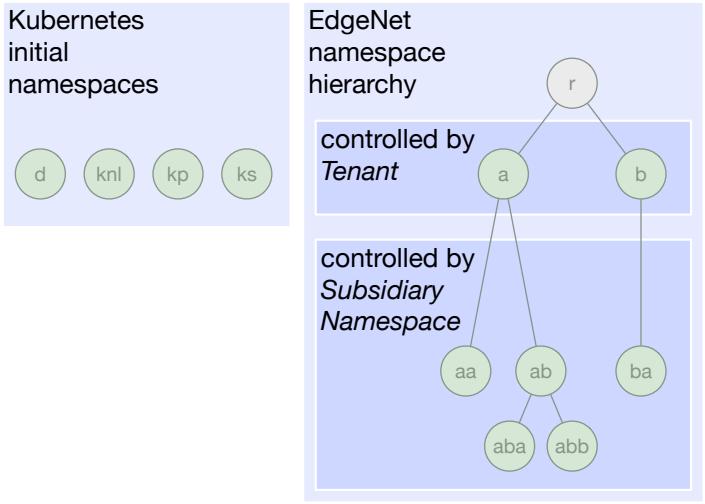


Figure 3.3: **Namespace Hierarchy in EdgeNet.** EdgeNet's multitenancy framework provides two principal controllers for managing its namespace hierarchy. The *Tenant* controller creates, updates, and deletes the tenant core namespaces at the top level of the hierarchy, while the *Subsidiary Namespace* controller handles all namespaces further down in the hierarchy.

In this example, *a* and *b* are tenant core namespaces, directly under the root of the hierarchy, *r*, which is not itself a namespace; the subsidiary namespaces are *aa*, *ab*, *aba*, *abb*, and *ba*.

Kubernetes' initial namespaces, `default` (*d*), `kube-node-lease` (*knl*), `kube-public` (*kp*), and `kube-system` (*ks*) are not included in the hierarchy and are not managed by these controllers.

The *Tenant* entity handles the creation of a tenant environment in a cluster. It starts by generating a namespace for this tenant, using a tenant name supplied by the tenant and checked for uniqueness among the cluster's namespaces. Because the tenant will be able to create its own hierarchy of namespaces rooted at this namespace, we distinguish this one, which will be at the root of any sub-tree that the tenant creates, by calling it the tenant's *core namespace*.

The controller also applies four labels to the namespace:

- *kind=<namespace-type>*, which is *core* in this case;
- *tenant=<tenant-name>*, the name supplied by the tenant;
- *tenant-uid=<tenant-uid>*, a locally-generated unique identifier for the tenant;
- *cluster-uid=<cluster-uid>*, the UID of the *kube-system* namespace.

UIDs are defined in Kubernetes as being 128-bit-long universally unique identifiers [83],[18] and the Kubernetes community suggests using the UID of the *kube-system* namespace as a cluster identifier.[19] The labels allow the tenant namespaces to be consumed by policies and other entities locally. This labeling model is also required for the inter-cluster object propagation mechanism.

Each tenant has an owner who has control over the tenant and its resources, including any subnamespaces that the tenant might create. Having created the core namespace, the

[18] Kubernetes documentation: *Object Names and IDs; UIDs* https://kubernetes.io/docs/concepts/overview/working-with-objects/names/#uids

[19] Cluster ID API discussion in the Kubernetes Architecture SIG mailing list https://groups.google.com/g/kubernetes-sig-architecture/c/mVGobfD4TpY/m/uEjVVsinAAAJ

Tenant entity uses the Kubernetes role-based access control (RBAC) mechanism to grant this control, while at the same time limiting the tenant owner's control to the scope of its core namespace, so that it may not interfere with other tenants' namespaces. The *Subsidiary Namespace* entity will be responsible for extending the scope of the owner's control to the subnamespaces. With their control over the core namespace, the owner can manage the tenant by, among other things: admitting users; granting roles, which are sets of permissions, for those users; and deploying workloads.

Kubernetes' network policies allow confining pod communication into a namespace or set of namespaces by using labels. In our multitenancy framework, the policies consume the UID labels, as specified earlier, attached to tenant namespaces. Since tenants have complete authorization on their network policies, an authorized user can, willingly or not, misconfigure network policies in a namespace, thus resulting in security threats. To overcome this vulnerability, we let a tenant enable or disable cluster-level network policy in the tenant specification, which confines the tenant's namespaces thanks to VMware's Antrea.[20]

3.2.2 Subsidiary namespaces

Authorizations are issued hierarchy-wise, establishing a chain of accountability. In other words, the permissions of a tenant owner to use the system are granted in the tenant's core namespace, applying to all its hierarchical namespaces. Each individual user in a tenant, in turn, is authorized by the tenant owner. According to permissions granted, the owner can create different roles in different subnamespaces as needed. For example, an owner can grant some users administrative rights to approve other users in core and subnamespaces. As their permissions are limited to their hierarchy tree, tenants cannot interfere with other tenants' environments. A tenant, at the same time, can use the system as if it has the authorization to create namespaces directly, thus having a relatively customizable environment.

The *subsidiary namespaces* custom resource, also known as a *subnamespaces*, is a software entity through which tenants can create Kubernetes namespaces without having the authorization to do so directly. Subnamespaces are indispensable for realizing the key features of our framework that are described in this chapter's introduction, as we see in our discussion of Tenancy Modes, Inheritance, Naming Convention, and Federation, below.

3.2.2.1 Consumer and vendor tenancy

The subnamespace entity relies on the parent-child relationship between the namespaces, starting from the core namespace of a tenant. Each subsidiary namespace can be both a parent and a child at the same time. The entity exists in one of two modes, *workspace* or *subtenant*, corresponding to the two forms of tenancy: consumer and vendor. The sequence diagram in Fig. 3.4 sketches out how the workspace and subtenant modes differ in creating a child namespace.

The hierarchical namespaces approach allows an organization to isolate *workspaces* of

[20] Antrea https://antrea.io/

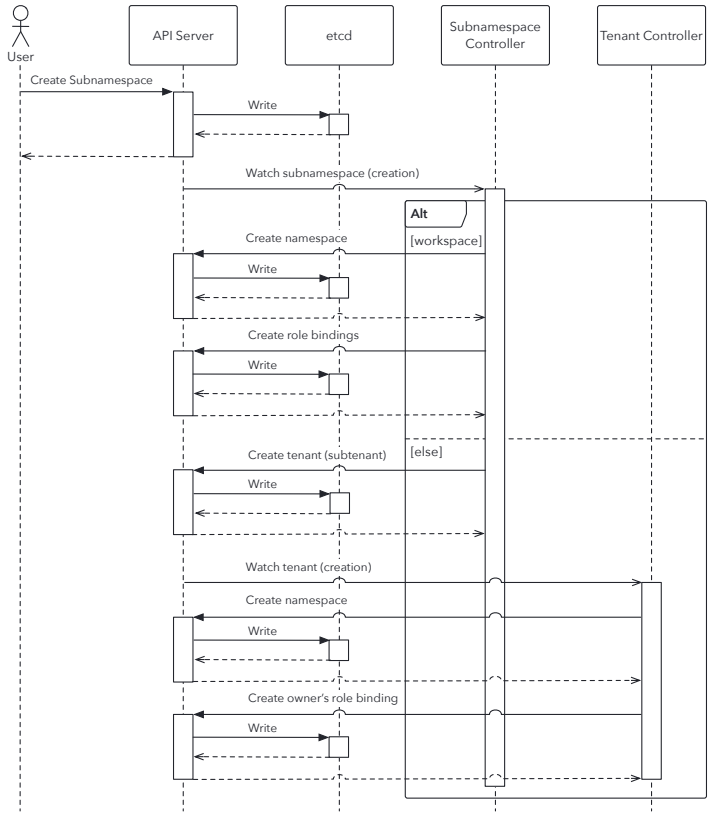


Figure 3.4: Sequence diagram of the *Subnamespace* entity performing namespace and permission creation.

different products by generating a subnamespace per product. Each child namespace assigned to a product can be used to isolate its teams, for example, backend and frontend, in the same way. Another real-world example consists in using subnamespaces to create isolated environments for multiple groups of students working on a laboratory experiment.

Fig. 3.5 demonstrates a tree of namespaces where a parent is blind to information about a child's namespace and its children. This shielding assists a tenant in subleasing the desired amount of resources to its customers, thus becoming a vendor. Accordingly, the customer of a vendor becomes a *subtenant*. A vendor can remove any of its subtenants when the customer-vendor relationship comes to an end.

A key characteristic of subnamespaces is enabling the choice of either mode, workspace or subtenant, at any depth of the hierarchy. By extension, subnamespaces allow a subtenant to be created in a child namespace with the workspace mode and another to be created with the subtenant mode, as shown in Fig. 3.5. Not only can these two modes co-exist in the same subtree, but they also reinforce each other's benefits. Last but not least, a subsidiary namespace can also be formed to be propagated across federated clusters. If so, it generates object names that are unique to the originating cluster and tenant to prevent name collisions during object propagation across the federation. Sec. 3.2.2.4 describes how our federation solution functions.

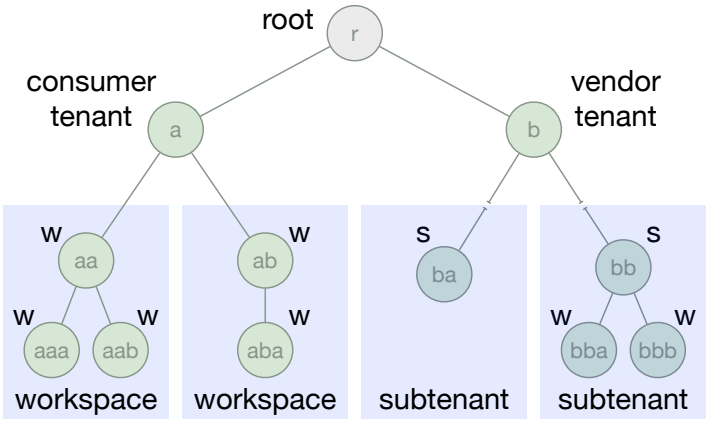


Figure 3.5: **Consumer and Vendor Tenancy in EdgeNet, showing workspace (w) and subtenant (s) modes.** EdgeNet uses its hierarchical namespace structure to build consumer and vendor tenancy. In this example, the namespace *a* belongs to a consumer tenant and the namespace *b* belongs to a vendor tenant that is reselling containers-as-a-service to its own customers.

The *Subnamespace* controller creates workspaces rooted at *aa* and *ab* for the **consumer tenant** by placing those namespaces into *workspace* mode. The consumer tenant has visibility into those workspaces.

The controller creates subtenants rooted at *ba* and *bb* for the **vendor tenant** by placing those namespaces into *subtenant* mode. The vendor tenant does not have visibility into its subtenants. Note that the subtenant that owns the sub-tree rooted at *bb* does have visibility into its own workspaces at *bba* and *bbb*.

3.2.2.2 Inheritance

In the subnamespace specification, an authorized user can declare which objects are passed by inheritance from parent to child. The Kubernetes resource kinds that can be inherited are currently as follows:

- Role-based access control (RBAC): Roles and Role Bindings; both together adjust permissions of users.
- Network policies; make a namespace restricted to defined ingress/egress rules.
- Limit ranges; set a resource quota per pod.
- Secrets; keep sensitive information such as credentials to be consumed by pods.
- Config Maps; configuration to be used by pods.
- Service Accounts; an entity that allows applications and services to authenticate with the Kubernetes API.

If RBAC objects are not inherited, the specification must include the owner of the subnamespace for management purposes. Further, it is possible to declare continuous inheritance. In this case, the controller constantly syncs objects from a parent to its child.

Note that a resource quota is not an entity subject to inheritance, so as to avoid overconsumption by a tenant, which could get around quotas by generating subnamespaces at will. The logic ensures that the aggregated child resource quotas cannot exceed their parent's initial resource quota, including the core namespace. Each subnamespace creation taxes its parent's resource quota so that the aggregation of resource quotas in the parent and child namespaces remain the same, as we discuss in Sec. 3.2.3. In other words, a tenant's resource quota is a cake to be shared out, and each subnamespace gets a piece of cake from its parent's cut.

3.2.2.3 Naming convention

The naming convention has been conceived so as to enable federation deployments. As mentioned in Sec. 3.2.1, a core namespace shares the same name with its tenant. Independent of its depth, a subnamespace follows the pattern of <*subnamespace-name*>-<*hash*>. We feed the hash function with the parent namespace and subnamespace name. This naming convention reduces the chance of name collisions while creating subnamespaces. If a collision nonetheless occurs, the subnamespace object enters a failure state, indicating a collision status. This is vital to the interoperability of multiple clusters. The reason is that tenants or namespaces holding the same names in different clusters probably occur in many clusters. Consequently, conflicts will inevitably arise while propagating objects, unless there is an adjustment mechanism such as the one described here.

3.2.2.4 Federation

In our federation vision, each cluster, even before it is federated, is a multitenant cluster, making its worker nodes available to multiple tenants, and federation further opens the cluster to the workloads of tenants from other clusters. (As we have discussed in Sec. 2.3.1, this differs from the approach of the Liqo framework, based on Virtual Kubelet, in which clusters only achieve multitenancy by federating.) We have developed a proof-of-concept federation architecture with a prototype implementation, which works jointly with our multitenancy framework (See Sec. 4.2). The source code of the prototype is publicly accessible via our repository.

We see each tenant gaining access to a federated set of clusters via what we might term a home cluster or local cluster. For example, a company that has developed an application that serves vehicles in several countries might need to deploy its workloads to the edge clusters of mobile operators in each of those countries, and it can do so via a cluster in its home country that is federated with these other clusters. To obtain access to a local cluster, it might contract with a cloud provider that has a commercial presence in its home country, leaving the cloud provider to manage the commercial relationships with the other providers in the federation. Information regarding the identity of the company and its contract with its local provider remains local, while only the workload-related objects necessary for the deployment of the application get propagated to remote clusters. Propagating as few objects as possible has three significant benefits: (1) it avoids replication of tenant information across clusters, thus reducing bandwidth consumption and unnecessary traffic; (2) it enhances data privacy and sovereignty and mitigates security risks; and (3) it significantly reduces overhead that could stem from running a control plane or worker nodes per tenant at the scale of a federation.

In EdgeNet, the deployment scope of any subnamespace can be set to either *federated* or *local*. If federated, the subnamespace controller adds the UID of the kube-system namespace as a prefix to the namespace name, and this cluster UID is also fed into the hash function described just above (Sec. 3.2.2.3). This ensures the uniqueness of each name across all of the federated clusters.

In our prototype federation, a tenant deploys its workloads to remote clusters by creating

a *Selective Deployment* [129] that targets the remote clusters using affinities, such as locations and connected devices.[21] A manager entity, called the *Federation Manager*, is informed by the local cluster for federation-scoped Selective Deployments. When it receives one, it searches for remote clusters that satisfy the affinities, in order to deploy the workload there on behalf of the tenant. To move towards a production federation architecture, issues such as caching and scheduling will need to be tackled.

3.2.3 Tenant resource quota

As described in Sec. 2.3.1.4, Kubernetes provides the ability to associate resource quotas with namespaces, but in the context of independent namespaces. Since our multitenancy framework extends Kubernetes namespaces to work in a hierarchical fashion, we need to extend the quota mechanism to take into account the dependency of each namespace on other namespaces above it and below it in the hierarchy. The EdgeNet quota mechanism is designed to allow for a given resource to be shared out between a namespace and its child namespaces, and for the parent namespace to recoup each child's portion when it is relinquished. Child namespaces can in turn share out their quota with their children, and so on, recursively. Our framework covers the following resources: CPU, memory, local storage, ephemeral storage, and bandwidth, each accounted for individually.[22]

We model tenant resource quotas by representing the tree of a hierarchical namespace as a graph $T = (V, E)$ composed of vertices V and parent-to-child edges E. For our purposes, each vertex $v \in V$ is a namespace, except for the root node. The tenant of a namespace v is entitled to construct a subtree T_v rooted at that namespace v, which is also called a core namespace. Denote $q(T)$ the resource quota of tree T, and each namespace $v \in V$ has a resource quota $q(v)$. Here, we assume that there is only a quota for different types of resources for simplicity. In fact, different quotas can be set for different resources, such as CPU and memory.

Let $\sigma(v) = \{w_1, w_2 \ldots\} \subset V$ represent the subnamespaces of v. Likewise, assume $\sigma(w) = \{z_1, z_2 \ldots\} \subset V$ represent the subnamespaces of w. The hierarchical resource quota problem here is twofold. First, we must ensure that a tenant resource quota $q(T_v)$ is equal to aggregated resource quota across all its namespaces: $q(v) + \sum_{w \in \sigma(v)} q(w) + \sum_{z \in \sigma(w)} q(z)$. The latter is to guarantee that the resource quota allocated to a subtree rooted at a namespace w is also equal to aggregated resource quota across the namespaces of that subtree, thus $q(T_w) = q(w) + \sum_{z \in \sigma(w)} q(z)$.

We solve this problem by partitioning resource quotas among parents and their children while keeping with the container orchestration tool's declarative approach. A tenant resource quota works by applying an identical resource quota, a Kubernetes resource, to the tenant's core namespace. Then, each subsidiary namespace in the core namespace takes its portion

[21] This is an extension to the *Selective Deployment* mechanism in our previous work [129]. There, workloads could be deployed to remote nodes within a single geographically dispersed cluster. Now, workloads can be deployed to entire remote clusters within a federation of clusters.

[22] Tenant resource quotas will be expanded to include other resources in the future, such as namespaces, pods, and configmaps.

from that resource quota, as shown in Fig. 2.5a.

As mentioned above, when resources are constrained, ensuring a fair share of them is essential. Static allocation of quotas, however, may lead to inefficient use of the resources. There are two sides to this problem. Such resource quotas that are allocated to tenants, assuming some tenants' resource consumptions are inferior to their quotas, may result in suboptimal utilization of compute resources in clusters. Likewise, the resource quotas that are allocated statically to subnamespaces by tenants, assuming some subnamespaces consume fewer resources than their quotas, may provoke less-than-ideal use of their tenant resource quotas. Even though our system allows temporary addition to and removal from tenant resource quotas as well as manually updating subnamespace quotas, this solution cannot scale when there are many clusters. Sec. 3.4 introduces how we plan to address this problem.

3.2.4 Slice and slice claim

Two software entities enable node-level slicing; *slice* and *slice claim*. Slice, a cluster-scoped entity, forms a subcluster by slicing among nodes, as its name signifies. A slice isolates the nodes within it from multitenant workloads once it is established. These nodes are chosen via a selector composed of fields that denote labels, number of nodes, and desired resources. On the other hand, a slice claim is a namespaced entity that tenants may create for their subnamespaces.

Nodes in a slice remain in the pre-reserved status until a subnamespace uses that slice. Once a subnamespace is bound to a slice, the multitenant workloads that runs on the nodes in this slice are terminated within a grace period of a minute. That is to say, workloads created in that subnamespace are isolated from other tenants. Thus, the container runtime configuration within such subnamespaces becomes available to tenants.[23] Regarding the termination grace period, we have set it to one minute by default, as twice the default grace period of 30 seconds in Kubernetes. However, providers can adjust this termination grace period according to their requirements.

A slice claim has two working modes; *dynamic* and *manual*. The dynamic mode permits a tenant to automatically create a slice if the resource quota in the slice claim's namespace is sufficient. In contrast, the manual mode prevents a slice claim from generating a slice even if the slice claim's namespace has an adequate resource quota. In this case, a cluster administrator must satisfy the tenant's request. This kind of behavior can be desirable if the number of nodes in a cluster is scarce. Fig. 3.6 depicts how a tenant can receive node-level isolation. We discuss the need for a daemon to improve isolation in Sec. 3.4.

3.2.5 Admission control webhook

An admission control webhook is a software entity that allows for enforcing custom policies. It can mutate and validate object operation requests of users. Such mutating and validating

[23] Resource-constrained environments may compel CaaS to operate on bare metal. We will, therefore, assess the performance of Kata with a specific experiment setup described in Sec. 3.4.

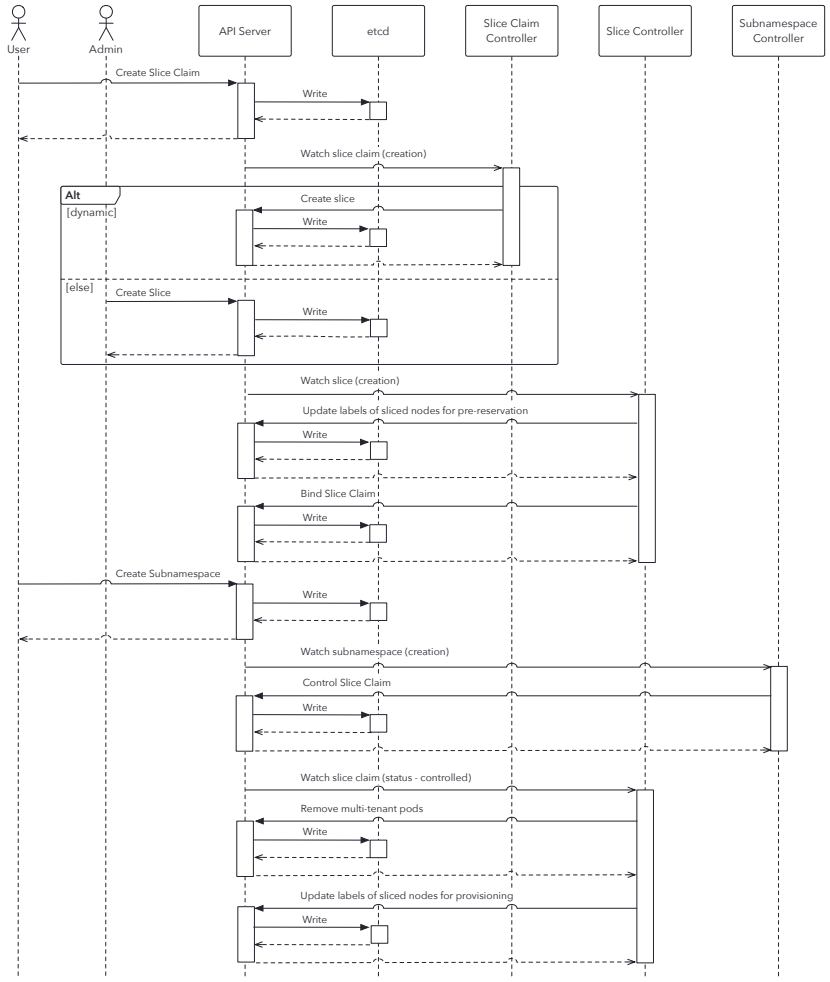


Figure 3.6: Sequence diagram of a tenant acquiring node-level isolation. For the dynamic mode, we assume the tenant has enough quota in the namespace.

operations are critical so as to ensure that users adhere to framework-specific policies. We enforce custom policies for subnamespaces, slice and slice claim, role requests, tenant requests, cluster role requests, as well as pods.

Kubernetes, by default, lets users pick runtime classes that they desire for their pods. Likewise, in our framework, a tenant can select the container runtime for the containers running on the nodes in its slice. However, this is not the preferred behavior unless a tenant acquires entire nodes through node-level slicing. For the purpose of better isolation, we constantly mutate the runtime class to employ Kata (See Sec.3.1.2) for multitenant workloads through admission control.

3.2.6 Role request

This feature facilitates permission management at namespace scope. Thanks to its design, this entity provides granular control over tenant users. A user can request a specific role in a core namespace or any subnamespace of a tenant. This role can be one of the cluster roles offered by the cluster provider or a role in that namespace. Once a request is made,

an authorized user in that namespace can approve or deny it. That is to say, tenant owners and admins can delegate responsibilities to team leaders in child namespaces. When a tenant represents a large organization, delegation becomes crucial to facilitate management.

3.2.7 Other entities

There are two more assistant entities. A *tenant request* stands for tenant registration. A central administrator or a trust strategy, for example, credit card verification, can approve the establishment of tenants and their owners. In our implementation, a cluster admin may approve a request or deny it. Another option is, as mentioned above, a provider can integrate a credit card verification-like mechanism with our framework to avoid the manual administration of clusters, supporting CaaS to operate with many clusters at scale. There are four pieces of information in the request; the organization, the owner, the tenant resource quota, if desired, and whether or not to apply a cluster-level network policy. A *cluster role request* is an entity that allows a user to claim to hold a role at the cluster scope. This entity eases shaping a cluster administration team and encourages the platform users to ask for the roles that they need.

3.2.8 Authentication

Our general design approach is to build, wherever possible, upon what is already available for Kubernetes, as we do by adopting OpenID Connect (OIDC)[24] running on top of OAuth 2.0[25] as our authentication method. A feature that is still under development is to extend OIDC with Pinniped[26] so as to access resources across clusters. This allows a user to authenticate once to access namespaces and objects, for which the user has access rights, in all of the clusters to which the objects have propagated.

3.3 Benchmarking

This section analyzes the performance of our EdgeNet single-instance native Kubernetes multitenancy framework. One of our goals is to assess to what extent native and multi-instance approaches are suitable for edge computing use cases. To this end, we compare our framework to single cluster per tenant offerings with the help of Rancher Kubernetes Engine (RKE)[27] in order to automate cluster creations and to the VirtualCluster [170] code that realizes a multi-instance-based multitenancy framework. That is to say, to represent the multi-instance through multiple clusters approach, we pick RKE, which is widely known for installing Kubernetes; VirtualCluster for the multi-instance through multiple control planes approach, which is a Kubernetes working group framework that is described in the scientific literature [170]; and our own EdgeNet framework is single-instance.

[24] OpenID Connect https://openid.net/connect/
[25] OAuth 2.0 https://oauth.net/2/
[26] Pinniped https://pinniped.dev/
[27] RKE https://rancher.com/products/rke

Both RKE and VirtualCluster perform well when the compute resources are nearly unlimited, or scalability with regard to the number of tenants is less of a concern. Compared to RKE, VirtualCluster is well-adapted to address the issues of the single cluster per tenant solution, such as high overhead. However, as we shall see, there is a tradeoff between performance and isolation, which means that existing solutions are not ideal for edge computing.

We used the GENI infrastructure [94] to spawn four Ubuntu 20.04 LTS virtual machines with 8 CPUs and 16 GB of memory in order to conduct experiments with EdgeNet and VirtualCluster. Using these virtual machines, we created a Kubernetes v1.21.9 cluster consisting of one control plane node and three worker nodes. The control plane node is completely isolated from any workloads.

For the VirtualCluster experiments, we reserved a worker node for running the manager, syncer, and agent components. Likewise, the per-VirtualCluster-tenant entities, which are apiserver, etcd, and controller-manager, are deployed on a dedicated worker node. For the EdgeNet experiment, an isolated worker node was sufficient to run the entities. A separate worker node hosted monitoring tools in both cases. We used the default configuration settings for both frameworks, including the number of workers that process concurrently and the execution period that triggers the controller.

We compared the frameworks' performance for tenant creation and for pod creation. For VirtualCluster tenant creation, inter-arrival times of 0, 8, 16, and 32 seconds were used for creating 2, 4, 8, 16, 32, and 64 tenants, respectively. For EdgeNet, inter-arrival times of 0, 2, 4, 8, 16, and 32 seconds were used for creating up to 10,000 tenants. (We discuss the reasons for the disparity in the number of tenants below.) For both framework, pods created were 1,000, 2,500, 5,000, and 10,000. Timeout is two minutes to create tenants and pods separately.

To measure the performance of a cluster per tenant method, reserved resources for tenant entities, a virtual machine with 8 CPUs and 16 GB of memory, were divided evenly among four Ubuntu 20.04 LTS virtual machines with 2 CPUs and 4 GB of memory on GENI. 2 CPUs were chosen because cluster provisioning repeatedly failed with VMs having a single CPU. We repeated measurements at least three times for each case.

3.3.1 Tenant creation

As discussed throughout this thesis, besides security, overhead is a noteworthy factor in qualifying a multitenancy framework, especially for edge clouds. Our experiments measure a framework implementation's ability to handle simultaneous creation requests; the time it takes to create a tenant; entities' resource consumption; and consumption per tenant, if it exists. Each request is considered successful if the framework returns a success status within two minutes after the control plane receives the request.

3.3.1.1 VirtualCluster

The experiments show a correlation between request inter-arrival time and tenant creation success rate. For example, with a 32 s inter-arrival time for 32 creation requests, the number of successfully created tenants ranges from 26 to 32; when the inter-arrival time is lowered to 8 s, the successes decrease to between 13 and 18, as shown in Fig. 3.7a. It is possible that VirtualCluster's difficulties in handling simultaneous requests stem from an implementation issue that starves tenants of the compute resources necessary to establish their control planes in these circumstances.

Similarly, as seen in Fig. 3.7b, decreasing the request inter-arrival time increases the tenant creation time. At a 32 s inter-arrival time, the median creation time is 76 s; put another way, it would take more than an hour to create 128 tenants. Furthermore, as the figure shows, the creation time fluctuates more widely as inter-arrival time decreases.

The most critical scaling weakness for VirtualCluster is that every tenant introduces additional overhead in terms of memory and CPU usage due to the per tenant isolation of control plane components: apiserver, etcd, and controller manager. Fig. 3.7c presents the regular memory usage for 2, 4, 8, 16, and 32 tenants. For example, a thousand tenants would consume around 300 GB of memory just to be present in the cluster. This limitation ultimately affected our experiment, which could not reach a high number of tenants on the single node that we had reserved for tenant components; the maximum number of tenants that we could create stably was approximately 40.

In addition to this, a tenant starting to use the cluster results in an increase in resource consumption. We also noticed that a successful status message for the tenant control plane does not imply that all its components are present and functioning properly. Therefore, we only considered the cases where control plane components per tenant were all created successfully.

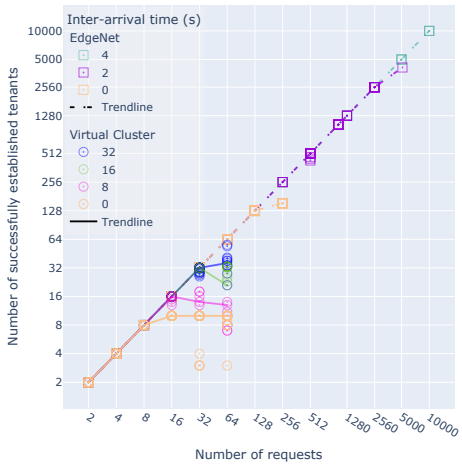
3.3.1.2 EdgeNet

As opposed to VirtualCluster, EdgeNet supported the creation of 128 tenants simultaneously with an almost zero failure rate across experiments. It also scaled well beyond this number, stably generating 2,560 and 10,000 tenants when the request inter-arrival time was set to 2 s and 4 s respectively, as shown in Fig. 3.7a. This is as far as one can go before running into Kubernetes' maximum namespace threshold[28] of 10,000 in a cluster; if tenants are allowed to have around ten namespaces each, the number of tenants per cluster is limited to around 1,000.

