Automatic resource management in geo-distributed multi-cluster environments

Mulugeta Ayalew Tamiru

DOCTORAT BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par

Mulugeta Ayalew TAMIRU

Automatic Resource Management in Geo-Distributed Multi-Cluster Environments

Thèse présentée et soutenue à Rennes, le 20 septembre 2021
Unité de recherche : IRISA (UMR 6074)

Rapporteurs avant soutenance :

Ivona BRANDIC	Professeure, Vienna University of Technology
Alexandru IOSUP	Professeur, Vrije Universiteit Amsterdam

Composition du Jury :

Président :	Olivier BARAIS	Professeur, Université de Rennes 1
Examinateurs :	Ivona BRANDIC	Professeure, Vienna University of Technology
	Alexandru IOSUP	Professeur, Vrije Universiteit Amsterdam
	Ling LIU	Professeure, Georgia Institute of Technology
	Prashant SHENOY	Professeur, University of Massachusetts Amherst
Dir. de thèse :	Guillaume PIERRE	Professeur, Université de Rennes 1
Co-dir. de thèse :	Erik ELMROTH	Professeur, Umeå University
	Johan TORDSSON	Maître de conférences, Umeå University

This work is part of a project that has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 765452. The information and views set out in this publication are those of the author(s) and do not necessarily reflect the official opinion of the European Union. Neither the European Union institutions and bodies nor any person acting on their behalf may be held responsible for the use which may be made of the information contained therein.

Experiments presented in Chapters 5 and 6 of this thesis were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

To my late father *Aleka* Ayalew Tamiru.
For standing up and fighting for faith, truth and justice.

Acknowledgement

First of all, I would like to thank my supervisors Guillaume Pierre, Johan Tordsson and Erik Elmroth for their continuous mentoring, feedback and encouragement throughout my journey. Johan and Erik, thank you for dedicating a lot of time for feedback and discussions during my 15-month stay at Elastisys, which has helped me immensely to understand my thesis' topic and identify important problems. Guillaume, thank you for the inspiration, guidance and continuous feedback that went into this thesis. It was a great honor for me to be supervised by great researchers like you all.

Next, I would like to thank members of the jury: Olivier Barais, Ivona Brandic, Alexandru Iosup, Prashant Shenoy and Ling Liu for accepting to review my thesis.

Many thanks to the members of the CSID committee Anne-Cécile Orgerie and Etienne Rivière for your feedback and suggestions throughout the years.

I would like to thank Robert, Henrik and Linda for creating a great working atmosphere at Elastisys. Furthermore, I would like to thank all colleagues at Elastisys for making my stay at Elastisys fun both at in the office and outdoors.

Thank you to friends I made in Umeå: Ewnetu, Selome, Aemiro, Samrawit and Abel.

I would also like to thank my colleagues at the FogGuru project: Lilly, Davaa, Mozhdeh, Hamid, Felipe, Paulo, Dimi, Julie and Nena. In particular, I appreciate the time I spent with Lilly, Hamid and Felipe in Umeå, and with Lilly, Davaa and Mozhdeh in València.

To my friends: the three Samuel's, Ul, Tom, Hailu, Said, Nassim, Deb, Ravish, Vikki, Najeeb, Mesfin, Wendmu, Girum and Solomon, thank you for being such great friends!

I am greatly indebted to my late parents *Aleka* Ayalew Tamiru and Rebeka Lisanework because I am the person I am today thanks to your love, nurturing and sacrifice. You will live in my heart forever. I thank you!

My siblings Semret, Senetsehay, Tsion, Paulos, Roman, Frehiwot, Habtamu, Michael, Yonathan, Bethlehem, Mahilet, Addis and Nathaniel have been constant sources of love, friendship, hope and strength. Thank you for being there for me whenever I needed your support. I would also like to thank all my nieces and nephews. I love you all very much.

A very special thank you to my beloved wife Tigisté. Words do not suffice to express my gratitude for your love, support and encouragement. I love you very much.

Résumé

Les plates-formes de cloud computing se composent d'un vaste réservoir de ressources de calcul, de réseau et de stockage virtualisées qui sont situées dans des centres de données très grands, mais centralisés. Cela leur permet d'offrir des ressources à la demande abordables et évolutives à leurs utilisateurs. En conséquence, les entreprises de toutes tailles déplacent de plus en plus leurs services vers des centres de données cloud.

Les applications cloud de différents secteurs doivent répondre à une ou plusieurs exigences non fonctionnelles telles que la proximité, la haute disponibilité, la tolérance aux pannes, l'évolutivité, la conformité à la confidentialité et aux réglementations régionales et la neutralité des fournisseurs. Par exemple, les applications de réseaux sociaux en ligne, de streaming vidéo et de jeux en ligne doivent maintenir une faible latence de bout en bout pour satisfaire les attentes des utilisateurs finaux. De même, les applications sensibles à la latence telles que l'assistance cognitive, la réalité virtuelle et la réalité augmentée nécessitent une latence de bout en bout ultra-faible inférieure à 10–20 ms. Les applications de l'Internet des objets dans les maisons, les villes intelligentes et les industries génèrent une grande quantité de données qui seraient lentes et coûteuses à transporter vers des centres de données cloud distants pour traitement. D'un autre côté, les applications sensibles à la confidentialité dans les domaines de la santé et de la finance doivent se conformer aux réglementations régionales qui exigent que les applications et les données s'exécutent dans certaines régions. En outre, les applications exécutées dans des centres de données de cloud privé peuvent avoir besoin d'acquérir des ressources supplémentaires auprès de centres de données de cloud public pour gérer les pics temporaires de demandes de ressources qu'un centre de données de cloud privé ne peut pas satisfaire. Enfin, les entreprises peuvent choisir d'éviter le verrouillage du fournisseur qui peut résulter de l'utilisation des ressources d'un seul fournisseur de cloud.

Afin de répondre à ces exigences, les applications cloud doivent être déployées sur des ressources informatiques suffisamment réparties géographiquement à des emplacements stratégiques dans diverses régions du monde. Certaines applications nécessitent encore plus de diffusion au niveau des villes, des quartiers et des installations. Par conséquent, des modèles de déploiement géo-distribués ont émergé pour répondre à ces exigences.

Ces environnements informatiques géo-distribués peuvent être classés en trois architectures principales, à savoir le cloud hybride, le multi-cloud et le fog computing, selon des degrés de distribution croissants. Le cloud hybride agrège les ressources des centres de données cloud privés et publics. Les déploiements multi-cloud couvrent des centres de données dans plusieurs régions d'un ou plusieurs fournisseurs de services de cloud public. Le fog computing place un grand nombre de ressources de calcul, de mise en réseau et de stockage à la périphérie du réseau pour couvrir des zones géographiques encore plus larges.

Malgré les avantages de l'informatique géo-distribuée, le déploiement d'applications et la gestion des ressources dans ces environnements sont difficiles en raison de plusieurs défis tels qu'un grand nombre de ressources, une large répartition géographique, des réseaux hétérogènes, de mauvaises conditions de réseau, de l'hétérogénéité des charges et de l'hétérogénéité des ressources. Ces défis entraînent des situations indésirables qui affectent la qualité d'expérience des utilisateurs finaux et la rentabilité des fournisseurs de services.

Dans cette thèse, nous abordons trois défis de déploiement d'applications et de gestion des ressources dans des environnements informatiques géo-distribués. Premièrement, des demandes de ressources des charges hétérogènes pourraient entraîner un sous-provisionnement ou un sur-provisionnement en fonction de la configuration du système d'autoscaling et du degré de coordination entre les niveaux des conteneurs et des machines virtuelles. Deuxièmement, les paramètres de configuration statiques associés à de mauvaises conditions de réseau dans des environnements informatiques géo-distribués pourraient entraîner des retards et des inexactitudes dans la détection des pannes. Cela entraînerait à son tour une instabilité, une indisponibilité et une dégradation des performances lorsque les applications sont déployées à grande échelle. Troisièmement, la distribution géographique à grande échelle des ressources informatiques pourrait conduire à une fragmentation des ressources où certains des clusters de charge sont sur-alloués tandis que d'autres sont à peine utilisés. Cela conduit à une dégradation des performances au niveau des clusters sur-alloués tandis que les clusters à peine utilisés sont sous-exploités.

Malgré ces défis de gestion des ressources, un gestionnaire de ressources pour les environnements informatiques géo-distribués doit répondre aux exigences de qualité de service des applications en prenant des décisions optimales concernant l'heure et l'emplacement de l'approvisionnement des ressources et du placement des applications, ainsi que la quantité

de ressources allouées aux applications. En outre, il doit fournir un modèle d'application approprié afin que les développeurs puissent déployer efficacement leurs applications.

Compte tenu de l'échelle et du niveau de distribution des environnements géo-distribués, il est difficile de les gérer manuellement. La gestion manuelle serait chronophage, sujette aux erreurs ou coûteuse. Par conséquent, il est important d'adopter des principes de gestion autonome des ressources et d'optimisation pour leur gestion.

Nous proposons trois contributions qui répondent aux défis mentionnés ci-dessus. Bien que nous ayons utilisé Kubernetes et ses écosystèmes pour valider nos contributions, nous soutenons que nos contributions peuvent également être généralisées à d'autres cadres d'orchestration de conteneurs actuels et futurs.

Contribution 1 : Évaluation des performances de l'autoscaler de cluster Kubernetes

L'élasticité est une caractéristique clé des infrastructures cloud qui permet de maintenir un niveau de performance acceptable malgré les fluctuations du trafic des utilisateurs. L'élasticité dans les clouds est rendue possible par le provisionnement dynamique des ressources ou l'autoscaling, où la quantité de ressources de calcul allouées aux applications est ajustée en réponse à l'évolution des demandes. Dans les plates-formes d'orchestration de conteneurs de pointe telles que Kubernetes, l'autoscaling peut être effectué au niveau des conteneurs et des machines virtuelles. Cependant, l'hétérogénéité des charges, telles que la durée des tâches et la quantité de ressources requises, peut entraîner un sous- ou sur-provisionnement en fonction de la configuration du système d'autoscaling, de la disponibilité des ressources et du degré de coordination entre les niveaux des conteneurs et des machines virtuelles. Alors que le sous-approvisionnement entraîne une dégradation des performances, le sur-approvisionnement entraîne des dépenses inutiles. Par conséquent, il est important de comprendre en profondeur comment ces systèmes d'autoscaling fonctionnent sous différentes configurations, applications et charges.

Dans la première contribution, nous analysons en profondeur les performances du Kubernetes Cluster Autoscaler à l'aide de coût monétaire et de métriques standards de performances d'autoscaling . Nous évaluons Cluster Autoscaler sous deux configurations qui déterminent la taille des nœuds. Nous évaluons l'impact de différentes applications et charges sur les performances de coût et d'autoscaling et montrons quantitativement que ces métriques sont affectées par la nature des applications. De plus, nous démontrons les

opportunités de gain de performances qui peuvent être obtenues en ajustant davantage de paramètres de configuration.

Contribution 2 : Améliorer la stabilité dans les environnements multi-clusters géo-distribués

Dans les environnements informatiques géo-distribués, la latence du réseau inter-cluster et la probabilité de perte de paquets sont des ordres de grandeur supérieurs à ceux d'un seul centre de données cloud. En conséquence, la détection des défaillances dans ces environnements peut souffrir de retard et d'imprécision. Cela affecte d'autres parties du système telles que les vérifications de l'état, le déploiement et la mise à l'échelle des applications. À son tour, cela peut entraîner une instabilité, une indisponibilité et une dégradation des performances.

Dans cette contribution, en utilisant Kubernetes Federation, nous analysons d'abord l'impact de différents paramètres de configuration sur la stabilité du système. Nous identifions ensuite le seul paramètre qui a le plus d'impact. Ensuite, nous identifions le compromis entre les détections inexactes des défaillances, qui conduit à l'instabilité, et les retards de détections. Nous montrons également l'impact de l'évolution des charges et des conditions du réseau sur le choix des valeurs des paramètres. Enfin, nous concevons, implémentons et évaluons un contrôleur proportionnel qui ajuste dynamiquement le paramètre de configuration concerné. Notre contrôleur améliore la stabilité du système de 83 à 92% dans le cas non contrôlé à 99,5 à 100% en utilisant le contrôleur.

Contribution 3 : Orchestration de conteneurs pour des environnements multi-clusters géo-distribués

Les frameworks d'orchestration de conteneurs à la pointe de la technologie ne répondent pas à toutes les exigences de déploiement d'applications et de gestion des ressources dans les environnements informatiques géo-distribués, car ils sont limités à un seul cluster dans un seul centre de données. Ceux proposés spécifiquement pour les environnements informatiques géo-distribués ont également plusieurs limitations telles que (1) des politiques de placement limitées ou manuelles ; (2) des politiques d'autoscaling qui ne prennent pas en compte la disponibilité des ressources dans les clusters voisins lorsque les clusters locaux manquent de ressources ; (3) travail limité sur le routage réseau pour les applications

qui s'étendent sur plusieurs clusters ; et (4) le manque de mécanismes pour compléter les clusters saturés avec des ressources supplémentaires, par exemple, du cloud public.

Pour relever ces défis, notre troisième contribution est une plate-forme d'orchestration de conteneurs pour les déploiements multi-clusters géo-distribués. Notre contribution s'appuie sur Kubernetes Federation et l'étend pour prendre en charge les politiques de placement tenant compte du réseau et des ressources, de l'autoscaling multi-clusters, du routage réseau inter-clusters, du provisionnement transparent, du déprovisionnement et de l'autoscaling des clusters cloud. Notre système proposé vise à être une plate-forme complète d'orchestration de conteneurs qui prend en charge les applications de plusieurs cas d'utilisation sur le continuum fog-cloud. Notre plate-forme améliore l'utilisation globale des ressources du système géo-distribué en réduisant le pourcentage de pods en attente à 6% contre 65% dans le cas de Kubernetes Federation pour la même charge.

Abstract

Cloud computing platforms consist of a large pool of virtualized compute, network and storage resources that are located in very large, yet centralized, *Data Center*s (DCs). This allows them to offer affordable and scalable on-demand resources to their users. As a result, enterprises of all sizes are increasingly moving their services to cloud DCs.

Cloud applications in different industries must fulfil one or more non-functional requirements such as proximity, performance, high availability, fault tolerance, scalability, cost efficiency, compliance with privacy and regional regulations and vendor neutrality. For instance, applications such as real-time gaming, video conferencing, wearable devices and traffic monitoring require low end-to-end latency below 100 ms to satisfy end users' expectations. Similarly, latency-critical applications such as cognitive assistance, virtual reality and augmented reality require ultra-low end-to-end latency below 10–20 ms. *Internet of Things* (IoT) applications in homes, smart cities and industries generate a vast amount of data which would be slow and expensive to transport to remote cloud DCs for processing. On the other hand, privacy-sensitive applications in healthcare and finance need to comply with regional regulations that require the applications and data to run in countries or regions with certain legislations. Furthermore, applications running in private cloud DCs may need to acquire additional resources from public cloud DCs to handle temporary spikes in resource demands that a private cloud DC cannot meet. Lastly, enterprises may choose to avoid vendor lock-in that may arise from using resources from a single cloud provider only.

In order to fulfil these requirements, cloud applications should be deployed on computing resources that are sufficiently geographically distributed at strategic locations in various regions of the world. Some of the applications require even more distribution at the level of cities, neighborhoods and facilities. Consequently, geo-distributed deployment models have emerged to address these requirements.

These geo-distributed computing environments can be categorized into three main architectures, namely hybrid cloud, multi-cloud and fog computing, with increasing degrees of distribution. Hybrid cloud aggregates resources from private and public cloud DCs. Multi-cloud deployments span DCs in multiple regions of a single or multiple public

*Cloud Service Provider*s (CSPs). Fog computing places a large number of compute, networking and storage resources at the edge of the network to cover even wider geographical areas.

Despite the benefits of geo-distributed computing, application deployment and resource management in these environments are difficult because of several challenges such as a large number of resources, wide geographical distribution, heterogeneous networks, poor networking conditions, heterogeneity of workloads and heterogeneity of resources. These challenges result in undesirable situations that affect end users' *Quality of Experience* (QoE) and service providers' profitability.

In this thesis, we address three application deployment and resource management challenges in geo-distributed computing environments. First, heterogeneous workload resource demands could lead to under- or over-provisioning depending on the configuration of the autoscaling system and the degree of coordination between the container and *Virtual Machine* (VM) levels. Second, static configuration parameters coupled with poor network conditions in geo-distributed computing environments could lead to delays and inaccurate failure detection. This would, in turn, lead to instability, unavailability and performance degradation when applications are deployed at a large scale. Third, large scale geographical distribution of computing resources could lead to resource fragmentation where some of the workload clusters are over-allocated while others are barely used. This results in performance degradation at the over-allocated clusters while the other clusters are under-utilized.

Despite these resource management challenges, a resource manager for geo-distributed computing environments needs to meet applications' *Quality of Service* (QoS) requirements by making optimal decisions about the time and location of resource provisioning and application placement as well as the amount of resources to allocate to applications. Furthermore, a resource manager needs to provide a suitable application model so that developers can deploy their applications efficiently.

Given the scale and level of distribution of geo-distributed environments, it would be time-consuming, error-prone or expensive to manage them manually. Therefore, it is important to adopt autonomous resource management and optimization principles for their management.

We propose three contributions that address the challenges mentioned above. We use tools in the very popular Kubernetes ecosystem to validate our contributions, and

argue that they can be generalized to other current and future container orchestration frameworks as well.

Contribution 1: Performance evaluation of Kubernetes cluster autoscaler

Elasticity is a key characteristic of cloud infrastructures that allows an acceptable performance level to be maintained despite fluctuations in demand. Elasticity in clouds is enabled by dynamic resource provisioning or autoscaling, where the amount of resources allocated to applications is adjusted in response to changing demands. In state-of-the-art container orchestration platforms such as Kubernetes, autoscaling may be done at the container and host(VM) levels. However, heterogeneity in workloads such as duration of tasks and amount of required resources could lead to under- or over-provisioning depending on the configuration of the autoscaling system, availability of resources and degree of coordination between the container and VM level autoscaling. Upon incorrect resource allocation, under-provisioning leads to performance degradation whereas over-provisioning leads to unnecessary expenses. Rapid fluctuations between the two should also be avoided as that could impact application robustness. Therefore, it is important to understand in depth how these autoscaling systems perform under different configurations, applications and workloads.

In the first contribution, we analyze the performance of the Kubernetes *Cluster Autoscaler* (CA) in depth based on monetary cost and standard autoscaling performance metrics. We evaluate CA under two configurations that determine the size of worker nodes. We evaluate the impact of different applications and workloads on cost and autoscaling performance and show quantitatively that these metrics are affected by the nature of the applications. Moreover, we demonstrate the potential for performance gains that can be achieved by tuning more autoscaling parameters.

Contribution 2: Improving stability in geo-distributed multi-cluster environments

In geo-distributed computing environments, the inter-cluster network latency and probability of packet loss are orders of magnitude greater than those within a single cloud DC. As a result, failure detection in these environments may suffer from delay and

inaccuracy. This affects other parts of the system such as health checks, application deployment and scaling. In turn, this may lead to instability, unavailability and performance degradation.

In this contribution, using Kubernetes Federation, we first analyze the impact of different configuration parameters on the stability of the system. We then identify the single parameter that has the most impact. Next, we identify the trade-off between inaccurate detection of failures, which leads to instability and detection delay, which may cause downtime. We also show the impact of changing workloads and network conditions on the choice of parameter values. Finally, we design, implement and evaluate a proportional controller that dynamically adjusts the concerned configuration parameter at run-time. Our controller improves the stability of the system from 83 – 92% in the uncontrolled case to 99.5 – 100% using the controller.

Contribution 3: Container orchestration for geo-distributed multi-cluster environments

State-of-the-art container orchestration frameworks do not meet all the requirements for application deployment and resource management in geo-distributed computing environments because they are limited to a single cluster in a single DC. Those proposed specifically for geo-distributed computing environments also have several limitations such as (1) limited or manual placement policies; (2) autoscaling policies that do not take into account availability of resources in neighboring clusters when local clusters run out of resources; (3) limited work on network routing for applications that span multiple clusters; and (4) lack of mechanisms to complement saturated clusters with additional resource, for example, from the public cloud.

To address these challenges, our third contribution is a container orchestration platform for geo-distributed multi-cluster deployments. Our contribution builds upon Kubernetes Federation and extends it to support network- and resource-aware placement policies, multi-cluster autoscaling, inter-cluster network routing and transparent provisioning, de-provisioning and autoscaling of cloud clusters. Our proposed system aims to be a comprehensive container orchestration platform that supports applications from multiple use-cases on the fog-to-cloud continuum. Our platform improves the overall resource utilization of the geo-distributed system by reducing the percentage of pending pods to 6% as opposed to 65% in the case of Kubernetes Federation for the same workload.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the past 15 years, the emergence and increasing adoption of cloud computing has significantly impacted enterprise computing. Enterprises, large and small, have adapted their *Information Technology* (IT) strategies to benefit from the flexibility, on-demand resource availability, scalability, high performance and low cost offered by cloud *Data Center*s (DCs). As a result, many enterprises have migrated most or all of their applications and data to cloud DCs [1]. Others, on the other hand, have kept their private DCs and complemented them with resources leased from the public cloud. Moreover, the ease of acquiring compute, storage and networking resources has allowed startups and new companies to deploy their applications in the cloud without the need to purchase hardware or build private DCs [2], [3].

Cloud computing allows its users to tap into a large pool of virtualized compute, network and storage resources that are located in very large, yet centralized, DCs. Users can lease *Virtual Machine*s (VMs) of very small to large capacity using flexible pricing schemes such as on-demand, reserved or spot. For instance, *Google Compute Engine* (GCE) offers VMs having resources from as little as 1 *Central Processing Unit* (CPU) core and 0.6 GB *Random Access Memory* (RAM) to 416 CPU cores and 5.75 TB RAM [4]. Moreover, in addition to handling regular workloads, cloud services have proven to be capable of handling large scale and urgent needs for scientific computing that require resources to the order of tens of thousands of *Graphics Processing Unit*s (GPUs) [5].

Although cloud computing offers seemingly infinite capacity, elasticity, flexibility and low-cost, the traditional single-DC model does not fulfil the requirements of several classes of applications. For instance, applications such as real-time gaming, video conferencing, wearable devices and traffic monitoring require low end-to-end latency below 100 ms [6], [7]. Other latency-critical applications such as virtual reality, augmented reality and cognitive assistance require ultra-low end-to-end latency, including communication and processing delays, below 10 - 20 ms [8]–[11]. *Internet of Things* (IoT) applications in factories, oil rigs, restaurants and smart cities generate a large amount of data which would be ex-

pensive and slow to transport to cloud DCs [12], [13]. Moreover, some of these applications are mission-critical and cannot tolerate any disconnections from the cloud [13]. Privacy-sensitive applications in some industries such as healthcare and finance need to comply with regulations that determine where their data can be stored [14]. Applications running in private cloud DCs may need to acquire additional resources from public cloud DCs to handle temporary spikes in resource demands which the private cloud DC cannot meet. Finally, enterprises may choose to avoid vendor lock-in that may arise from using resources exclusively from a single cloud provider.

The traditional single-DC application deployment approach fails to fulfil these requirements due to the distribution of end-users and end-devices at the edge of the network worldwide, and the predominant use of wireless networking, which results in high latency, low bandwidth and intermittent connectivity [10], [13], [15]. These application requirements can only be fulfilled if applications are deployed on computing resources that are sufficiently geographically distributed at strategic locations in various regions of the world. Some of the applications require even more distribution at the level of cities, neighborhoods and facilities.

Cloud providers are expanding their reach globally by building DCs in different parts of the world in response to the increasing demand. As of 2021, the three major *Cloud Service Provider*s (CSPs) Microsoft Azure, *Amazon Web Services* (AWS) and Google Cloud have DCs in 53 [1], 25 [2] and 25 [3] regions, respectively. Figure 1.1 depicts the distribution of DCs of the three major public CSPs in different parts of the world. The global distribution of cloud DCs coupled with growing network access and improved bandwidth has reduced end-user latency so that cloud services such as Facebook can be reached within 40 ms [16]. In fact, a recent study has shown that 80% of the probes used to measure *Round Trip Time* (RTT) to cloud DCs in Europe and North America can access a cloud DC within 20 ms thanks to the relatively higher concentration of cloud DCs in these regions [7].

To exploit this geographical distribution, hybrid cloud, multi-cloud and fog computing architectures have emerged as a natural evolution from consuming resources from single DCs [9], [10], [17], [18]. (1) A hybrid cloud deployment integrates resources from a private DC and one or more public cloud DCs [17]. (2) A multi-cloud deployment integrates resources from multiple zones or regions of a single or multiple public cloud provider(s) [15], [18], [19]. (3) Fog computing deployments are far more distributed than hybrid and multi-

1. Microsoft Azure regions – `https://bit.ly/3yxjeT8`
2. AWS regions – `https://amzn.to/3q25Oe0`
3. Google Cloud regions – `https://bit.ly/3bPQ4VM`

Figure 1.1 – Locations of cloud DCs of the three major CSPs Microsoft Azure (violet), AWS (orange) and Google Cloud (blue). Source: https://cloud-providers.jumpintothe.cloud/

cloud deployments and may encompass private clouds, public clouds and resources at the edge of the network closer to end-users [9]–[11], [20].

Enterprises make increasing use of hybrid and multi-cloud deployments to satisfy the non-functional requirements of modern applications such as performance, proximity to end-users, high availability, fault tolerance and compliance with privacy and legal constraints [14], [19], [21]–[24]. Furthermore, hybrid and multi-cloud deployments may offer cost reduction and vendor neutrality. These deployments have allowed the emergence and widespread use of applications such as web services, e-commerce, *Content Delivery Networks* (CDNs), online social networks, video streaming and video conferencing [25]–[29].

Fog computing addresses proximity, privacy and compliance requirements by placing compute, storage and networking resources in large numbers dispersed across wide geographical areas, much wider than cloud DCs. Fog computing resources can be found in factories, hospitals, restaurants, oil rigs, cell towers, streets and even airplanes [13]. These resources complement resources from the cloud DCs and provide services to sensors and end users. As a result, enterprises can pre-process the data generated from sensors and devices, get immediate insights from the data, and discard unnecessary or redundant data, thus saving time and money. Similarly, applications that rely on ultra-low response times may function and provide their services while fulfilling the expected quality of experience.

Resources in geo-distributed hybrid, multi-cloud and fog computing environments are organized as a large number of independent, medium- to small-sized clusters that collaborate to provide services to sensors and end users [30]–[32]. Taking fog computing as a representative multi-cluster environment, Figure 1.2 shows the architecture consisting of multiple workload clusters coordinated by a management cluster. Unlike centralized cloud

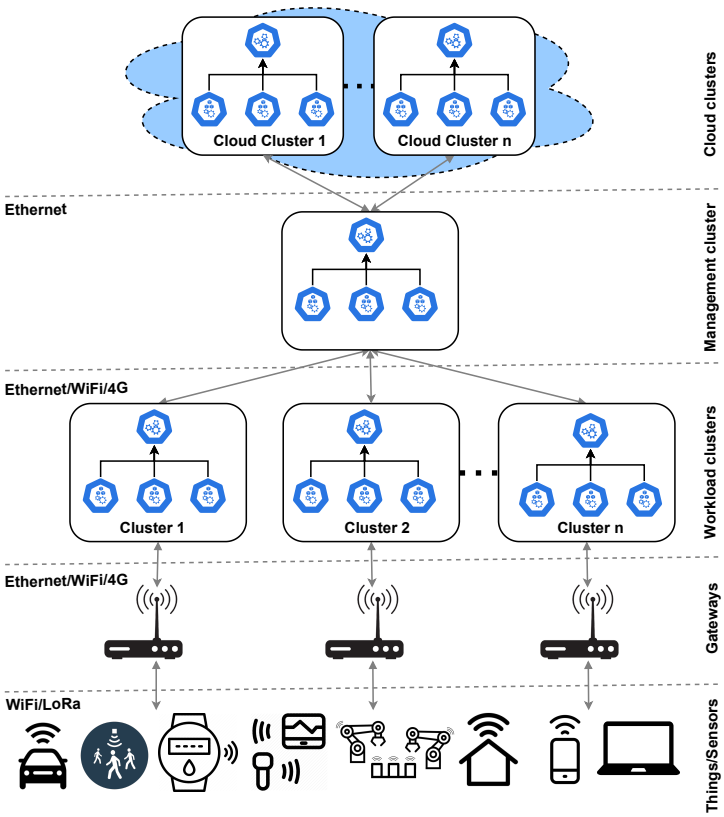


Figure 1.2 – Architecture of a typical geo-distributed computing platform. Resource management and application deployment on the workload clusters is done from a management cluster situated in a higher hierarchy or in the cloud. Different networking technologies are used for communication between the management cluster and the workload clusters.

computing deployments, geo-distributed computing is characterized by wider geographical distribution, heterogeneity in resources, limited capacity of resources and predominant use of wireless networking technology between the end-user devices and the workload clusters [9]. Individual workload nodes may have small to large compute capacity. For instance, devices such as Raspberry Pi's are often used to prototype fog computing nodes, whereas hybrid and multi-cloud environments may have medium- or large-sized nodes [33], [34]. However, because of the large number of workload nodes and wider geographical distribution, geo-distributed computing provides a large amount of aggregate compute and storage capacity. Even though workload nodes may be interconnected using high-speed wired network within a cluster, they are often connected to other workload clusters and to the cloud using a variety of networking technologies including ethernet, WiFi, cellular and satellite [13] that may be characterized by intermittence, high latency and low bandwidth.