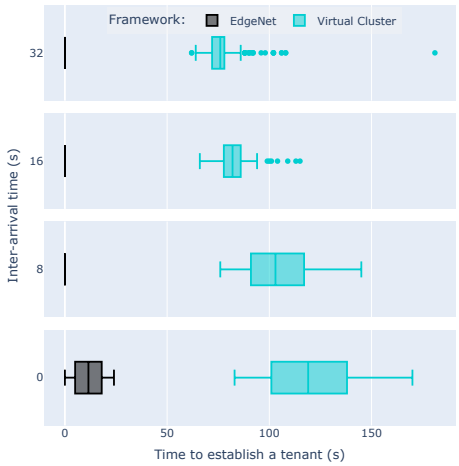
When requests arrive simultaneously, the median time for EdgeNet to create a tenant object in the control plane increases with the number of tenants: 38 ms, 48 ms, 63 ms, 68 ms, 106 ms, 175 ms, 216 ms, and 270 ms for 2, 4, 8, 16, 32, 64, 128, 256 tenants respectively. Another pattern of results is obtained with an inter-arrival time of 2 s: creation times are

[28] Kubernetes Scalability SIG documentation: *Kubernetes Scalability thresholds* https://github.com/kubernetes/community/blob/master/sig-scalability/configs-and-limits/thresholds.md

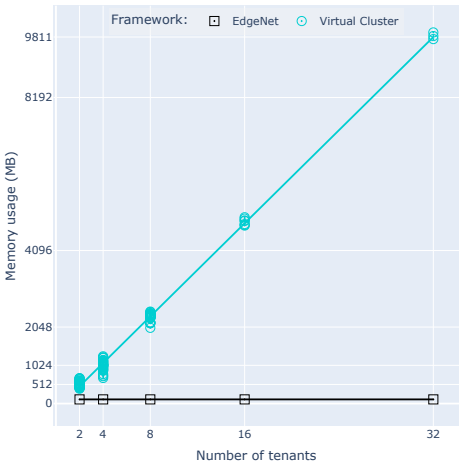
11 ms for 1,280 tenants, 11 ms for 2,560 tenants, and we tested as far as 5,120 tenants, also clocking in at a median of 12 ms. For 10,000 tenants, the median value is still 12 ms when inter-arrival time is set to 4 s. However, the maximum values increase as a function of the number of requests.



(a) Symbols represent frameworks, and each inter-arrival time is colored. The longer the time between arrivals, the higher the number of successfully created tenants. Stably, VC manages to create around 40 tenants at most when inter-arrival time is set to 32 s, while EdgeNet reaches 10,000 with 4 s.



(b) Box spans from quartile 1 to 3, a line inside indicates the median, and outliers are shown as single points. Inter-arrival time affects tenant creation time. The shorter it is, the longer the creation, with higher variation. EdgeNet consistently outperforms VC. The results are for 32 tenants.



(c) Base resource consumption rises with the number of tenants due to the multi-instance approach in VC. EdgeNet does not introduce such overhead per tenant. Resource consumption of framework-specific software entities is insignificant for both. No user activity is involved.

Figure 3.7: Experiment results for VirtualCluster and EdgeNet.

This suggests that concurrent or many requests saturate the shared API server, controller manager, and etcd moderately. Thus, when arrivals are simultaneous, the average time to fully establish a tenant increases as follows: 500 ms for 2 tenants, going up to 937 ms for 128 tenants. But Fig. 3.7b reveals that the time to fully establish a tenant drops when requests are spread out in time. For 32 tenants, the median times are 11.5 s for simultaneous arrivals,

271 ms for 8 s, 274 ms for 16 s, and 274 ms for 32 s.

Good results are seen for EdgeNet since it configures the state of the cluster rather than replicating the components, it does not generate per-tenant overhead, as shown in Fig. 3.7c. Given that the resource consumption of controllers is negligible, it is fair to state that there is no significant overhead in our framework.

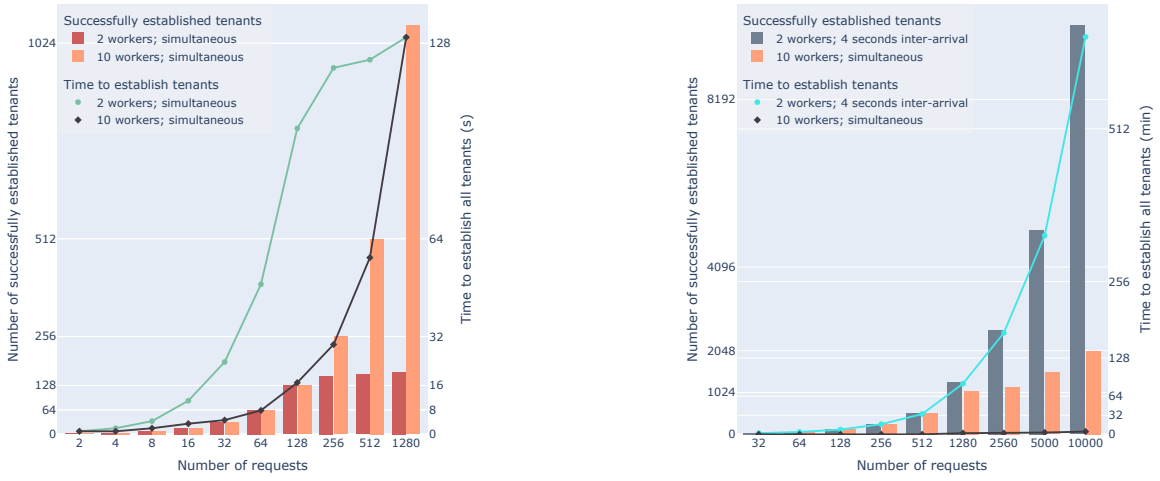
It takes EdgeNet approximately 1 min 41 s to create 128 tenants. Furthermore, EdgeNet's creation time can be shortened if needed by adjusting the number of workers and the running period. By default, the tenant controller uses two workers with a running period of 1 s, and the client's query per second (QPS) rate and burst size are set to 5 and 10, respectively. We tried altering the setup to have ten workers with a 500 ms running period, setting QPS and burst to 1,000,000 each. With these settings, it takes just 17 s to fully create 128 tenants, as seen in Fig. 3.8a. The same figure shows that EdgeNet can handle simultaneous requests if a cluster welcomes around 1,000 tenants. The time it takes to establish all tenants eventually converges towards two minutes for both settings, thereby satisfying the success criteria we described at the beginning of Sec. 3.3.1. However, we noticed it surpasses two minutes when simultaneous requests are more than 1,280. We presume that this may be due to client or control plane saturation resulting in the API server receiving delayed requests, which we need to investigate further. Fig. 3.8b shows that EdgeNet with default settings can scale up to 10,000 tenants when inter-arrival time is set to 4 s, but it takes more than ten hours in total.

3.3.1.3 Comparison

Our findings on tenant creation at least hint that better isolation provided by the multi-instance approach comes at the cost of performance loss. What can be clearly seen is that EdgeNet surpasses VirtualCluster on scalability and speed. The peak number of tenants in a cluster is 10,000 for EdgeNet but around 40 for VirtualCluster, even with longer inter-arrival times. VirtualCluster offers a separate control plane per tenant, meaning an increase in base resource consumption, which is one of the major limitations. In contrast, EdgeNet can scale up to the cluster namespace threshold thanks to the native approach discussed in Sec. 2.3.1.1.

Scalability is only one aspect of evaluating a framework's performance, especially for edge-specific workloads. Speed, stability, and overall reliability are also important. EdgeNet is considerably faster than VirtualCluster at tenant establishment for all inter-arrival times. Fig. 3.8a shows how optimizing the number of workers, running period, QPS, and burst can further improve EdgeNet's performance. Furthermore, when arrivals are not simultaneous, EdgeNet handles each request in microseconds, whereas VirtualCluster takes seconds, even minutes. On the one hand, such a brief tenant provisioning time is essential, especially for short-lived workloads that require a quick start-up time or use cases where workloads need to be moved across edge clouds. On the other hand, speed is an important contributing factor to establishing many tenants concurrently or in sequence, but stability and reliability are also critical.

VirtualCluster cannot adequately address simultaneous requests or requests with a short inter-arrival time, even if they are not many. Because of this issue, we observe a marked fall in the success rate of tenant establishment in such cases. We speculate that an imple-



(a) Default settings vs. optimization.

(b) Inter-arrival time vs. optimization.

Figure 3.8: Effect of the number of workers, running period, QPS, and burst on EdgeNet's performance, including comparison with the effect of time between arrivals.

mentation issue might be provoking resource starvation in tenant control planes. The time it takes to finish establishing all tenants is significantly more deterministic for EdgeNet than for VirtualCluster; EdgeNet exhibits almost no variation, irrespective of whether 128 tenants or 2,560 tenants are being created. However, EdgeNet's performance is tied to the control plane capacity as well. When many requests with little time between arrivals oversaturate the control plane, it has difficulty establishing all tenants properly. Nonetheless, EdgeNet can process 1,000 simultaneous requests, allowing tenants to use ten namespaces for each, as discussed above.

The multi-instance approach limits VirtualCluster's scalability since the base resource consumption increases as tenant numbers grow; providing one control plane per tenant costs about 285 MB of memory each. It is a large memory consumption, especially for edge computing use cases. This would also increase cloud computing expenses per tenant. Such overhead is not present in EdgeNet.

VirtualCluster, with its multi-instance approach, falls on the isolation side of the isolation-performance trade-off. Thus, performance degradation is expected. Considering this information is crucial to interpreting results correctly. Table 3.1 shows how much better VirtualCluster performs than a single cluster per tenant system. Regardless, our framework produced a more robust outcome with a significant performance advantage. Based on the results, EdgeNet is better suited for edge computing, as well as for cloud-edge collaborations.

3.3.2 Pod creation

To examine the effect of VirtualCluster's syncer on performance, we measure the time that it takes to create a representation of a pod as an object. The syncer gathers pod objects from the tenant control plane and creates them in the host control plane, called a *supercluster*. The time that we measure is the time that it takes for pods to show a pending status in the supercluster.

Table 3.1: Quick comparison of native and multi-instance approaches

	Max Tenants	Creation Time (s)		Per Tenant
		Median	Max	Overhead (MB)
EdgeNet	10,000	0.616	1.741	None
VC	40	82	93	285
RKE	4	343	395	711

Max number represents the maximum number of successfully established tenants that can be stably reached with respect to allocated resources for tenant creation.
Creation time is the time that it takes to establish a tenant for four simultaneous requests.
Per tenant overhead refers to the fixed proportion of resources each tenant consumes in an average manner, regardless of activity. VC consumption was measured using pods that deliver control planes to tenants, and RKE consumption was measured through containers that provide clusters to tenants. Traditional VM-based overhead is not included in RKE.

We focus on pods instead of containers as that helps us reveal the framework's capabilities, as the creation of containers is dependent upon many factors such as available resources on the host, container runtime, and volume type. There are already many papers that evaluate these aspects for different container runtimes.

3.3.2.1 VirtualCluster

Request inter-arrival time affects the number of successfully created pods similarly to how it affects the number of successfully created tenants. This is because, as described above, created tenants struggle to enter a healthy state when inter-arrival time is short. Also, increasing the number of tenants that create pods degrades pod creation performance. One reason for this is the increased resource use in tenants' control planes. Pod creation success, for 16 or 32 tenants, is 100% for 1,250 and 2,500 pods; performance drops slightly for the creation of 5,000 pods; for 10,000 pods, a median of 9,431 are successfully created. The time it takes to create pods also increases with the number of pods created.

3.3.2.2 EdgeNet

Up to 10,000 pods can be created simultaneously for up to 300 tenants in a deterministic manner. Performance for 10,000 pods started to degrade at 384 tenants due to saturation of the Kubernetes API server in processing requests from different core namespaces. Pod creation time does not spike, as there is no intermediate layer syncing the objects. A linear relationship is observed between the median creation times and the number of pods in Table3.2.

3.3.2.3 Comparison

In VirtualCluster, the syncer is an intermediate layer between the supercluster and tenant control planes in order to sync pod objects. The disadvantage of this approach is that every pod operation introduces synchronization overhead, both on the supercluster and tenant

Table 3.2: Time in seconds, median values, to create a representation of a pod as an object in the host control plane. The number of tenants used for the experiments is set to 32 for both VirtualCluster and EdgeNet

Number of pods	1,250	2,500	5,000	10,000
VirtualCluster	12	22.5	46	134
EdgeNet	2.5	6	12	27

sides. We should emphasize that every synchronization process causes a delay for a pod to be up and running. This may raise concerns about running VirtualCluster at scale; however, it can be mostly overcome by providing more computing resources to the framework, leading to higher costs. In contrast, EdgeNet allows tenants to directly make use of the same control plane so as to create pods. Its performance is directly related to the capabilities of the control plane. Thus, EdgeNet produces superior results, where VirtualCluster takes at least three times as much time as EdgeNet to create 1,250 pods, 2,500 pods, 5,000 pods, and 10,000 separately. Table 3.2 shows how far VirtualCluster's synchronization of objects between the supercluster and tenant control planes causes significant delays while achieving better isolation.

3.4 Conclusion and future work

We have presented EdgeNet, a Kubernetes-based multitenancy framework for Containers as a Service (CaaS) that, because it is native, i.e., serves all tenants through a single control plane and a single data plane per cluster, is a more efficient alternative to the current multi-instance manner in which cloud providers offer CaaS. Our benchmarking results demonstrated good scalability and response times for EdgeNet as compared to a leading multi-instance alternative. Though, in our framework, tenants are not isolated into separate control planes, their containers nonetheless receive the high level of isolation that is provided by Kata containers. For edge computing to succeed, we believe that security and isolation must be handled natively in software so that workloads can be moved between distant clusters within short delays.

There are, of course, still many questions to be answered. What are the most optimal ways to establish a robust CaaS federation that is composed of ubiquitous clusters offered by numerous providers? In order for clusters to join and leave such a federation seamlessly and securely, what trust mechanisms must be in place? How can users get reliable and transparent billing systems in such an environment? Chapter 4 discusses our federation strategy that addresses some aspects of such a robust CaaS federation.

Anyone may avail themselves of our liberally-licensed, free, open-source code to enable multitenancy in a Kubernetes cluster. It is already in production use in the EdgeNet edge cloud testbed, for which the tenants are research groups around the world. And it is particularly suited for edge clouds, where resources are limited, as well as for the cloud. Because of its federation features, we see this framework as paving the way for tenants to deploy their services across edge clouds operated by many different operators worldwide.

Although the work presented in this chapter goes a long way to establishing a Kubernetes

multitenancy framework that is suitable for the edge cloud, there is still considerable room for improvement. We describe areas for future work below.

Sub-node-level VIP Slicing. In order for tenants to receive guaranteed access to resources that are both available and dedicated to them, node-level slicing is currently the only option. By adding a new point to the slice spectrum, it will be possible to do so at sub-node-level granularity. We will deploy a pod that consumes almost no real resources on a node to ensure that resources are secured. Priority classes will enable the reservation mechanism for pods.

Resource Quota Reallocation. Resource usage is sub-optimal if tenants reserve quotas that they do not then fully use. We plan to develop an algorithm to redistribute unused quotas in a best-effort fashion. Tenants that consume all of their quotas would receive additional resources. Quotas can also be redistributed among the namespaces of each individual tenant. This would improve overall efficiency at little cost, provided that tenants and namespaces can recoup their assigned quota if and when it is needed.

Storage. Sharing storage among containers securely at the edge is a challenge due to the security issues discussed in the Rationale section (see Sec. 2.2.2). We plan to develop an agent that runs on every node and is ready, upon tenant demand, to prepare a disk partition that the tenant can use as a storage volume for its Kata containers.

Security. We plan to encrypt each tenant's data separately, across its namespaces and cluster-scoped resources. In this way, even if a tenant's data leaks, another tenant will not be able to read it. This improvement may affect the performance of our framework slightly.

Customization. Tenants cannot currently create cluster-scoped resources independently. We plan to develop a namespace-scoped custom resource that allows users to dynamically create cluster-scoped resources. This entity will be using the namespace name as a prefix in generating cluster-scoped resource names to avoid collisions.

Subnamespaces. A user may want to attach labels to subnamespaces. There is a risk, however, of a malicious actor breaching another tenant's network policies if labels are defined independently. For example, one can launch a brute-force attack to correctly guess the namespace labels used in a tenant's network policies. By using the name of the subsidiary namespace as a prefix, we plan to solve this issue. Inheritance will then allow labels to be passed down from parent to child.

Container Isolation. Based on the reasons outlined at the end of our discussion of lightweight hardware virtualization (see Sec. 3.1.2), we will use a specific experiment setup to assess how Kata, gVisor,[29] and runC perform. We will examine a setup in which Kata and gVisor run on a physical server while runC runs on a virtual machine created on that server.

Isolation Daemon. Kubernetes garbage collection removes unused images. However, our slicing feature provides on-demand node-level isolation, so we need to instantly clean the node from multi-tenant pods and container images. We also consider clear iptables rules during this process. An isolation daemon that runs on each node will be further developed to fulfill these operations.

[29] gVisor https://gvisor.dev/

Chapter 4

A federation that spreads by local action

Cloud computing has changed how applications are developed and deployed over the last decades. The adoption of cloud computing has been stably growing as it protects customers from upfront infrastructure investments and provides on-demand access to scalable resources with the charging model of pay-as-you-go. The more organizations take advantage of cloud computing, the more additional services likely are to be delivered by CSPs with the aim of satisfying customer requirements, as it already happens with the cloud providers constantly bringing out new self-specific services [81]. These provider-specific offerings hinder interoperability [47]. Furthermore, such diverse services that are specific to a particular CSP cause vendor lock-in problems such that customers cannot easily move between CSPs. With time, this becomes a burden for customers as they cannot effortlessly benefit from the infrastructure, service, and maybe better prices of other CSPs. It is through the development of intercloud solutions that this problem can be overcome.

Resource provisioning and application deployment in an intercloud environment is a long-lasting topic in both research and industrial communities [157, 10, 104, 163, 97, 118]. The main motivations are to improve scalability, reduce downtime, optimize costs, avoid vendor lock-in, benefit from various services, leverage more geographical locations, provide enhanced security, and comply with legal obligations [9, 105, 64, 3]. Five main delivery models are available, which can be used in combination to achieve these goals: hybrid clouds, multi-clouds, sky computing, multi-clouds tournament, and cloud federations [78].[1] Still, the fast-evolving nature of cloud computing requires continuous adaptation of these intercloud solutions to encompass new APIs, services, and concepts [46]. The provider-centric techniques, therefore, fail to facilitate client-centric and flexible mixtures of cloud services [140].

As these problems are being dealt with in the cloud, edge computing infrastructure is also being deployed. With the requirements imposed by the internet of things (IoT), network function virtualization (NFV), and mobile computing, it is no longer debatable whether distributed cloud infrastructures are needed [124]. Interoperability between these clouds and

[1] Alongside hybrid clouds, multi-cloud and federation are the areas of greatest interest, to the best of our knowledge.

edge clouds becomes a fundamental requirement for some edge computing use cases [37, 55]. We think the CaaS service model can facilitate establishing such interoperability since most offerings are built upon Kubernetes, a de facto industry standard container orchestration tool for clouds.[2] However, there remains the question of which delivery model to adopt.

Table 3.1 shows that spinning up a cluster per location introduces high overhead and slow startup. Such slow startup hinders workload mobility, and high overhead results in inefficient use of constrained resources in edge clouds. This can be manageable for large organizations that can have clusters ready at required locations but can quickly become tedious and costly for a small-sized edge cloud customer. In adherence with our assumption that multiple operators will provide edge clouds in many locations, we assert that a CaaS federation that employs our multitenancy framework can incentivize edge cloud customers, especially small and medium-sized ones, with three benefits: (1) lower costs, (2) abstraction of cluster management per location, and (3) quicker tenant environment preparation.[3] This inclusive approach can also help edge cloud providers achieve their infrastructure cost-efficiency despite widely dispersed site locations.

That being the case, we have developed our federation solutions so as to extend Kubernetes, which helps maintain interoperability between many clouds and edge clouds at plentiful locations. Another advantage of using Kubernetes is its declarative approach: users declare their desired states of resources to which corresponding controllers converge the actual state within a continuous control cycle. This principle provides a robust failure resiliency for distributed systems. However, Kubernetes is not designed to run on a geographically distributed environment like edge computing infrastructure but to run on centralized data centers.

Based upon our analysis of the scientific literature, we argue that an integrated federation strategy is required for CaaS built upon Kubernetes to work for clouds and edge clouds sustainably, thus identifying three delivery models of federation:

- *Node-wise*: multiple providers offer nodes to a cluster. The cluster's control plane becomes a control plane for federation, and resources that are federated are the nodes. This approach allows small-size providers to offer their resources. It is also suitable when resources are tightly constrained and are probably needed at various locations, so an additional control plane overhead should be avoided.
- *Cluster-wise*: multiple providers open their clusters to each others' tenants. It requires a mechanism to propagate objects and move workloads across clusters securely, and resources that are federated are the clusters. This approach is reasonable when resources can relatively scale and are presumptively needed at multiple regions, so an additional cluster control plane is acceptable.
- *System-wise*: multiple providers can interconnect and interoperate their systems. Each system can be a standalone federation itself, and the form of resources that are federated are the systems. Such an approach is required if and when providers employ

[2] Kubernetes, being a standard tool for container orchestration, mostly overcomes the continuous adaptation problem occurring in intercloud efforts.

[3] Even if a large-sized edge cloud customer wants to acquire a set of dedicated clusters across a number of regions, our proposed federation approach is still suitable to interoperate these clusters located in edge clouds offered by multiple providers.

different tools and systems and when resources are likely to be needed at various spots.

Our reasoning behind an integrated federation strategy is that we contend there is no one-size fits all solution to federate resources in the edge continuum. This is evidenced by the fact that WAN deployment made in a geographically distributed node-wise federation causes a high latency between the control plane and worker nodes, resulting in performance degradation for some regards, and that a centralized federation control plane brings about limitations and weaknesses, e.g., the direct interaction of local actors to member clusters can result in their misconfiguration in a split-brain scenario [93]. These two approaches also share a deficit in scalability; Kubernetes suggests having 5000 nodes in a single cluster at the maximum[4], and the KubeFed-based cluster-wise federation option targets dozens of clusters [82], which is relatively low for envisaged edge infrastructure.

Below are our contributions and the sections of the chapter that address them:

- We have devised an integrated federation strategy that we believe will foster a future in which CaaS can grow to help establish a federated edge infrastructure at scale by enabling multiple providers to offer compute resources in the form of nodes (Sec. 4.1), clusters (Sec. 4.2), and systems (Sec. 4.3) in many locations while preserving the lightweight and low overhead essence of our multitenancy framework.
- We have implemented a node deployment procedure in which providers run a bootstrap script to make their nodes join the cluster (Sec. 4.1.2) and have brought it into operation in the EdgeNet testbed (See Sec. 5.3.2 for details).
- After analyzing the Kubernetes federation tools, we have developed a functioning prototype of cluster-wise federation (Sec. 4.2.1), which aims at federating a large number of clusters while covering different use case scenarios through its three federation manager deployment model (Sec. 4.2.1.1). Our proposed architecture also removes the necessity for a centralized federation control plane that manages member clusters, and by this means, it addresses the single point of failure and the split-brain problems (Sec. 4.2.1.2).[5]

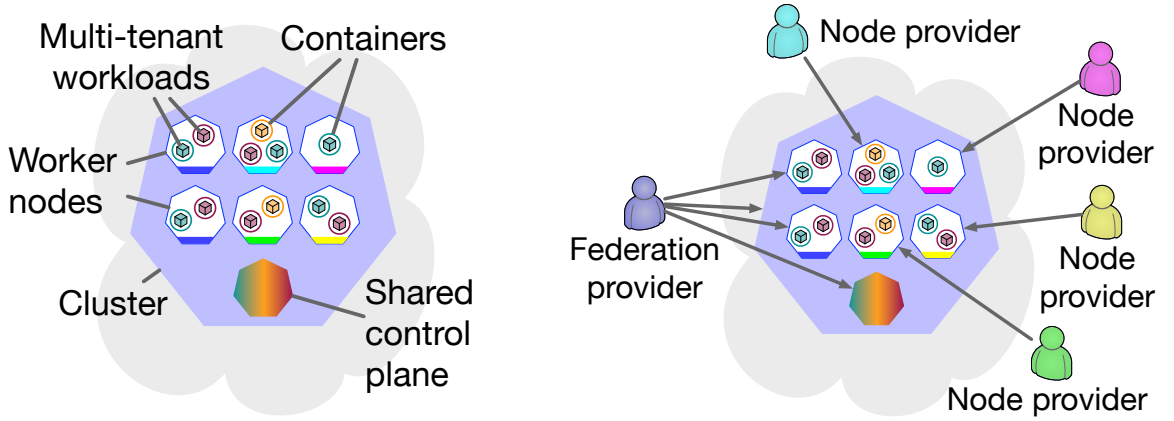
The chapter is organized as follows. Sec. 4.1 describes how to establish a node-wise federation and how we have implemented the node deployment procedure. Sec. 4.2 introduces a conceptual architecture for cluster-wise federation, argues it, and explains how we have developed a prototype on top of this architecture. In Sec. 4.3, we provide a vision for system-wise federation and discuss why it can be needed, and we conclude the chapter along with a future work in Sec. 4.4.

4.1 Node-wise federation

We believe that a federation of edge nodes, each node being offered by a different provider, is essential to establish heterogeneity in compute resources and devices as well as to enable numerous vantage points. Such a federation can virtually address two problems that are re-

[4] Kubernetes documentation: *Large Clusters* https://kubernetes.io/docs/setup/best-practices/cluster-large/

[5] The architecture of cluster-wise federation has not yet been validated through the in-depth experiments and analysis, which we will do through future work (Sec. 4.4).



(a) A cluster that is made up of a control plane that is shared by multiple tenants and of compute nodes that run multi-tenant workloads.

(b) A federation provider delivers a cluster consisting of a control plane with two compute nodes, and multiple node providers contribute with four compute nodes to it.

Figure 4.1: A depiction of the establishment of a node-wise federation.

source provisioning and maintenance. As multiple providers deliver edge nodes, a federation can ensure an infrastructure at scale. It also facilitates the maintenance of the nodes through physical operation, thanks to a shared workload between the providers.

4.1.1 Architecture

This delivery model is based on a regular control plane with a set of compute nodes that form a federation, as shown in Fig. 4.1a. A federation is initiated by a federation provider, which can be a single institution or an association, that delivers a cluster that consists of a control plane and conceivably compute nodes. Multiple providers supply compute nodes to such a cluster, scaling out the federation infrastructure, as seen in Fig. 4.1b. All compute nodes inside clusters, in this case, are fully accessible to their clusters' control plane for the purpose of supervision.

A federation provider is responsible for cluster administration, from maintaining the control plane nodes as well as compute nodes that it provides to auditing. If multiple providers deploy their nodes to their sites, this model alleviates the federation provider's workload in terms of node maintenance, as providers share the maintenance burden.[6] Regarding who can provide nodes, it depends on the policies of federation provider. It is possible to have control over who can offer nodes, or a federation provider may choose to allow anybody to supply nodes.[7] In any case, tenants who deploy workloads on nodes should be able to target them by their providers.

[6] If federation activity is for profit, node providers, in return, can receive a payment according to how much customers use their nodes. Our work does not cover such an aspect.

[7] Within the EdgeNet testbed that supports not-for-profit research, we allow anonymous contributions. Their contribution can be acknowledged through a website or a leaderboard in the future.

4.1.2 How providers deploy nodes?

Our procedure consists of three main components for node deployment: a bootstrap script, a node agent, and a controller. A provider is only required to run the bootstrap script on the target node. This script is designed to install the container orchestration tool, the container runtime, along with a dedicated *node agent*, as well as to set up a secure communication channel through which the control plane nodes can access the node to be joined. Once the node agent starts, it configures hostname and VPN, and then creates a node contribution object in the cluster. Following the creation of a node contribution object in the control plane, the controller gets informed, then generates a token for the join command that is invoked remotely on the node.

We have implemented this procedure to extend Kubernetes.[8] A node is a machine, physical or virtual, that runs a container runtime and the Kubernetes agent, *kubelet*. Our implementation uses *containerd* as the container runtime. While installing containerd and kubelet is relatively easy for a user comfortable with the command line, we seek to make the process as easy and error-proof as possible in order to encourage contributions.

To do so, we describe a configuration through a set of Ansible playbooks. Ansible is a popular configuration management tool and is commonly used to deploy Kubernetes clusters (see, for example, the Kubespray project[9]). By using Ansible, we can reuse community-maintained playbooks for deploying containerd and kubelet, and we can benefit from the ecosystem of tools that integrate with Ansible. Most notably, we make use of the Packer tool[10] to build ready-to-use virtual machines from the playbooks. We currently support the deployment of nodes on the major Linux distributions (CentOS, Fedora, and Ubuntu) on x86 machines. We will extend support to ARM hosts in the future. The current implementation is very flexible:

- The bootstrap script works on any recent Debian or RedHat-based Linux distribution, on aarch64 or x86_64 architectures, and it doesn't require any preinstalled software.
- The bootstrap script URL can be passed to cloud-init[11] to automatically set up the instances on first boot.
- The Ansible playbooks can be used in a standalone fashion for bulk deployment or node maintenance.
- The Ansible playbooks can be used together with Packer to create VM images with our software pre-installed.

In order to make this process as easy and as transparent as possible, a set of Ansible playbooks automatically perform some of the node deployment steps. The node agent is written in Go, and the controller is a Kubernetes operator, node contribution,[12] that consists of a custom resource and its custom controller. Fig. 4.2 presents a flowchart of our current deployment procedure, and its main steps are summarized as follows:

[8] EdgeNet node provisioning https://github.com/EdgeNet-project/node.

[9] Kubespray https://github.com/kubernetes-sigs/kubespray

[10] Packer https://www.packer.io

[11] cloud-init https://cloud-init.io

[12] Since our edge cloud testbed, which is described Chapter5, uses this feature, we name it *node contribution*.

- A provider runs the `bootstrap.sh` script on the target node. This script installs Ansible, fetches the deployment playbook, and runs it.
- The playbook setups SSH access and installs the container runtime, Kubernetes, and a dedicated *node agent* written in Go.
- The node agent starts and configures the node hostname and the VPN as described in Sec. 5.3.1, and it creates a node contribution object in the cluster.
- The controller connects to the node through SSH, generates a token, and runs a join command using this token. If the node joins the cluster, the controller then adds a DNS record for it. If not, the controller retries its step until the failure count reaches back off-limit.

Our node contribution procedure can spawn a node in under 5 minutes.

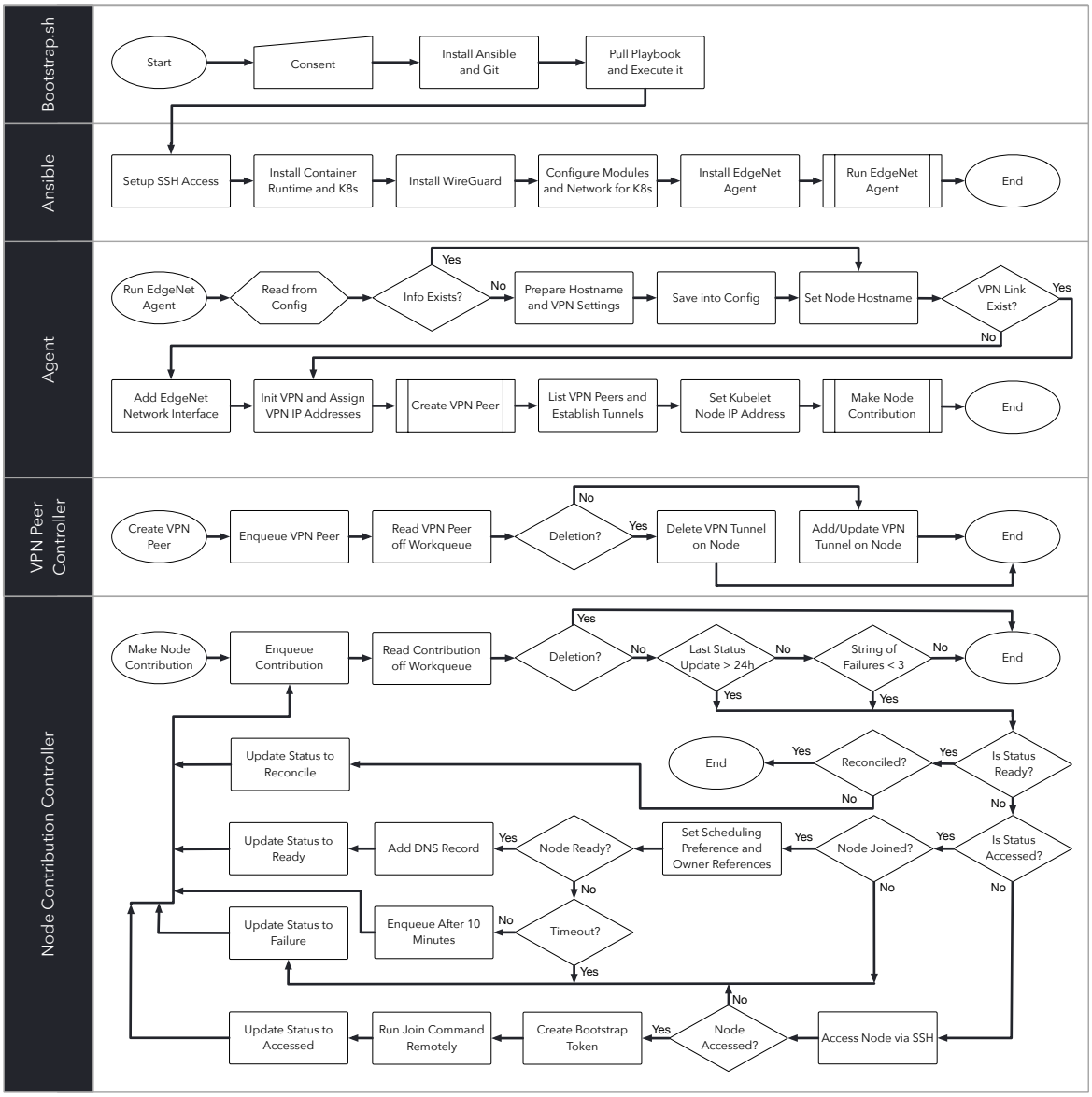


Figure 4.2: Flowchart of the node contribution procedure. A controller typically resyncs objects at an interval to converge the actual state to the desired state in a continuous loop. In the diagram, *Node Contribution Controller* includes two ends for clarity and view.

In our current implementation, no input is required from the user, and anyone can deploy a node without authentication. However, a cluster administrator can manually configure the cluster to require providers to authenticate first to make their nodes join that cluster. As our edge cloud testbed, described in Chapter 5, adopts a node-wise federation delivery model, we further discuss how this architecture is used to provide a globally distributed testbed infrastructure, along with other features such as node labeler and selective deployment (See Sec. 5.2).

4.1.3 Time to deploy a node

A node can be deployed in two ways: from a pre-built cloud image, or from a *bootstrap* shell script in a dedicated virtual machine. We perform our measurements for both methods on an AWS (Amazon Web Services) *t2.small* instance with 1 vCPU and 2 GB of memory, in the *eu-west-1* (Ireland) region. We neglected setting up VPN during these measurements. The measurements were carried out on the EdgeNet testbed in 2021.

With the bootstrap script, counting from the launch of the script, it takes 2 minutes and 50 seconds to install Docker and Kubernetes, 3 minutes and 5 seconds for the node to be detected by the cluster, and 3 minutes and 50 seconds to be ready to deploy containers.

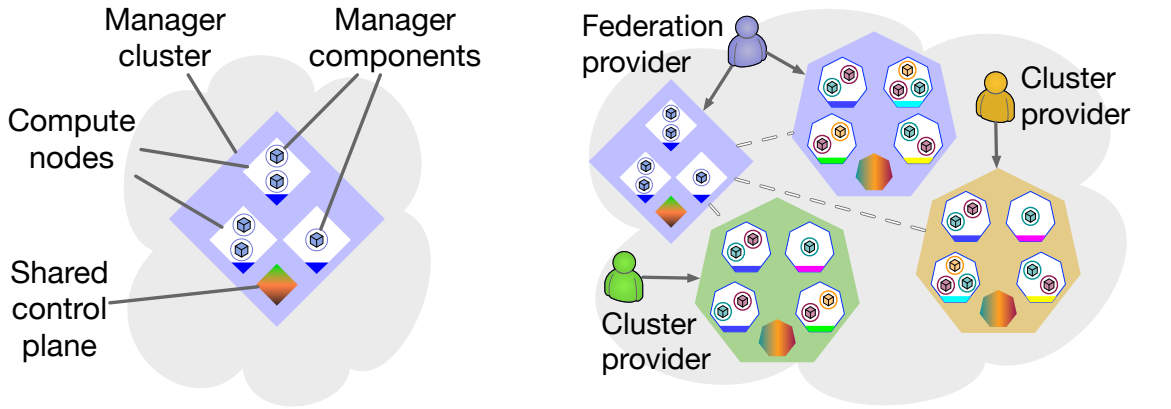
With the prebuilt AMI (Amazon Machine Image), counting from the instance creation, it takes 40 seconds for the node to be detected by the cluster, and 1 minute 20 seconds for the node to be ready to deploy containers.

4.2 Cluster-wise federation

We aim at a federation of a large number of clusters that are hosted, besides in clouds, mainly in edge clouds that are geographically dispersed and that multiple operators provide. As multiple providers deliver clusters, similar to the node-wise federation, an infrastructure at scale can be accomplished. In other words, the more operators supply clusters to the federation, the more coverage of regions in which an operator's customer can run containers.[13] Thanks to a division of labor in maintaining federated clusters between the providers, this approach facilitates the maintenance of the clusters in many locations through physical operation.

We have conceived an architecture to federate clusters with the goal of overcoming the single point of failure problem, ensuring scalability, and reducing attack vectors. We have also implemented a prototype, which works in concert with our multitenancy framework discussed in Chapter 3, that deploys tenants' containers to remote clusters. This architecture has not been validated yet through rigorous experiments and analysis, which we have started to work on, as explained in Sec. 4.4. Below, we describe our proposed architecture and argue it.

[13] We have an architecture where tenants contract with only one operator they prefer; federated resources are accessible through a worker cluster of this chosen operator.



(a) A cluster that is made up of a control plane that multiple providers share and of compute nodes that run the federation manager's software components.

(b) A federation provider delivers a federation consisting of a federation manager with one compute cluster, and two cluster providers each contribute a compute clusters to it.

Figure 4.3: A depiction of the establishment of a cluster-wise federation.

4.2.1 Architecture

A federation provider is conceived to be both a single institution, like a prominent cloud provider, or an association, such as small data center owners. It delivers a federation that can consist of one or more federation managers, as well as compute clusters that run tenants' workloads (Fig.4.3). Multiple providers then can contribute to this federation with their own compute clusters, as depicted in Fig.4.3b. These providers can also bring their own manager clusters to the federation.

A cluster provider can be a cloud provider, operator, and small data center owner. It can even be a small or medium-sized enterprise that wants to access federated resources as well as rent out resources on its on-premises infrastructure. A federation provider is responsible for ensuring the satisfaction of federation-level agreements and service-level agreements that these providers offer.

We describe our architecture through five primary aspects: federating clusters, resource discovery, accessing federated resources, scheduling, and autoscaling. Our prototype implementation, a CaaS federation tool, is designed as a set of custom resources and custom controllers that extend Kubernetes from within.[14] This prototype works together with our multitenant CaaS framework so that it enables multiple tenants to share federated clusters besides their local clusters. Below are seven new entities that our prototype introduces to address the first four primary aspects of our architecture:

- *Cluster* is the fundamental entity through which providers can federate their clusters (Sec.4.2.1.1).
- *Selective deployment anchor* helps propagate federation-scoped deployment information across federation managers (Sec.4.2.1.2).
- *Selective deployment* is a feature through which users can target clusters on which their containers will be run (Sec.4.2.1.2).

[14]We follow the same approach that we use for our multitenancy framework to implement our prototype.

- *Federation agent*, named as *Fedlet*, runs on compute clusters and sends their available resources to their managers (Sec. 4.2.1.3).
- *Manager cache* contains minimal information for the federation scheduler to make decisions (Sec. 4.2.1.3).
- *Cluster labeler* is a service, which is derived from node labeler (See Sec. 5.2.1), that attaches labels to clusters according to their location information (Sec. 4.2.1.4).
- *Federation scheduler* is a simple scheduler that selects worker clusters (Sec. 4.2.1.4).

We benefit from the existing mechanisms for autoscaling purposes, which is the fifth aspect of our architecture (Sec. 4.2.1.5).

4.2.1.1 Federating clusters

We conceived of two types of clusters: (1) the compute clusters (Fig. 4.1a), namely the worker clusters, which run tenants' workloads; and (2) the manager clusters (Fig. 4.3a), known as the federation managers, which discover federated resources, make federation-level scheduling decisions, and propagate objects. Both of these clusters employ our multi-tenancy framework.

Compute clusters. As their name suggests, compute clusters are designated for running tenants' workloads and are therefore called worker clusters as well. These clusters can be on the cloud, service provider edge, or user/device edge. Each worker cluster is managed and maintained autonomously by its provider. Providers make their clusters part of the federation through the use of federation managers.

Multiple tenants share these worker clusters, thus multi-tenant, and access the federation resources through the worker clusters with which they are registered. Our multitenancy framework allows these clusters to receive federation deployments from other worker clusters without collision (See Sec. 3.2.2.4). When a worker cluster receives a federation deployment instruction, the cluster is not dependent on any additional information that comes from any central federation component in order to make in-cluster scheduling, pod autoscaling, and cluster autoscaling decisions. This, aside from preventing the concentration of computational load on centralized components, allows cluster providers to apply their own strategy to maximize their profits, whether by shutting down nodes or changing outsource/insource balance [57].

Manager clusters. A federation is initiated with a federation provider delivering the core federation manager. A manager cluster consists of a control plane with which providers communicate to federate their clusters and of compute nodes that run software entities that fulfill the federation manager's responsibilities. In this way, a manager cluster can scale out and down as a function of demand in the federation.

Multiple providers share a federation manager to federate their own federation managers

or worker clusters with the federation.[15] Registration of providers with federation managers is subject to the administrator approval. Once approved, each provider is allocated an isolated environment in their requested manager cluster's control plane. Having their own environment, providers can start generating a token for each cluster to be federated and put these tokens in their environments. Federation managers consume these tokens to establish bidirectional communication with clusters to be federated.

Providers declare their clusters through our *cluster* entity's specification, which consists of the following:

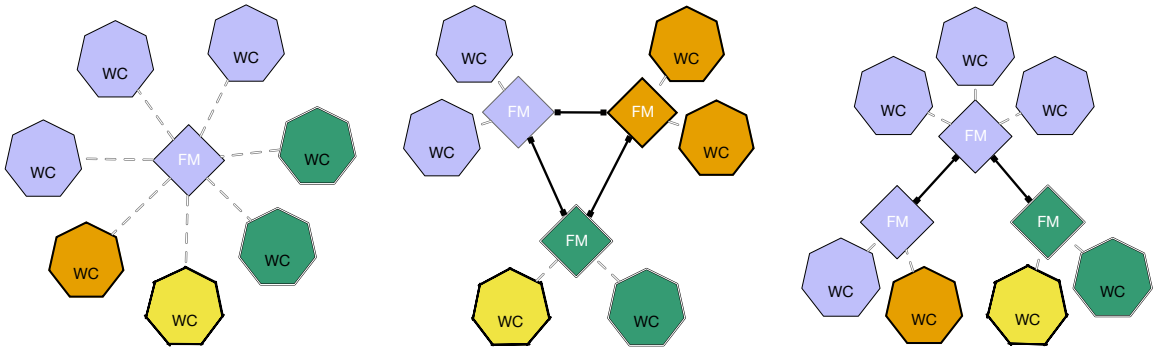
- *UUID* of the cluster such that the federation manager can verify it.
- The *role* to determine whether it is a worker cluster or a manager cluster.
- An *IP address* through which the federation manager can communicate with the cluster's *API server*.
- The *name of the secret* that stores the *token*, which allows the federation manager to authenticate with the target cluster.
- *Visibility* of the cluster where public means open to federation deployments and private indicates closed.
- *Sharing preferences*, through the use of an allowlist or a denylist, that restrict who can access the cluster resources.
- *Enabled* that allows making the cluster open or close to the incoming or outgoing federation deployments.

If the visibility of a cluster is set to private, it means that tenants of this cluster can access federated resources, but the cluster does not accept federation deployments. We consider such an option valuable as a small-size enterprise may want to benefit from the federated resources while being ensured that the federation workloads do not run on its infrastructure.

It is also not uncommon for a provider to demand control over what resources are shared and with whom. Our architecture gives providers the ability to configure sharing preferences at two layers. The former allows a provider to share a set of nodes in a worker cluster, while the rest of the nodes does not accept federation deployments. This can be done simply by labeling the nodes to be shared with federation workloads. The latter is to form an allow list or deny list with which providers can open or close their clusters, both manager and worker, to certain operators and tenants.

Edge computing, as discussed in Sec. 2.1.2.2, comes with heterogeneity in locations, network settings, hardware, software resources as well as resource owners. Some use cases will likely mandate different federation deployments. We hereby suggest that there is no one-size fits all solution regarding the federation deployment model for edge computing. We look at the earlier work that discusses various federation deployment scenarios [15] and distill out three of them to be adapted for the edge cloud infrastructure that this thesis envisages. Fig. 4.4 depicts these three deployment models: (1) Standalone; (2) Peer-to-peer; and (3) Hierarchical.

[15] No end user communicates with federation managers to create federation-scoped deployments.



(a) Standalone deployment. (b) Peer-to-peer deployment. (c) Hierarchical deployment.

Figure 4.4: A depiction of three federation manager deployment scenarios. Each color represents a different provider.

Standalone deployment. A federation provider can launch a standalone federation manager to be shared by multiple operators with the goal of federating their worker clusters (Fig. 4.4a). Clusters provided by different operators are inter-communicated through a single federation manager.[16] Such a deployment can be contained in a region, which results in preventing customers from accessing resources outside of the region. It can also consist of clusters scattered around the world to overcome this issue.

Lightweight as this deployment model is, it has some disadvantages as well. First of all, a single, standalone federation manager can federate a limited number of worker clusters, thus causing scalability deficiency. Although the worker clusters are autonomous, having one federation manager, even if it can have multiple control plane nodes to ensure HA, also introduces a single point of failure in the context of the federation. In other words, a down federation manager prevents users from accessing federated resources.

Peer-to-peer deployment. This model is based on federation managers peering with each other (Fig.4.4b). Multiple operators can provide their own federation managers, becoming federation providers, then peer with others for their tenants to gain access to resources in different regions. In this deployment model, each federation manager is more likely to retain worker clusters in a particular region rather than having them distributed to a wide area. This approach lowers the latency between federation managers with their worker clusters. This low latency enables a federation manager to quickly move workloads between its worker clusters as well as to receive available resource updates from these clusters with a brief delay, which improves scheduling decisions' accuracy.

This model addresses two drawbacks of standalone deployments. First, it overcomes the single point of failure as multiple federation managers peer with each other. If a federation manager in a region is down, the others can keep on sharing their resources without interruption. Having multiple federation managers also partially alleviates scalability weakness. However, the more peers there are, the more direct connection will occur, which may result in slower performance. Regarding security, each federation manager bears responsibility for its own protection as well as the collective safety of the federation. A security downside of peer-to-peer connections is that it might be tedious for a federation manager provider to

[16] No worker cluster establishes peer-to-peer communication with another unless they host identical deployments.

detect and cut off the flow of malicious activity in order to keep such traffic in a particular region in case of an organized attack.

Hierarchical deployment. This is a federation that is hierarchical in nature, forming a general tree that is owned and maintained by its federation provider (Fig. 4.4c). Federation managers construct a parent-child relationship where parents have control over their children. Multiple operators share federation managers, similar to the previous deployment models, with one difference: they can also federate their own federation managers as a child besides their worker clusters. In our estimation, a hierarchical deployment will cover one region, with a root-level federation manager running in the regional data center of the region. Then each federation located in a different region can form a federation of federations by peering with each other through the core network.[17]

There is no hard limit to the number of levels a hierarchy can employ, but the more levels, the more delays may occur in moving workloads between different subtrees. A federation provider can enrich these levels by putting in new ones or can refine them according to the physical infrastructure. Every level has one or more federation managers that may hold worker clusters. Direct communication occurs neither between federation managers at the same level nor between worker clusters, except for the worker clusters that host the replicas of the same services and applications. This limited access to federation managers reduces attack vectors, thus mitigating the security issues that arise from peer-to-peer deployments. A provider can cut malicious traffic that comes from a subtree off by disabling its corresponding child federation manager. The rest of the hierarchical federation remains serving well in this case. We believe that having fewer direct connections than solely peer-to-peer deployments contributes to better scalability. Although this deployment model addresses such disadvantages of previous ones, lacking direct connections might result in slight delays in moving workloads across federated clusters.

4.2.1.2 Accessing federated resources

Sec. 3.2.2.4 describes a lightweight mechanism through which every worker cluster can receive deployment instructions from tenants of other clusters without collisions. Tenants make deployments on remote worker clusters in the federation through the use of their local worker clusters. No tenant communicates with a federation manager in order to make such deployments. It means that a network disconnection of a worker cluster from the federation does not prevent its cluster admins and local tenants, as well as its worker nodes, from using its control plane locally, thus ensuring robustness [23].

Let us consider an example shown in Fig. 4.5 that illustrates a federation deployment:[18] A given federation F consists of one federation manager FM and three worker clusters, $WC1$, $WC2$, and $WC3$, one of which is used by a tenant T in order to make deployment on remote clusters in the federation. First of all, this tenant needs to be registered with one of these

[17] An example could be telco operators that run their business in different regions can build their own hierarchical federations and then peer them to achieve high utilization of their infrastructures.

[18] For clarity, we use the deployment term for *selective deployment* and the deployment package term for *selective deployment anchor*.

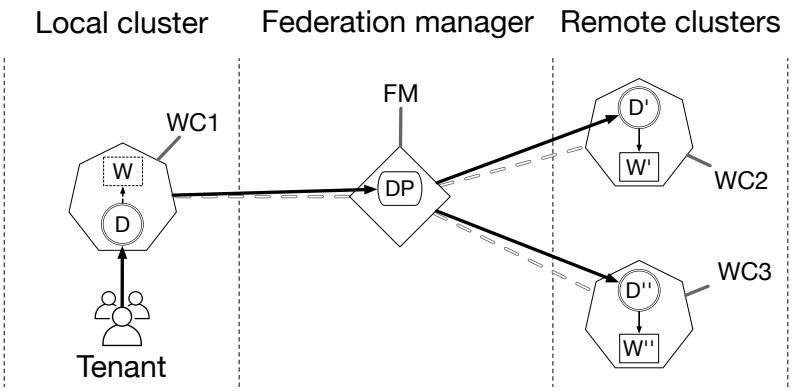


Figure 4.5: A tenant deploys three identical workloads on a local worker cluster and two remote worker clusters.

worker nodes. Let us assume that tenant T has registered with $WC1$, which is now the local worker cluster to the tenant.

T needs to prepare a specification for deployment D in order to run a workload W on remote clusters. This specification targets remote clusters, for example, by their locations, on which workload W will run. In our scenario, two remote clusters, $WC2$ and $WC3$, are targeted to receive D instruction; local cluster $WC1$ could also be targeted along with the other two.

Once D is made, it creates workload W on $WC1$ if it is included as a target, and then prepares a package DP for deployment D to be transmitted to federation manager FM.[19] DP contains as little information as possible regarding deployment, which is required for FM to make a scheduling decision at the level of federation. It also carries a secret of access credentials that FM uses to retrieve further information from D and that worker clusters $WC2$ and $WC3$ uses to update the status of D.

FM makes a scheduling decision based on information from DP, so selecting remote clusters $WC2$ and $WC3$. Once these two remote clusters are selected, the deployment procedure then begins with FM obtaining missing information from D. This missing information is used to create a deployment D' in $WC2$ and a deployment D'' in $WC3$, where $D \equiv D' \equiv D''$.

These three deployments have the same specification that describes how to run workloads. Each deployment in a separate worker cluster creates its own workloads: D, D', and D'' create W, W', and W'' respectively, where $W \equiv W' \equiv W''$. Upon the creation of these workloads, deployment D continuously receives updates on the status of W' and W'' from deployment D' and D''.

[19] If multiple federation managers collaborate, the foremost manager makes a scheduling decision and sends the deployment package to the selected manager through the intermediary managers if any.

4.2.1.3 Resource discovery

Resource discovery occurs in two flows: first from worker clusters to their federation managers, then between federation managers. Every worker cluster, through our *fedlet* entity, sends resource updates to its federation manager at an interval if there are no significant changes regarding resource availability. The status stanza of each cluster resource in a federation manager holds information related to the readiness and allocatable resources of the corresponding cluster. Our *manager cache* entity then consolidates these announcements and makes a cache from them.

In the context of peer-to-peer deployment, this cache is sent to all peers at an interval. If there is a hierarchical deployment, this cache is conveyed to the children managers at an interval. Subsequently, this federation manager pulls cache information from its children managers to synchronize with up-to-date caches made by others. These caches are used in order to search for feasible worker clusters during scheduling.

4.2.1.4 Scheduling

Sec. 2.1.3 glances through state-of-the-art scheduling solutions and states that scheduling algorithms are not in the scope of our thesis but that our federation architecture allows compute clusters to employ their own scheduling algorithms. This is because our architecture adopts a two-level scheduler approach that separates federation-level scheduler from cluster-level scheduler. The upper-level scheduler at federation managers selects feasible *worker clusters*, whereas the lower-level scheduler at worker clusters chooses feasible *worker nodes*.

The upper-level scheduler, namely the *federation scheduler*, first checks with the federation manager's clusters to see if any of them meets the requirements. This way, we aim to shorten the time it takes to move a workload from one cluster to another. Another advantage is that the federation scheduler has more up-to-date state information about these clusters, which may lead to more accurate scheduling decisions. If there are no feasible worker clusters, the federation scheduler acts on cached states of clusters that belong to other federation managers and prioritizes location over available resources.[20]

Once a worker cluster is selected, it receives the required information about the deployment to make its own scheduling decision to choose which nodes to run containers. We believe that there is no one-size fits all scheduling algorithm and that worker clusters can make the best scheduling decisions taking the workload context into account, compared to any central scheduling algorithm that runs on a federation scale.

4.2.1.5 Autoscaling

Sec. 4.2.1.1 presents the worker clusters that providers manage and maintain autonomously. It comes with its own advantages and disadvantages. For example, it does not require a

[20] Clusters are given labels, which contain their location information, programmatically by the *cluster labeler* entity.

central mechanism that manages autoscaling, which allows employing existing autoscaler tools to scale out and down pods and nodes. In other words, a compute cluster or a pod can independently be scaled out and down as a function of user requests, even if its connection to the federation is lost. When user requests are high, autoscalers that run locally can add new nodes to the cluster as well as can spin up new pods to meet the demand. However, this approach lacks a general view of federation resources, which may lead to inefficiency in utilizing federated resources.

4.2.2 Evaluation

This section assesses the performance of our cluster-wise federation architecture through our prototype. We aim to reveal how much overhead locating a federation manager cluster between worker clusters introduces and to what extent our multitenancy framework promotes workload mobility. To this end, we compare our prototype with native Kubernetes regarding pod deployment times. We measure the time it takes to make a deployment within a cluster in native Kubernetes experiments, whereas this is the time it takes to make a selective deployment from a local cluster to a remote cluster for our prototype. The measurements were carried out in 2023.

The results demonstrate that a tenant can deploy a single pod to a remote cluster, which includes creating the tenant's workspace, with a brief delay of approximately 1.5 s (median time) compared to native Kubernetes deployments. This indicates that introducing a federation manager between worker clusters does not entail a significant increase in pod scheduling time. For 20 concurrent deployments, the maximum time required for pods to be deployed and scheduled on a remote cluster is shorter than the median time required for creating a tenant environment through VirtualCluster when 4 simultaneous tenant creation requests are made. These outcomes offer additional evidence supporting our claim that our single-instance framework promotes workload mobility. Nevertheless, we observed a decrease in performance when concurrent deployment requests were made, which was also evident in the native Kubernetes experiments. This finding highlights the need for further optimization studies regarding federation components.

In Sec. 4.2.2.1, we describe the setups that we use for experiments, and Sec. 4.2.2.2 provides a step-by-step account of the experiments we conducted.

4.2.2.1 Setup

We prepared two experiment setups by using the GENI infrastructure to spawn four Ubuntu 20.04.6 LTS virtual machines: a VM with 8 CPUs and 12 GB of memory, a VM with 4 CPUs and 6 GB of memory, two VMs with 2 CPUs and 3 GB of memory each. We installed containerd version 1.7.0, Kubernetes 1.26.3 using kubeadm, and Calico version 3.25.0 to these VMs. Each VM hosts a separate one-node Kubernetes cluster.

For the native Kubernetes experiments, we assigned the VM with 8 CPUs and 12 GB of memory. The experiments were conducted on the Kubernetes cluster running on this VM.

For the experiments on our federation prototype, we reserved the remaining three VMs: two VMs with 2 CPUs and 3 GB of memory for the federation manager and for the worker-1 cluster, and one VM with 4 CPUs and 6 GB of memory for the worker-2 cluster. We created the VMs so that the resources allocated to the VM used for native Kubernetes experiments are equal to the combined resources allocated to the three VMs used for experiments on our prototype. 4 CPUs and 6 GB of memory were chosen for the worker-2 cluster because containers run on it. We federated the worker clusters through the use of the federation manager cluster, as explained in Sec. 4.2.1.1.

For the native Kubernetes experiments, we implemented a script that creates and deletes deployments while watching and recording events of deployments and pods. The deployments run a busybox container with the command to sleep indefinitely. This is to have containers consume as low as possible resources at the host. We conducted, for each case, 50 measurements to create deployments with the pod replica count set at 1, 5, and 20. We also conducted, for each case, 50 measurements to simultaneously create 1, 5, and 20 deployments with 1 pod replica each.

For the experiments on our federation prototype, we prepared a script that creates and deletes selective deployments, as well as watches and records events of selective deployments on the worker-1 and worker-2 clusters, selective deployment anchors on the federation manager cluster, and deployments and pods on the worker-2 cluster. We conducted, for each case, 50 measurements to create selective deployments with the pod replica count set at 1, 5, and 20. We also conducted, for each case, 50 measurements to simultaneously create 1, 5, and 20 selective deployments with 1 pod replica each. Similar to the native Kubernetes experiments, we used a busybox image with the command to sleep indefinitely for our containers.

4.2.2.2 Findings

We measure the time it takes to schedule pods, counting from the API call. For a single deployment, in Kubernetes, the median times are 272 ms for 1 pod, 470 ms for 5 pods, and 836 ms for 20 pods. An outlier is evident for the 1 pod case, where the maximum time is 4762 ms. In this case, long response times occur concerning the API server, controller manager, and etcd; it takes, respectively, 1155 ms and 4474 ms when creating a deployment and a pod. Additional investigation is required to reveal the underlying reason for such performance decline of these Kubernetes components. In EdgeNet, the median times are 1868 ms for 1 pod, 2285 ms for 5 pods, and 3129 ms for 20 pods, as shown in Fig. 4.6. This signifies that it takes 1868 ms in median time to create a single pod at the remote cluster and then have this pod scheduled. Regarding median times, EdgeNet introduces a brief delay of 1596 ms, 1815 ms, and 2293 ms for 1, 5, and 20 pods respectively. Since the selective deployment created on the local cluster is conveyed through the federation manager to the remote cluster, we may conclude that the federation manager component introduces negligible overhead, especially in the case of a single deployment.

Examining the maximum time spent deploying pods, we observe unexpected fluctuations for 1 pod and 5 pods. It clocks in at a maximum time of 6068 ms for 1 pod, 6578 ms for 5 pods, and 7613 ms for 20 pods. A rise is anticipated for deployment with multiple pods;

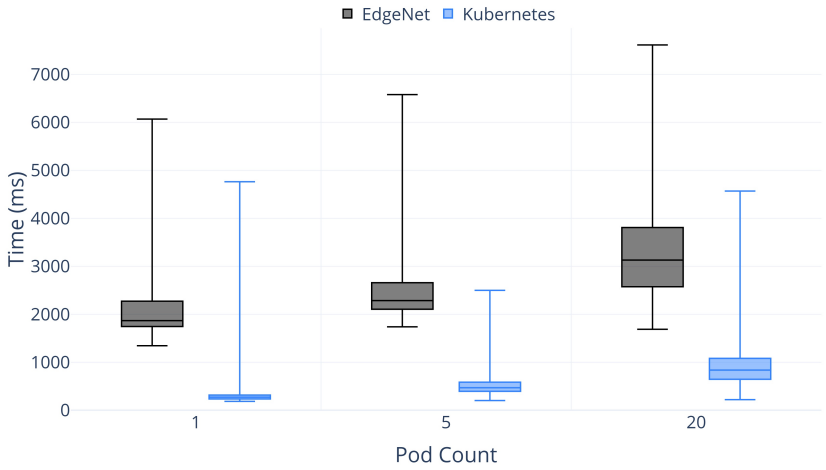


Figure 4.6: Pod scheduling time comparison for a single deployment. For EdgeNet, the time it takes to schedule pods on a remote cluster through a single selective deployment. For Kubernetes, the time it takes to schedule pods on a local cluster through a single deployment.

the Kubernetes scheduler assigns a node for each pod individually following its creation, which results in more time for the last pod to be scheduled. However, this is not expected behavior, especially in the case of a single deployment with 1 pod replica. The subsequent paragraphs further investigate how much time is expended on each component with the aim of elucidating the underlying cause.