Enterprises that rely on geo-distributed computing resources to deploy their applications and deliver their services to their customers need to manage the resources to ensure

high performance and availability of the services while reducing the cost of running the infrastructure. Some of the resource management challenges that arise are resource provisioning, scheduling, allocation, scalability, availability, configuration and optimization [35]. As the scale and heterogeneity of servers and applications grows, so does the management complexity of the systems. At large scales, it becomes expensive, time consuming, error prone and even impossible for human operators to manage the systems manually [36], [37]. The concept of autonomic computing has therefore been proposed to allow systems manage themselves according to high-level objectives defined by system administrators. Inspired by the autonomic nervous system of the human body, autonomic computing relies on actuating or changing the system parameters based on information gathered by continuously sensing (monitoring) the system and its environment. [38]–[43].

Modern geo-distributed applications are composed of multiple self-contained components called microservices that are deployed in large numbers, require fast starting, scaling and upgrading. This paradigm shift has resulted in the emergence of containers such as Docker as an application packaging and delivery mechanism as well as a unit of deployment. Containers allow packaging of microservices along with all their dependencies as self-contained and isolated entities that communicate with other microservices using *Application Programming Interfaces* (APIs). This has brought about increased portability of microservices across DCs, servers and *Operating Systems* (OSs). Moreover, containers allow individual microservices to start, restart, upgrade and scale independently of other microservices [44], [45].

Although the emergence of container technology as an application-packaging mechanism as well as an application-deployment unit has made packaging and deploying microservices easier, the challenge of resource management has become even more difficult [46]. As containers are deployed repeatedly in large numbers across multiple hosts and may fail often, the need for autonomic deployment, self-healing and scaling mechanisms has never been higher. As a result, container orchestration platforms such as Docker Swarm [47], Marathon [48] and Kubernetes [49] were developed to rise to the challenge.

Kubernetes, which was originally developed inside Google as an open-source version of their in-house container orchestration platform Borg [50], has grown in popularity quickly and has now become the most popular container orchestration platform for managing resources in private DCs as well as in the cloud [44]. By hiding the heterogeneity in the underlying hardware, OSs and networking, Kubernetes gives the appearance of a unified computing platform. Moreover, Kubernetes achieves true portability by exposing the same

interfaces irrespective of where the clusters are deployed. Following Kubernetes' success in the cloud, *Kubernetes Federation* (KubeFed) has been proposed for automating application deployment and resource management in multi-cluster environments [51]. Building upon Kubernetes, KubeFed introduces the abstractions and building blocks that can be used to build container orchestration platforms for multi-cluster environments.

In this thesis, we address three challenges in resource management and application deployment in geo-distributed computing environments.

First, autonomic resource managers such as Kubernetes are expected to provide mechanisms by which applications maintain a stable *Quality of Service* (QoS) despite fluctuations in the number of users and requests. One approach to achieve this is through dynamic addition and removal of resources in response to changes in the number of requests. To this end, several horizontal and vertical autoscaling mechanisms have been proposed [52]–[59]. Horizontal autoscaling allows adjusting the number of VMs or containers, whereas vertical autoscaling allows adjusting the amount of resources allocated to a VM or a container at run time.

One limitation of using horizontal autoscalers from the state of the art is that additional VMs are selected from a homogeneous pool of VMs that all have the same amount of CPU and RAM [53]. This works well for clusters that are used to deploy workloads with similar resource requests. In this case, the right size of the VMs can be selected by operators of the system a priori, and a pool with the same size of VMs can be used when scaling horizontally. However, many applications such as big data, machine learning, IoT or scientific computing are made up of workloads with heterogeneous resource requests [60]–[62]. These kinds of workloads generally require different types of VMs with heterogeneous resource allocations optimized for memory, computation or storage [63]. Therefore, it is not possible to know in advance the size of the VMs that can fulfill the resource requirements of all workloads.

In container orchestration platforms such as Kubernetes, autoscaling is performed at two levels, i.e., the application (container) level and the infrastructure (VM) level [64]. For containerized workloads that exhibit wide diversity in the resources allocated to them, it is important to select the right size of VMs for efficient bin-packing of containers onto VMs and to avoid over- or under-provisioning. Moreover, this has implications on costs as the major public cloud providers charge their customers by the amount of resources allocated to their VMs.

On the other hand, CSPs offer a wide variety in VM sizes that can be exploited to optimize for cost or provisioning efficiency [53]. Moreover, Kubernetes and major cloud providers have recently introduced support for horizontal autoscaling with multiple VM pools [65]–[67]. Therefore, it is important to evaluate the benefit of using multiple VM sizes during autoscaling and understand their impact on cost and provisioning accuracy.

The second issue addressed in this thesis is stability in geo-distributed multi-cluster deployments. Geo-distributed multi-cluster federations are usually managed from a control plane that may be located in a higher hierarchy of the network or in the cloud. From this central control plane, human operators define deployment objectives for their applications such that the scheduler can deploy, allocate resources for and scale applications at a regional or global scale. Several networking technologies are used for communication between the management and workload cluster layers, which are often characterized by high latency, low bandwidth and intermittent connectivity that lead to long message delays and high probability of message loss [13].

The fact that several implementations of distributed systems such as KubeFed come with static configuration parameters, coupled with the nature of the network, sometimes leads to premature timeout of control messages [68], [69]. In [70], using KubeFed as a reference platform for geo-distributed multi-cluster federations, we show that multi-cluster application deployments suffer from instability due to premature timeouts of control messages from the control plane to the workload clusters. This, in turn, affects the availability and performance of the applications deployed on the multi-cluster platform. Therefore, the control plane needs to mitigate this problem adaptively to ensure the stable functioning of the overall multi-cluster computing system.

The last issue addressed by this thesis is the orchestration of containerized applications in geo-distributed multi-cluster environments. State-of-the-art resource management platforms for multi-cluster deployments, such as KubeFed, are good candidates for geo-distributed multi-cluster computing environments because they allow managing multiple geo-distributed resource clusters from a single management cluster. For example, KubeFed introduces the necessary concepts, abstractions and application model needed to manage applications across multiple clusters. It also proposes a scheduling mechanism for multi-cluster applications. However, this scheduling mechanism is largely manual and lacks the policy-rich higher-level scheduling mechanisms that are required in geo-distributed computing environments where several types of applications are deployed and several use cases reside. In [71] using KubeFed as a reference resource management platform

in geo-distributed computing environments, we show that the limited manual scheduling mechanism results in resource fragmentation where some of the workload clusters are over-utilized and some others are under utilized, while application containers wait for resources to be released in the over-utilized clusters. Moreover, bursting and dynamic provisioning mechanisms are required to provision resources from the public cloud when the workload clusters run out of resources [72].

Although several attempts have been made in the research community to address the resource management problem in geo-distributed computing environments [73], [74], most of the work focuses on placement algorithms and simulation-based evaluations. As a result, currently there are not many autonomous container orchestration platforms for geo-distributed computing environments that have been implemented and experimentally evaluated. In contrast, we propose and evaluate in a real decentralized testbed, an autonomous container orchestration platform for multi-cluster geo-distributed computing environments with rich scheduling and autoscaling policies to make the most efficient use of the geo-distributed resources. This work demonstrates the feasibility of platforms for geo-distributed container orchestration, and hopefully inspires other researchers to evaluate their proposed tools in real systems.

1.1 Contributions

We propose three contributions that address the challenges in application deployment and resource management of containerized applications in geo-distributed multi-cluster environments. Our contributions aim at bringing state-of-the-art resource management solutions for multi-cluster containerized applications closer to being mature application deployment and resource management platforms for geo-distributed multi-cluster computing environments. Because of its rich set of APIs, simple design and ease of extension, we validate our contributions in Kubernetes and its ecosystem. However, we argue that our contributions can be generalized and extended to work on other current and future container orchestrators.

1. **Evaluation of the performance of Kubernetes cluster autoscaler**

   Autoscaling is one of the widely studied resource provisioning topics in cloud computing. Kubernetes, arguably the most popular cloud platform today, has introduced autoscaling mechanisms at the container and infrastructure levels. The Kubernetes *Cluster Autoscaler* (CA) that scales the infrastructure can be configured to select

nodes either from a single node pool (CA) or from multiple node pools (*Cluster Autoscaler with Node Auto-Provisioning* (CA-NAP)).

In this contribution, we evaluate and compare these configurations using two representative applications and workloads on *Google Kubernetes Engine* (GKE). We report our results using monetary cost and standard autoscaling performance metrics (under-and over-provisioning accuracy, under-and over-provisioning timeshare, instability of elasticity and deviation from the theoretical optimal autoscaler) endorsed by the *Standard Performance Evaluation Corporation* (SPEC) Cloud Group.

Our results show that, overall, CA-NAP outperforms CA and that autoscaling performance depends mainly on the composition of the workload. We compare our results with those of the related work and point out further configuration tuning opportunities to improve performance and cost-saving.

2. **Improving stability in a geo-distributed multi-cluster computing environment**

   In geo-distributed multi-cluster computing environments, the workload clusters are dispersed in different geographical regions, whereas the management cluster could be located in a central location such as a cloud DC. The network infrastructure between the management and workload clusters exhibits heterogeneity in the type of networking technology, latency and bandwidth. As a result, delays and transient network failures are common between the management layer and the remote workload clusters.

   In this contribution, we show that delays and transient network failures coupled with static configuration, including the default configuration parameter values, can lead to instability of application deployments in KubeFed, making applications unavailable for long periods of time. To address this problem, first, we evaluate the impact of the different configuration parameters of the KubeFed control plane on the stability of the system. Next, we identify the configuration parameter that has the biggest impact on the stability of the system. Lastly, we design, implement and evaluate a feedback controller to dynamically adjust the concerned configuration parameter to improve the stability of application deployments without slowing down

the detection of hard failures.

We evaluate the effectiveness of our controller in a geo-distributed setup where a management cluster and five workload clusters were deployed across five sites of Grid'5000. Our results show that using our controller the stability of the system is improved to 99.5–100% as opposed to 83–92% with no controller.

3. **Container orchestration for geo-distributed multi-cluster computing environments**

   Geo-distributed deployments suffer from resource fragmentation, as the resources in certain locations are over-allocated while others are under-utilized. Orchestration platforms such as Kubernetes and KubeFed offer conceptual models and building blocks that can be used to build integrated solutions that address the resource fragmentation challenge. However, the orchestration platforms in the state of the art offer very few and often manual placement policies and lack automated placement mechanisms with multiple placement policies that maximize the utilization of geo-distributed resources. On the other hand, the solutions proposed from the research community in resource provisioning and allocation focus mainly in placement algorithms and simulation studies. As a result, there is a lack of fully autonomous container orchestration platforms for geo-distributed computing environments that have been evaluated experimentally in real testbeds.

   In this contribution, we propose *mck8s* – an orchestration platform for multi-cluster applications on multiple geo-distributed Kubernetes clusters. It offers controllers that automatically place, scale and burst multi-cluster applications across multiple geo-distributed Kubernetes clusters. *mck8s* allocates the requested resources to all incoming applications while making efficient use of resources. We designed *mck8s* to be easy to use by development and operation teams by adopting Kubernetes' design principles and manifest files.

   We evaluate *mck8s* in a geo-distributed experimental testbed in Grid'5000. Our results show that *mck8s* balances the resource allocation across multiple clusters and reduces the fraction of pending pods to 6% as opposed to 65% in the case of KubeFed for the same workload.

1.2 Published papers

The following papers have been published:

1. *An Experimental Evaluation of the Kubernetes Cluster Autoscaler in the Cloud*, **Mulugeta Ayalew Tamiru**, Johan Tordsson, Erik Elmroth, and Guillaume Pierre, 12th IEEE International Conference on Cloud Computing Technology and Science (CloudCom), 2020.
2. *Instability in Geo-Distributed Kubernetes Federation: Causes and Mitigation*, **Mulugeta Ayalew Tamiru**, Guillaume Pierre, Johan Tordsson, and Erik Elmroth, 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), 2020.
3. *mck8s: An Orchestration Platform for Geo-Distributed Computing Environments*, **Mulugeta Ayalew Tamiru**, Guillaume Pierre, Johan Tordsson, and Erik Elmroth, 30th International Conference on Computer Communications and Networks (ICCCN), 2021.

1.3 Organization of the thesis

This thesis is organized into 7 chapters.

In Chapter 2, we provide the technical background of the thesis. We start by providing an overview of cloud computing, cloud deployment models, cloud service models and cloud architecture. Next, we discuss the limitations of the centralized cloud model and the geo-distributed computing models that have emerged to address those limitations. We also introduce the non-functional requirements that lead to the emergence of geo-distributed computing models. Next, we discuss the virtualization technologies that enable cloud computing and geo-distributed computing. Finally, we present Kubernetes – the popular open-source container orchestrator – and its ecosystem, on which we rely for validating and evaluating our contributions.

Chapter 3 presents the state of the art in academic research as it relates to our contributions. First, we review the literature on autoscaling, autoscaling performance evaluation techniques and metrics used to quantify autoscaling performance. Next, we explore the literature on automatic configuration tuning where we identify the negative consequences of misconfigurations, sampling methods, automatic configuration tuning methods with a focus on failure detection and recovery. Finally, we present the literature on the resource

management challenges in geo-distributed computing environments, namely, placement, autoscaling, network traffic routing, dynamic VM provisioning and container orchestration frameworks.

In Chapter 4, we present an experimental performance evaluation of the Kubernetes *Cluster Autoscaler*. We start by describing the experimental testbed in the cloud, the applications and workloads and the metrics used for the evaluation. Finally, we present the results of the evaluation and an in-depth discussion of the results followed by a discussion of further performance enhancement opportunities.

In Chapter 5, we propose a proportional controller that dynamically improves the stability of KubeFed-based geo-distributed computing environments following automatic configuration tuning. First, we experimentally demonstrate the instability problem that arises in KubeFed-based geo-distributed computing environments due to poor networking conditions. We then design and implement a proportional controller that dynamically improves the stability of the system by adjusting the necessary configuration parameter at run-time. Finally, we conclude the chapter by presenting the results of the performance evaluation of our controller.