We provide some context regarding operations done throughout a deployment period to lay the basis for an explanation. Once an API call is made, it takes some time to create an origin selective deployment in the local cluster. The selective deployment controller processes it and then creates a selective deployment anchor in the federation manager. Our simple federation scheduler then selects a cluster for this selective deployment. Upon assignment of a cluster, the selective deployment anchor controller conveys the selective deployment to the remote cluster. In order to accomplish this, the controller first communicates with the local cluster to obtain necessary information related to selective deployment and then creates it in the remote cluster. Subsequent to its creation, the selective deployment controller creates the deployment in the remote cluster, and then the deployment creates pods.

Regarding the median times for a single deployment with 1 pod, it takes 215 ms for *origin selective deployment creation*, 530 ms for *selective deployment anchor creation*, 245 ms for *federation manager scheduling*, 185 ms for *follower selective deployment creation*, 83 ms for *deployment creation*, 189 ms for *pod creation*, and 107 ms for *pod scheduling*. Fig. 4.7 demonstrates that the variation is considerably greater for operations that require communicating with other clusters, as is for selective deployment anchor creation and follower selective deployment creation. Among all, follower selective deployment creation presents the most pronounced fluctuations. We think the underlying reason for this is that it communicates both the local and the remote clusters during the operation. Consistent with these findings, we observe reduced fluctuations in the outcomes of operations that are handled locally, such as origin selective deployment creation, federation manager scheduling, and pod scheduling.

We encounter extended time spans when we analyze the simultaneous creation of mul-

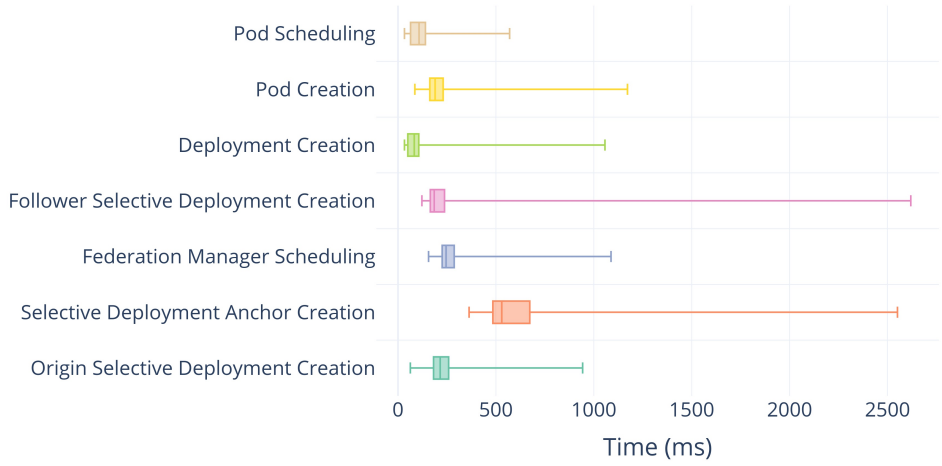


Figure 4.7: Time usage comparison for cluster-wise federation components. The case of a single deployment with 1 pod.

tiple deployments with one pod replica, as seen in Fig. 4.8. For 1, 5, and 20 deployments in EdgeNet, respectively, the median times are 1868 ms, 9600 ms, and 31,932 ms, and the maximum times recorded are 6068 ms, 19,153 ms, and 72,651 ms. Kubernetes experiments exhibit a similar outcome; for 1, 5, and 20 deployments, respectively, the median times are 272 ms, 769 ms, and 2874 ms, and the maximum times are 4762 ms, 7701 ms, and 16812 ms. This behavior aligns with our analysis in Sec. 3.3.1.2; the controllers process a higher quantity of objects, which includes both creation and modification, leading to an elevated number of API calls that saturate the API server, controller manager, and etcd moderately. As can be observed in Fig. 3.8a, changing the number of workers, running period, QPS, and burst values can improve the performance of these controllers.

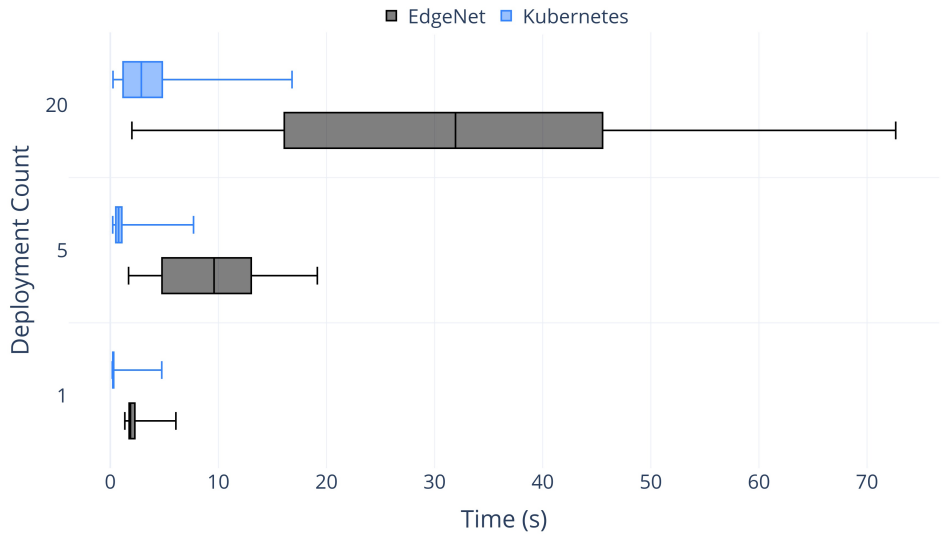


Figure 4.8: Pod scheduling time comparison for deployments that are created simultaneously. For EdgeNet, the time it takes to schedule pods on a remote cluster by using selective deployments. For Kubernetes, the time it takes to schedule pods on a local cluster by using deployments.

Fig. 4.9 shows the amount of time devoted to each component in the case of 5 simultaneous selective deployment creations. The three components appearing in the middle of the bars consume more time than others for each deployment, whether the first or last. This is

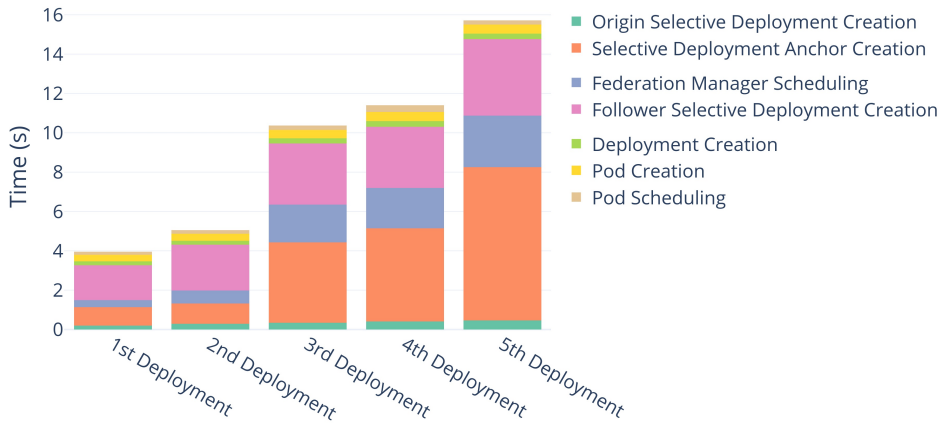


Figure 4.9: Time-based comparison of five concurrently made selective deployments from a local cluster to a remote cluster. Average time spent on each component.

also related to the controllers set to process two objects concurrently, which explains augmentation between the second and third deployments as well as between the fourth and fifth.

For the first deployment, the time allocated to scheduling at the federation manager is 363 ms on average, which is 158 ms to pod scheduling. For the last one, the federation manager scheduling records at 2624 ms, whereas pod scheduling measures at 218 ms. Our analysis suggests that the federation manager's components cause delays when the creation of 5 selective deployments occurs simultaneously, which results in the creation of pods at the remote cluster with time intervals. As the pods are created non-concurrently, we do not observe substantial variations, comparing all deployments, between the average pod creation times as well as between the average pod scheduling times. In addition, we noticed an implementation issue of the selective deployment anchor controller regarding status updates, which may also lead to a delay. Further investigation is needed to reveal the precise influence of this implementation issue on the deployment time.

When making 20 selective deployments concurrently, a discernible pattern is noticeable. Fig. 4.10 depicts that the time consumed by federation manager components takes a greater percentage of the overall time than other components for the last selective deployment in the series. Although this time taken by the federation manager components can be shortened with the optimization, these components will still account for a greater fraction of the total time for the last deployment than for the first deployment. As some workloads are more sensitive to time constraints than others are, prioritization must be enabled for selective deployments, as does Kubernetes for pods. In this way, a selective deployment with priority over others can be scheduled earlier.

In conclusion, we assert that the results indicate that our cluster-wise federation architecture introduces a reasonable overhead to run workloads on remote clusters. Through an optimization study, it is possible to reduce the time spent on federation managers when requests are concurrent. Our findings also show that a fully developed scheduling algorithm for the federation must include selective deployment priorities so that time-sensitive and time-critical workloads can be deployed on remote clusters with brief delays. With our federation prototype working in concert with our multitenancy framework, a single deployment for a tenant's pod onto a remote cluster, including scheduling and tenant workspace creation,

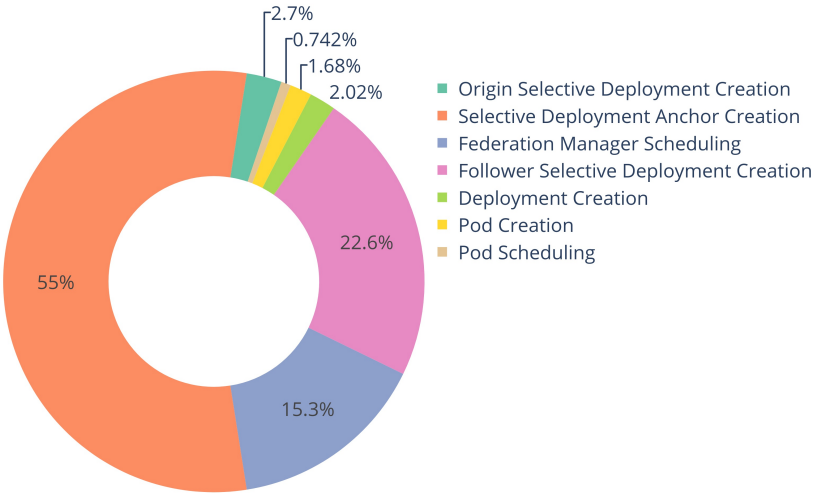


Figure 4.10: The case wherein twenty concurrent selective deployments are initiated from a local cluster to a remote cluster. Average time that is spent on each federation component for the 20th deployment.

takes under 2 s (median time). It even takes approximately 72 s at maximum for a tenant to make 20 concurrent deployments on a remote cluster. For VirtualCluster, as seen in Table 3.1, only preparing a tenant environment at a remote cluster would take 82 s (median time) when 4 simultaneous creation requests are made, excluding the time it would take to make 20 deployments that originate from a local cluster onto a remote cluster. This comparison proves that our EdgeNet multitenancy framework, which is single-instance, promotes workload mobility in federated environments. It is our contention that these findings also show that, in EdgeNet, tenants of a local cluster can quickly move their workloads across federated clusters while imposing low overhead.

4.3 System-wise federation

With edge computing, we are witnessing heterogeneity not only in resources and devices but also in architectures and their implementations, including APIs, and we will continue to see these unfold. In the context of container orchestration, what we mean by the system-wise federation is the interconnection and interoperability of the tools and platforms that can manage containers' life-cycle. It is, therefore, necessary to standardize container orchestration, as has been done for containers through OCI[21] from our perspective.

CNCF[22] endeavors to provide such standardization for container orchestration through Kubernetes, today's de facto industry standard container orchestration tool.[23] These valuable efforts can lead to standardization, but a biased assumption may be that alternative tools (See Sec. 2.1.3 for details) are not suitable for orchestrating containers at the edge and that they will remain unsuitable even if they are so now. We believe there is no one-size fits all

[21] Open Container Initiative https://opencontainers.org/

[22] CNCF https://www.cncf.io/

[23] Kubernetes, which is Google's third container orchestration system built on top of the knowledge acquired from Borg and Omega [19], had been developed under the roof of Google until being donated to CNCF and open-sourced.

solution that can address the diverse requirements of many edge use cases. Furthermore, the more operators who offer edge clouds, the more likely they are to employ various container orchestration tools to satisfy the needs of their customers in the future.

For standardization, foundational properties[24] that a container orchestration tool should possess must be identified. We think that it can pave the way toward developing new container orchestration tools, each to address a set of different requirements. It could facilitate interoperability between these tools. Thus, edge clouds can still interoperate with minimal effort even if edge cloud providers adopt different container orchestration tools for their CaaS solutions.

Regarding the system-wise federation, we have developed an aggregate manager (AM), which empowers researchers to use our globally distributed edge cloud testbed, EdgeNet, through Fed4FIRE+[25] as they do for the other federated testbeds. This AM removes the requirement of registering with the testbed to conduct measurements on our globally distributed edge cloud. Sec. 5.4 explains how we have implemented it.

4.4 Conclusion and future work

We have devised an integrated federation strategy for Containers as a Service (CaaS) that incorporates compute resources in the forms of nodes, clusters, and systems that providers offer in many locations, which is a more comprehensive approach than the scope of current federation tools. Our strategy encourages the inclusivity of all sizes of providers to mitigate the resource provisioning challenges that stem from the highly distributed and heterogeneous infrastructure that edge computing envisages. This, we believe, can ensure a federated edge infrastructure at scale.

We also contend that for edge computing infrastructure to be successfully built at scale while being sustainable, compute resources must be offered in various forms changing from nodes to clusters by multiple providers as well as must be shared by multiple tenants (See Sec. 3.4 for details). Such that shared and federated clusters allow tenants' workloads to be seamlessly moved between edge clouds operated by multiple providers with low overhead. Our CaaS federation toolset that works in concert with our CaaS multitenancy framework provides a basis to achieve establishing such a sustainable edge infrastructure.

Scheduling decisions occur at the two levels at most in our prototype that establishes the cluster-wise federation. This approach prevents repetitive decisions from being made, avoiding a possible overhead. As our prototype works with workload resources such as deployments, existing mechanisms are employed locally to manage pod and node autoscaling, improving scalability. The removal of a centralized federation control plane with which users communicate addresses the single point of failure and split-brain problems as well as contributes to scalability.

Our measurement results show that it takes under 5 minutes to provide a node to a cluster

[24] Namespaces, Pods, Service, Ingress, and Workload Resources are some of the Kubernetes primitives that can be the basis for standardization effort.

[25] Fed4FIRE+ https://www.fed4fire.eu/

with our deployment procedure. Sec. 3.3 demonstrates that with RKE, it still takes around 5 minutes to start a cluster that runs in Docker containers if the host environment is ready. Furthermore, the results of our experiments on the cluster-wise federation architecture show that it does not cause a significant increase in the time it takes to deploy a pod to a remote cluster. Our findings also support our assertion that our single-instance multitenancy framework enhances workload mobility.

Everyone has access to our free, open-source code to apply our federation strategy for Kubernetes, which is particularly tailored for edge clouds, where providers and locations are many. The node-wise federation is actively being used in the EdgeNet edge cloud testbed, for which the providers are research centers, universities, and individual contributors across the globe. An instance for system-wise federation is also currently operational, which allows Fed4Fire+ users to make use of the EdgeNet testbed. As future work, the following assignments remain to be addressed.

Node sharing preferences. A provider who offers nodes should have control over what nodes are shared with whom, as does in the cluster-wise federation. We will develop a mechanism that allows node providers to declare allowlist or denylist for each node that they provide.

Authentication for node providers. In our vision, multiple node providers will constantly offer nodes in many locations to clusters. Although the operators of these clusters can manually enforce authentication to node providers, it is not a scalable approach for both these operators and node providers. We envisage a process in which node providers authenticate through a system, using a command line interface or web application, so as to retrieve their provider-specific bootstrap script to be run on the target node.

Experiments on federated clusters. Our cluster-wise federation architecture has not been validated yet through rigorous experiments and analysis. Our first step is to reveal how many worker clusters a federation manager support to scrutinize scalability competency. Henceforth, peer-to-peer and hierarchical deployment models will be subject to experiments with the same purpose. Then we will deploy federation managers as well as worker clusters in an array of dissimilar locations to assess which deployment model is suitable for what kind of edge use case. Further experiments will probe the competence of features such as the federation scheduler and will benchmark our architecture with other solutions.

Federation scheduler algorithm. The current federation scheduler employs a simple algorithm that prioritizes the locations of clusters. Resources of clusters are partially included in these decisions, which might lead to inaccurate decisions under challenging scenarios. Along with a cache optimization study, an algorithm that the federation scheduler will use is another research direction.

Consensus algorithm for failover. As part of our architecture, tenants contract with a single operator they choose; they can access federated resources through a worker cluster of the preferred operator. Tenants must be able to continue using the system if a worker cluster that they belong to goes down. HA can be achieved through multiple control plane nodes, but it does not address a disaster scenario in which the site that hosts this worker cluster can go down. We plan to handle this by replicating tenant data toward multiple worker clusters in different sites and forwarding tenant traffic to available clusters if a worker cluster

is down. The same method can be applied to federation managers with regard to peer-to-peer and hierarchical deployments in order to keep worker clusters accessing federated resources if their federation manager is down. If these clusters are mobile, it introduces another complexity, which may require a consensus algorithm to be developed.

Orchestration tool properties. It is important to identify and characterize the foundational properties of container orchestration tools for standardization. This is also an open research question that requires intense collaboration with the communities that develop container orchestration tools.

Cluster-level slicing. Besides node-level slicing and sub-node-level slicing, a tenant may need to acquire an entire cluster in the federation. Our slice feature introduced in Sec. 3.2 will be extended to include this functionality.

Networking in cluster-wise federation. There are a number of available tools to establish inter-cluster networking, such as Submariner,[26] Skupper,[27] and Istio.[28] We will analyze these tools to reveal their pros and cons in the context of our cluster-wise federation architecture. A VPN solution, as is in the node-wise federation, can also be developed.

[26] Submariner https://submariner.io/
[27] Skupper https://skupper.io/index.html
[28] Istio https://istio.io/latest/

Chapter 5

A real-world instance as a globally distributed testbed

Traditional cloud architectures are concerned with providing on-demand access for external users to compute and storage resources located in centralized data centers. This model is challenged with the emergence of new applications, such as content delivery, peer-to-peer multicast, distributed messaging, and the Internet of Things (IoT). These applications are sensitive to latency and they benefit from compute resources that are geographically close to the user.

Edge clouds complement centralized clouds by placing computation and storage resources close to users or data sources, to offer high bandwidth and low latency between cloud computing resources, data producers, and data consumers. For well over a decade, the networking and distributed systems research communities have deployed a series of wide-area edge cloud testbeds, such as PlanetLab Central [107], PlanetLab Europe [43], GENI [94], G-Lab [98], V-Node [101] and SAVI [84]. These testbeds degraded over time for two reasons: they relied on dedicated hardware, which required on-site support, and they used custom control frameworks.

The requirement for dedicated hardware has led to maintenance and scalability issues. On one hand, the cost of purchase and replacement of servers discouraged people from contributing to the testbeds. On the other hand, the human resources required to maintain servers over the long term were costly. In the long term, the testbeds were not able to scale.

The testbeds also relied on custom software for managing the nodes and the experiments. These control frameworks were typically written and maintained by a small team of researchers, and used by a relatively small community of distributed-systems experimenters. Such software gets quickly outdated, and is only improved by the small communities of the original developers and dedicated experimenters. This lack of standardization resulted in a waste of resources, as each testbed has to be documented individually, and experimenters had to learn testbed-specific knowledge.

We argue that the solution to make the next generation of distributed testbeds viable

is to rely on widely used control frameworks and on inexpensive virtual machines. This approach reduces the cost per site, as the virtual machines can be created for free on an existing infrastructure, and requires almost no maintenance. This solves the scalability problem by lowering the entry barrier for contributing new nodes. In addition, there is no need to maintain extensive testbed-specific documentation and software, as most of it is reused from external projects. This also benefits the users of a testbed, as by learning how to use it, they gain industry-valuable knowledge. However, today's common cloud frameworks treat nodes as homogeneous entities in a centralized data center, whereas a key feature of edge testbeds is their heterogeneity and geo-diversity. Up until now been no production-ready framework for edge cloud testbeds.

This chapter provides further details on the EdgeNet free, open-source software that allows an edge cloud to be deployed onto virtual machines as worker nodes, with Kubernetes as the orchestration framework. EdgeNet offers a novel architecture for edge computing, which directly addresses the sustainability and maintenance issues described above. Since an EdgeNet worker node is typically a VM running at a site's local cloud, the expense of maintaining a dedicated hardware resource disappears; in fact, an EdgeNet VM is just another VM among many running at that cloud, requiring no marginal maintenance commitment. Using Kubernetes directly addresses the maintenance, upgrade, and training issues of control frameworks mentioned above. A worldwide community of developers maintains and extends Kubernetes, and extensive documentation and training resources are available on the internet.

The challenge that EdgeNet overcomes is that the use of Kubernetes as an edge cloud orchestration framework breaks Kubernetes' central design assumptions in three important areas:

- Kubernetes was designed for a single-tenant deployment. In our cluster, there are mutually-untrusting multiple tenants.
- Kubernetes was designed for homogeneous nodes, where computation could be rapidly moved from one node to another in the cluster. For EdgeNet, a node's physical location is a first-class design parameter, and so nodes are heterogeneous in physical location. Further, experimenters must be able to choose where their worker nodes are placed.
- Kubernetes was designed so that control plane nodes and all worker nodes were within the same cluster, so communication was on layer 2 and latencies were on the order of microseconds. In our deployment, layer-2 connectivity is not available, and inter-node latencies are on the order of 10s of milliseconds.

Kubernetes is widely used in central data centers for container organization. EdgeNet brings it to the edge with three key contributions: (1) Multi-tenancy at the edge, providing isolation between tenants and sharing limited resources fairly, so that multiple organizations can concurrently benefit from the edge cloud (Sec. 5.1). (2) A node selection feature that makes it possible to deploy containers on nodes based on their locations. It is easy to configure deployments to schedule pods to cities, countries, continents, or latitude-longitude polygons (Sec. 5.2). (3) A single-command node installation procedure lowers the entry barrier for the institutions wishing to contribute nodes to the cluster, thus simplifying the establishment of an edge cloud (Sec. 5.3). This procedure includes a virtual private network solution that

makes it possible for nodes located behind NATs to join a cluster.

Other noteworthy contributions that we make are as follows. (4) We have developed an aggregate manager (AM), which allows users from the Fed4FIRE+ federation to make use of the EdgeNet testbed as they do for the other federated testbeds (Sec. 5.4). (5) The EdgeNet has supported more than 10 experiments since its start (Sec. 5.5).

The following is how we organize the sections of this chapter. In Sec. 5.1, we explain how we use our multitenancy framework in the context of the testbed. Sec. 5.2 describes a feature that allows location-based node selections. Sec. 5.3 provides details about how our node deployment procedure is employed in production. In Sec. 5.4, we present how EdgeNet is federated with Fed4FIRE+. In Sec. 5.5, we describe the current status of the platform along with several experiments that have been conducted on it. Sec. 5.6 evaluates the performance of EdgeNet. Sec. 5.7 explains how we deploy observability tools in EdgeNet. In Sec. 5.8, we showcase EdgeNet's ability with its geographically distributed nodes to uncover an experimental CDN framework's performance issues through experiments that we ourselves conduct, and the conclusion and future work are discussed in Sec. 5.9.

EdgeNet's open-source software is freely available on GitHub[1], and the testbed that it supports[2] is open to researchers worldwide.

5.1 Multitenancy

EdgeNet employs the multitenancy framework introduced in Chapter 3 to allow multiple research groups to conduct experiments in parallel, thus opening up its infrastructure to be shared by a broader community with the aim of achieving high resource utilization in the cluster. An EdgeNet tenant can be a not-for-profit research organization, a research team, or even a teaching unit supervised by a professor. Through tenant resource quotas, we ensure that the testbed's resources are fairly shared among them. The subnamespaces feature allows organizations to assign a namespace per research team, as does research teams per experiment and as does teaching units per class. EdgeNet users can participate in multiple tenants and subnamespaces, and this approach facilitates collaboration between different research groups. Our multitenancy framework, however, does not allow users of other testbeds to use EdgeNet without enrollment by default, a concern that we address in Sec. 5.4.

In the testbed context, a shared node can be either a VM or dedicated hardware, like ODROIDs,[3] to which multiple researchers deploy containers. Lightweight virtualization is still required to better isolate experimenters' containers: VM nodes must support the nested virtualization for this purpose, causing increased overhead. Additionally, a noisy neighbor is a major problem, as is in the cloud, that degrades neighboring experiments' performance, which may result in varied experiment outcomes. Although resource quota enforcement fairly mitigates this problem, not every resource is detected by Kubernetes. Depending on the requirements of experiments, there may be a need for more isolation and guaranteed

[1] The EdgeNet software https://github.com/EdgeNet-project

[2] The EdgeNet testbed https://edge-net.org/

[3] On the home networks, we deploy ODROID nodes (Sec. 5.3.1).

access to those resources at a specific location. To do so, our variable slice granularity allows experimenters to acquire entire nodes that are isolated from multi-tenant workloads for a given time.

5.2 Location-based node selection

EdgeNet's main value as compared to pure Kubernetes is its ability to deploy containerized software to a widely distributed set of nodes rather than to nodes that are all grouped in a centralized datacenter. To do so, the users must be able to deploy containers based upon a node's location. Note that Kubernetes offers the ability to choose specific nodes based on *labels*, but it is up to the cluster administrators to attach the relevant labels to the nodes. EdgeNet achieves location-based deployments with two components: (1) a service that geolocates nodes and attaches the appropriate labels to them; (2) a *selective deployment* resource, which allows the users to select amongst nodes based on geographic criteria.

5.2.1 Node labeler

In order to be able to select nodes by their location, EdgeNet attaches multiple labels to the nodes according to their city, state/region, country, continent, and coordinates. This is done by the *node labeler*, a controller that watches the cluster for new nodes, or for node IP updates, and geolocates the nodes. By default, the nodes are located by IP address, using the MaxMind GeoLite2 database. If the node is running at a known cloud provider, we use the location of the data center in which the node is running, obtained from the instance metadata. For example, we assign the following labels to a node located at Stanford:

- edge-net.io/city=Stanford
- edge-net.io/state-iso=CA
- edge-net.io/country-iso=US
- edge-net.io/continent=North_America
- edge-net.io/lat=n37.423000
- edge-net.io/lon=w-122.163900

5.2.2 Selective deployment

Once the labels are attached to the nodes, they can be used to select specific nodes when creating resource objects such as deployments. However Kubernetes' built-in selectors are limited, consisting only of equality comparison (*city = Stanford*), and set inclusion (*city in (Paris, Stanford)*). In order to enable additional selection criteria, we introduce the *selective deployment* resource.

A selective deployment is comprised of a resource type (a deployment, for example), and of geographic queries associated with a number of nodes. It can target nodes by con-

tinent, country, region, and city, as well as polygons that are described using latitudes and longitudes. In order to efficiently determine which nodes lie within a polygon, we use the Point-in-Polygon algorithm [48]. When a selective deployment is created, the controller finds the relevant nodes and creates the appropriate resource objects. If a node goes down, the controller re-configures its resource objects in order to start a new pod on a new node in the same geographic area.

5.3 Node contributions

The edge computing paradigm captures the attention of researchers in launching experiments from the edge of the network. While providing an edge cloud testbed at scale is possible to satisfy this demand, it poses a challenge since the nodes are located and maintained at the edge and they require physical and remote access, thus incurring additional costs. If these nodes are located behind network address translation (NAT) boxes, it introduces an extra burden in terms of remote access. Taken together, the issues of delivering, maintaining, and accessing nodes at the edge, also considering hindrances that come from NAT boxes, impair the long-term viability of edge cloud testbeds due to economic constraints.

We grouped the challenges of delivering edge nodes into three categories: provision, access, and maintenance. Geographically distributed infrastructure conceived for edge computing causes an increase in the complexity of provisioning nodes. In order to alleviate this complexity, an easy node deployment procedure is necessary. However, this kind of procedure does not address economic and organizational obstacles to a provider establishing ubiquitous edge clouds that run at scale. To overcome such economic and organizational difficulties, a collaboration between edge cloud providers, where they open up their infrastructures to each other's users, possibly through a federation, can be an efficient method, as discussed in Chapter4.

The edge infrastructure mentioned above also limits physical access to the nodes. This physical access issue is primarily related to maintenance and also urges collaboration across organizations providing nodes. On the other hand, remote access is a technical problem. Nodes behind NATs do not hold a public IP address but a private IP address to communicate; thus, they become unreachable outside the local network remotely by default. That is to say, a system's control plane cannot reach such a node to take necessary actions such as service deployment. Being physically inaccessible and having connectivity issues raises maintenance challenges for edge nodes.

We tackle these three issues on EdgeNet: providing, accessing, and maintaining edge nodes:

- **Home Networks:** With the advent of edge computing, home networks have drawn heavy attention from internet researchers for the past few years. With home networks, the presence of NAT boxes is an important deterrent to deployment. Thus, it is not a trivial task to manage the life-cycle of an experiment, such as deployment and version updates, that launches measurements from a node behind a NAT box.

  We offer a virtual private network solution that makes it possible for nodes at home

networks to take part in a cluster, handling the access issue, as explained in Sec. 5.3.1. An agent running on each node configures the VPN network by actions such as assigning an IPv4/IPv6 pair of addresses. Current deployment ensures a VPN tunnel is established between two public nodes or between a public node and a NATted node. In future releases, we plan to support communication between two NATted nodes through the VPN network.