In Chapter 6, we present *mck8s* – a container orchestration platform for geo-distributed computing environments. We start by demonstrating experimentally the resource fragmentation problem that arises due to the lack of automated policy-based placement mechanisms in KubeFed-based geo-distributed computing environments. Next, we present *mck8s*'s application deployment model. Then, we discuss the system design and implementation details. Finally, we evaluate *mck8s* in a realistic testbed with representative applications and workloads and present the results of the performance evaluation.

We conclude this thesis with Chapter 7. First, we restate the resource management challenges that we address in this thesis. Then, we summarize the main aspects of our contributions and their limitations. Finally, we identify a few perspectives and directions for future research.

Chapter 2

Background

In this chapter, we explore the shift from the centralized cloud computing model to decentralized and geographically distributed multi-cluster computing environments. We discuss the motivations behind this shift, the benefits and the resource management challenges that arise from it. The contributions in this thesis address some of these challenges.

As geo-distributed computing environments are natural extensions of the cloud computing model, we start this chapter by discussing the fundamentals of cloud computing, its architecture, deployment models and service models.

Next, we explain the motivations for geo-distributed multi-cluster environments such as hybrid cloud, multi-cloud and fog computing, the architectures proposed for these types of deployments, and how the geo-distributed multi-cluster architectures address the requirements of the applications that need them.

Then, we discuss the evolution of application deployment and resource management in cloud systems from *Virtual Machine*s (VMs) to containers. We also argue for the need for automatic resource management in geo-distributed computing environments. In addition, we identify the automatic resource management challenges that we address in this thesis and the autonomic management components that we utilize in our contributions to address the resource management challenges.

Furthermore, we discuss Kubernetes and *Kubernetes Federation* (KubeFed), the two orchestration platforms for containerized workloads, that we have used as reference platforms to validate our contributions. We also briefly describe other technologies in the Kubernetes ecosystem that are used in our implementations.

Finally, we discuss our vision of a container orchestration platform for geo-distributed multi-cluster environments that provides generic placement, autoscaling and resource provisioning mechanisms that apply to several applications and use cases.

2.1 Cloud computing

Cloud computing is arguably one of the most consequential developments in Computer Science in the last two decades [75]. Cloud computing has enabled several innovations and has proved to be a strong driving force behind the digital economy. Enabled by advances in hardware and software technologies, broadband internet, economies of scale, and a utility-based business model, cloud computing has emerged as the computing model of choice not only for Internet applications but also enterprise applications. According to the RightScale 2019 State of the Cloud Report that presented results from a survey 786 technical executives, managers and practitioners across a broad cross-section of organizations of varying sizes across many industries, 94% of respondents use cloud computing [76], [77]. On the other hand, the Eurostat statistics from the European Commission that surveyed 146,000 EU enterprises shows that 36% used cloud computing in 2020, a 12 percentage point increase from 2018 [78].

Over the years, advances in computer hardware technologies have made servers with high computing power increasingly affordable [79]. Moreover, purchasing servers at the scale of *Data Center*s (DCs) offers significant cost reductions. Some public *Cloud Service Provider*s (CSPs) such as Google and OVHcloud build their own servers, which at the scale of very large DCs that host hundreds of thousands of servers, offers significant reductions in cost [80].

On the software side, the GNU/Linux *Operating System* (OS) has emerged as the OS of the Internet because of its free and open-source nature as well as its high modularity, performance and power efficiency [81]. Moreover, the licensing model of most GNU/Linux distributions reduces licensing fees.

The other critical development in software and arguably the main enabler of cloud computing is virtualization technology [79], [82], [83]. Virtualization has made cloud computing possible by abstracting the underlying resources such as processing, memory, storage, networking and OSs so that multiple tenants can coexist on a shared physical infrastructure. It simplifies automated provisioning and lifecycle management of resources. Moreover, virtualization improves server consolidation and utilization, making cloud services affordable at large scale [80], [84]–[86].

What is equally important as the technological innovations that enabled cloud computing is the business model that allowed computing to be available as a utility like other utilities such as electricity, gas and water [87], [88]. Cloud business models have allowed

consumers to acquire compute power without the upfront capital expenditures but paying only for the amount used. This has enabled enterprises to focus on their core business processes rather than owning and operating DCs and hiring the human power necessary to operate the infrastructure.

2.1.1 Cloud deployment models

Cloud providers follow different cloud deployment models that differ mainly by the ownership of the cloud resources and whether the cloud service is reserved for few private entities or available for the general public.

Cloud deployment models can be categorized into four main groups, namely private, public, hybrid and community clouds. According to the Flexera State of the Cloud Report 2021, enterprises that use cloud computing run 50% of their workloads in the public cloud. Moreover, among all the respondents, 82% follow the hybrid cloud approach, whereas 10% use multiple public clouds [89].

A public cloud offers its services to the general public via different pricing models such as usage-based fixed pricing, subscriptions or reserved service contracts [90]. The cloud resources are owned and managed by public CSPs and may be located in the cloud DCs in different regions of the world owned by the public cloud provider. The public cloud offering is interesting for enterprises which want to outsource their *Information Technology* (IT) operations and focus on their core business. In addition, if the demand for applications is highly variable or not known in advance, the public model is ideal to benefit from the elasticity offered by it. Some other applications such as temporary batch analytics, *Machine Learning* (ML) model training and scientific applications may also benefit from provisioning cloud resources only for the needed duration. Moreover, the public cloud model is suitable for startups because it lowers the barrier of entry into business by significantly reducing capital expenditure. However, although it is widely believed that the public cloud model is cheaper than the private alternative, pricing models might be complex and costs may become unpredictable in some instances [91], [92].

In the private cloud model, enterprises own all the cloud infrastructure which may be located in their DC or a co-location center. This model is interesting for enterprises that are concerned about security and data privacy, and would like to have a higher degree of control and customization. Some enterprises in regulated industries such as finance and healthcare need high levels of data security and are good candidates for the private cloud.

The private cloud model might be more expensive than the public model especially if the utilization of the DC is low [93].

Moreover, building and operating a cloud DC can be a daunting task. For example, in 2015 *The Guardian* [1] announced its decision to move all its IT operations to *Amazon Web Services* (AWS) because of the significant challenges faced while operating a private DC based on OpenStack [2] [94]. Although owning a private cloud DC may be beneficial in terms of data privacy, control and flexibility, it incurs significant costs [95]. The capital expenditures include not only the servers but also network equipment, storage devices, and infrastructure costs such as electric power, cooling, physical security and building. Moreover, software license fees and the salary of human experts should be considered. The operating expenditures include maintenance, upgrades and depreciation costs. Servers and equipment need to be maintained and replaced every few years. Moreover, software needs to be upgraded, patched for security, and license fees paid. Therefore, for enterprises to choose the private cloud avenue, the reasons not related to cost should outweigh the cost of building and operating a cloud DC.

In the community cloud model, cloud resources are shared exclusively between a group of organizations. For instance, collaborating universities and research institutes within a country may use this model. The resources are owned and managed by one or more of the organizations, or a third party, and may be located in the premises of one or more of the organizations. This model allows organizations to enjoy the benefits of cloud computing with a high level of control, security, privacy and flexibility while sharing the cost of owning and maintaining the DCs. The community cloud model is similar to grid computing in its vision and architecture but different in various aspects such as programming model, compute model, service model, business model, applications and abstractions [96]–[98].

The hybrid cloud model is a combination of two or more of the models discussed above. In most cases, organizations which operate private clouds acquire resources using the public cloud model to offload additional load that cannot be met by resources in the private cloud [99]. This allows organizations to allocate extra resources without the need to invest in purchasing more hardware and software.

As geo-distributed multi-cluster deployments are achieved by combining resources provisioned according to the different cloud deployment models discussed above [100], the resource management problems that we study in this thesis and the contributions we

1. The Guardian – https://www.theguardian.com/
2. OpenStack – https://www.openstack.org/

propose to address these problems apply to all four of the cloud deployment models. We study autoscaling in a public cloud environment in Chapter 4, stability of geo-distributed multi-cluster deployments in a private or community cloud in Chapter 5, and container orchestration in geo-distributed hybrid clouds involving private and public cloud models in Chapter 6.

2.1.2 Cloud service models

One of the differentiating factors between cloud computing and other computing models such as grid computing is the cloud service model that allows users to consume cloud resources at different levels of abstraction [101], [102]. Cloud service models play an important role in geo-distributed deployments by making it easier to acquire resources across different cloud providers.

Cloud resources can be consumed using different service models such as *Infrastructure-as-a-Service* (IaaS), *Container-as-a-Service* (CaaS), *Platform-as-a-Service* (PaaS) and *Software-as-a-Service* (SaaS). The most common model consists of acquiring VMs and other forms of virtualized hardware such as storage and network from cloud providers, which is referred to as IaaS. When using IaaS customers have the freedom to deploy their preferred OS and software stack on the servers they acquired from the CSP, but are responsible for managing the servers. Therefore, it is important to design the infrastructure well and monitor the usage of the resources from the IaaS provider to maximize performance, while avoiding over- and under-provisioning and overspending.

Enterprises that want to deploy their applications in the cloud with the flexibility of choosing their preferred programming language and framework but without the burden of managing VMs themselves, can choose the PaaS or the CaaS service models, where the virtualized hardware is managed by the CSP. At the other end of the service models spectrum, the SaaS allows consuming full applications such as email, word processing, file storage, or business applications without the need to manage any infrastructure, platform or containers. In many cases, enterprises use a combination of cloud service models. For instance, an enterprise may use IaaS for its business-critical applications and use SaaS for email or *Enterprise Resource Planning* (ERP) applications.

In this thesis, we study resource management problems mainly in the IaaS and CaaS service model layers of geo-distributed multi-cluster deployments. Particularly, we study autoscaling and stability problems in a CaaS model in Chapters 4 and 5, and geo-

distributed container orchestration problems in an environment that involves both CaaS and IaaS in Chapter 6.

2.1.3 Cloud architecture

Unlike local applications, those in the cloud are accessed over the network, usually over the Internet. This is because the cloud servers that host the applications are usually found in remote DCs operated by the CSPs. In many cases, the cloud DCs are found in another country or even another continent than where the customer is located. However, advances in broadband internet technology allow accessing most cloud applications without major issues. The architecture of a typical cloud DC is shown in Figure 2.1.

A cloud DC may house tens or hundreds of thousands of servers that are organized in clusters and racks [103], [104]. The servers in a rack are interconnected using high-speed network switches in the access layer, with uplinks to switches in the aggregate layer that interconnects different clusters. Eventually, the cloud servers are connected to the Internet via routers at the core layer, through which they are accessed from the outside by customers and end users. As customers and end users access the cloud servers remotely over the Internet they incur significant network latency in the order of tens to hundreds of milliseconds [7], [16], [105]. Moreover, end users may experience unpredictable network performance over the public Internet. However, since cloud servers in the same DC are interconnected using high-speed networks, the exact cluster and rack of the servers hosting the applications is not relevant for most cloud applications [20].

2.2 Geo-distributed computing models

As shown in Figure 1.1 in Chapter 1, cloud DCs are distributed in different parts of the world. This is driven by the ever-increasing customer demand to deploy their applications in a distributed manner. For instance, a geo-distributed deployment is required by large-scale social media applications that serve dynamic content and need to handle a wide range of demands from different geographical regions [100]. Moreover, general purpose compute DCs are often complemented by a set of geo-distributed cache servers in the form of *Content Delivery Network*s (CDNs) [106]. We discuss below some of the major non-functional requirements that led to geo-distributed application deployments.

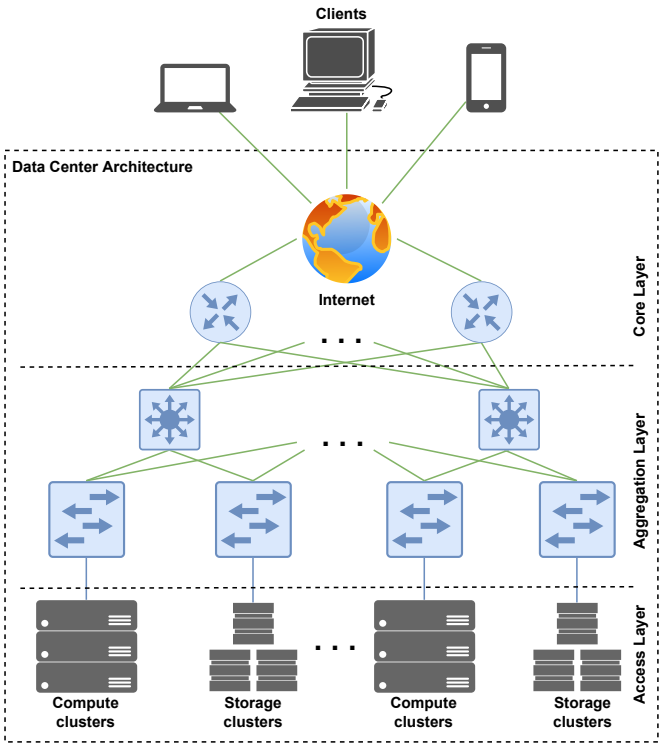


Figure 2.1 – Cloud DC architecture.

— **Proximity:** Since applications deployed on the cloud are accessed over the Internet, the location of the cloud DC that hosts the application has an impact on the response time perceived by end users. It is reported that, in general, end users experience response times of 20 ms - 150 ms when accessing cloud applications over different types of networking technologies [7], [105]. As a result, many application service providers deploy their applications in several cloud DCs depending on the concentration of their end users to be closer to end users and reduce user-perceived latency [21]–[23]. In doing so, they satisfy their end users expectations and improve their profitability. Similarly, CSPs are constantly growing their reach by building DCs in different parts of the world.

— **High availability and disaster recovery:** Modern Internet services are expected to achieve high degrees of availability, typically at least "four-nines" or 99.99% of uptime [80]. Any downtime could lead to business losses in the form of lost sales. One way of assuring a highly available service is replicating the different components of a service not only in the same DC but also across multiple DCs in different regions [19]. This ensures that the service continues to serve users even in the event of human-

induced or natural disasters in one or more DCs. Recently, a fire that broke out in one of the DCs of OVHcloud, the largest European cloud provider, destroyed the DC, and partially damaged another one [107]. Two more DCs in the region were temporarily shut down. This incident affected thousands of customers and millions of services. However, OVHcloud was able to restore much of the services in its remaining DCs and to move its clients to other sites [108].

— **Scalability:** Some applications may need to scale vertically or horizontally, ending up using a large number of resources. This may happen due to spikes in user traffic, batch jobs, upon ML model training, or as part of scientific applications. However, in some cases, especially in private DCs, the cloud DC hosting the application may run short of resources. The increasing number of DC locations also leads to resource fragmentation where each individual private cloud DC may contain only a modest amount of resources. In such cases, additional resources may be acquired from another DC and the applications may be allowed to over-flow to the new DC. This situation is referred to as cloud bursting. On the other hand, some systems suffer from performance degradation on a large scale. For instance, Kubernetes may run into performance degradation and other issues when scaled to thousands of nodes [109]. One way of mitigating this problem is by running multiple clusters in different cloud DCs.

— **Compliance with regional regulations:** Regulations in some countries, regions, or business domains require businesses to store and process the personal data of their customers in a private DC, or in a DC located in a certain region, and/or operated by a vendor incorporated under certain local legislation [24]. Therefore, to comply with these regulations, enterprises may store and process user data or deploy sensitive applications in DCs in certain regions and deploy the less sensitive applications in other regions. For instance, the data sovereignty provision of the *General Data Protection Regulation* (GDPR) requires that service providers store all data collected on *European Union* (EU) citizens either in the EU or a jurisdiction that has similar levels of protection [110]. Other countries also have similar legislation. Therefore, service providers need to offer multiple DC locations and the corresponding data orchestration mechanism to comply to local regulatory requirements.

— **Vendor lock-in avoidance:** Relying heavily on specialized services from a single public cloud provider might be costly in monetary terms or in migration time if an enterprise decides to migrate its application and data from one public cloud provider

to another or a private DC [111], [112]. For instance, after acquiring Instagram in 2012, it took Facebook about a year to migrate Instagram's photo-sharing service from AWS to Facebook's DC [113]. To reduce the time or effort to migrate services between DCs, and to stay vendor-neutral, some enterprises run applications in or migrate them between multiple private or public DCs from different providers.

Driven by the need to satisfy these requirements, different deployment models for geo-distributed computing environments have emerged. In particular, three major trends could be identified, namely hybrid cloud, multi-cloud and fog computing.

The resource management problems that we study in this thesis and our contributions to addressing these problems apply to all three geo-distributed computing environments.

2.2.1 Hybrid cloud

A hybrid cloud deployment integrates resources from a private DC and one or more public clouds such as Microsoft Azure, Google Cloud and AWS. The main drivers of hybrid cloud are scalability, availability, disaster recovery and regulatory compliance [114]. A hybrid cloud deployment leverages the benefits of the private and public cloud deployments: benefit from flexibility, scalability and availability of the public clouds while capitalizing on the security, controllability and customizability of the private cloud [115].

A hybrid cloud deployment can be achieved by creating a secure network link such as *Virtual Private Network* (VPN) between the private DC and the public cloud DCs. The major public cloud providers have a service called *Virtual Private Cloud* (VPC), which is a pool of resources isolated from other tenants' resources, and to which an enterprise can connect their private DC securely. Alternatively, some of the major public CSPs have hybrid cloud offerings such as Google Anthos [3] and AWS Outposts [4], which are managed services that extend the services of the public cloud to the private DC. Some enterprises may choose to design their hybrid cloud implementations using open-source technologies to avoid relying on specialized services such as Anthos and Outposts from the public cloud providers. For instance, container technologies such as Docker and container orchestration platforms such as Kubernetes can be used [116].

To fully deliver on their promises, hybrid cloud deployments must address a number of difficult resource management issues such as transparent cloud bursting, efficient autoscaling and automated network traffic routing [115].

3. Google Anthos – https://cloud.google.com/anthos
4. AWS Outposts – https://aws.amazon.com/outposts/

Transparent cloud bursting allows provisioning resources from the public cloud automatically without human intervention to handle workload spikes that require more resources than available in the private DC. This requires continuous monitoring of the private DC resources to forecast or identify the number of resources to be provisioned from the public cloud and the right moment to provision those resources. Moreover, the right placement algorithms should be chosen to schedule services in the public cloud. This might involve the movement of data between the private and public clouds, for which efficient methods need to be sought [117]. Moreover, scheduling services and migrating data to the public cloud is not sufficient in itself as user traffic needs to be routed to the public cloud and load-balanced between the private and public clouds. Lastly, to avoid unnecessary spending when the amount of workload has decreased to the point where all workloads can be handled by the private cloud alone, there should be mechanisms to migrate applications and data back to the private cloud, reroute network traffic, re-configure load balancers, and eventually tear down resources provisioned in the public cloud.

2.2.2 Multi-cloud

A multi-cloud deployment integrates resources from multiple zones or regions of a single or multiple public cloud provider(s). This deployment may be motivated by any of the requirements discussed above. Using multi-cloud deployments, enterprises can benefit from the proximity that can be gained by using the combined presence of multiple public cloud providers in different regions [21]–[23], [25]. Furthermore, they can leverage the specialized services and different pricing schemes offered by multiple CSPs [118].

The most straightforward multi-cloud approach is to use the multi-zone or multi-region solutions offered by a single public cloud provider and deploy in multiple zones or regions of that cloud provider. For example, major public cloud providers Microsoft Azure and Google Cloud have multi-region offerings [114], [119]. The upsides of this approach are easy and fast deployment thanks to the same *Application Programming Interfaces* (APIs), VM types, pricing schemes and other services. Moreover, enterprises can benefit from secure, highly available and high-speed network connections between the different zones or regions. For instance, Google Cloud and AWS have deployed high-speed fiber cable and undersea cables to interconnect their DCs from which their customers can benefit [120], [121]. The major downside of this approach is a high dependency on a single cloud provider, which can have impacts in terms of vendor lock-in and possible downtime

in case of disasters. Some other downsides could be missing out on better locations, pricing schemes and specialized services offered by other cloud providers.

The other multi-cloud approach is combining resources and services from multiple cloud providers. To make this approach a reality, it is necessary to use third-party solutions such as IBM multi-cloud management platform [5], Rancher [6], Crossplane [7], or open-source solutions such as KubeFed [8]. The major advantage of this approach is the flexibility that arises from being vendor-neutral, which makes it easy to deploy and redeploy applications across different platforms. On the downside, different sets of management tools, APIs, pricing schemes and VM types make it difficult and time-consuming to make this approach a success. Moreover, enterprises which choose to go on this avenue need to interconnect their resources across different providers over the public Internet, and, therefore, need to set up secure tunnels such as VPNs.

Some of the resource management challenges in multi-cloud environments are resource provisioning and orchestration across multiple cloud administrative domains, efficient network traffic routing, proximity-aware placement, proximity-aware autoscaling, cost optimization and right-sizing of VMs, some of which overlap with those of the hybrid cloud scenario discussed in Section 2.2.1.

2.2.3 Fog computing

Mainly driven by proximity and privacy concerns, fog computing deployments are far more distributed than hybrid and multi-cloud deployments and may encompass private clouds, public clouds and resources at the edge of the network closer to the end users [9], [10].

The reason for wider geographical distribution in fog computing is the need to fulfill the requirements of applications that require ultra-low latency, high bandwidth, and uninterrupted network connectivity between end users / devices and workload cluster nodes by placing computing, storage, and networking resources closer to end users and end devices [9], [20], [122]. Some modern applications such as autonomous vehicles and *Virtual Reality* (VR) require consistent ultra-low latency between end users and the cloud instances serving them below 10 ms [7]. Some other applications such as *Internet of Things*

5. IBM Multi-Cloud – https://www.ibm.com/services/cloud/multicloud/management
6. Rancher – https://bit.ly/3oLC18T
7. Crossplane – https://crossplane.io/
8. KubeFed – https://github.com/kubernetes-sigs/kubefed

(IoT) and video analytics generate a large amount of data that would congest the network links and could be costly to send over long-range network links to cloud DCs for processing. Other business-critical applications in restaurants, oil rigs and healthcare facilities cannot tolerate network partitions even though they can tolerate network delays [11], [13]. As these requirements cannot be met by cloud computing platforms because of their location and speed-of-light limitations, fog computing has emerged to accommodate these requirements by placing resources closer to the devices and end users [9], [10].

Although fog computing is a hot research topic at the moment, there are no large-scale public fog deployments to date. However, there are initiatives by major CSPs to deploy their managed hardware and software at the edge of the network outside their DCs [123], [124]. There are also initiatives such as MobiledgeX to integrate public clouds, Telco clouds and 5G technology to deliver resources at the edge of the network [125]. Enterprises which would like to deploy their own fog computing infrastructure would have to use third-party or open-source solutions such as Kubernetes, KubeFed, or KubeEdge [9] to integrate resources from private and public clouds as well as resources at the edge [126]. Kubernetes-based systems are currently the most popular choice for prototyping fog computing deployments [20], [122], [127]–[130].

The main resource management challenges in fog computing environments are resource provisioning and orchestration in a wide geographical area, proximity-aware network traffic routing, proximity-aware placement, proximity-aware autoscaling and cost optimization, some of which overlap with those of the hybrid cloud and multi-cloud scenarios discussed in the previous sections.

This thesis addresses a number of resource management challenges in geo-distributed cloud environments. Specifically, we study cloud autoscaling issues in Chapter 4, stability of geo-distributed cloud deployments in Chapter 5, and orchestration of containerized application in geo-distributed multi-cluster environments, transparent cloud bursting, and de-provisioning of cloud resources in Chapter 6.

2.3 From virtual machines to containers

As discussed before, virtualization is one of the main enablers of cloud computing. Virtualization allows multi-tenancy, improved server consolidation and improved resource

9. KubeEdge – https://kubeedge.io/en/

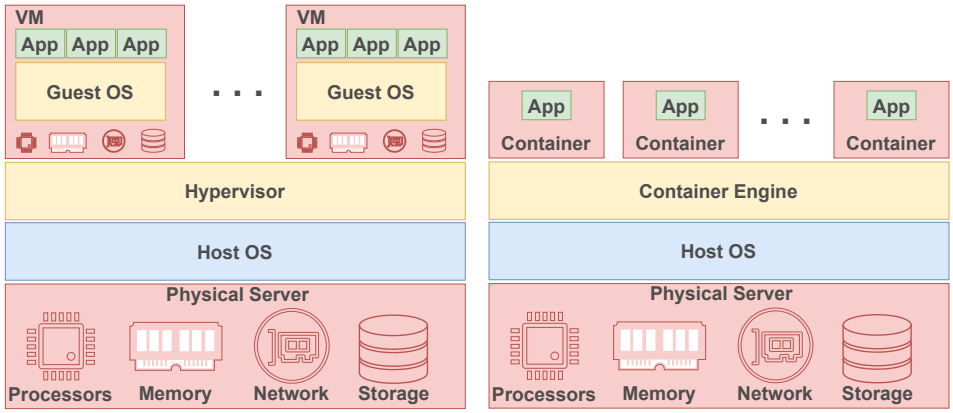
utilization by abstracting processing, memory, networking, storage and OS from applications and end users [82]. There are two main types of virtualization, namely, hardware-level virtualization and OS-level virtualization. These two techniques offer slightly different benefits and often complement each other [131].

Hardware virtualization revolutionized computing by allowing running different OSs in VMs in the same *Physical Machine* (PM) as is common in all the cloud providers. A hypervisor or *Virtual Machine Monitor* (VMM) is used in hardware virtualization to allocate, control and multiplex physical resources for the VMs. Some examples of hypervisors include KVM [10], Xen [11], Microsoft Hyper-V [12] and VMware ESXi [13]. The major public cloud providers rely on modified versions of these hypervisors to enable their cloud services. For instance, Google Cloud uses a modified version of KVM [132], AWS uses a combination of Xen and KVM [133], whereas Microsoft Azure relies on the Azure hypervisor system that is based on Windows Hyper-V [134].

In IaaS clouds today, resources are provisioned primarily in the form of VMs to which the desired amounts of *Virtual Central Processing Unit* (vCPU) and memory are allocated. As shown in Figure 2.2(a), traditional cloud applications are usually based on a monolithic architecture and are deployed on VMs along with all their dependencies and a suitable OS. Load balancers may be used to expose the applications to the end users. To make the deployment of the applications repeatable and for scaling, images of the VMs have to be maintained. Configuration management tools such as Chef [14], Puppet [15] and Ansible [16] are used to automate the installation and configuration of OSs and applications on VMs. However, scaling applications based on VMs is challenging because it takes a considerable amount of time, in the order of minutes, until a VM is provisioned, is fully up, is registered with a load balancer, and starts serving requests.

Deploying applications using VMs in geo-distributed multi-cluster environments poses even more challenges. Because of the differences in the underlying infrastructure of different cloud providers, portability issues may arise where VM images that work in one environment might not work in another [135]. Therefore, it may be necessary to convert VM image formats or prepare VM images for each of the cloud environments the applica-

10. KVM – https://www.linux-kvm.org/page/Main_Page
11. Xen – https://xenproject.org/
12. Microsoft Hyper-V – https://bit.ly/3fNjODN
13. VMware ESXi – https://www.vmware.com/products/esxi-and-esx.html
14. Chef – https://www.chef.io/
15. Puppet – https://puppet.com/
16. Ansible – https://www.ansible.com/



(a) Hardware virtualization. (b) OS virtualization.

Figure 2.2 – Comparison of application deployment using (a) VMs and (b) containers .

tions run in. Another issue is that because they bundle the OS, application dependencies, application data and the application itself, VM image sizes tend to be very large in size, in the order of GBs, which makes it difficult to move them around cloud providers. This may cause significant delays in cloud bursting or migrating applications from one cloud provider to another.

More recently, container technologies such as Docker [17] have gained popularity for application packaging and deployment in the cloud and multi-cluster environments. Containers rely on OS virtualization in which applications or processes run in isolated user spaces. OS virtualization relies on resource isolation features of the Linux kernel such as namespaces and cgroups [136]–[138]. Containers package the application and all its dependencies in an image that can be instantiated in any environment that has the container runtime. As containers share the same OS, as shown in Figure 2.2(b), there is no need to include a full OS image in the container, which significantly reduces the size of the container images and makes them lightweight. Moreover, Docker's incremental approach to develop container images favorizes image layer reuse and reduces the size of images. As a consequence, containers take only a short amount of time, usually in the order of seconds, to start and become ready to accept user requests [138]. Additional techniques may be used to further reduce container startup times [139]–[141]. Moreover, many containers can run inside a VM or PM at the same time.

Although containers are more lightweight than VMs, they provide weaker isolation and security against the noisy-neighbor's effect, which is an anomaly caused by other co-

17. Docker – https://www.docker.com/

located containers interfering in the resource usage of the concerned container [142]. As a result, multi-tenancy is sometimes considered too risky for containers [143]. On the other hand, as the performance overhead of running processes inside containers is negligible as compared to running them directly on the OS [144], the most common practice in public clouds nowadays is to run containerized applications on top of VMs [145]–[147].