- **Node Deployment:** A fundamental necessity is a simple procedure that safely sets up a node and makes it join a cluster. By providing nodes in such a forthright manner, a contributor is less likely to give up during the process.

  EdgeNet facilitates node contributions to the cluster via the deployment procedure that Sec. 4.1.2 describes. In Sec. 5.3.2, we show that anyone can contribute a node to the EdgeNet public cluster using either our bootstrap script or pre-built cloud image, and Sec. 5.3.3 discusses how we plan to achieve node robustness in the future.

5.3.1 Home networks

Kubernetes is designed for centralized data centers, and on that basis, the system assumes that cluster nodes share a local network. Put another way, it does not provide an off-the-shelf solution for nodes on different networks without public IP addresses. This introduces two communications problems:

- A node behind a NAT box can access control plane nodes, but a control plane node cannot access that node.
- Containers on a node behind a NAT box are unreachable from other cluster nodes.

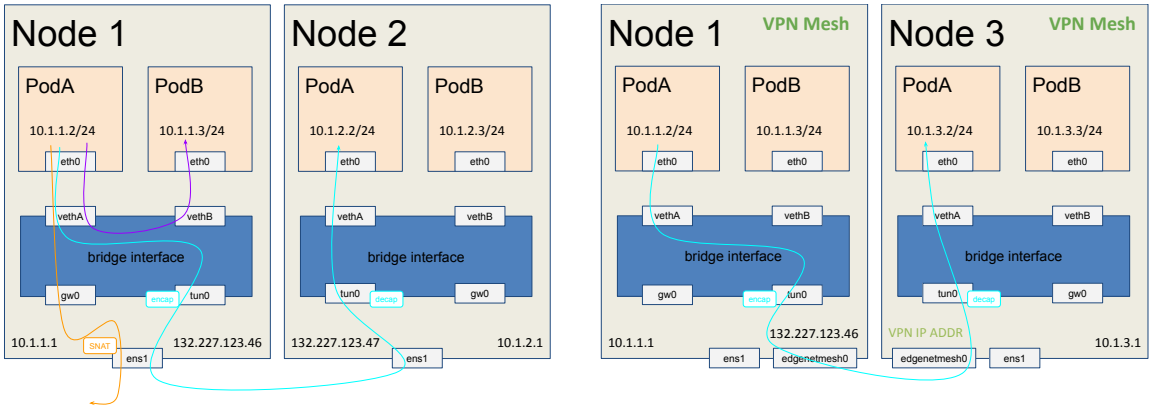
Kubernetes has an extensible architecture that allows developing and using plugins. A container network interface (CNI) plugin typically establishes networking between pods. EdgeNet employs VMware's Antrea CNI[4] for this purpose. Antrea uses the Open vSwitch (OVS) bridge[5] on every node. Furthermore, it forms a virtual ethernet (veth) pair for each pod, a gateway (gw) to the node subnet, and a tunnel (tun) for inter-node communications.[6]

The OVS bridge forwards packets using veth pairs on the node regarding local pod traffic. If traffic is toward an external destination, packets to be routed are forwarded through the gateway port. Antrea benefits from source network address translation (SNAT) so that the pod IP address is preserved. In terms of inter-node communication, tunnels encapsulate and decapsulate packets. Fig. 5.1a depicts the traffic flow where every node has a public IP address.

However, if a node is behind a NAT box on another network, it blocks inter-node communication. To overcome this problem, we set up a VPN tunnel between every pair of nodes in the cluster. We settled on using the WireGuard VPN [38] for multiple reasons:

- its performance: it offers throughput 5 times higher than OpenVPN and 1.1 times

[4] VMware Antrea https://antrea.io/
[5] Open vSwitch https://www.openvswitch.org/
[6] The VMware Antrea architecture https://antrea.io/docs/main/docs/design/architecture/



(a) Pod traffic where every node has a public IP address. Orange is for pod-to-external, cyan is for inter-node, and purple is for intra-node traffic.

(b) Pod traffic where a node has a private IP address. Cyan shows that traffic is routed through edgenetmesh both on the source and destination nodes.

Figure 5.1: Traffic flow in EdgeNet. The drawings are inspired by VMware architecture documentation.[6]

higher than IPsec on the same configuration;[7]

- its simplicity: it only requires generating of public/private pair of keys for each client and does not requires a PKI infrastructure as OpenVPN certificate-based authentication does;
- its integration in the Linux kernel: it is natively integrated in the kernel since Linux 5.6, requiring no additional kernel modules.

Our solution provides IPv4 and IPv6 peer-to-peer communication for every nodes of the cluster (with the exception of NAT-to-NAT communications, see Sec. 5.3.1.2) over the public IPv4 internet, as seen in Fig. 5.1b.

5.3.1.1 Bootstrapping VPN peers

A node must have established VPN connectivity with the rest of the cluster before Kubernetes starts. To achieve this, an agent present on each node performs the following actions on boot:

1. It checks if a public/private key pair has ever been generated. If none, it generates one and saves it for subsequent boots.
2. It checks if an IPv4/IPv6 pair of address has ever been generated. If none, it queries the cluster to get the list of used IP address in the VPN network, and chooses a random pair of addresses amongst the ones available. Randomization allows to reduce the risk of two new nodes choosing the same IP address if booted at the same time.
3. It publishes its public key, its private IP address pair, and its public IP address to the cluster by creating a *VPNPeer* Kubernetes resource.
4. It queries the list of *VPNPeer* resources and configures the tunnel interface to add each peer.

[7] The WireGuard benchmarking https://www.wireguard.com/performance

5.3.1.2 NAT-to-NAT communications

In our current deployment, a VPN tunnel can be established between two public nodes, or between one public node and one NATted node. Establishing a connection between two NATted nodes requires the use of an external server to exchange port numbers and perform UDP hole punching. We will implement such a technique in future iterations.

5.3.2 Node deployment

There are two ways forward to contribute a node to the EdgeNet cluster: Bootstrap script and pre-built cloud images.

For users who want to deploy nodes on their own machines, we provide a *bootstrap* script that installs Ansible, downloads the playbooks, and runs them. Note that users comfortable with Ansible can directly use the EdgeNet playbooks to deploy a node.

We provide prebuilt cloud images for the major cloud providers (Amazon Web Services, Google Cloud Platform and Microsoft Azure). These images allow any user of these clouds to deploy an EdgeNet node with no configuration required. On first boot, a *NodeContribution* object is created and the node is allowed to join the cluster.

5.3.3 Node robustness

In comparison to other testbeds such as PlanetLab, EdgeNet nodes are not expected to be maintained by system administrators. Besides VM nodes, a physical node can be deployed in a user's home with limited debugging time and knowledge. As such, we must ensure that the nodes are able to self-heal in case of problems. We have currently identified two main issues: unresponsive nodes and file system corruption. We describe below two tentative solutions that we will try to implement in the next iteration of EdgeNet.

5.3.3.1 Unresponsive nodes

A node can become unresponsive if some application consumes all of its resources, or if an excessive amount of network traffic saturates the network interface and the CPU. We are investigating the use of the hardware watchdog present on Raspberry Pi and ODROID single-board computers to automatically reboot unresponsive nodes. The kernel periodically sends heartbeats to the watchdog. If the watchdog stops receiving heartbeats, it power cycles the node. This procedure can fix unresponsive nodes without any user intervention.

5.3.3.2 File system corruption

Single-board computers often use flash-based memory such as SD cards or eMMCs. These memories are prone to failure as they are usually not designed for continuous random writes

over long period of times. When these memories fail, the file system is corrupted and the systems stop working properly. It is also possible that a user improperly changes the configuration of a node.

To handle this issue, we are exploring the possibility of booting nodes over the internet. The petitboot[8] bootloader can boot a kernel and a live disk image over the internet. The disk image and the kernel would live in RAM, and the node's flash storage would only be used to store container data. If the flash memory fails, the node would still be accessible and the user would only have to replace the SD card.

This method also has the benefit of making system updates very easy: deploy a new disk image on the server and reboot the remote nodes. If the update fails, roll back the disk image on the server. The main concern with this approach is security and how to authenticate the boot server as well as the disk images.

5.4 Federating EdgeNet with Fed4FIRE+

The EdgeNet testbed provides nodes scattered around the world, with access to the internet and private network connectivity between each nodes, and thereby constitutes a valuable platform for running Next-Generation Internet (NGI) experiments. For this reason, we received funding from the European Commission sponsored Fed4FIRE+ project to integrate EdgeNet into a federation alongside many other computer networking testbeds in Europe. The project, in conjunction with the GENI project in the United States, provided a unified way of accessing heterogeneous testbeds through a common API called the GENI Aggregate Manager (AM) API v3. We implemented this API for EdgeNet, thereby granting access to our testbed to experimenters who presented electronic Fed4FIRE+ credentials, allowing them to deploy and access containers on nodes of their choice.

This federation with Fed4FIRE+ presented an interesting challenge in reconciling two very different system control approaches. The Kubernetes API is natively a declarative API: users define the desired state of a resource (e.g., a container) and a control loop (also called the *controller*) keeps the resource in sync. In contrast, the AM API is imperative by nature: users perform actions that change the state of a resource, such as *allocate*, *provision*, *shutdown*, etc. In order to reconcile the two paradigms, we have created an AM API that manages Kubernetes objects on the behalf of the users.

5.4.1 Mapping GENI resources to Kubernetes resources

The AM specification defines three main kinds of resources: users, slices, and slivers. Slivers are collections of compute resources, and users are given rights to create slivers in slices. In our case, we seek to offer experimenters SSH access to Docker containers running on EdgeNet. Thus, a sliver maps to a collection of three Kubernetes objects:

- A *Deployment* object defines the specification of the container: image, node and CPU

[8] petitboot https://github.com/open-power/petitboot

architecture. Kubernetes will ensure that a container matching these specifications will always be running.

- A *Service* object maps an available TCP port of the host node, to the SSH port of the container.
- A *ConfigMap* object holds the SSH keys of the user and is mounted on *~/.ssh/authorized_keys*.

Users can choose a specific Docker image, node, and CPU architecture (aarch64 or x86_64). If none are specified, the AM API will choose a default image and Kubernetes will create the container on any available node.

5.4.2 Resource expiration

Slivers have an expiration time, which can be extended by performing the renew action. When a sliver expires, the associated resources must be deleted. Kubernetes has currently no way of specifying expiration dates for objects and automatically deletes them (excepted for Jobs resources). To work around this, we run a garbage collector goroutine which periodically checks for the expiration of the resources and deletes them.

5.4.3 Object naming

Object names are derived from the first 8 bytes of the SHA512 hash of the sliver name. This allows the creation of objects with names that are valid in the GENI AM specification, but not in Kubernetes which allows only alphanumeric chars.

5.4.4 Non-standard TLS certificated workaround

As per the specification, clients are authenticated using client TLS certificates. The certificates provided by Fed4FIRE+ contain non-standard OIDs (Object IDentifiers) which cannot be parsed by the Go X.509 parser. Specifically, the *Authority Information Access* OID contains numbers larger than 32-bits. This makes it impossible to authenticate clients using Go code and prevents the use of reverse proxies written in Go such as the popular Caddy[9] and Traefik[10] proxies. Upon discussion with the Fed4FIRE+ administrators, it became clear that there is no immediate plan for the Fed4FIRE+ OID format to change. To work around this issue, we place an NGINX[11] reverse proxy in front of our AM API server. This proxy performs client TLS authentication and forwards the request to the AM API server by including the certificate in a custom `X-Fed4Fire-Certificate` HTTP header. This information is then passed to external tools by the AM API, such as `xmlsec1` to validate credentials and authorize users.

[9] Caddy https://caddyserver.com
[10] Traefik https://doc.traefik.io/traefik
[11] NGINX https://www.nginx.com

5.4.5 Deployment

Our AM API is publicly deployed[12] and its source code is available on GitHub.[13] It can easily be deployed on any Kubernetes cluster to federate that cluster with Fed4FIRE+. That is to say, the API that we provide is general to Kubernetes and is not specific to just the EdgeNet Kubernetes cluster.

5.5 Platform status

As of April 2023, EdgeNet is up and running at over 40 nodes worldwide including 7 in France, 1 in Germany, 1 in Greece, 2 in Japan and the others in the United States.[14] The current nodes are hosted by universities and the GENI [94] testbed in the USA. In addition, several experiments have been conducted on EdgeNet over the past year:

CacheCash (NYU Tandon School)
CacheCash [4] is a blockchain-based CDN that involves the end users themselves into the network to serve content through their own machines. More than 30 EdgeNet nodes have been used to deploy CacheCash and perform extensive latency, throughput, and resource usage measurements.

PacketLab (CAIDA)
CAIDA's PacketLab, which made a demonstration using EdgeNet in IMC'22, allows vantage point sharing among measurement researchers [165]. PacketLab offers EdgeNet nodes as external endpoints to its users.

Internet scale topology discovery (Sorbonne Université)
The Multilevel MDA-Lite Paris Traceroute [161] tool, an evolved version of the well known `traceroute` tool, was used to survey the internet from EdgeNet nodes continuously. Also, Diamond-Miner [160], which conducts high speed internet-scale route traces, has been deployed on 7 EdgeNet nodes as part of a production internet topology measurement system.

Neuropil (pi-lar)
An open-source project for cyber security mesh is Neuropil.[15] The experimenters put their implementation to the test regarding scalability and functionality in a real-world environment through EdgeNet. Thanks to these experiments, they detected memory leaks and scalability issues due to their implementation.

Cyberlab Honeypot Experiment (University of Ljubljana)
The honeypot experiment uses EdgeNet to expose fake SSH servers on the internet and detect malicious activities.

[12] EdgeNet AM API https://fed4fire.edge-net.io
[13] EdgeNet AM software https://github.com/EdgeNet-project/fed4fire
[14] In 2021, EdgeNet was up and running at over 40 nodes worldwide including 5 in Europe, 1 in Brasil, 1 in Australia, and the others in the United States.
[15] Neuropil https://www.neuropil.org/

Reveal topologies of remote networks
(Université de Liège - Institut Montefiore)
This experiment sends ICMP probes from EdgeNet nodes to perform internet topology discovery.

Darknet Watch (University College Dublin)
This bandwidth-intensive experiment conducts measurements on the I2P anonymous network.

NDT Client (M-Lab)
EdgeNet supports continuous measurements by the M-Lab NDT (Network Diagnostic Tool) client that measures download and upload speeds [92].

Murakami (M-Lab)
Being a tool for automating Internet measurements, Murakami supports NDT, Neubot DASH, Ookla's speedtest-cli, and OONI Probe [91]. EdgeNet supports continuous measurements by Murakami.

RIPE Atlas (RIPE)
RIPE Atlas, as an Internet measurement network, provides real-time insight into the state of the IP Layer across the globe [142]. RIPE Atlas network measurement software is run on EdgeNet nodes, adding to their collection of software nodes.

The EdgeNet cluster, as it stands in April 2023, includes a resource pool of 130 vCPUs and 248 GB of memory, with 51 registered tenants representing research institutions, groups, and teams, as well as individual researchers. Since its start, EdgeNet has supported more than 10 experiments and up to 7 parallel experiments. Several class exercises have been done through EdgeNet as well. Scaling the system is ongoing work: currently, if a new experiment requires a node that does not have enough available resources to handle a new experiment, the system does not deploy the experiment on that node. In future work, we plan to fix this limitation by automatically instantiating a new EdgeNet node on the overloaded site (if resources are available). The node contribution procedure (Sec. 5.3) that automates the deployment of a new node goes in that direction.

5.6 Benchmarks

EdgeNet is a global Kubernetes cluster with nodes all over the world communicating over the internet. This differs from the classic Kubernetes use case with nodes located in well-interconnected data centers. When we started work on EdgeNet, it was not at all clear that Kubernetes would be able to support such a highly distributed cluster.

Kubernetes is designed such that there is a brief delay between its control plane nodes and worker nodes within a cluster. Worker nodes are typically expected to have more resources than are in EdgeNet. EdgeNet, which has a resource-constrained, geographically dispersed cluster where RTT between control plane nodes and a worker node reaches 215 ms, puts Kubernetes' capabilities to the test. In the event of Kubernetes malfunctioning under these conditions, we would have needed to alter its core code rather than only extend it. Our study

has revealed that Kubernetes can operate properly in these circumstances, as evidenced by the fact that EdgeNet has been stressed and tested by various experiments introduced in Sec. 5.5, which were carried out successfully. Sec. 5.8 also demonstrates that such a geo-distributed cluster can be used to build a CDN, which is a specific type of distributed system.

In this section, we study how the cluster performs in terms of deploying containers and networking. The measurements described here were conducted in 2021.

5.6.1 Time to deploy an experiment

We measured the time necessary to schedule and run pods using selective deployments in 5 use cases: 1 pod anywhere in the European Union, 5 pods anywhere in the United States, 20 pods anywhere in the United States, and 1 and 20 pods in a polygon with 18 vertices provided in the GeoJSON format. We also simulated node failures by stopping the Kubernetes agent, *kubelet*, on them. The results are presented in Table5.1.

Table 5.1: Time in seconds to schedule and run pods on EdgeNet using selective deployments.

Selection	Creation Time			Node Failure Detection	Recovery Time	
	Selective Deployment	Daemon Set	Pod		Daemon Set	Pod
1 pod in the EU	0.42	3.9	3.4	36.7	4.6	3.5
5 pods in the US	0.19	10.1	10.5	45.5	79.9	79.2
20 pods in the US	0.47	60.6	79.6	39.8	8.2	8.3
1 pod in a polygon	0.18	6.7	6.2	41.1	8.3	8.3
20 pods in a polygon	0.33	57.2	56.8	37.4	9.8	9.8

Selective Deployment is the time to create a selective deployment object and select the nodes. *Daemon Set* is the time for the selective deployment to create the daemon set and its pods. *Pod* is the time between the creation of the first pod and having the last pod in the running state. *Node Failure Detection* is the time to detect a node failure. The last two columns are the time to reconfigure the daemon sets and to recreate the pods after the node failure.

We first see that creating a selective deployment, including the node selection, is done in a very short time, always under half a second. Thus, selecting nodes geographically incurs almost no overhead over the classical Kubernetes selector.

Next, when the number of pods is small, the time to create the daemon set and to get all the pods up and running is under 10 seconds. However, when 20 pods are requested, this time jumps up to 80 seconds. This is explained by the lack of resources on some nodes of the cluster. Pods typically are scheduled in under 10 seconds, but if the node where a pod is scheduled has a bottleneck of the resources in terms of CPU and memory, it spikes the time for the pod to be up and running.

The cluster can detect a node failure in under 45 seconds, and in all cases but one that we have tested, restart the pods on a new node in under 10 seconds. In the case of 5 pods in the US, it took 80 seconds to recover the pods. In that instance, the Kubernetes scheduler removed all pods from nodes and redeployed them even though only one node was changed and others remained the same in the selector. We assume that changing the selector, the node affinity, in the workload spec caused this unexpected behavior. Further investigation is needed to understand the underlying cause.

These findings raise questions about the ability of the default Kubernetes scheduler to meet globally distributed edge cloud requirements and indicates the need for an edge scheduler for edge-specific applications. A health-check mechanism between the control plane and worker nodes could also be put in place to reduce the time needed to detect a node failure.

5.6.2 Cluster network performance

At the time that experiments were conducted, EdgeNet was relying on the Calico network plugin to enable intra-cluster communications, that is, communications between the pods of the cluster. Calico establishes a private *pod network* on each node, and exchanges routes through full-mesh BGP peerings. Packets between the pod networks are sent over the internet using an IP-in-IP encapsulation. In this section we investigate the performance impact of this encapsulation in terms of throughput and RTT.

Throughput We used the standard iPerf3 tool to measure the throughput between nodes of the cluster and a node in Paris, and between nodes and a public server hosted by an ISP in France. This allows us to assess the impact of the IP-in-IP encapsulation on the network throughput. The results are presented in Fig. 5.2.

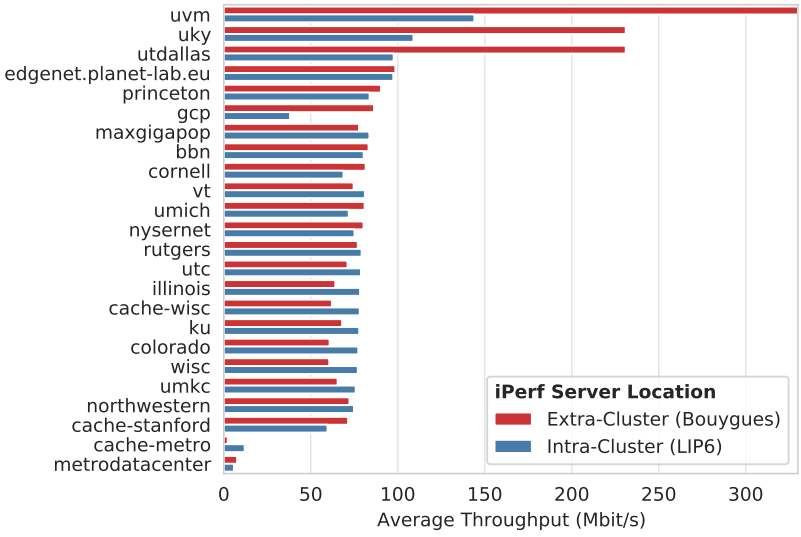


Figure 5.2: Average throughput between intra and extra-cluster targets. The throughput was measured over 10 seconds with iPerf 3. *Bouygues* is a 10Gbit/s iPerf server hosted by an ISP, and *LIP6* is an EdgeNet node, both located in Paris, France. Measurements towards the EdgeNet node are routed through the Calico IP-in-IP tunnel.

In almost every case, the intra and extra cluster throughput is similar. However, for the *uvm*, *uky*, and *utdallas* nodes the extra-cluster throughput is much higher than the intra-cluster one. Similarly, the *gcp* node located in the Google Cloud Platform, has significantly worse intra-cluster performance. Further investigations are needed to determine if this is due to congestion in the network, different IP routes for the two destinations, or to the IP-in-IP encapsulation.

Round-trip time Similarly to the throughput measurements, we measure the round-trip time between the nodes of the cluster and a node at our laboratory in Paris, France. The measurements are done towards the external IP of the node as well as towards the internal IP of the node. The measurements towards the external IP involve the packets being directly sent over the internet. On the other hand, the measurements toward the internal IP entail the packets being encapsulated before transmission. The results are presented in Fig. 5.3.

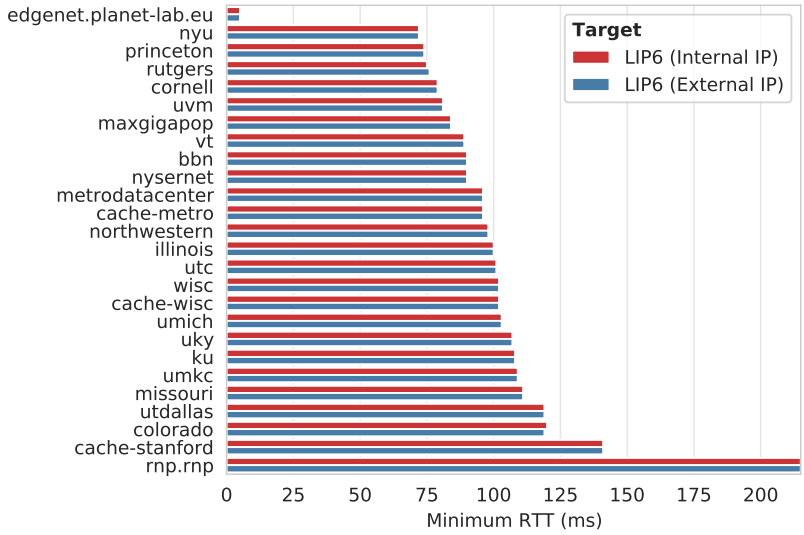


Figure 5.3: Minimum RTT between EdgeNet nodes and a node in Paris, France. The RTT was computed over 100 ICMP ping measurements. Pings towards the external IP are directly sent over the internet, while pings towards the internal IP are routed through the Calico IP-in-IP tunnel.

In all cases the minimum RTT is identical between the external and internal IP, indicating that the IP-in-IP encapsulation has no effect on the RTT. This makes EdgeNet suitable for research on computer networks, as the platform overhead is minimal. We have also done this measurement towards 3 nodes in the USA, and 1 node in Brazil. We observe no overhead for the nodes in the USA, but for the node in Brazil the intra-cluster delay is consistently longer by 4 milliseconds. Further investigation is needed to understand why this is the case for this node.

5.7 Observability

As the EdgeNet testbed is free for non-profit research, we employ a policy that provides complete transparency in such a way that observability tools are publicly accessible. Besides the cluster administration, this way, providers can monitor activity on their nodes by using these tools, and so do users for their experiments. There are two toolsets we run on EdgeNet: (1) Prometheus,[16] cAdvisor,[17] and Grafana[18] for system monitoring[19] and (2) VMware's Antrea

[16] Prometheus https://prometheus.io
[17] cAdvisor https://github.com/google/cadvisor
[18] Grafana https://grafana.com
[19] EdgeNet monitoring https://grafana.edge-net.org

for network flow visibility.[20] A dedicated node in the cluster, under EdgeNet's authority, stores the data gathered by these tools.

5.8 Experimenting within EdgeNet

With container orchestration capabilities at the edge, we will see design changes in the architectures of services and applications. These capabilities, for example, enable us to move services and applications across many locations to reduce their response time and optimize their bandwidth consumption. As an edge cloud testbed, EdgeNet is particularly suitable for putting these edge-adapted, new designs to the test. We believe that CDNs, in particular, may benefit from container orchestration at the edge, leading to architectural improvements for delivering large chunks of content more optimally.

We had the opportunity to participate in the testing of an experimental CDN framework on the EdgeNet testbed, not just by supporting the experiment as a testbed provider, but also by being directly involved in the performance evaluation of the CDN. In this subsection, we report on this work, which showcases EdgeNet's ability to support advanced experiments. For confidentiality reasons, we are not allowed to disclose the name of the experimental CDN.

The CDN framework that was tested on EdgeNet is designed around three agents: (1) a publisher that offers content to clients, (2) caches that serve the publisher's content to clients in exchange for payment, and (3) clients that consume content retrieved through caches. Two steps must be completed for successful content delivery. A client requests content from the publisher, and the publisher returns instructions that name the caches that the client must contact in order to retrieve content. As a security measure for untrusted nodes in the network, caches deliver data chunks to clients in an encrypted state, and those can be decrypted when a client has a preset number of chunks.

Below we first discuss how we configured deployment of the experimental CDN on EdgeNet (Sec.5.8.1) and how we prepared six different CDN setups (Sec.5.8.2) to be tested. We examine the extent to which EdgeNet is suitable for supporting proper testing of the CDN (Sec.5.8.3) and describe our findings regarding the CDN itself (Sec.5.8.4).

5.8.1 Deployment

The requirements of testing the experimental CDN led us to adjust the EdgeNet testbed's infrastructure. At the time of the experiments, most of the testbed's nodes had a dual-core processor and 2 GB of RAM. This amount of resources is adequate for running caches but not for running daemons. We included a node with a quad-core processor and 8 GB of RAM that hosts the main daemons as they consume more resources than an ordinary EdgeNet node would have. Once the namespace dedicated to experiments is created, the remaining step is developing the configuration files for the deployment.

[20] EdgeNet auditing https://audit.edge-net.org

To secure communication between the daemons, we used cert-manager[21] instead of managing certificates manually. As part of EdgeNet's design, which is based on Kubernetes, we benefit from several secrets and a config map[22] to keep the credentials and initial settings within the cluster. Additionally, a daemonset is used to create a required configuration file across the nodes being used in the experiments. Our selective deployment feature allowed us to straightforwardly deploy them on the desired nodes. The same strategy is applied to the cache deployments as well. Pods are put in communication within the cluster through the use of services besides the external IP addresses of the nodes.

To monitor and harvest data from the experiments on EdgeNet, we could choose from a number of available tools. The ones that we deployed can be grouped into two categories: (1) the operational logs and (2) the resource usage and performance metrics. Elasticsearch,[23] Filebeat,[24], Kibana,[25] and Jaeger[26] are employed to collect, explore, and visualize operational logs, whereas cAdvisor, Prometheus, and Grafana enable resource-based performance analysis. A script written in Golang automates data collection from these tools so measurement data can be published in spreadsheet format.

5.8.2 CDN setups

Below discusses which setups we prepared, where caches and daemons are deployed in those setups, and what resources are incorporated into the nodes that host these caches and daemons.

Our experiments aim to reveal the performance of the CDN framework regarding time to first byte served (TTFB) and content access duration (CAD) and analyze the factors on which these parameters depend. TTFB refers to the total amount of time it takes between a client issuing a request to a server and the moment that it receives the first byte of the data. In this CDN framework, TTFB has an added twist, as the client receives the data in an encrypted state and must perform a decryption operation. CAD can be described as the time between a client requesting content from a publisher and having it decrypted entirely.

There are six setups in which the caches are deployed in different locations while the daemons are run at our laboratory in Paris. In five setups, the caches run in one specific location, whereas the last one includes distributed geolocations. The geodiversity of the EdgeNet nodes contributes to the setup's ability to obtain metrics for real-world clients across different locations. These setups have been crafted separately as below:

- Paris case; the caches and daemons both running in Paris. Over 750 measurements are taken, benefiting from more than 30 vantage points.

[21] cert-manager https://cert-manager.io/

[22] ConfigMap is a native Kubernetes resource that allows storing configuration information to be consumed by pods.

[23] Elasticsearch https://www.elastic.co

[24] Filebeat https://www.elastic.co/beats/filebeat

[25] Kibana https://www.elastic.co/kibana/

[26] Jaeger https://www.jaegertracing.io

- Rio case; 5 caches are running in Rio de Janeiro, daemons running in Paris. Over 550 measurements are taken from over 30 vantage points.
- Wisconsin case; 5 caches running in the state of Wisconsin in the United States, daemons running in Paris. More than 650 measurements are taken from over 30 vantage points.
- Ohio case; 5 caches running in the state of Ohio in the United States, daemons running in Paris. Approximately 850 measurements are taken from more than 30 vantage points.
- Los Angeles case; 5 caches running in Los Angeles, daemons running in Paris. More than 700 measurements are taken from over 30 vantage points.
- Distributed case; one cache per location in Paris, Los Angeles, Ohio, Wisconsin, and Rio, daemons running in Paris. More than 900 measurements are taken from more than 30 vantage points.

In Table 5.2, we show which nodes are used for what purpose in these six cases and the resources of these nodes. In all cases, the experiments use a 5.52 MB video file.

In the next subsection, we examine the network performance of the EdgeNet nodes before looking at the metrics associated with the CDN framework.