The lightweight nature of containers, their fast-starting times and the fact that they can be shared using a container image registry [148] makes them a good application deployment tool in geo-distributed multi-cluster environments. As containers can be launched in any environment that has the underlying container runtime, this breaks the portability barrier and allows running applications consistently in private clouds, public clouds, or at the edge of the network [131]. Moreover, there is no longer a need to migrate a VM containing the application, as the application's container images can be easily downloaded from a container registry. This makes scaling and bursting from one cloud environment to another easier compared to cloud bursting using VMs.

As several containers may run on a single host because of their lightweight nature, there is a high probability of them being shut down or becoming unhealthy simultaneously due to several reasons such as resource contention from other containers [149]. Therefore, there should be mechanisms to continuously monitor and restart them in such situations. On the other hand, the process of starting, scaling and stopping them needs to be automated. The process of automatically scheduling, starting, scaling and stopping containers is called orchestration. In the last few years, several container orchestration platforms have been proposed. Kubernetes, which came out of Google in 2014, has gained popularity very quickly and is the most used platform today [50], [150], [151].

2.4 Automatic resource management

In geo-distributed computing environments, resources should be managed efficiently to improve their utilization, ensure application performance and to reduce costs. To this end, resource management involves dynamically allocating compute, networking and storage resources to a set of applications in a manner that seeks to jointly accomplish the objectives of applications, the service providers and the users of the resources. There are several resource management problems such as discovery, monitoring, estimation, modeling, provisioning, allocation, mapping, placement, adaptation, optimization and brokering [152]–

[154]. Figure 2.3 presents an overview of the elements of resource management in cloud and geo-distributed computing environments.

In this section, we only discuss some of the resource management challenges that we address in this thesis.

Scheduling / Placement: Scheduling / placement is a critical part of any resource management system [155]. A scheduler ensures that a container or VM is placed in a way that satisfies the needs of the user and the constraints of the application and infrastructure provider. A scheduler takes into account several factors such as resource availability and application priority when making placement decisions. In geo-distributed computing environments, a scheduler needs to consider the location of resources and network conditions such as inter-cluster latency in addition to the usual considerations such as resource availability [122], [156]. Moreover, as computing resources in one particular location are often constrained, a scheduler should consider offloading or bursting the application replicas to neighboring resources when those resources are fully used. Offloading may be done horizontally to other neighboring clusters with sufficient resources or vertically to a cloud DC. In the latter case, the scheduler should interact with a provisioning system to transparently and dynamically provision resources from cloud DCs.

Autoscaling: Autoscaling allows applications to provide the expected *Quality of Service* (QoS) by dynamically adjusting the resources allocated to them match the changes in user traffic [157]. An autoscaler continuously monitors the application's resource usage or the amount of user traffic it serves and adjusts the amount of resources allocated to the application to determine an appropriate level which allows it to provide the QoS specified by the user. Autoscaling can be done in two directions (horizontal and vertical) and at the container and VM levels [158]. Moreover, autoscaling systems differ in the metrics they use for making their decisions, their timing, scaling methods and architectures [157]. In geo-distributed computing environments, autoscalers need to take into account the location of resources and network conditions to make their decisions [129]. The autoscaler's decisions are materialized by the provisioner and scheduler.

Provisioning: Scheduling / placement decisions are materialized by the underlying infrastructure by calling the appropriate APIs of cloud providers or open-source systems such as OpenStack and Kubernetes. These systems provision the requested containers and

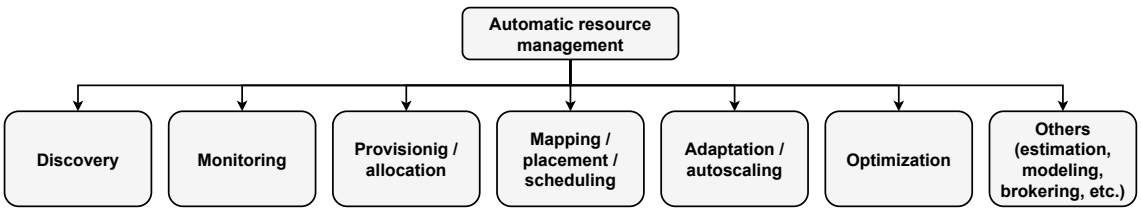


Figure 2.3 – Components of automatic resource management.

VMs on the physical infrastructure using virtualization technology. A resource manager for geo-distributed computing environments should be able to dynamically and transparently provision resources from different cloud providers when the existing infrastructure runs out of resources.

In addition to their distribution and scale, geo-distributed computing environments are dynamic with clusters and nodes being added and removed frequently. These environments are also characterized by changing network conditions, failures, workloads and user traffic. This complexity makes it practically impossible to manage them manually [39], [43]. Therefore, it is necessary to adapt the infrastructure automatically to changing conditions to ensure resource efficiency and meet user's QoS.

The MAPE-K (Monitor- Analyze-Plan-Execute-Knowledge) model is extensively used as a reference architecture for automatic resource management and optimizations in cloud management systems [38], [41]. For example, this approach is extensively used in Kubernetes for building controllers that continuously monitor the system and bring the current state to the desired state expressed by users. Similarly, we have used this approach in our contributions, particularly in Chapters 5 and 6, for automatic resource management in geo-distributed computing environments.

The MAPE-K loop consists of five components as shown in Figure 2.4.

Monitor: Scheduling, placement, autoscaling and provisioning policies rely on an accurate knowledge of the status of the underlying infrastructures. Therefore, it is important to continuously monitor the status of hardware and software resources such as clusters, PMs, VMs, containers and applications. The status information includes metrics such as *Central Processing Unit* (CPU) utilization, memory usage, network traffic, number of requests and rate of request arrival. However, in geo-distributed environments it is also important to monitor inter-cluster network conditions such as network latency and reliability to make network-aware decisions. There are several monitoring solutions for

geo-distributed computing environments including open-source solutions such as *Serf* [18] for monitoring inter-cluster latency and *Prometheus* [19] for monitoring resource usage.

Analyze: The information gathered by the *Monitor* component is processed by the *Analyze* component to determine whether it is necessary to perform placement, scaling or provisioning actions. This component compares the actual status of the system to the desired state and decides which actions can bring the system to the desired state. For instance, it may decide to adjust the number of replicas of an application that are necessary to provide a certain level of QoS. If changes are to be made, the *Plan* component is triggered.

Plan: The *Plan* component is responsible for defining the exact actions to be executed. For instance, for placement this could mean selecting the cluster or node to place containers. For autoscaling, it refers to estimating the number of replicas or amount of resources to be allocated to an applications. For provisioning, it refers to estimating the number of VMs and the amount of resources to be allocated to them. This can be done using rule-based policies, predictive algorithms or utility functions that optimize a given metric.

Execute: The *Execute* component executes the actions decided by the *Plan* component by calling the APIs of the underlying infrastructure. These APIs could belong to orchestrators such as Kubernetes, resource managers such as OpenStack or CSPs. In this thesis, we rely on Kubernetes for container orchestration and *Cluster API* [20] for interfacing with OpenStack.

Knowledge: The *Knowledge* component serves as a database for data that is shared by the different components. In this thesis, we rely on *etcd* [21] which is a stateful key-value store for persisting the results from the other components of the autonomic system. *etcd* is mainly used by Kubernetes for persisting cluster state.

In this thesis, we study the resource management problems in geo-distributed multi-cluster environments based on containerization. Our contributions use Kubernetes as a

18. serf – https://www.serf.io/
19. Prometheus – https://prometheus.io/
20. Cluster API – https://bit.ly/3zy9Lvt
21. https://etcd.io/

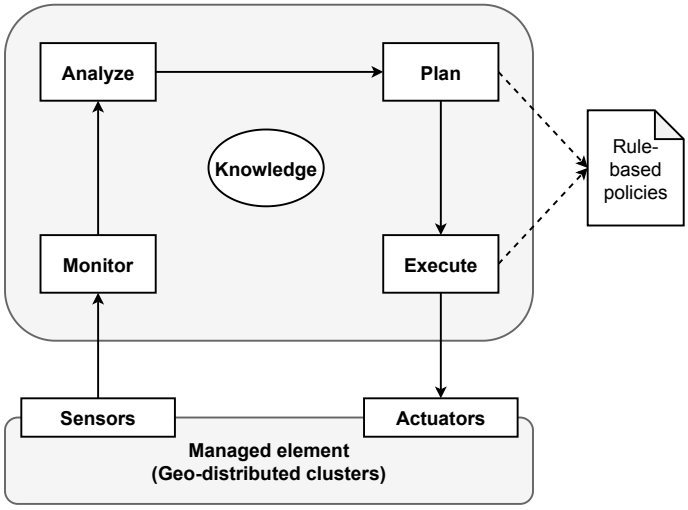


Figure 2.4 – MAPE-K control loop.

reference platform because of its simplicity, rich set of APIs and easy extensibility. Moreover, it is popular and widely accepted by the research community and industry alike. However, adapting these contributions to other orchestrators should arguably be feasible.

2.5 Kubernetes

As we discussed in Section 2.3, challenges related to slow VM start times, differences in VM image formats across different providers and environments, and large VM image sizes have led to the emergence of containers that offer a more lightweight, faster, reproducible, portable and scalable way of packaging and running applications in the cloud. We also mentioned that when containers are deployed at a large scale, as it is usually done in cloud deployments [151], there is a high probability of them crashing at some point in their lifecycle. Therefore, their packaging, deployment, scaling and failure recovery need to be automated. Docker has gained popularity in automating container lifecycle and offering a standard way of packaging and storing them. On the other hand, orchestration platforms that automate the management, scaling and maintenance of containerized applications such as Docker Swarm and Kubernetes have been proposed.

Kubernetes is the most popular open-source container orchestration platform and cluster manager which was inspired by the Borg cluster management system from Google and later donated to the Cloud Native Foundation [50], [159]. In the last few years, Kubernetes has been widely adopted by enterprises for deploying applications in private

DCs, public cloud and hybrid cloud environments. As of November 2020, more than half of enterprises that use containers do so using Kubernetes [151].

2.5.1 Motivations for Kubernetes

In this thesis, we use Kubernetes and its ecosystem for validating our contributions primarily because of certain features that make it possible to extend it into managing geo-distributed computing environments. In addition, its popularity would likely increase the potential adoption of our results.

1. **Consistent API:** Kubernetes presents a consistent API for all clusters whether they are deployed in public or private cloud environments [160]. This makes it easy to provision resources or deploy applications across different providers, and saves time and effort when implementing additional features on top of Kubernetes.
2. **Abstraction:** Kubernetes abstracts the differences between the resources in different cloud providers by standardizing resource types and units [161]. This allows users to focus on their services, and let Kubernetes automate the selection of resources.
3. **Network model:** The Kubernetes network model makes it easy to interconnect containers in one cluster, and can easily be extended to multiple clusters [162], [163].
4. **Portability:** As Kubernetes is based on containerization, it enhances portability across different environments [44], [160]. Services can be easily deployed from a developer's laptop to multiple geo-distributed clusters without significant changes or issues.
5. **Interoperability:** Kubernetes enhances interoperability between different cloud providers and breaks the administrative domain barrier that was a hindrance for prior works in this domain [164].
6. **Extensibility:** Unlike its competitors, Kubernetes is modular in its design and implementation, allowing easy extensions in the form of *Custom Resources* (CRs), custom controllers or operators [165], [166].

These properties of Kubernetes have allowed us to use it as a reference platform based on which we study the resource management problems in containerized geo-distributed multi-cluster environments, and to implement and evaluate our contributions. However, our contributions may arguably apply in other container orchestration platforms such as

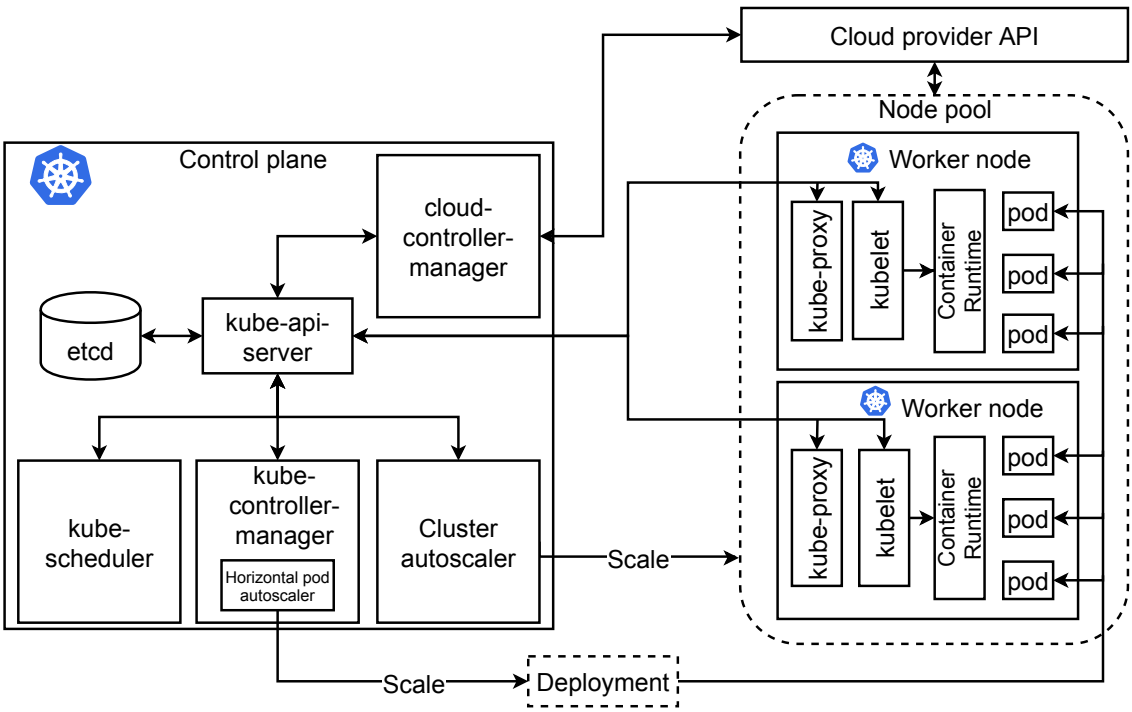


Figure 2.5 – A simplified architecture of Kubernetes. It has two main components, the control plane and worker nodes. The control plan consists of kube-api-server, kube-controller-manager, kube-scheduler, cloud-controller-manager and cluster autoscaler. All cluster state is stored in the etcd key-value store. The worker nodes are responsible for executing application pods and consist of kube-proxy, kubelet and a container runtime such as Docker. The lifecycle of pods is managed by a Deployment controller whereas the horizontal pod autoscaler automatically adjusts the number of replicas. Similarly, the cluster autoscaler is responsible for automatically adjusting the number of worker nodes. Kubernetes nodes are provisioned from infrastructure providers using the cloud-controller-manager via their APIs.

Docker Swarm and Hashicorp Nomad, or any future orchestration platform for container-like abstractions.

2.5.2 Scheduling in Kubernetes

Figure 2.5 shows the overall Kubernetes architecture. In Kubernetes, containerized applications run on a set of worker nodes that are managed by a control plane. In production environments, the control plane usually runs across multiple master nodes for fault tolerance and high availability.

A pod – which consists of one or more containers and data volumes sharing networking and storage namespaces – is the smallest unit of execution in Kubernetes. Kubernetes also provides higher-level controllers such as Deployment, StatefulSet and Job for managing a group of pods that belong to the same service. When a user requests *kube-api-server* to create pods on Kubernetes, *kube-scheduler* selects the most suitable worker node(s) in the cluster to place the pod(s). *kube-scheduler*'s default policy is to place pods on

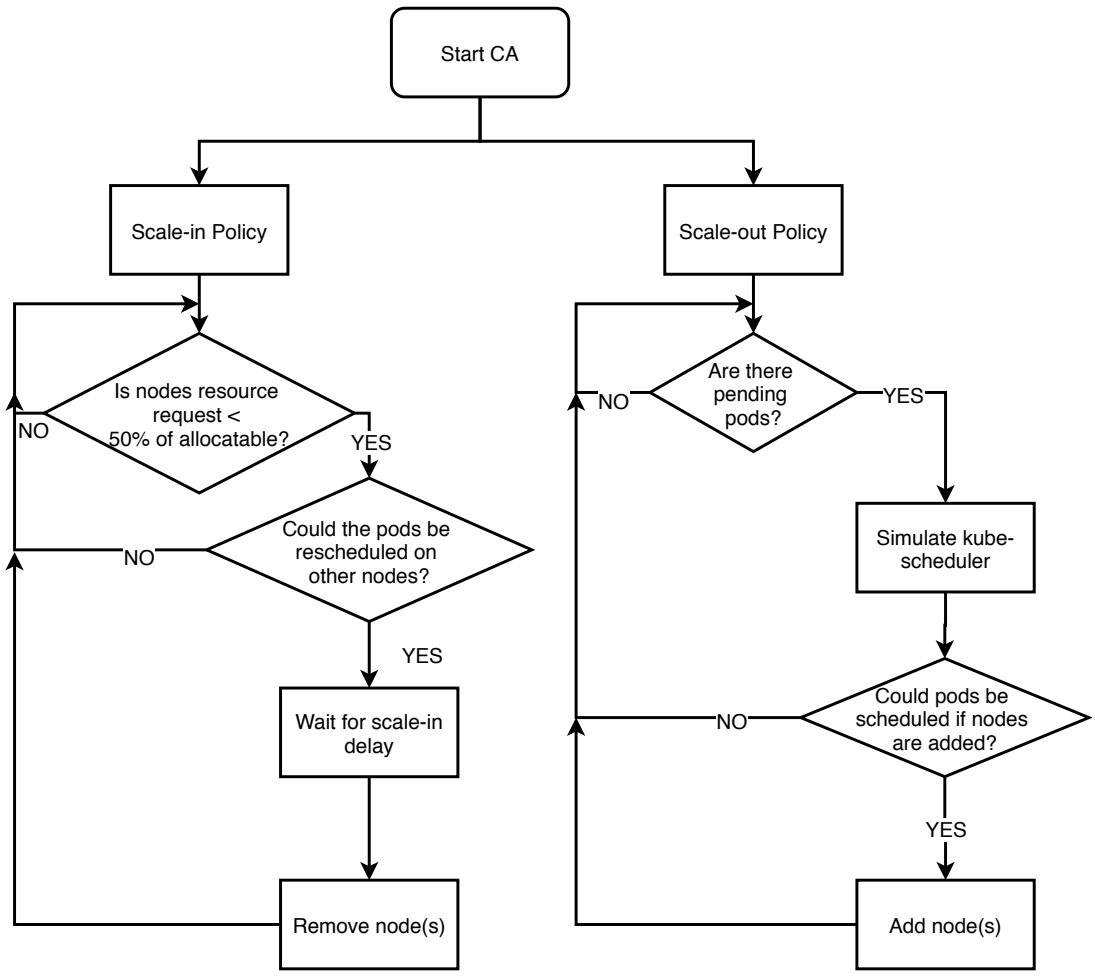


Figure 2.6 – Kubernetes Cluster Autoscaler (CA) algorithm flowchart.

worker nodes that have the most free resources while spreading out pods from the same deployment across different worker nodes. By doing so, *kube-scheduler* tries to balance out resource utilization of the worker nodes in the cluster.

2.5.3 Autoscaling in Kubernetes

Kubernetes supports autoscaling at two different levels. At the application (container) level, the *Horizontal Pod Autoscaler* (HPA) adjusts the number of pod instances based on CPU and memory utilization or other metrics such as response time, whereas the *Vertical Pod Autoscaler* (VPA) adjusts the CPU and memory request of pods based on past and present resource utilization. At the infrastructure level, Kubernetes offers the *Cluster Autoscaler* (CA) for adding/removing worker nodes to/from the cluster. CA's algorithm flowchart is shown in Figure 2.6.

CA watches *kube-api-server* periodically (by default every 10 seconds) for pods that are not scheduled due to resource shortage or other reasons. It assesses the specifications of the pods and simulates *kube-scheduler* to check whether adding more worker nodes to the cluster would enable placing the unscheduled pods. If so, CA adds new worker node(s) to the cluster. These worker nodes may be created for example by starting new VMs in a public or private cloud.

CA also periodically checks the resource utilization of the worker nodes. A worker node becomes a candidate for removal if the total sum of CPU and memory requests of its pods are less than 50% of the node's allocatable resources. The allocatable resource is defined as the number of computing resources available for pods, excluding resources needed for the OS and system daemons. If the pods running on the node can be rescheduled on other worker nodes, the node gets removed from the cluster after the scale-in time (by default 10 minutes).

One of the many configurable parameters for CA defines how it manages worker node pools. A node pool is a set of worker nodes of identical size. The CA, by default, adds all worker nodes from a single node pool, resulting in a cluster where all worker nodes have the same size. On the other hand, CA can be configured with *Cluster Autoscaler with Node Auto-Provisioning* (CA-NAP) which manages multiple node pools [167]. For example, at the time of writing, CA-NAP can create node pools in Google Cloud with machines from N1 machine types with 1 up to 64 vCPUs [168]. CA-NAP dynamically selects the minimal size of the worker node to be added based on the total resource request of the unscheduled pods. As a result, the cluster may have differently-sized worker nodes. CA uses the concept of *expanders* which provides different strategies for selecting the node pool out of multiple node pools from which new worker nodes will be added [169]. With *expanders* operators have the possibility to select node pools randomly or based on certain criteria such as ability to schedule most pods, having the least node price, or having the highest assigned priority.

As traditional CPU-usage-based autoscalers offered by cloud providers are not concerned about pods when scaling up and down, they may add a worker node that does not have any pods or remove worker nodes that have system-critical pods on them. CA makes sure that all pods in the cluster have a place to run irrespective of CPU load. Moreover, it tries to ensure that there are no unneeded worker nodes in the cluster. However, for correct CA operation, developers need to explicitly specify the right amount of resources for their workload. It is also important to design workloads that can tolerate the transient

disruptions that may result when pods are moved from one worker node to another during scale down.

2.5.4 Custom Resource Definitions (CRDs)

In the Kubernetes API, a resource is an endpoint that stores API objects. For instance, the built-in pods resource contains a collection of Pod objects. The Kubernetes API can be extended using a *Custom Resource Definition* (CRD) that defines new object kinds and lets the Kubernetes API server handle their entire lifecycle. When a new CRD is created, the Kubernetes API server creates a new RESTful resource path for it. This approach allows using custom objects like any other native Kubernetes objects.

As Kubernetes has emerged as a platform on top of which to build other platforms, several sub-projects have been created around it by the open-source community. These sub-projects extend Kubernetes to address specific use cases and develop the tools that can be used for these extensions. In the following sections, we discuss some of the open-source projects in the Kubernetes ecosystem that we have used as reference platforms or as part of the implementations of our contributions.

2.5.5 Kubernetes Federation

As discussed in the previous sections, Kubernetes is a container orchestration platform that automates the deployment, scaling and management of containerized applications in centralized large-scale computing infrastructures such as a cluster and a datacenter [160]. To extend it to multi-site deployments, KubeFed supports resource management and application deployment on multiple Kubernetes clusters from a single control plane, thus making it suitable for managing geo-distributed resources [170].

KubeFed's implementation builds upon the concept of CRDs from Kubernetes. In KubeFed terminology, a single *host cluster* runs the *federation control plane* which controls any number of *member clusters* where applications may be deployed. The host cluster is also the central point where the federation's configuration parameters are defined. The KubeFed *controller manager* runs as a Deployment resource on the host cluster. It runs several controllers to manage the member clusters, scheduling, deployments, services and other resources.

KubeFed introduces three concepts for each resource:

— *Template* defines the common specification of a resource across all member clusters;

— *Placement* specifies which member cluster(s) will get the resource;

— *Override* defines per-cluster variations of the template.

Using these concepts, users can define their deployments and services and decide how many application containers of a deployment should appear in which cluster(s). Figure 2.7 shows the KubeFed architecture with a host cluster and three member clusters where an "app 1" *federated service* is configured to be deployed on only two of the three member clusters.

KubeFed also offers *Replica Scheduling Preference* (RSP) which is an automated mechanism to distribute and rebalance federated deployments across member clusters. This is useful when scaling an application across several clusters. As shown in Figure 2.8, users only need to specify the target resource to be controlled by the RSP and the total number of replicas to be distributed in the federation. By default, RSP distributes the replicas evenly across all member clusters if they have sufficient resources. RSP does this in multiple iterations proportional to the number of total replicas. After calculating how many replicas should go to each member cluster, RSP modifies the Federated Deployment object to update the number of replicas on each cluster, which in turn pushes or reconciles the changes to the member clusters via the *sync controller*. The sync controller is responsible for propagating changes from the Federation Control Plane on the host cluster to the member clusters and maintaining the desired state of resources across member clusters.

Several configuration parameters control the behavior of the *sync controller*. The most important for our work is the *Cluster Health Check Timeout* (CHCT), which has a default value of 3 seconds. This parameter determines the duration after which sync requests and cluster health checks time out. In a geo-distributed deployment where large network latencies and packet losses are commonplace, it is important to choose the right value for this parameter to ensure the correct functioning of the *sync controller* and RSP. A low CHCT value ensures fast detection of cluster failures, allowing RSP to rebalance the deployment away from failed clusters onto healthy clusters. However, fast detection increases the probability of false positives due to delays and transient network problems. Therefore, it may be necessary to increase the value of CHCT to reduce the number of false positives. However, increasing the CHCT value may lead to delayed cluster failure detection. It is, therefore, important to consider the trade-off between false positives and slow detection of cluster failures when choosing the CHCT parameter value. As extensively discussed in Chapter 5, manually selecting an optimal value for CHCT is challenging as the

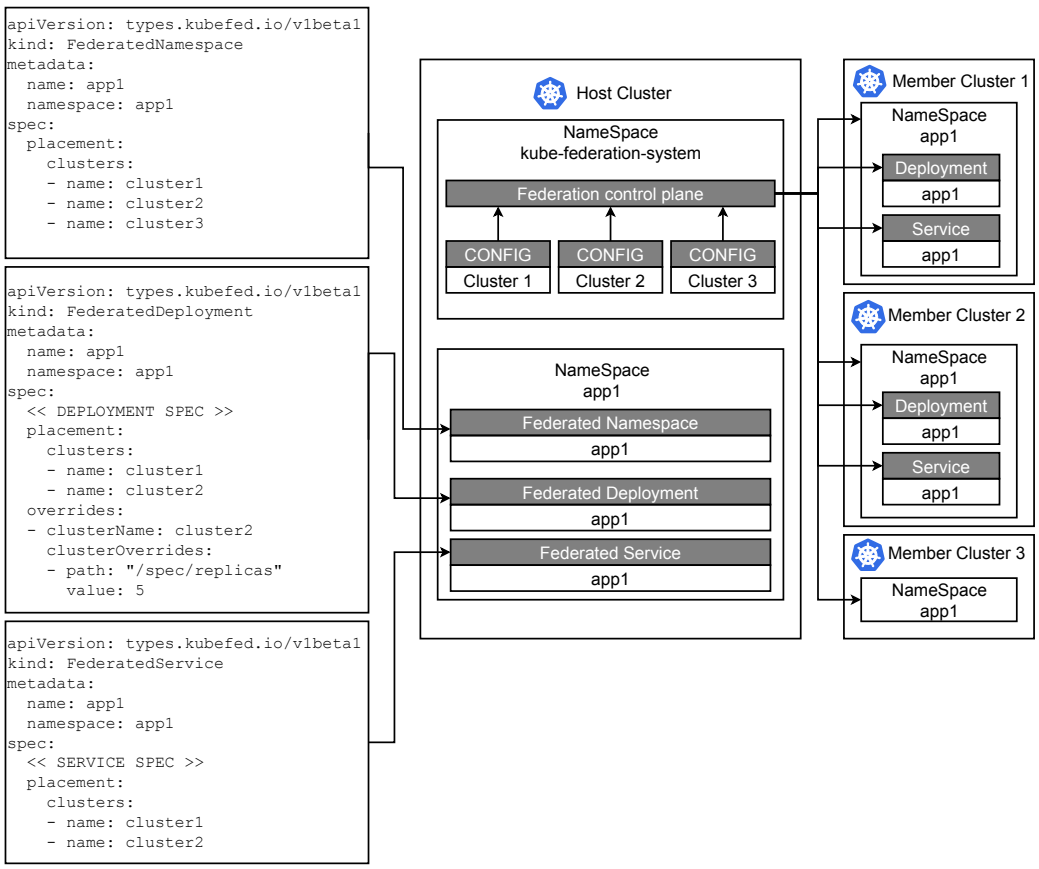


Figure 2.7 – A simplified view of KubeFed architecture with a host cluster and three member clusters and propagation of federated resources of a sample application *app1*.

choice may depend on the computing and networking environment as well as application workload dynamics.

2.5.6 Limitations of Kubernetes and Kubernetes Federation

Kubernetes provides basic low-level primitives such as *Pods, Jobs, Deployments* and *Services* to orchestrate applications. Similarly, KubeFed provides low-level multi-cluster primitives such as *FederatedDeployments, FederatedJobs,* and *FederatedServices* that allow orchestrating federated deployments across multiple clusters. However, for various reasons such as limitation to a single cluster, lack of network and location awareness and lack of sufficient level of automation, neither Kubernetes nor KubeFed meet all the non-functional requirements of scalable multi-cluster applications identified in Section 2.2.