5.8.3 Evaluation of the infrastructure being used

To begin with, we examine RTT and geographic distance between clients and caches to provide us with insight when evaluating TTFB and CAD in the CDN framework. An RTT value between a client and a cache is determined by taking the minimum value over 100 pings made from the client to the cache. Our reasoning for taking the minimum value is to minimize the negative impact of network congestions and delays on any network devices between these clients and caches. The results show that the RTT values of a group of nodes, which make up most of the cluster, range from 10 milliseconds to 60 milliseconds, as seen for the Wisconsin case in Table5.3.

In almost all cases, the results are identical for external and internal IP addresses; the intracluster networking shows slightly lower or higher RTTs for several cases. Such deviations may be caused by network congestion in intermediate routers. However, a delay of around 3 milliseconds on average constantly appears in the Rio case. There needs to be further investigation to find out why this is the case. Considering the results are identical for external and internal IP addresses in almost all cases, we think these findings indicate that it is possible to establish a distributed Kubernetes cluster with reliable cluster network performance as we have done with EdgeNet.

Physical proximity matters for achieving low RTT because installing direct fiber connections that form a mesh among all endpoints is not feasible, and each hop between a source and a destination introduces some overhead [125]. In analyzing geographic distance, we consider two parameters that can influence the outcome. First, we use MaxMind's GeoLite2 Database to programmatically find the latitude/longitude coordinates of the nodes in the cluster, even though the database can sometimes be misleading about the precise locations of the

Table 5.2: Information about the nodes that are used in the experiments, including their roles.

Node	Location	Client	Cache	CPU (Cores)	RAM (GB)
bbn-1	MA, US	●		2	2
cache-metro	OH, US	●	●	3	3
cache-stanford	CA, US	●		1	1
cache-ucla	CA, US	●	●	3	3
cache-wisc	WI, US	●	●	3	3
case-1	OH, US	●		2	2
colorado-1	CO, US	●		2	2
cornell-1	NY, US	●		2	2
edgenet.planet-lab.eu	IDF, FR	●		4	8
illinois-1	IL, US	●		2	2
ku-1	KS, US	●		2	2
lip6-lab.cache	IDF, FR	●		4	8
lip6-lab.ple-1	IDF, FR	●	●	2	6
lip6-lab.ple-2	IDF, FR	●		4	6
maxgigapop-1	MD, US	●		2	2
metrodatacenter-1	OH, US	●		2	2
missouri-1	MO, US	●		2	2
northwestern-1	IL, US	●		2	2
nps-1	CA, US	●		2	2
nysernet-1	NY, US	●		2	2
nyu-1	NY, US	●		2	2
princeton-1	NJ, US	●		2	2
rnp.rnp-1	RIO, BR	●	●	4	4
rutgers-1	NJ, US	●		2	2
uky-1	KY, US	●		2	2
umich-1	MI, US	●		2	2
umkc-1	MO, US	●		2	2
utc-1	TN, US	●		2	2
utdallas-1	TX, US	●		2	2
uvm-1	VT, US	●		2	2
vcu-1	VA, US	●		2	2
vt-1	VA, US	●		2	2
wisc-1	WI, US	●		2	2

All setups used the node *lip6-lab.cache* to run the daemons.
The nodes of *case-1* and *nps-1* were used in a few experiments.

Table 5.3: Minimum RTTs between clients and caches. The RTT was computed over 100 ICMP ping measurements. Pings towards the external IP are directly sent over the internet. Time in milliseconds.

Node	Paris	Rio	Wisconsin	Ohio	Los Angeles
bbn-1	90	149	60	29	84
cache-metro	96	148	40	0.088	55
cache-stanford	141	184	60	59	9
cache-ucla	140	166	49	58	0.148
cache-wisc	102	146	0.092	40	49
case-1	Undetermined	138	44	29	53
colorado-1	119	163	28	34	33
cornell-1	79	136	21	25	58
edgenet.planet-lab.eu	5	215	121	101	148
illinois-1	100	138	16	15	48
ku-1	108	143	17	45	34
lip6-lab.cache	0.294	215	102	96	140
lip6-lab.ple-1	0.044	215	102	96	139
lip6-lab.ple-2	0.309	215	102	96	140
maxgigapop-1	84	137	21	26	59
metrodatacenter-1	96	148	40	1	55
missouri-1	111	146	20	24	37
northwestern-1	98	135	20	12	44
nysernet-1	90	147	53	24	78
nyu-1	72	141	25	22	62
princeton-1	74	125	27	26	64
rnp.rnp-1	215	0.053	146	153	171
rutgers-1	76	142	24	25	62
uky-1	107	135	16	27	54
umich-1	103	146	12	19	49
umkc-1	109	146	17	32	34
utc-1	101	132	33	28	65
utdallas-1	119	142	28	35	40
uvm-1	81	138	35	31	65
vcu-1	97	129	40	28	72
vt-1	89	142	33	29	67
wisc-1	102	146	0.429	40	49

The *nps-1* node is excluded from the list due to its non-utilization in RTT-related analysis.

nodes. In the second place, we calculate the distance in kilometers between any two EdgeNet nodes by applying the Haversine formula, which assumes the world is a perfect sphere. The Haversine formula's impact on the results is less considerable than the first precise location issue. It is, therefore, important to consider geographic distance information is not totally reliable when interpreting Fig. 5.4.

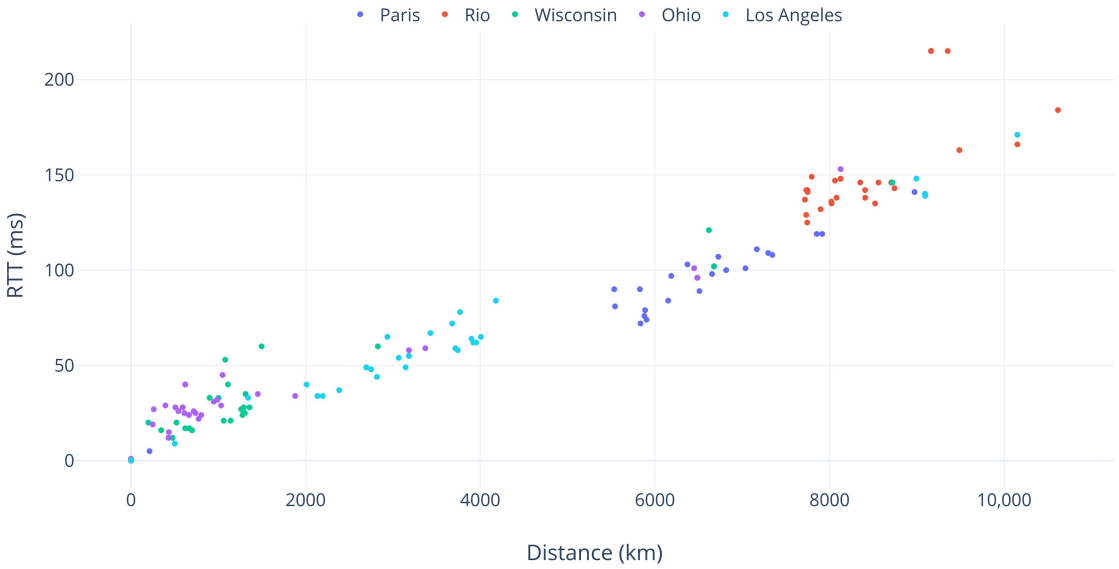


Figure 5.4: Scatter plot of geographic distance and network delay for five setups.

RTT outcome depends on many factors. During a measurement, congestion on routers between the source and destination may result in varied RTT values. Another key determinant is the transmission medium. Fig. 5.4 shows that RTT typically rises as the distance between the nodes increases substantially. Although we observe such a positive correlation, possibly due to the number and spatial distribution of EdgeNet nodes, outliers that can be seen in the Rio case are evident. RTT between the Paris nodes and the Rio node presents a notable spike even though the distance in kilometers between these two points is shorter than in a few other cases. We assume that the underlying reason for this behavior is the Internet backbone.

This scatter plot also provides insight into the geographic distribution of EdgeNet nodes; we can infer that many nodes are scattered around Ohio and Wisconsin from this plot, verified by Fig. 5.5. This map demonstrates the minimum RTT between EdgeNet nodes used in the experiments and a node in Wisconsin, US.

Based on our interpretation of the data, we verify that physical proximity matters when it comes to end-to-end latency. However, it is not the only deciding factor, as we discussed above. We do not use geographic distance as a criterion for any further comparison or analysis since it is unreliable for analyzing network performance. The following subsection investigates the correlation of RTT with TTFB and CAD.

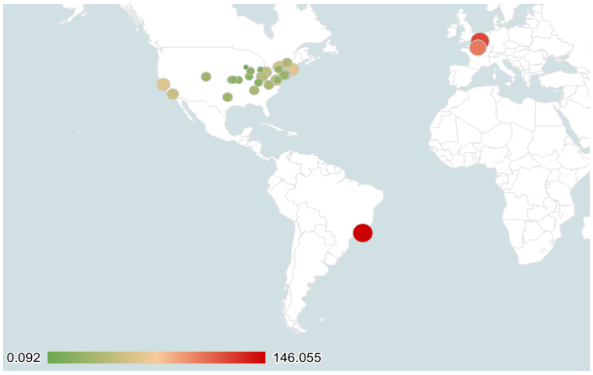


Figure 5.5: Map demonstrates minimum RTT between EdgeNet nodes used in the experiments and a node in Wisconsin, US.

5.8.4 Findings from the data

A consumer starts watching a video on their browser and experiences a delay. If the connection is not dramatically slow, this consumer is probably distant from the node hosting the content. There are some factors in this CDN framework, such as TTFB and CAD, which are crucial to ensure a better consumer experience. Assessing these factors via experiments that are conducted on a single machine or a cluster of nodes connected via LAN is unattainable. Powerful machines can mask the significance of free CPU resources for this framework, and LAN communication does not provide an accurate assessment of whether the framework can deliver low TTFB and CAD in a real-world deployment. An emulation setup could be established to appraise the framework's performance in a real-world deployment scenario, but this would be fraught with complexities and require a significant amount of work. Whereas, EdgeNet, with its geographically distributed, resource-constrained edge nodes, excels in addressing these limitations and allows the evaluation of this framework accurately. In the subsequent paragraphs, we describe the experiments we ourselves carried out on this CDN framework by using our own edge cloud testbed and showcase EdgeNet's function in uncovering its performance issues, primarily related to decryption operations.

Our anticipation was that the cryptographic operations in this CDN framework would introduce minimal overhead. However, our experiments in a real-world environment that EdgeNet delivers revealed that any CPU shortage can result in extended decryption times at clients, negatively affecting TTFB and CAD. The results demonstrate that the percentage of overall time consumed by decryption operations is higher when a client is closer to the caches, as compared to when the client is farther away. The framework can still provide a TTFB below 600 ms depending on cache geo-distribution, but decryption times undermine this promising approach's benefits. Thus, we assert that it requires more lightweight and efficient cryptography techniques. Below we discuss how we come to this conclusion.

As a starting point, we expect CAD to be typically lower in the distributed case than in other cases, as the distributed case has 5 caches in all locations that other cases separately cover. However, we confront an unexpected outcome showing that the distributed case presents longer CAD than some others, as seen Fig. 5.6. The underlying reason is that the publisher daemon does not choose caches that serve content to clients in such a way as to result in a shorter RTT. For example, it should have chosen the caches with lower RTT for the

client location of *edgenet.planet-lab*, but the selection includes the cache in Rio, resulting in higher CAD. The distributed case is not included in our further analysis for this reason. The following analysis also discerns the effect of the cryptography operations on TTFB and CAD by separating it from any network delays that may have influenced fluctuations in the datasets.

Figure 5.6: CAD comparison of four client locations for six setups.

We find that TTFB readings are typically shorter when clients are closer to the caches regarding RTT, as seen in Fig. 5.7. Shorter TTFBs result in consumers accessing content more quickly. From another standpoint, the more effectively the background tasks in the CDN framework work without causing major delays, the greater the correlation between RTT and TTFB. Although we can see a positive correlation between them, TTFB exhibits fluctuations in many cases. To comprehend if the decryption operations at the clients are the root cause, we subtract the time spent on decryption on the client side from TTFB, which is called *Time To Encrypted Data Chunks (TTEDC)*.

TTEDC measures the complete time it takes between a client making a request from the caches and the moment that it finishes receiving the data chunks in an encrypted state. In this case, no decryption operation is included. Since we exclude a parameter on which TTFB depends in the CDN framework, the correlation between RTT and TTEDC should be more assertive.

Fig. 5.8 portrays that TTEDC results in a shorter time than TTFB for all cases. We can assume that this time would decrease more if we remove the encryption process on the cache side. However, the correlation does not change dramatically even if fluctuations decrease for several cases. It creates the perception that the encryption and decryption operations do not cause a break in the correlation between RTT and TTFB as long as there is no overload of CPU resources of the nodes that run caches or of the clients. If there is a bottleneck in CPU resources for the client or the nodes that run caches, such notable variability occurs.

As we examined the data further to see the impact of decryption operations on CAD,

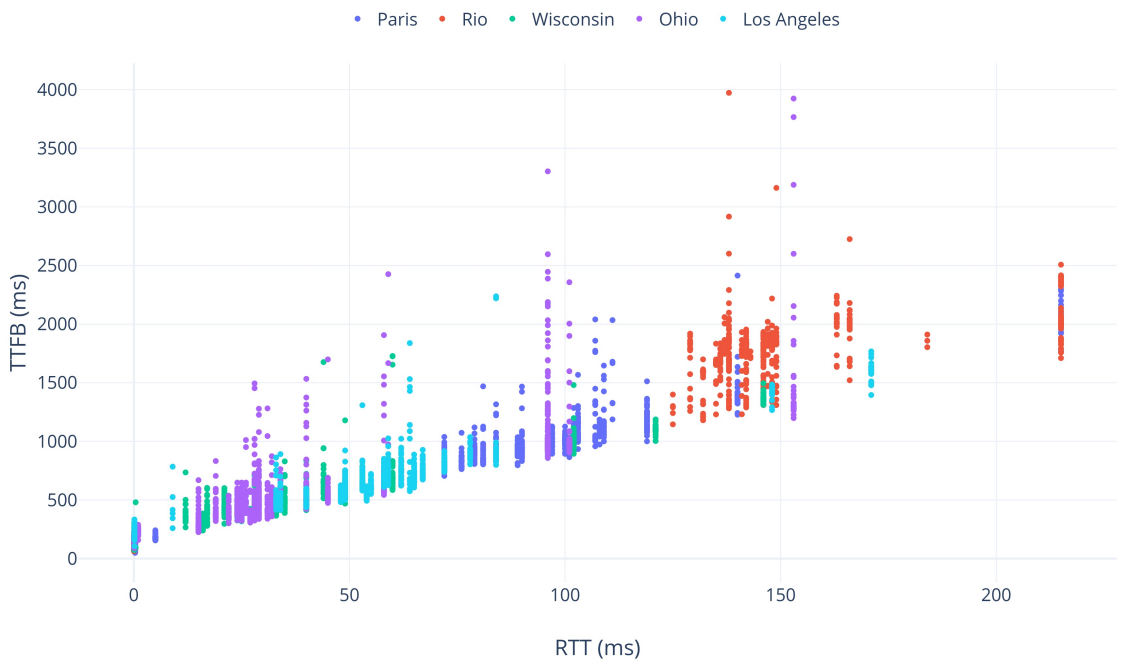


Figure 5.7: TTFB vs RTT for five setups. Minimum RTT between clients and caches.

we noticed an implementation issue. For small content deliveries, retrieving the first data chunks dramatically affects CAD. This implementation error adds a delay of approximately 850 milliseconds on average, which can reach 8 seconds. If a CAD is less than 5 seconds, then a delay of 850 milliseconds is considerable.

Say a consumer opened a tab on the browser, went to a video streaming platform, and picked a movie without any issue. Once the consumer pressed the play button, however, she had to wait for an extra second for the movie to start. If she tries to watch another movie, she would again have to wait for a second before watching. Given this waiting time can reach 8 seconds, it is a reasonable inference that this behavior is unacceptable.

The data presented in Fig. 5.9 illustrates the effect of this delay on CAD. Not only is the time shorter, but also lowered variability is observed compared to CAD. This is our expected result, but when we look at the correlation, it still appears to be weaker than between TTFB and RTT. This is likely due to two factors: (1) Clients must contact the publisher multiple times to receive instructions that allow retrieving all data chunks of content through selected caches, and (2) the number of cryptographic operations that must be done throughout this timespan.

On the one hand, our previous arguments have emphasized that decryption does not have a significant impact on the TTFB outcome as long as there are free CPU resources on the client and the nodes running the cache. On the other hand, we state it negates the correlation between TTFB and RTT if there is a shortage or inconsistency in CPU resources. When this occurs, the magnitude of the negative impact of the decryption operations is more severe for CAD. In the Ohio case, such a CPU shortage adds up to a delay of 3 seconds in CAD. Fig. 5.10 demonstrates a significant number of outliers even when the publisher delay is excluded from CAD. Considering the likelihood that network congestion contributes to these fluctuations, we conduct a deeper analysis of the effects of decryption operations below.

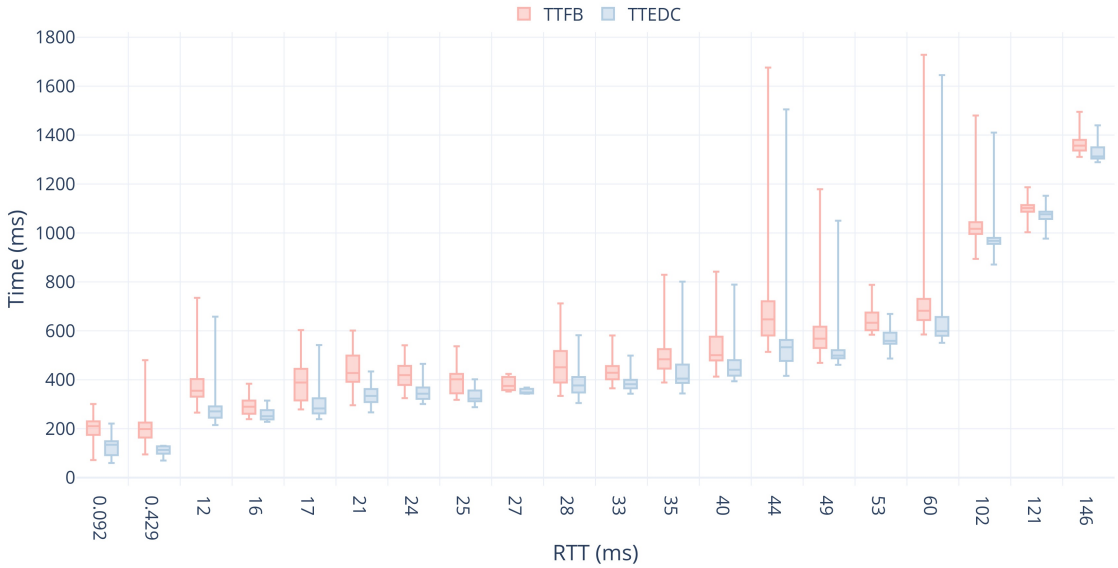


Figure 5.8: Comparison of TTFB and TTEDC for the Wisconsin case. Minimum RTT between clients and caches.

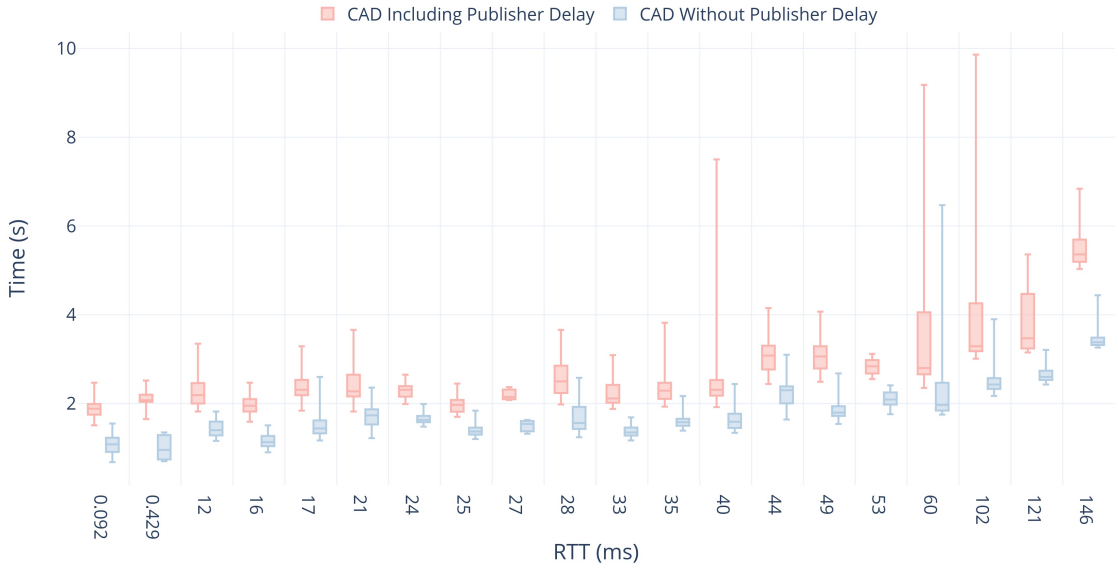


Figure 5.9: Wisconsin case, the influence of the publisher delay on CAD. The trace of CAD without publisher delay neglects the delay caused by an implementation issue. Minimum RTT between clients and caches.

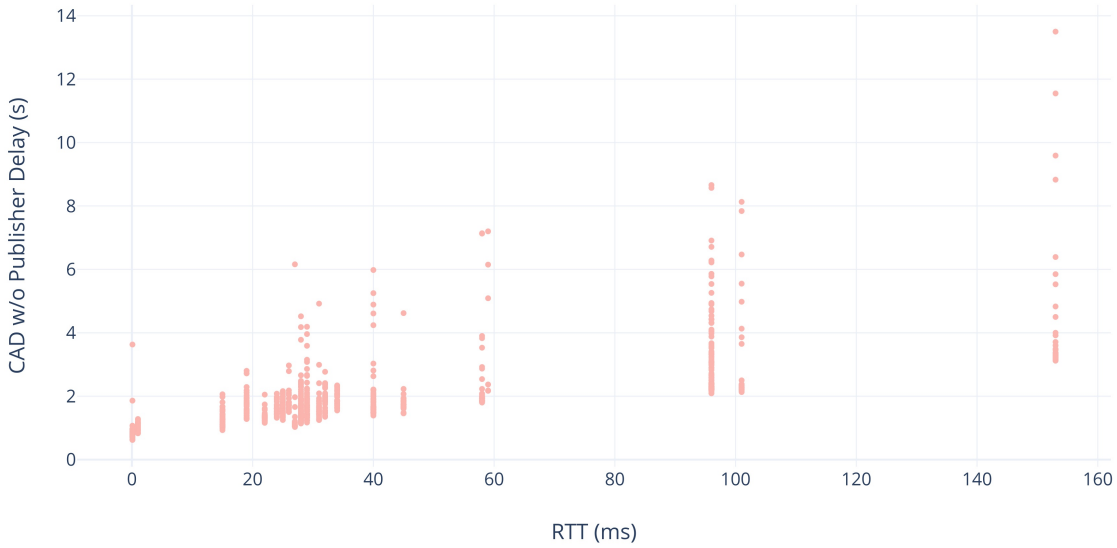


Figure 5.10: Ohio case, CAD without the publisher delay vs RTT. Minimum RTT between clients and caches.

It is important to know that there is no independent causality of RTT with TTFB and CAD in this CDN framework because of its cryptography techniques. TTFB is dependent on three factors: RTT, data chunk encryption in caches, and decryption operation time bonded to the client's free resources. CAD depends on these three factors plus on the publisher daemon preparing instructions for clients to retrieve content.

Our analysis above suggests that data encryption and decryption do not have to be an obstacle to achieving a TTFB below 600 ms. Fig. 5.11 highlights that the decryption operation requires a small amount of time in comparison to the total time. A decryption time of around 70 ms is acceptable where the TTFB is around 400 ms. The node in LA has a TTFB of 223 ms, the Stanford node's TTFB is lower than 500 ms, the node in Illinois has a TTFB of 552 ms, and the TTFB of the node in Paris is longer than a second. This bar chart shows that closer proximity to the caches results in a shorter TTFB for clients. However, the shorter TTFB is, the larger the percentage of time dedicated to decryption operations typically.

The question is, what is the underlying reason behind the high cryptography overhead in CAD when everything appears acceptable in TTFB? We need to provide some context regarding the design details of the CDN framework to provide a basis for an explanation. For retrieval of a 5.52 MB file, the publisher breaks this content into four bundles, each typically owning three handling operations. In turn, each handling operation generally contains four requests from caches so that the client receives four data chunks. Simple math says we typically have twelve data chunks per bundle. Each handling operation also includes a decryption process. As it requires four data chunks present at the client, there are three data decryption operations to complete a bundle, except for one. The final bundle has only two data decryption operations because it contains seven data chunks instead of twelve. In total, a client executes 11 decryption operations to have the entire content in a decrypted state.

Looking at the percentage of decryption cost in CAD is a better way to understand its effect rather than looking at the raw numbers. Fig. 5.12 compares the impact of decryption operations on TTFB and CAD. It is evident that decryption presents an aberrant behavior for CAD as it takes a significantly greater percentage of time compared to TTFB. Fig. 5.12a

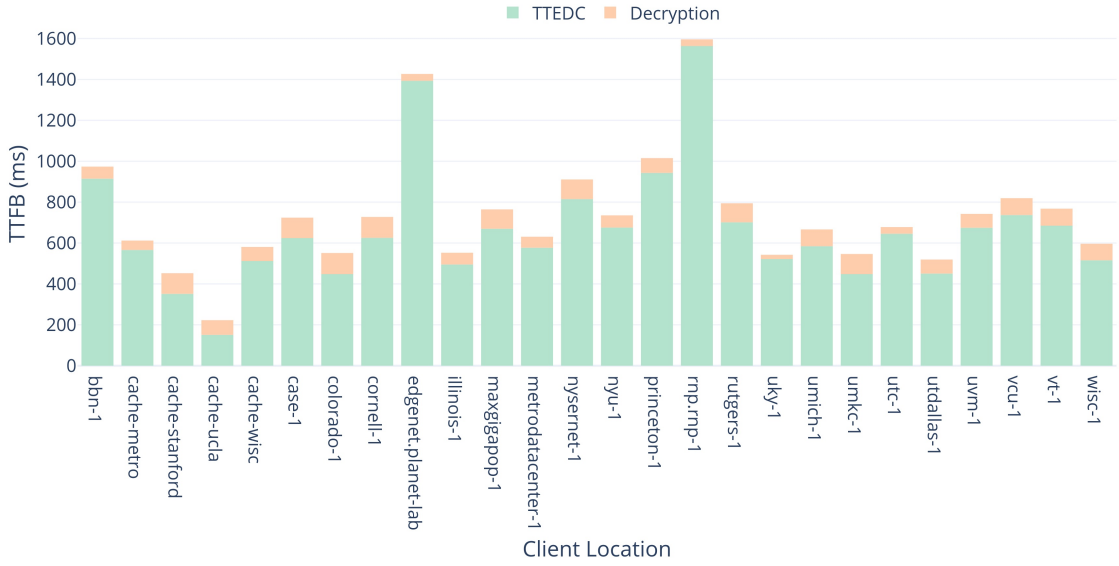


Figure 5.11: The implications of the decryption time on TTFB for the Los Angeles case.

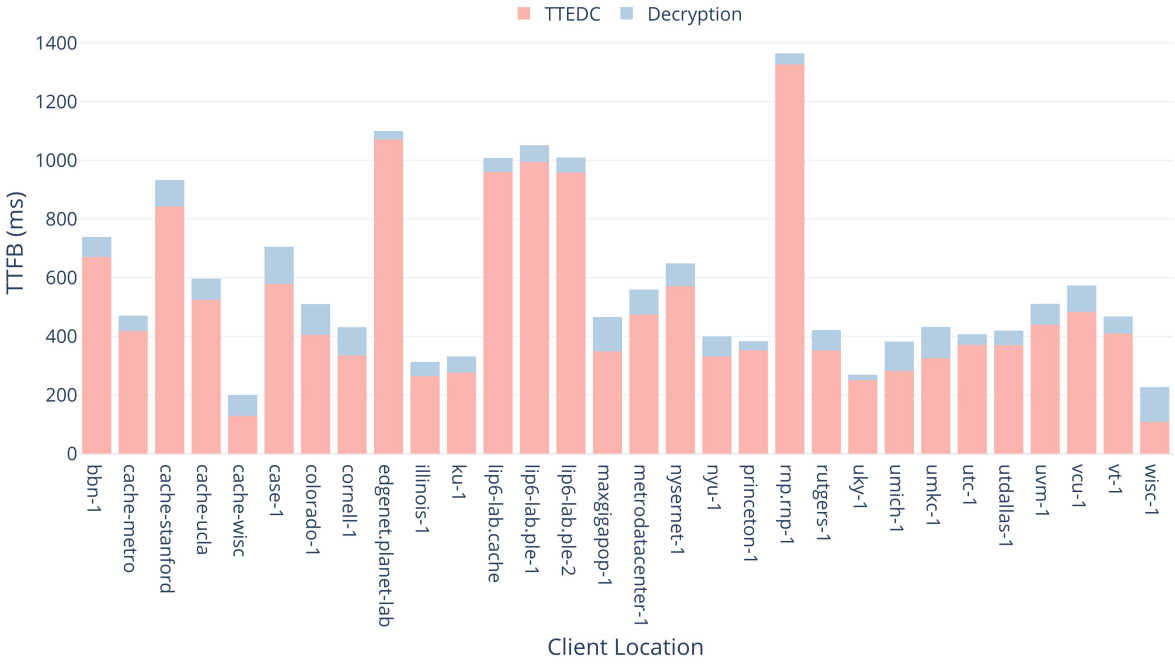
shows that clients dedicate a concise timeframe to decryption operations in TTFB, as do the clients in the Los Angeles case discussed above. By way of contrast, almost half of the time in CAD for several cases is spent on decryption, which nearly doubles the time, as seen in Fig. 5.12b.

To comprehend this unanticipated conduct, we investigate how much time is spent on each decryption operation. In Fig. 5.13, it is apparent that the time of the third decryption nearly quadruples the first one. A more severe occurrence exists in the Los Angeles case for the *maxgigapop-1* node; the first decryption time is 18 ms, while the slowest decryption is at 673 ms. We also observe that the mean time spent on the decryption operations following the first one is typically higher. This contributes to our interpretation of how pivotal free CPU resources at the client are for this CDN framework. It also explains why the decryption operations take a greater percentage in CAD than TTFB as well as shows that data decryption plays a role in the fluctuations that we observe in Fig. 5.10 besides network congestion.

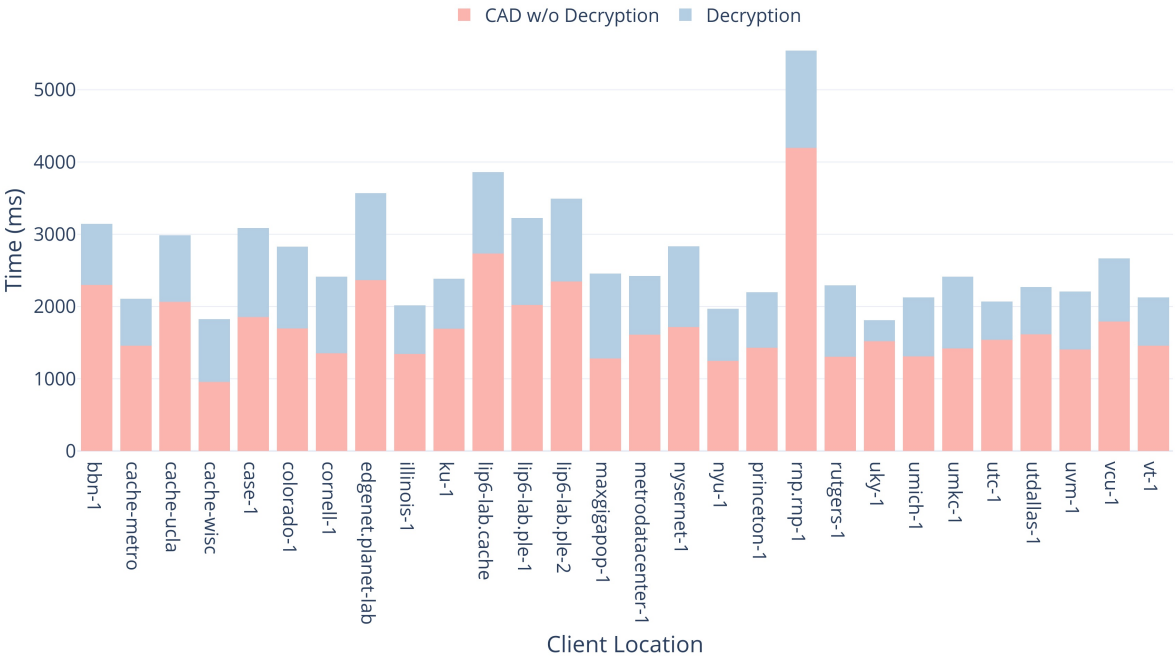
It is clear that the amount of decryption time needs to be reduced. We now look for solutions after discussing how the decryption step significantly slows down performance. TTFB can be improved by approximately in the range of 30 ms and 40 ms by decreasing the number of data chunks the client needs to start data decryption. Another option is reducing the requests clients make from caches to retrieve content, and lowering the number of decryption operations in a bundle can also shorten CAD. The level of security that is provided by cryptography techniques needs to be kept in mind when assessing these options.

Alternatively, it can be done by using client machines with more powerful CPU resources, as there is a correlation between decryption operations and CPU usage. Another possibility is putting caches closer to the end-user, resulting in shorter RTT. These all are future research questions to be tackled regarding this CDN framework. We examine below to what degree changing bundle design can shorten CAD and then conclude this work.

As mentioned before, there are four data chunks in a handling operation of a bundle, and these must be present at the client to start decryption. We thought of lowering the number



(a) The implications of the decryption time on TTFB.



(b) The implications of the decryption time on CAD.

Figure 5.12: Comparison of the effect of the decryption time on TTFB and CAD for the Wisconsin case.

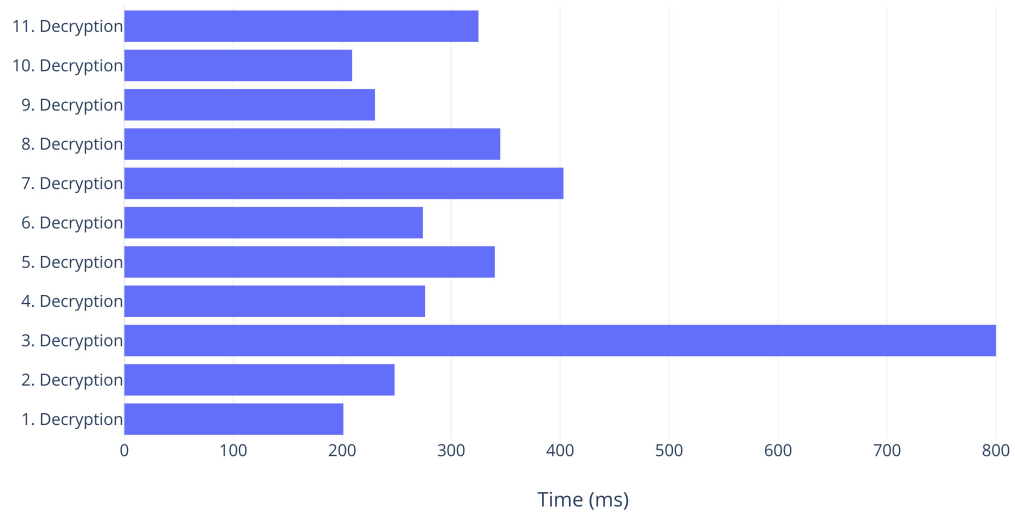


Figure 5.13: Comparison of time spent on each performed decryption during content retrieval for the Rio case. The bars represent the maximum time expensed on decryption at the client of *rutgers-1*.

of chunks to initiate the decryption operation in order to decrease time.[27] With this aim in mind, we measure the difference in time between the fastest and second-fastest data chunk retrievals as well as between the slowest and second-slowest ones.

In certain cases, longer time differences occur between the fastest chunks than the slowest ones, as seen in Fig. 5.14, while in other instances, vice versa. Our analysis points to no guiding pattern existing in this behavior. We assume either using the first three chunks or the last three chunks over four chunks does not make a significant difference in reducing time loss. However, based on data collected from clients at a close distance, we can see a pattern where differences in time are longer for the handling operations in the middle. This pattern may indicate the consistency in data chunk retrieval.

At the beginning and end of content retrieval, there are typically fewer differences in time, whether for the fastest chunks or the slowest chunks. But in the middle of the process, time differences almost doubled. Moreover, these time differences become inconsistent, which means they are variable, as noticed in Fig. 5.14. More interesting is that the same pattern, which is a bigger time difference range in the middle as opposed to the beginning and end of the content retrieval, occurs in data collected from some distant clients as well. The cause might be the encryption processes on the cache side, which requires further investigation.

Taken together, using three data chunks instead of four saves around 40 ms on average to start the decryption operation, thus shortening TTFB. Is it worth changing the bundle design? This issue needs to be addressed while considering its impact on security. Another question is, does it help us to significantly shorten CAD by decreasing the effect of decryption operations in total? A quick answer is no.

Fig.5.15 demonstrates that the decryption operations append a considerable time to CAD.

[27] While this approach might improve TTFB, it could also reduce the system's security. This area of inquiry needs further scrutiny.

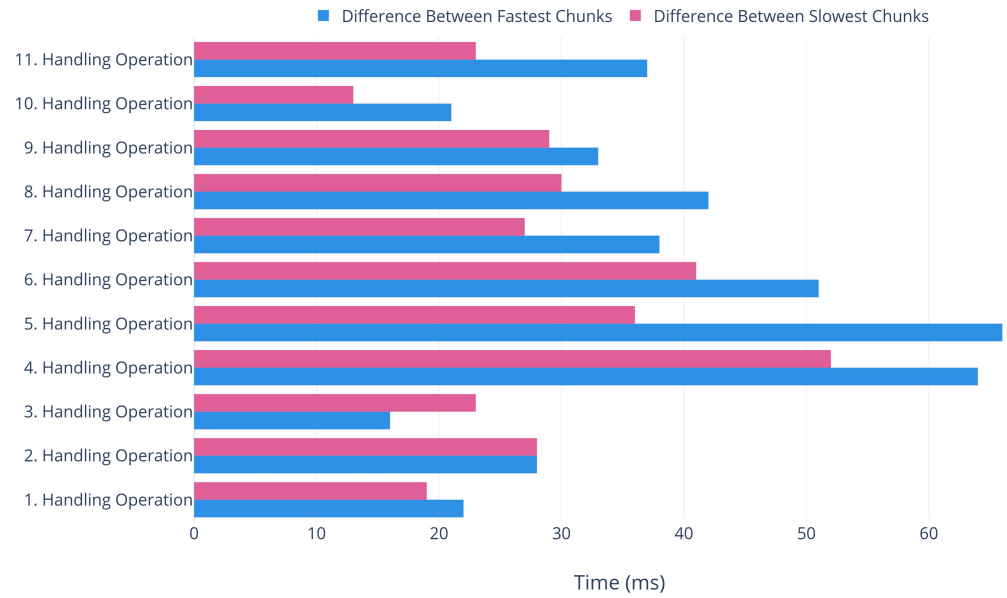


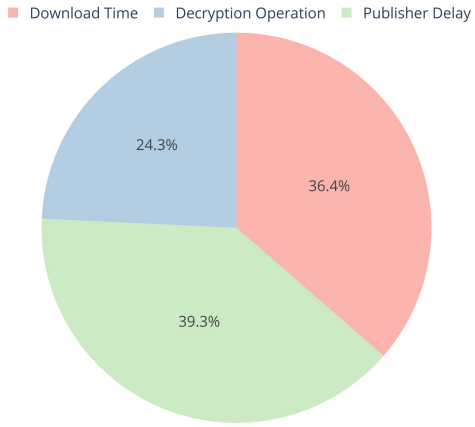
Figure 5.14: The difference in elapsed time between the fastest and second fastest data chunk retrievals, as well as between the slowest and second slowest ones. Los Angeles case, the client is in Illinois.

When the client is closer to caches, as is in Fig. 5.15b, the decryption operations occupy a larger proportion of the overall time than when the client is distant from caches, as shown in Fig. 5.15a. In light of current circumstances, we assert that a need for a new bundle design emerges for this CDN framework. We argue that rather than reducing the number of chunks needed to start decryption, having fewer total decryption operations is a better option. Let us assume a new bundle design for the same content at a 5.52 MB size. Each bundle has twelve data chunks, there are still three handling operations per bundle, and clients can start decryption with the presence of three chunks. In this particular scenario, instead of eleven, there will be four decryption operations, one per first handling operation in a bundle. Such an approach may reduce both TTFB and CAD, but it should be examined through the lens of security.

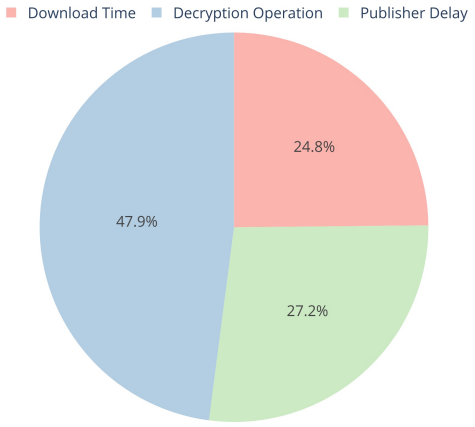
This CDN framework employs cryptography techniques to involve untrusted nodes to serve content. With this approach, this framework offers an inspiring design change as conventional CDN architecture is built upon trusted nodes. On the one hand, it can shorten TTFB and CAD since caches can be both many and located closer to consumers. On the other hand, this framework's current cryptography techniques introduce a notable overhead, which impairs these gains. We conclude that more lightweight cryptography mechanisms, which ensure encryption and decryption operations expend less time than it takes now, should be investigated to unlock this framework's full potential.

5.9 Conclusion and future work

In this chapter, we introduced EdgeNet, a multi-tenant and multi-provider edge cloud testbed based on Kubernetes. EdgeNet extends Kubernetes through custom resources and con-



(a) The client is *rnp.rnp-1*. Among the two, the minimum RTT between the client and the cache is longer, and the decryption operations take up a smaller portion of the overall time.



(b) The client is *maxgigapop-1*. Among the two, the minimum RTT between the client and the cache is shorter, and the decryption operations take up a larger portion of the overall time.

Figure 5.15: Time analysis to ascertain the percentage of CAD that each process consumed in the Wisconsin case. Download time includes the time spent for the publisher to send instructions as well as for caches to convey data chunks in an encrypted state to the client.

trollers, making it usable with nothing other than the standard Kubernetes tools. In its current state, the EdgeNet cluster offers reasonable performance with minimal overhead, and is suited to all kinds of experiments on networked systems. EdgeNet shows that a public Kubernetes cluster with nodes distributed over the world works.

We have introduced challenges associated with nodes at the edge that distributed testbed providers face: provision, access, and maintenance. Three contributions, home networks, node deployment, and federation, addressed these issues. A node agent configures a virtual private network, thus enabling nodes behind NAT boxes to participate in a cluster with the help of the native VPN peer controller in Kubernetes. A node can join a cluster in less than ten minutes via our node deployment procedure that includes installing the above-mentioned agent. Finally yet importantly, our aggregate manager (AM), which integrates EdgeNet into the European Commission financed Fed4FIRE+ federation of computer networking testbeds, shows that our testbed is capable of running in concert with other testbeds. In conclusion, these three features allow the testbed to reach out to a wider community and scale up the cluster, including edge nodes typically blocked by NAT boxes.

Several steps can be taken to improve EdgeNet, and more generally Kubernetes at the edge.

Remote maintenance. Individuals and institutions that contribute nodes to the clusters should be isolated from operational burdens to the extent possible. Aside from running a quick and easy node deployment script, the work required for maintaining a node should be minimal, so as to better motivate contributors to provide nodes. Sec.5.3.3 described our plan to tackle this issue.

Reproducibility. EdgeNet can improve its way of providing the reproducibility of experiments, which is a major challenge for computer science [123], by providing immutable infrastructure through node replacements instead of mutations [121]. We use Ansible for

node deployments, contributing to creating reproducible Kubernetes clusters [58]. However, our current approach does not follow immutable infrastructure principles as we mutate the nodes as requirements change. In addition to cloud nodes, every VM node that joins the cluster can be spawned using pre-built machine images. This can contribute to having reliable repeatability regarding the environment for experiments. We can base future work on EdgeNet's reproducibility strategy on a research study that tackled the reproducibility of environments in distributed systems through a tool called NixOS [59].

Edge-specific scheduler. An edge-specific scheduler for Kubernetes could be designed so as to minimize the deployment times and to better take into account the limitations of each edge node.

IPv4/IPv6 dual-stack. The EdgeNet cluster currently provides support for allocating only IPv4 addresses to pods and services. We will enable dual-stack networking to support both IPv4 and IPv6 allocations.[28]

DNS cache. EdgeNet currently runs in-cluster DNS services according to the regional distributions of its nodes. We will employ the DNS cache feature of Kubernetes and conduct measurements on it to assess its performance for a cluster with geographically distributed nodes.[29]

[28] Kubernetes documentation: *IPv4/IPv6 dual-stack* https://kubernetes.io/docs/concepts/services-networking/dual-stack/

[29] Kubernetes documentation: *Using NodeLocal DNSCache* https://kubernetes.io/docs/tasks/administer-cluster/nodelocaldns/

Chapter 6

Conclusion

Presented in this thesis are our contributions to container orchestration for the edge cloud. These are organized around three fundamental elements: a native multitenancy, a federation that spreads by local action, and an edge cloud testbed. Sec. 6.1 summarizes the contributions, and Sec. 6.2 introduces the perspectives looking forward from this thesis.

6.1 Summary of contributions

The field of container orchestration for the edge cloud is enriched by the work that has been done throughout this thesis. At the outset, we have investigated cloud multitenancy and framed CaaS multitenancy frameworks into single-instance and multi-instance approaches. The findings of these earlier studies on cloud multitenancy, accompanied by the resource constraints in edge clouds, have brought us to choose the single-instance approach for our multitenancy framework to introduce low overhead. The single-instance multitenancy framework that we have developed improves the state-of-the-art via new mechanisms that activate vendor mode, establish variable slice granularity, set tenant resource quota for hierarchical namespaces, and enable clusters to accept federation workloads from remote clusters without collisions. Through empirical investigation and analysis, we have revealed the strengths and weaknesses of the single-instance and multi-instance approaches as well as demonstrated that single-instance is more lightweight and is faster in creating tenants and pods, which has important implications for enabling multitenancy in edge clouds. Pertaining to our framework, being of a lightweight build enables a resource-constrained edge cloud to accommodate many tenants with minimal resource usage, and being time-saving to create tenants and pods improves workload mobility across edge clouds.

Accounting for the heterogeneous essence of edge computing, we have developed an integrated federation strategy for CaaS that allows multiple providers to offer compute resources in the forms of node, cluster, and system to a federation. This strategy, which is more comprehensive than existing state-of-the-art approaches, also serves as a means to prioritize local action to form a federation. Taken together, it can ensure a federated infrastructure at scale and facilitates the maintenance of such infrastructure through physical operations by

virtue of a shared workload between the providers. We have developed a toolset to enact our federation strategy in which the proposed federation architectures work in concert with our multitenancy framework, thus benefiting from its advantages of being lightweight and fast.

Last but not least, we have combined our multitenancy framework with our federation architectures that are node-wise and system-wise to enable an edge cloud testbed for Internet researchers, on which researchers have conducted more than 10 experiments. We have further developed innovative features, such as location-based node selection, that facilitate the use of the testbed by researchers. The EdgeNet testbed removes the dependency on dedicated hardware and custom control frameworks that its precedents suffered from, lowering maintenance costs and supporting the sustainability of an edge cloud testbed.

Overall, by developing a new vision, a novel multitenancy framework, a new federation strategy, a new federation toolset, new procedures, mechanisms, and algorithms, by investigating multitenancy approaches, and by enabling an edge cloud testbed, this thesis has contributed to the field of container orchestration for the edge cloud. All our code is publicly available, and our framework, tools, and systems are free for all to use.

6.2 Perspectives

We conclude our dissertation on *"Container Orchestration for the Edge Cloud"* with the perspective section that reports the state of the field after this thesis and how we envision it heading in the future.

6.2.1 CaaS at the edge

Prior to this thesis, limited standardization with respect to the scientific literature on cloud multitenancy was observed in classifications of multitenancy frameworks in the context of CaaS. There already existed an understanding of the fundamental features a CaaS multitenancy framework should offer, but no study was conducted to characterize these features for CaaS to thrive in clouds and edge clouds considering envisaged edge computing infrastructure. This thesis presented a novel classification of these frameworks by analyzing the scientific literature on cloud multitenancy, with a set of fundamental features that are conceived for upcoming edge cloud infrastructure and are consolidated in a single instance multitenancy framework that we developed. Even though there was already a common understanding of which kind of frameworks introduce low or high overhead, this thesis has benchmarked three multitenancy approaches and revealed their pros and cons from a tenancy-centered edge computing standpoint.

As for CaaS federation, the community around Kubernetes has already been offering several solutions that are typically based upon a centralized federation control plane or a Virtual Kubelet abstraction. The ones using the centralized federation control plane method establish a federation at the level of clusters, whereas Virtual Kubelet-based ones can launch a federation in connection with both clusters and systems. This thesis has suggested an integrated federation strategy, which benefits from the lightweight build of our multitenancy

framework, in which providers can offer compute resources in the shape of nodes, clusters, or systems. Node-wise federation allows small-sized providers to contribute to infrastructure, cluster-wise federation introduces a new method based on lightweight federation managers that spread deployments across federated clusters without taking control of worker clusters, and system-wise federation interconnects different systems.

We envision the future of edge computing infrastructure to be geographically widely distributed and offered by multiple providers and accordingly provide a well-defined and inspiring vision that presents a future in which CaaS thrives to contribute to the economic feasibility of the edge computing paradigm. By utilizing our multitenancy framework, many tenants can share a single cluster, which can support up to 10,000 tenants, as well as federated resources, and our federation strategy ensures infrastructure scalability. By this means, customers that tolerate less than perfect isolation from others can make use of geographically dispersed resources at lower prices. Regarding providers that offer compute resources to the edge infrastructure, they can ensure high resource utilization of these resources with this inclusive approach.

There is still plenty of room for improvement. First, although we choose the Kata runtime to isolate multi-tenant workloads, our multitenancy framework can be enhanced by new techniques to discover underlying hardware in order to select the best-fitting container runtime, which can be gVisor for some scenarios. Second, such a discovery mechanism can also improve the slicing mechanism that an isolation daemon will complement to make sure of complete isolation in terms of processes and networking. Third, the integrated federation strategy cannot be based on trust between parties in the context of a commercial partnership. In order to prevent malicious actors from tampering with federation data or ensure agreements between parties are being satisfied, it requires putting relevant trust mechanisms in place, which can benefit from smart contracts. Next, providers adopting a single instance multitenancy framework imply that they manage the infrastructure on behalf of customers. This may not be suitable for all customers, but providers can offer clusters with different versions to appeal to a broader customer base. In this way, the multi-tenant CaaS would support the provisioning of tenants and their migration onto specific cluster versions based on their preferences. Furthermore, tenants can make deployment choices based on cluster version in a federation. Lastly but not least important, regarding the workloads that need to move as a function of client/end-user requests, we believe the choice of locations in which containers to be run must be abstracted from customers if they desire. A scheduling study is needed to tackle this aspect; it can extract locations that requests come from and accordingly move or replicate workloads at available locations across the federation in accordance with users' budgets.

6.2.2 Edge cloud testbed

Prior to this thesis, the networking and distributed systems research communities have provided geographically distributed testbeds that have been particularly successful in the past decades. However, the requirement for dedicated hardware and bespoke software development hampered the efficiency and sustainability of these testbeds. Our edge cloud testbed embraces an alternative philosophy that enables providers to contribute with VM nodes in-

stead of requiring dedicated hardware and steers the software development in a way that researchers just implement a set of extensions to Kubernetes, a de facto industry standard container orchestration tool. Our multi-tenant and multi-provider edge cloud testbed, with globally distributed nodes, ensures an infrastructure at scale with lower hardware and maintenance costs as well as allows many researchers to make use of this infrastructure.

We believe, in addition to the current EdgeNet testbed, there are two more ways forward to enable different types of edge cloud testbeds while upholding a similar philosophy. The first case employs the same node deployment model as does the EdgeNet testbed, geographically dispersed nodes, but only deploys nodes to home networks. As individuals are more vulnerable than organizations in case they go under investigation due to malicious traffic on their home networks, such a testbed will only be open to vetted experiments and be managed by the testbed administrators. We have created a cluster to launch such a testbed but have yet to scale it up to a large number of nodes. The focus of the second case lies in leveraging the existing clusters of universities and research centers. Universities and research centers typically have their own Kubernetes clusters for their researchers to conduct experiments. Except for the periods of measurement campaigns, the resources in these clusters are generally in an idle state. These clusters can be federated for better use of these resources devoted to research, forming a global resource pool for not-for-profit researchers. Cluster-wise federation described in Sec. 4.2 is conceived, including sharing preferences, with this perspective in mind.

Bibliography

[1] Ayesha Abdul Majeed et al. "Performance Estimation of Container-Based Cloud-to-Fog Offloading". In: *Proc. UCC Companion*. 2019. DOI: 10.1145/3368235.3368847.

[2] Vaibhav Aggarwal and B Thangaraju. "Performance Analysis of Virtualisation Technologies in NFV and Edge Deployments". In: *Proc. CONECCT*. 2020. DOI: 10.1109/CONECCT50063.2020.9198367.

[3] Usama Ahmed, Imran Raza, and Syed Asad Hussain. "Trust Evaluation in Cross-Cloud Federation: Survey and Requirement Analysis". In: *ACM Computing Surveys* 52.1 (2019). DOI: 10.1145/3292499.

[4] G. Almashaqbeh. "CacheCash: A Cryptocurrency-based Decentralized Content Delivery Network". PhD thesis. Columbia University, 2019.

[5] Ganesh Ananthanarayanan et al. "Real-Time Video Analytics: The Killer App for Edge Computing". In: *Computer* 50.10 (2017), 58–67. DOI: 10.1109/MC.2017.3641638.

[6] Hallvard Andersen et al. "NATO Federated Coalition Cloud with Kubernetes: A National Prototype Perspective". In: *26th International Command and Control Research and Technology Symposium (ICCRTS)*. 2021. URL: https://ffi-publikasjoner.archive.knowledgearc.net/handle/20.500.12242/2991.

[7] Atakan Aral et al. "Addressing application latency requirements through edge scheduling". In: *Journal of Grid Computing* 17.4 (2019), pp. 677–698. DOI: 10.1007/s10723-019-09493-z.

[8] William Arms. *Early Timesharing*. 2015. URL: https://www.cs.cornell.edu/wya/AcademicComputing/text/earlytimesharing.html.

[9] Marcio Roberto Miranda Assis and Luiz Fernando Bittencourt. "MultiCloud Tournament: A cloud federation approach to prevent Free-Riders by encouraging resource sharing". In: *Journal of Network and Computer Applications* 166 (2020), p. 102694. DOI: 10.1016/j.jnca.2020.102694.

[10] M.R.M. Assis and L.F. Bittencourt. "A survey on cloud federation architectures: Identifying functional and non-functional properties". In: *Journal of Network and Computer Applications* 72 (2016), pp. 51–71. DOI: 10.1016/j.jnca.2016.06.014.

[11] A. Bavier, R. McGeer, and G. Ricart. "PlanetIgnite: A Self-Assembling, Lightweight, Infrastructure-as-a-Service Edge Cloud". In: *Proc. ITC*. 2016. DOI: 10.1109/ITC-28.2016.125.

[12] Alexander Benlian and Thomas Hess. "Opportunities and risks of software-as-a-service: Findings from a survey of IT executives". In: *Decision support systems* 52.1 (2011), pp. 232–246. DOI: 10.1016/j.dss.2011.07.007.

[13] David Bernstein. "Containers and cloud: From LXC to Docker to Kubernetes". In: *IEEE Cloud Computing* 1.3 (2014), pp. 81–84. DOI: 10.1109/MCC.2014.51.

[14] Ngo Huy Bien and Tran Dan Thu. "Hierarchical multi-tenant pattern". In: *Proc. ComManTel*. 2014. DOI: 10.1109/ComManTel.2014.6825597.

[15] Robert Bohn, Craig Lee, and Martial Michel. *The NIST Cloud Federation Reference Architecture*. Special Publication. NIST, 2020. DOI: 10.6028/NIST.SP.500-332.

[16] Irena Bojanova, Jia Zhang, and Jeffrey Voas. "Cloud Computing". In: *IT Professional* 15.2 (2013), pp. 12–14. DOI: 10.1109/MITP.2013.26.

[17] Raphaël Bolze et al. "Grid'5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed". In: *IJHPCA* 20.4 (2006), pp. 481–494. DOI: 10.1177/1094342006070078.

[18] Flavio Bonomi et al. "Fog Computing and Its Role in the Internet of Things". In: *Proc. MCC*. 2012. DOI: 10.1145/2342509.2342513.

[19] Brendan Burns et al. "Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade". In: *Queue* 14.1 (2016), pp. 70–93. DOI: 10.1145/2898442.2898444.

[20] Rajkumar Buyya and Karthik Sukumar. "Platforms for Building and Deploying Applications for Cloud Computing". In: (2011). DOI: 10.48550/ARXIV.1104.4379.

[21] Martin Campbell-Kelly. "Historical reflections the rise, fall, and resurrection of software as a service". In: *Communications of the ACM* 52.5 (2009), pp. 28–30. DOI: 10.1145/1506409.1506419.

[22] Centaurus Infrastructure Project. *Arktos*. software: v1.0 of March 2022. Owner: LF Projects, LLC. Website: https://www.centauruscloud.io/. 2020. URL: https://github.com/CentaurusInfra/arktos.

[23] Ronan-Alexandre Cherrueau et al. "Edge Computing Resource Management System: a Critical Building Block! Initiating the debate via OpenStack". In: *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*. 2018. URL: https://www.usenix.org/conference/hotedge18/presentation/cherrueau.

[24] Mung Chiang and Tao Zhang. "Fog and IoT: An Overview of Research Opportunities". In: *IEEE Internet of Things Journal* 3.6 (2016), pp. 854–864. DOI: 10.1109/JIOT.2016.2584538.

[25] Frederick Chong, Gianpaolo Carraro, and Roger Wolter. *Multi-tenant data architecture*. MSDN article. Microsoft, 2006. URL: https://pdfs.semanticscholar.org/9a0c/3466190d83fb5fbc18730413b232f3fdae06.pdf.

[26] Andrew Chung, Jun Woo Park, and Gregory R. Ganger. "Stratus: Cost-Aware Container Scheduling in the Public Cloud". In: *Proc. SoCC*. 2018. DOI: 10.1145/3267809.3267819.

[27] Clastix Labs. *Capsule*. software: v0.3.1 of March 2023. Website: https://capsule.clastix.io/. 2021. URL: https://github.com/clastix/capsule.

[28] Clastix Labs. *Kamaji*. software: v0.2.1 of February 2023. Website: https://kamaji.clastix.io/. 2022. URL: https://github.com/clastix/kamaji.

[29] McKinsey & Company. *Cloud's trillion-dollar prize is up for grabs*. 2021. URL: https://www.mckinsey.com/capabilities/mckinsey-digital/our-insights/clouds-trillion-dollar-prize-is-up-for-grabs.

[30] McKinsey & Company. *New demand, new markets: What edge computing means for hardware companies*. 2018. URL: https://www.mckinsey.com/industries/technology-media-and-telecommunications/our-insights/new-demand-new-markets-what-edge-computing-means-for-hardware-companies.

[31] IDC Corporate. *New IDC Spending Guide Forecasts Double-Digit Growth for Investments in Edge Computing*. 2022. URL: https://www.idc.com/getdoc.jsp?containerId=prUS48772522.

[32] Datadog. *9 Insights on Real-World Container Use*. Presentation: https://www.datadoghq.com/zip/container-report-2022-ppt.zip. 2022. URL: https://www.datadoghq.com/container-report/.

[33] Wesam Dawoud, Ibrahim Takouna, and Christoph Meinel. "Infrastructure as a service security: Challenges and solutions". In: *Proc. INFOS*. 2010. URL: https://ieeexplore.ieee.org/abstract/document/5461732.

[34] Piet Demeester et al. "Fed4FIRE: The largest federation of testbeds in Europe". In: *Building the future internet through FIRE*. River Publishers, 2016.

[35] *Description of Network Slicing Concept*. Requirements and Architecture team deliverable. NGMN Alliance, 2016. URL: https://ngmn.org/wp-content/uploads/160113_NGMN_Network_Slicing_v1_0.pdf.

[36] Aaron Yi Ding and Marijn Janssen. "Opportunities for Applications Using 5G Networks: Requirements, Challenges, and Outlook". In: *Proc. ICTRS*. 2018. DOI: 10.1145/3278161.3278166.

[37] Chuntao Ding et al. "A Cloud-Edge Collaboration Framework for Cognitive Service". In: *IEEE Transactions on Cloud Computing* 10.3 (2022), pp. 1489–1499. DOI: 10.1109/TCC.2020.2997008.

[38] Jason A. Donenfeld. "WireGuard: Next Generation Kernel Network Tunnel". In: *Proceedings Network and Distributed System Security Symposium*. 2017. DOI: 10.14722/ndss.2017.23160.

[39] Shreya Mukesh Dubey. *Design of User Management and security Tools for Centaurus*. preprint under review. 2022. DOI: 10.21203/rs.3.rs-1517240/v1.

[40] C. Dupont, R. Giaffreda, and L. Capra. "Edge computing in IoT context: Horizontal and vertical Linux container migration". In: *Proc. GIoTS*. 2017. DOI: 10.1109/GIOTS.2017.8016218.

[41] J. Roberto Evaristo, Kevin C. Desouza, and Kevin Hollister. "Centralization Momentum: The Pendulum Swings Back Again". In: *Communications of the ACM* 48.2 (2005), 66–71. DOI: 10.1145/1042091.1042092.

[42] Francescomaria Faticanti et al. “An Application of Kubernetes Cluster Federation in Fog Computing”. In: *24th Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*. 2021. DOI: 10.1109/ICIN51074.2021.9385548.

[43] S. Fdida, T. Friedman, and T. Parmentelat. “OneLab: An Open Federated Facility for Experimentally Driven Future Internet Research”. In: *New Network Architectures: The Path to the Future Internet*. Ed. by Tania Tronco. Vol. 297. Studies in Computational Intelligence. Springer, 2010, pp. 141–152. DOI: 10.1007/978-3-642-13247-6.

[44] Wes Felter et al. “An updated performance comparison of virtual machines and Linux containers”. In: *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2015. DOI: 10.1109/ISPASS.2015.7095802.

[45] Jingyun Feng et al. “AVE: Autonomous Vehicular Edge Computing Framework with ACO-Based Scheduling”. In: *IEEE Transactions on Vehicular Technology* 66.12 (2017), pp. 10660–10675. DOI: 10.1109/TVT.2017.2714704.

[46] Nicolas Ferry et al. “CloudMF: Model-Driven Management of Multi-Cloud Applications”. In: *ACM Transactions on Internet Technology* 18.2 (2018). DOI: 10.1145/3125621.

[47] Nicolas Ferry et al. “Towards Model-Driven Provisioning, Deployment, Monitoring, and Adaptation of Multi-cloud Systems”. In: *IEEE Sixth International Conference on Cloud Computing*. 2013. DOI: 10.1109/CLOUD.2013.133.

[48] D. Rex Finley. *Point-in-Polygon Algorithm—Determining Whether a Point Is Inside a Complex Polygon*. 2021. URL: http://alienryderflex.com/polygon/.

[49] Olivier Flauzac, Fabien Mauhourat, and Florent Nolot. “A review of native container security for running applications”. In: *Procedia Computer Science* 175 (2020), pp. 157–164. DOI: 10.1016/j.procs.2020.07.025.

[50] Eric Fleury et al. “FIT IoT-LAB: The Largest IoT Open Experimental Testbed”. In: *ERCIM News* 101 (2015), p. 4. URL: https://hal.inria.fr/hal-01138038/.

[51] Xing Gao et al. “Containerleaks: Emerging security threats of information leakages in container clouds”. In: *Proc. DSN*. 2017. DOI: 10.1109/DSN.2017.49.

[52] Gartner. *Gartner Forecasts Worldwide Public Cloud End-User Spending to Reach Nearly $ 500 Billion in 2022*. 2022. URL: https://www.gartner.com/en/newsroom/press-releases/2022-04-19-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-reach-nearly-500-billion-in-2022.

[53] Gartner. *Gartner Predicts the Future of Cloud and Edge Infrastructure*. 2021. URL: https://www.gartner.com/smarterwithgartner/gartner-predicts-the-future-of-cloud-and-edge-infrastructure.

[54] Julien Gedeon et al. “What the Fog? Edge Computing Revisited: Promises, Applications and Future Challenges”. In: *IEEE Access* 7 (2019), pp. 152847–152878. DOI: 10.1109/ACCESS.2019.2948399.

[55] Anousheh Gholami and John S. Baras. "Collaborative Cloud-Edge-Local Computation Offloading for Multi-Component Applications". In: *Proc. SEC*. 2021. URL: https://ieeexplore.ieee.org/abstract/document/9709014.

[56] Tom Goethals, Filip De Turck, and Bruno Volckaert. "Extending Kubernetes Clusters to Low-Resource Edge Devices Using Virtual Kubelets". In: *IEEE Transactions on Cloud Computing* 10.4 (2022), pp. 2623–2636. DOI: 10.1109/TCC.2020.3033807.

[57] Inigo Goiri, Jordi Guitart, and Jordi Torres. "Characterizing Cloud Federation for Enhancing Providers' Profit". In: *IEEE 3rd International Conference on Cloud Computing*. 2010. DOI: 10.1109/CLOUD.2010.32.

[58] Nicolas Greneche et al. "A Methodology to Scale Containerized HPC Infrastructures in the Cloud". In: *Euro-Par: Parallel Processing*. Ed. by José Cano and Phil Trinder. 2022. DOI: 10.1007/978-3-031-12597-3_13.

[59] Quentin Guilloteau et al. "Painless Transposition of Reproducible Distributed Environments with NixOS Compose". In: *IEEE International Conference on Cluster Computing (CLUSTER)*. 2022. DOI: 10.1109/CLUSTER51413.2022.00051.

[60] Chang Jie Guo et al. "A framework for native multi-tenancy application development and management". In: *Proc. CEC-EEE*. 2007. DOI: 10.1109/CEC-EEE.2007.4.

[61] Red Hat. *What is IaaS?* 2022. URL: https://www.redhat.com/en/topics/cloud-computing/what-is-iaas.

[62] Qiang He et al. "A Game-Theoretical Approach for User Allocation in Edge Computing Environment". In: *IEEE Transactions on Parallel and Distributed Systems* 31.3 (2020), pp. 515–529. DOI: 10.1109/TPDS.2019.2938944.

[63] Benjamin Hindman et al. "Mesos: A platform for fine-grained resource sharing in the data center." In: *NSDI*. 2011. URL: https://www.usenix.org/legacy/event/nsdi11/tech/full_papers/Hindman_new.pdf.

[64] Jiangshui Hong et al. "An Overview of Multi-cloud Computing". In: *Web, Artificial Intelligence and Network Applications*. 2019. DOI: 10.1007/978-3-030-15035-8_103.

[65] Yang Hu et al. "Concurrent container scheduling on heterogeneous clusters with multi-resource constraints". In: *Future Generation Computer Systems* 102 (2020), pp. 562–573. DOI: 10.1016/j.future.2019.08.025.

[66] Yang Hu et al. "Ecsched: Efficient container scheduling on heterogeneous clusters". In: *European Conference on Parallel Processing*. 2018. DOI: 10.1007/978-3-319-96983-1_26.

[67] Marco Iorio et al. "Computing Without Borders: The Way Towards Liquid Computing". In: *IEEE Transactions on Cloud Computing* (2022), pp. 1–18. DOI: 10.1109/TCC.2022.3229163.

[68] Asad Javed et al. "IoTEF: A Federated Edge-Cloud Architecture for Fault-Tolerant IoT Applications". In: *Journal of Grid Computing* 18.1 (2020), pp. 57–80. DOI: 10.1007/s10723-019-09498-8.

[69] Isam Mashhour Al Jawarneh et al. “Container Orchestration Engines: A Thorough Functional and Performance Comparison”. In: *Proc. ICC*. 2019. DOI: 10.1109/ICC.2019.8762053.

[70] Qin Jia et al. “Smart Spot Instances for the Supercloud”. In: *Proceedings of the 3rd Workshop on CrossCloud Infrastructures & Platforms*. 2016. DOI: 10.1145/2904111.2904114.

[71] Ru Jia et al. “A systematic review of scheduling approaches on multi-tenancy cloud platforms”. In: *Information and Software Technology* 132 (2021), p. 106478. DOI: 10.1016/j.infsof.2020.106478.

[72] Jaap Kabbedijk et al. “Defining multi-tenancy: A systematic mapping study on the academic and the industrial perspective”. In: *J. Systems and Software* 100 (2015), pp. 139–148. DOI: 10.1016/j.jss.2014.10.034.

[73] Chanwit Kaewkasi and Kornrathak Chuenmuneewong. “Improvement of container scheduling for docker using ant colony optimization”. In: *9th international conference on knowledge and smart technology (KST)*. 2017. DOI: 10.1109/KST.2017.7886112.

[74] P. Karhula, J. Janak, and H. Schulzrinne. “Checkpointing and Migration of IoT Edge Functions”. In: *Proc. EdgeSys ’19*. 2019. DOI: 10.1145/3301418.3313947.

[75] Kuljeet Kaur et al. “KEIDS: Kubernetes-based energy and interference driven scheduler for industrial IoT in edge-cloud ecosystem”. In: *IEEE Internet of Things Journal* 7.5 (2019), pp. 4228–4237. DOI: 10.1109/JIOT.2019.2939534.

[76] Wazir Zada Khan et al. “Edge computing: A survey”. In: *Future Generation Computer Systems* 97 (2019), pp. 219–235. DOI: 10.1016/j.future.2019.02.050.

[77] Dongmin Kim et al. “TOSCA-Based and Federation-Aware Cloud Orchestration for Kubernetes Container Platform”. In: *Applied Sciences* 9.1 (2019). DOI: 10.3390/app9010191.

[78] Dimitrios G. Kogias, Michael G. Xevgenis, and Charalampos Z. Patrikakis. “Cloud Federation and the Evolution of Cloud Computing”. In: *Computer* 49.11 (2016), 96–99. DOI: 10.1109/MC.2016.344.

[79] Endah Kristiani et al. “Implementation of an Edge Computing Architecture Using OpenStack and Kubernetes”. In: *Information Science and Applications*. 2018. DOI: 10.1007/978-981-13-1056-0_66.

[80] Rakesh Kumar and B Thangaraju. “Performance analysis between runc and kata container runtime”. In: *Proc. CONECCT*. 2020. DOI: 10.1109/CONECCT50063.2020.9198653.

[81] Tobias Kurze et al. “Cloud federation”. In: *The Second International Conference on Cloud Computing, GRIDs, and Virtualization*. 2011. URL: https://www.aifb.kit.edu/images/0/02/Cloud_Federation.pdf.

[82] Lars Larsson et al. “Decentralized Kubernetes Federation Control Plane”. In: *IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. 2020. DOI: 10.1109/UCC48980.2020.00056.

[83] Paul Leach, Michael Mealling, and Rich Salz. *A Universally Unique IDentifier (UUID) URN namespace*. RFC 4122. http://www.rfc-editor.org/rfc/rfc4122.txt. IETF, 2005.

[84] A. Leon-Garcia and H. Bannazadeh. "SAVI Testbed for Applications on Software-Defined Infrastructure". In: *The GENI Book*. Ed. by R. McGeer et al. Springer, 2016. Chap. 22. DOI: 10.1007/978-3-319-33769-2_22.

[85] Xiaomin Li et al. "A Hybrid Computing Solution and Resource Scheduling Strategy for Edge Computing in Smart Manufacturing". In: *IEEE Transactions on Industrial Informatics* 15.7 (2019), pp. 4225–4234. DOI: 10.1109/TII.2019.2899679.

[86] Bo Liu et al. "A new container scheduling algorithm based on multi-objective optimization". In: *Soft Computing* 22.23 (2018), pp. 7741–7752. DOI: 10.1007/s00500-018-3403-7.

[87] Juan Liu et al. "Delay-optimal computation task scheduling for mobile-edge computing systems". In: *IEEE International Symposium on Information Theory (ISIT)*. 2016. DOI: 10.1109/ISIT.2016.7541539.

[88] Madhusanka Liyanage et al. "Driving forces for Multi-Access Edge Computing (MEC) IoT integration in 5G". In: *ICT Express* 7.2 (2021), pp. 127–137. DOI: 10.1016/j.icte.2021.05.007.

[89] Loft Labs, Inc. *kiosk*. software: v0.2.11 of November 2021. Website: https://loft.sh/. 2020. URL: https://github.com/loft-sh/kiosk.

[90] Loft Labs, Inc. *vcluster*. software: v0.14.2 of March 2023. Website: https://www.vcluster.com/. 2021. URL: https://github.com/loft-sh/vcluster.

[91] M-Lab. *Murakami*. 2023. URL: https://github.com/m-lab/murakami.

[92] M-Lab. *Network Diagnostic Tool*. 2021. URL: https://measurementlab.net/tests/ndt/.

[93] Karim Manaouil and Adrien Lebre. *Kubernetes and the Edge?* Research Report RR-9370. Inria Rennes - Bretagne Atlantique, Oct. 2020, p. 19. URL: https://hal.inria.fr/hal-02972686.

[94] R. McGeer et al., eds. *The GENI Book*. Springer, 2016. DOI: 10.1007/978-3-319-33769-2.

[95] Peter Mell and Tim Grance. *The NIST Definition of Cloud Computing*. Special Publication 800-145. NIST, 2011. DOI: 10.6028/NIST.SP.800-145.

[96] Dirk Merkel. "Docker: lightweight Linux containers for consistent development and deployment". In: *Linux Journal* 2014.239 (2014). URL: https://dl.acm.org/doi/10.5555/2600239.2600241.

[97] Rafael Moreno-Vozmediano, Ruben S. Montero, and Ignacio M. Llorente. "Multicloud Deployment of Computing Clusters for Loosely Coupled MTC Applications". In: *IEEE Transactions on Parallel and Distributed Systems* 22.6 (2011), pp. 924–930. DOI: 10.1109/TPDS.2010.186.

[98] P. Müeller and S. Fischer. "Europe's Mission in Next-Generation Networking With special emphasis on the German-Lab Project". In: *The GENI Book*. Ed. by Rick McGeer et al. Springer, 2016. Chap. 21. DOI: 10.1007/978-3-319-33769-2_21.

[99] *Multi-access Edge Computing (MEC); Framework and Reference Architecture*. Group Specification ETSI GS MEC 003 V2.2.1 (2020-12). ETSI, 2020. URL: https://www.etsi.org/deliver/etsi_gs/MEC/001_099/003/02.02.01_60/gs_MEC003v020201p.pdf.

[100] *Multi-access Edge Computing (MEC); Support for network slicing*. Group Report GR MEC 024 V2.1.1 (2019-11). ETSI, 2019. URL: https://www.etsi.org/deliver/etsi_gr/MEC/001_099/024/02.01.01_60/gr_MEC024v020101p.pdf.

[101] Akihiro Nakao and Kazuhisa Yamada. "Research and Development on Network Virtualization Technologies in Japan: VNode and FLARE Projects". In: *The GENI Book*. Ed. by Rick McGeer et al. Springer, 2016. Chap. 23. DOI: 10.1007/978-3-319-33769-2_23.

[102] C. Pahl and B. Lee. "Containers and Clusters for Edge Cloud Architectures – A Technology Review". In: *Proc. FiCloud '15*. 2015. DOI: 10.1109/FiCloud.2015.35.

[103] C. Pahl et al. "A Container-Based Edge Cloud PaaS Architecture Based on Raspberry Pi Clusters". In: *Proc. FiCloud '16*. 2016. DOI: 10.1109/W-FiCloud.2016.36.

[104] Fawaz Paraiso et al. "A Federated Multi-Cloud PaaS Infrastructure". In: *5th IEEE International Conference on Cloud Computing*. 2012. DOI: 10.1109/CLOUD.2012.79.

[105] Dana Petcu. "Multi-Cloud: Expectations and Current Approaches". In: *Proceedings of the 2013 International Workshop on Multi-Cloud Applications and Federated Clouds*. 2013. DOI: 10.1145/2462326.2462328.

[106] Larry Peterson et al. *Edge Cloud Operations: A Systems Approach*. Systems Approach, LLC, 2022. URL: https://ops.systemsapproach.org/.

[107] Larry Peterson et al. "Experiences Building PlanetLab". In: *Proc. 7th Symposium on Operating Systems Design and Implementation (USENIX OSDI)*. 2006. URL: https://dl.acm.org/doi/abs/10.5555/1298455.1298489.

[108] Quoc-Viet Pham et al. "A survey of multi-access edge computing in 5G and beyond: Fundamentals, technology integration, and state-of-the-art". In: *IEEE Access* 8 (2020), pp. 116974–117017. DOI: 10.1109/ACCESS.2020.3001277.

[109] Roger Alan Pick et al. "Shepherd or servant: centralization and decentralization in information technology governance". In: *International Journal of Management & Information Systems (IJMIS)* 19.2 (2015), pp. 61–68. URL: https://core.ac.uk/download/pdf/268112979.pdf.

[110] D. Pizzolli et al. "Cloud4IoT: A Heterogeneous, Distributed and Autonomic Cloud Platform for the IoT". In: *Proc. CloudCom '16*. 2016. DOI: 10.1109/CloudCom.2016.0082.

[111] Amit M Potdar et al. "Performance Evaluation of Docker Container and Virtual Machine". In: *Procedia Computer Science* 171 (2020), pp. 1419–1428. DOI: 10.1016/j.procs.2020.04.152.

[112] Rancher Labs, Inc. *k3v - Virtual Kubernetes*. proof-of-concept software: v0.0.1 of July 2019. Website: https://www.rancher.com/. 2019. URL: https://github.com/ibuildthecloud/k3v.

[113] Alessandro Randazzo and Ilenia Tinnirello. "Kata containers: An emerging architecture for enabling MEC services in fast and secure way". In: *Proc. IOTSMS*. 2019. DOI: 10.1109/IOTSMS48152.2019.8939164.

[114] Antonio Regalado. *Who Coined 'Cloud Computing'?* 2011. URL: https://www.technologyreview.com/2011/10/31/257406/who-coined-cloud-computing/.

[115] Robert Ricci and the Emulab Team. "Precursors: Emulab". In: *The GENI Book*. Springer, 2016. Chap. 2, pp. 19–33.

[116] Bhaskar Prasad Rimal, Eunmi Choi, and Ian Lumb. "A Taxonomy and Survey of Cloud Computing Systems". In: *Fifth International Joint Conference on INC, IMS and IDC*. 2009. DOI: 10.1109/NCM.2009.218.

[117] Bhaskar Prasad Rimal et al. "Architectural requirements for cloud computing systems: an enterprise cloud approach". In: *J. Grid Computing* 9.1 (2011), pp. 3–26. DOI: 10.1007/s10723-010-9171-y.

[118] B. Rochwerger et al. "The Reservoir Model and Architecture for Open Federated Cloud Computing". In: *IBM Journal of Research and Development* 53.4 (2009), 535–545. DOI: 10.1147/JRD.2009.5429058.

[119] John Fralick Rockart and Joav Steve Leventer. *Centralization vs decentralization of information systems: a critical survey of current literature*. Tech. rep. 1976. URL: https://dspace.mit.edu/bitstream/handle/1721.1/1906/SWP-0845-02293677-CISR-023.pdf.

[120] Luis Rodero-Merino et al. "Building safe PaaS clouds: A survey on security in multitenant software platforms". In: *Computers & Security* 31.1 (2012), pp. 96–108. DOI: 10.1016/j.cose.2011.10.006.

[121] Josh Rosso et al. *Production Kubernetes*. "O'Reilly Media, Inc.", 2021. URL: https://www.oreilly.com/library/view/production-kubernetes/9781492092292/.

[122] Peter Rost et al. "Network Slicing to Enable Scalability and Flexibility in 5G Mobile Networks". In: *IEEE Communications Magazine* 55.5 (2017), pp. 72–79. DOI: 10.1109/MCOM.2017.1600920.

[123] Cristian Ruiz, Olivier Richard, and Joseph Emeras. "Reproducible Software Appliances for Experimentation". In: *Testbeds and Research Infrastructure: Development of Networks and Communities*. 2014. DOI: 10.1007/978-3-319-13326-3_4.

[124] D. Sarmiento et al. "Multi-site Connectivity for Edge Infrastructures : DIMINET:DIstributed Module for Inter-site NETworking". In: *20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*. 2020. DOI: 10.1109/CCGrid49817.2020.00-81.

[125] Mahadev Satyanarayanan. "The emergence of edge computing". In: *Computer* 50.1 (2017), pp. 30–39. DOI: 10.1109/MC.2017.9.

[126] Mahadev Satyanarayanan et al. “The Case for VM-Based Cloudlets in Mobile Computing”. In: *IEEE Pervasive Computing* 8.4 (2009), pp. 14–23. DOI: 10.1109/MPRV.2009.82.

[127] Malte Schwarzkopf et al. “Omega: Flexible, Scalable Schedulers for Large Compute Clusters”. In: *Proceedings of the 8th ACM European Conference on Computer Systems*. 2013. DOI: 10.1145/2465351.2465386.

[128] Berat Can Şenel et al. “Demo: EdgeNet, a Production Internet-Scale Container-Based Distributed System Testbed”. In: *IEEE International Conference on Distributed Computing Systems (ICDCS), Demo session*. 2022. DOI: 10.1109/ICDCS54860.2022.00141.

[129] Berat Can Şenel et al. “EdgeNet: A Multi-Tenant and Multi-Provider Edge Cloud”. In: *EdgeSys’21 - 4th International Workshop on Edge Systems, Analytics and Networking, workshop held in conjunction with the 16th ACM European Conference on Computer Systems (EuroSys 2021), Best Paper Award*. 2021. DOI: 10.1145/3434770.3459737.

[130] Berat Can Şenel et al. “EdgeNet: the Global Kubernetes Cluster Testbed”. In: *IEEE INFOCOM International Workshop on Computer and Networking Experimental Research using Testbeds (CNERT), extended abstract, demo*. 2021. DOI: 10.1109/INFOCOMWKSHPS51825.2021.9484475.

[131] Berat Can Şenel et al. “Federating EdgeNet with Fed4FIRE+ and Deploying its Nodes Behind NATs”. In: *SLICES Workshop at International Federation for Information Processing (IFIP 2022) Networking Conference*. 2022. DOI: 10.23919/IFIPNetworking55013.2022.9829758.

[132] Berat Can Şenel et al. “Multitenant Containers as a Service (CaaS) for Clouds and Edge Clouds”. In: *arXiv preprint arXiv:2304.08927* (2023). DOI: 10.48550/arXiv.2304.08927.

[133] Berat Can Şenel et al. “Shared internet-scale measurement platforms”. In: *NSF Workshop on Overcoming Measurement Barriers to Internet Research (WOMBIR 2021), extended abstract*. 2021. URL: https://hal.archives-ouvertes.fr/hal-03113118.

[134] *Sharpening the Edge II: Diving Deeper into the LF Edge Taxonomy and Projects*. White Paper. The Linux Foundation, 2022. URL: https://www.lfedge.org/wp-content/uploads/2022/06/LFEdgeTaxonomyWhitepaper_062322.pdf.

[135] *Sharpening the Edge: Overview of the LF Edge Taxonomy and Framework*. White Paper. The Linux Foundation, 2020. URL: https://www.lfedge.org/wp-content/uploads/2020/07/LFedge_Whitepaper.pdf.

[136] Supreeth Shastri and David Irwin. “HotSpot: Automated Server Hopping in Cloud Spot Markets”. In: *Proc. SoCC*. 2017. DOI: 10.1145/3127479.3132017.

[137] Weisong Shi and Schahram Dustdar. “The promise of edge computing”. In: *Computer* 49.5 (2016), pp. 78–81. DOI: 10.1109/MC.2016.145.

[138] Weisong Shi et al. “Edge computing: Vision and challenges”. In: *IEEE internet of things journal* 3.5 (2016), pp. 637–646. DOI: 10.1109/JIOT.2016.2579198.

[139] Sachchidanand Singh and Nirmala Singh. “Containers & Docker: Emerging roles & future of Cloud technology”. In: *2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*. 2016. DOI: 10.1109/ICATCCT.2016.7912109.

[140] Mukesh Singhal et al. “Collaboration in multicloud computing environments: Framework and security issues”. In: *Computer* 46.2 (2013), pp. 76–84. DOI: 10.1109/MC.2013.46.

[141] Inés Sittón-Candanedo et al. “A review of edge computing reference architectures and a new global edge proposal”. In: *Future Generation Computer Systems* 99 (2019), pp. 278–294. DOI: 10.1016/j.future.2019.04.016.

[142] RIPE Ncc Staff. “Ripe atlas: A global internet measurement network”. In: *Internet Protocol Journal* 18.3 (2015), pp. 2–26. URL: http://book.itep.ru/depository/diagnostics/ipj18.3.pdf.

[143] *State of the Edge*. Tech. rep. The Linux Foundation, 2021. URL: https://stateoftheedge.com/reports/state-of-the-edge-report-2021/.

[144] *State of the Edge*. Tech. rep. The Linux Foundation, 2022. URL: https://stateoftheedge.com/reports/state-of-the-edge-report-2022/.

[145] Sari Sultan, Imtiaz Ahmad, and Tassos Dimitriou. “Container security: Issues, challenges, and the road ahead”. In: *IEEE Access* 7 (2019), pp. 52976–52996. DOI: 10.1109/ACCESS.2019.2911732.

[146] Lalith Suresh et al. “Automating Cluster Management with Weave”. In: *arXiv preprint arXiv:1909.03130* (2019). DOI: 10.48550/arXiv.1909.03130.

[147] Mulugeta Ayalew Tamiru et al. “Instability in Geo-Distributed Kubernetes Federation: Causes and Mitigation”. In: *28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 2020. DOI: 10.1109/MASCOTS50786.2020.9285934.

[148] Mulugeta Ayalew Tamiru et al. “mck8s: An orchestration platform for geo-distributed multi-cluster environments”. In: *International Conference on Computer Communications and Networks (ICCCN)*. 2021. DOI: 10.1109/ICCCN52240.2021.9522318.

[149] Tencent. *tensile-kube*. software: v0.1.1 of February 2021. 2021. URL: https://github.com/virtual-kubelet/tensile-kube.

[150] The Admiralty authors. *Admiralty*. software: v0.15.1 of March 2022. Website: https://admiralty.io/. 2018. URL: https://github.com/admiraltyio/admiralty.

[151] The HNC authors. *The Hierarchical Namespace Controller (HNC)*. software: v1.0.0 of April 2022. Project incubated by the Kubernetes Multitenancy Working Group. 2019. URL: https://github.com/kubernetes-sigs/hierarchical-namespaces.

[152] The Karmada authors. *Karmada*. software: v1.5.0 of February 2023. Website: https://karmada.io/. 2020. URL: https://github.com/karmada-io/karmada.

[153] The Liqo authors. *Liqo*. software: v0.7.2 of March 2023. A project from the Politecnico di Torino. Website: https://liqo.io/. 2020. URL: https://github.com/liqotech/liqo.

[154] The Virtual Kubelet authors. *Virtual Kubelet*. software: v1.8.0 of March 2023. A Cloud Native Computing Foundation sandbox project. Website: https://virtual-kubelet.io/. 2020. URL: https://github.com/virtual-kubelet/virtual-kubelet.

[155] The VirtualCluster authors. *VirtualCluster - Enabling Kubernetes Hard Multi-tenancy*. software: v0.1.0 of June 2021. Project incubated by the Kubernetes Multitenancy Working Group. 2021. URL: https://github.com/kubernetes-sigs/cluster-api-provider-nested/tree/main/virtualcluster.

[156] Muhammad Tirmazi et al. "Borg: The next Generation". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020. DOI: 10.1145/3342195.3387517.

[157] Orazio Tomarchio, Domenico Calcaterra, and Giuseppe Di Modica. "Cloud resource orchestration in the multi-cloud landscape: a systematic review of existing frameworks". In: *Journal of Cloud Computing* 9.1 (2020). DOI: 10.1186/s13677-020-00194-7.

[158] Guillaume Everarts de Velp, Etienne Rivière, and Ramin Sadre. "Understanding the performance of container execution environments". In: *Proc. WoC*. 2020. DOI: 10.1145/3429885.3429967.

[159] Abhishek Verma et al. "Large-Scale Cluster Management at Google with Borg". In: *Proceedings of the Tenth European Conference on Computer Systems*. 2015. DOI: 10.1145/2741948.2741964.

[160] K. Vermeulen et al. "Diamond-Miner: Comprehensive Discovery of the Internet's Topology Diamonds". In: *Proc. NSDI '20*. 2020. URL: https://www.usenix.net/system/files/nsdi20-paper-vermeulen.pdf.

[161] K. Vermeulen et al. "Multilevel MDA-lite Paris traceroute". In: *Proc. IMC '18*. 2018. DOI: 10.1145/3278532.3278536.

[162] William Viktorsson, Cristian Klein, and Johan Tordsson. "Security-Performance Tradeoffs of Kubernetes Container Runtimes". In: *28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 2020. DOI: 10.1109/MASCOTS50786.2020.9285946.

[163] David Villegas et al. "Cloud Federation in a Layered Service Model". In: *Journal of Computer and System Sciences* 78.5 (2012), 1330–1344. DOI: 10.1016/j.jcss.2011.12.017.

[164] Y. Xiong et al. "Extend Cloud to Edge with KubeEdge". In: *Proc. SEC 2018*. 2018. DOI: 10.1109/SEC.2018.00048.

[165] Tzu-Bin Yan et al. "PacketLab: Tools Alpha Release and Demo". In: *Proceedings of the 22nd ACM Internet Measurement Conference*. 2022. DOI: 10.1145/3517745.3563029.

[166] Shanhe Yi, Cheng Li, and Qun Li. "A Survey of Fog Computing: Concepts, Applications and Issues". In: *Proc. Mobidata*. 2015. DOI: 10.1145/2757384.2757397.

[167] Shanhe Yi et al. "Fog Computing: Platform and Applications". In: *Proc. HotWeb*. 2015. DOI: 10.1109/HotWeb.2015.22.

[168] Mark Zachry. "Constructing Usable Documentation: A Study of Communicative Practices and the Early Uses of Mainframe Computing in Industry". In: *Proceedings of the 17th Annual International Conference on Computer Documentation*. 1999. DOI: 10.1145/318372.318388.

[169] Lanfranco Zanzi, Fabio Giust, and Vincenzo Sciancalepore. "M2EC: A multi-tenant resource orchestration in multi-access edge computing systems". In: *Proc. WCNC*. 2018. DOI: 10.1109/WCNC.2018.8377292.

[170] C. Zheng, Q. Zhuang, and F. Guo. "A Multi-Tenant Framework for Cloud Container Services". In: *Proc. ICDCS*. 2021. DOI: 10.1109/ICDCS51616.2021.00042.