Kubernetes is designed to orchestrate containers and manage servers in a single cluster [51], [171]. As a result, the application model, resource abstractions, scheduling poli-

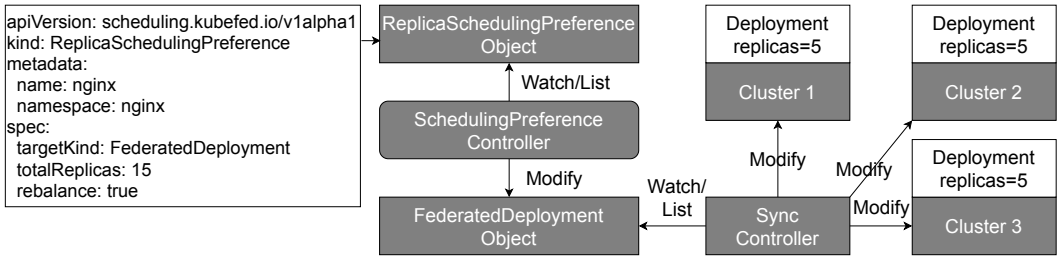


Figure 2.8 – Automatic distribution of deployment replicas using RSP on a Federation with a host cluster and three member clusters. Here, RSP evenly distributes a total of 15 replicas across the three member clusters, 5 replicas per cluster.

cies and autoscaling mechanisms in Kubernetes make sense only in a single-cluster environment. For instance, since Kubernetes' scheduling policy is based mainly on resource availability in nodes, it cannot place or autoscale Pods optimally in geo-distributed environments such as fog computing which require proximity-aware placement [122], [129]. Similarly, Kubernetes' round-robin-based service routing mechanism is not optimal if there is significant network latency between the worker nodes of the cluster as is often the case in geo-distributed deployments [20]. Therefore, Kubernetes is not suitable for managing geo-distributed multi-cluster environments out of the box without significant modifications.

KubeFed, on the other hand, is designed to provide low-level placement primitives in multi-cluster environments [51], [172]. However, the scheduling mechanisms it provides are mainly manual and it lacks automated policy-based placement mechanisms that are required in geo-distributed multi-cluster environments. The only available automated placement functionality, *Replica Scheduling Preference* (RSP), distributes replicas across clusters either evenly or based on manually-configured weights. Similar to Kubernetes, KubeFed lacks proximity-aware or other automated placement policies. Also, KubeFed's *Federated Service* and *Federated Horizontal Pod Autoscaler* features are limited to replicating Service and *Horizontal Pod Autoscaling* objects, respectively, across multiple clusters. As a result, KubeFed does not have any mechanisms to route traffic between clusters or burst replicas from one cluster to another. Moreover, as geo-distributed multi-cluster environments vary from one another in terms of scale, heterogeneity in networking technologies, KubeFed lacks in the automatic tuning of configuration parameters that affect various aspects of the system such as stability and availability. Therefore, KubeFed cannot perform optimal placement, service routing, or autoscaling functions in geo-distributed multi-cluster environments without major improvements.

On the upside, Kubernetes is modular and extensible with arbitrary CRs and custom controllers, using which one can transform Kubernetes into an optimal container orchestration platform for various use cases. In Chapter 5, we propose a controller that improves the stability of KubeFed by adapting one of its important configuration parameters to the changing environment. Additionally, in Chapter 6, we propose a container orchestration platform for geo-distributed multi-cluster environments, with controllers that drive the deployment, autoscaling and bursting of multi-cluster applications as well as provision, scale and de-provision cloud resources.

2.5.7 Custom Kubernetes controllers

As mentioned in the previous section, Kubernetes can be extended using CRs and controllers. The controller logic is an important part of many of the objects in Kubernetes itself, such as the control plane, ReplicaSets, Deployments, Horizontal Pod Autoscalers and Cluster Autoscaler. Following the *Monitor - Analyze - Plan - Execute over a shared Knowledge* (MAPE-K) principle, Kubernetes controllers repeatedly compare the desired state of the cluster to its actual state. Whenever there is divergence, the controllers take actions to bring the actual state of the cluster to the desired state specified by the user [173].

Similarly, operators can be implemented to orchestrate custom objects of a certain kind with domain-specific logic [174]. An operator extends the Kubernetes API by adding an endpoint called a CR. The CR data schema is defined by a CRD. The operator also adds the corresponding control plane components to monitor and maintain the CRs. Operators observe the state of the objects and react to any changes to bring them to the desired state. Operators communicate with Kubernetes-native or other objects via the Kubernetes API. Users can interact with CRs as they would interact with any other Kubernetes resource via the Kubernetes API using the Kubernetes *Command Line Interface* (CLI) or any other Kubernetes client.

The *Kubernetes Operator Pythonic Framework* (Kopf) [22] is a framework to build Kubernetes operators using the Python programming language. Kopf provides a toolkit to run operators, communicate with the Kubernetes cluster, translate Kubernetes events into pure Python functions and persist the state of objects. It also provides libraries to manipulate Kubernetes objects. By allowing the developer to focus on the operator logic, this framework makes it easy to develop new Kubernetes operators.

22. Kopf – https://kopf.readthedocs.io/en/stable/

In Chapters 5- 6, we propose controllers that interact with the Kubernetes API to improve the stability of geo-distributed deployments or enable container orchestration across geo-distributed clusters. One of the positive aspects of Kubernetes' extensibility using custom controllers such as operators is that it allows building more sophisticated platforms on top of it without the need to build a platform specific to a use-case from scratch. As a result, we have relied on operators developed using the Kopf framework, particularly in Chapter 6, to transform Kubernetes into a container orchestration platform for geo-distributed multi-cluster environments with policy-rich placement, autoscaling, bursting and auto-provisioning capabilities.

2.5.8 Cluster API

In a multi-cluster environment, it may be necessary to transparently provision or de-provision resources from different cloud providers, for instance in the case of cloud bursting. However, this is not an easy task due to interoperability issues that arise because of differences in APIs, management interfaces and resource types between providers [164], [175], [176]. As a result, a lot of effort is required to acquire resources from multiple providers.

Cluster API [23] is a Kubernetes sub-project that allows provisioning, scaling, upgrading and de-provisioning Kubernetes clusters from multiple cloud providers in a declarative way. It provides tools to provision VMs, networks, load balancers as well as Kubernetes clusters consistently from various private and cloud providers. These resources are defined and created in the same way other Kubernetes-native resources such as Pods or Deployments are. Moreover, Cluster API automates Kubernetes cluster lifecycle management including creating, upgrading and deleting a cluster.

In Cluster API, a management cluster manages the lifecycle of the clusters provisioned from cloud providers, called workload clusters. Some of the components of the management cluster include infrastructure providers, and CRDs such as Machine, MachineSet and MachineDeployment. Infrastructure providers are implementations of major bare metal infrastructure providers such as VMware [24] and OpenStack [25] as well as cloud providers such as AWS, Google Cloud and Microsoft Azure. Infrastructure providers hide the implementation details and allow users to consume resources from providers easily.

23. Cluster API – https://github.com/kubernetes-sigs/cluster-api
24. Cluster API Provider vSphere – https://bit.ly/3gHkKeF
25. Cluster API Provider OpenStack – https://bit.ly/2U7Jcgb

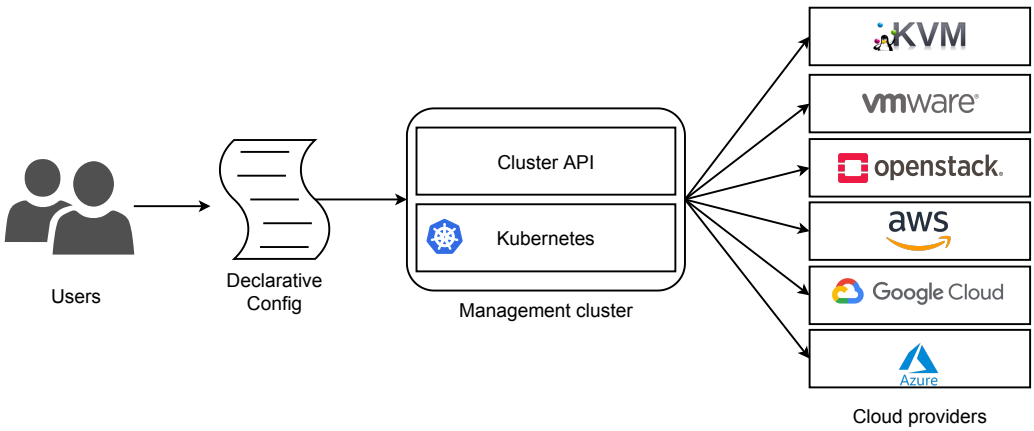


Figure 2.9 – Cluster API.

A Machine is an object that represents a Kubernetes node deployed on a VM from an infrastructure provider. When a new Machine object is created, a new VM with the provided specifications is created by the infrastructure provider and joined to the Kubernetes clusters. Similarly, updating or deleting a Machine object removes the corresponding node from the Kubernetes cluster and replaces or deletes the VM. A MachineSet is analogous to a Kubernetes ReplicaSet and ensures that the specified number of Machines are running all the time. Similarly, a MachineDeployment, which is analogous to a Kubernetes Deployment, reconciles the desired state of Machines and MachineSets.

In Chapter 6, our proposed container orchestration platform for geo-distributed multi-cluster environments integrates with Cluster API to be able to transparently provision clusters from cloud providers when the existing clusters run out of resources. Autoscaling of the additional cloud cluster or de-provisioning of the cluster when they are not needed anymore is also done via Cluster API.

2.5.9 Cilium cluster mesh

In geo-distributed deployments, it is important not only to be able to burst applications across multiple clusters but also distribute or load balance network traffic to the application instances residing on the clusters [177], [178]. For instance, in the case of cloud bursting, it is important to re-configure load balancers to distribute user traffic between the original cluster and the newly provisioned clusters. Moreover, there should be a mechanism to route network traffic in between the application containers that reside on the multiple clusters. Similarly, in fog computing scenarios it is often important to route user traffic to the closest cluster that hosts the application replicas [20].

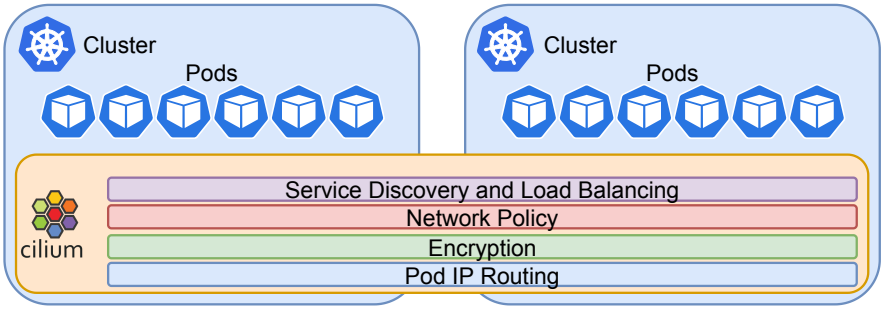


Figure 2.10 – Cilium cluster mesh.

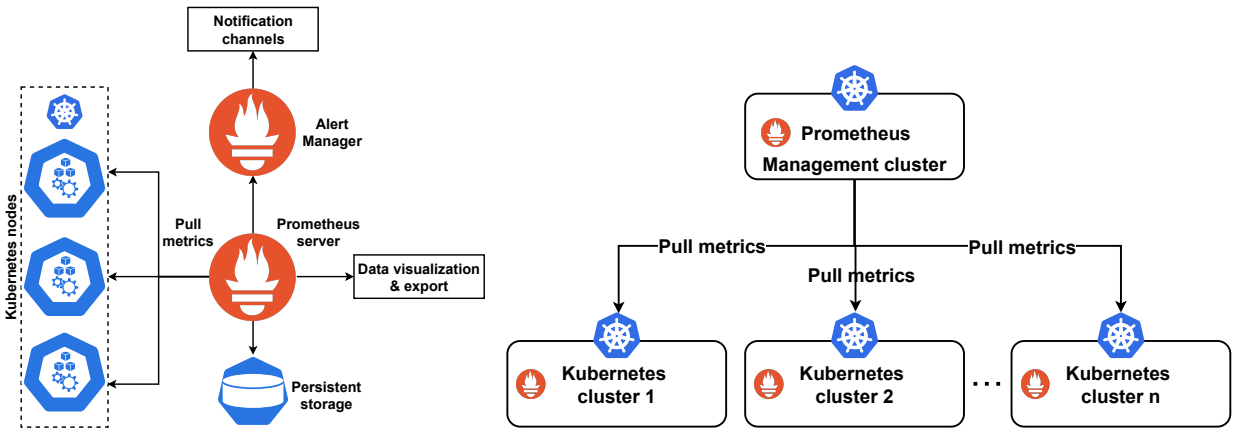
In container orchestration platforms such as Kubernetes, the networking model allows communication only between the containers in a single cluster. Therefore, additional tooling is required to allow inter-cluster network communication between containers that reside in multiple clusters.

Cilium [26] is an open-source container networking technology that can be used inside container orchestration platforms such as Kubernetes but also allows inter-cluster network connectivity. Cilium builds on *Berkeley Packet Filter* (BPF) which is a technology that can run sandboxed programs in the Linux kernel without modifying the kernel source code or kernel modules. BPF makes the Linux kernel programmable and allows one to build tools for networking, security, application tracing and performance troubleshooting [179].

Cilium Cluster Mesh [27] is Cilium's multi-cluster implementation that creates network connectivity between the nodes multiple clusters for load-balancing, observability and security. Cilium Cluster Mesh enables multi-cluster network connectivity by allowing pods in different clusters to reach each other using their Pod IP addresses without the need for proxies or gateways. Moreover, Cilium Cluster Mesh allows transparent service discovery, network policies spanning multiple clusters, and transparent encryption to secure all communication between multiple clusters [163].

In Chapter 4, we rely on Cilium Cluster Mesh to enable network traffic routing and load balancing in our proposed container orchestration platform for geo-distributed multi-cluster environments.

26. Cilium – https://cilium.io/
27. Cilium Cluster Mesh – https://cilium.io/blog/2019/03/12/clustermesh



(a) In a single cluster. (b) Across multiple clusters.

Figure 2.11 – Monitoring with Prometheus.

2.5.10 Prometheus

Prometheus [28] is an open-source monitoring solution for cloud systems. It is widely used in container orchestrators such as Kubernetes for monitoring the status of the entire cluster, nodes, pods and applications. As shown in Figure 2.11 (a) Prometheus deploys agents in each worker node that periodically gather information about the worker node and the services running in it. The Prometheus server pulls the metrics from all worker nodes and stores it in a time-series database. The monitoring data can be queried using Prometheus' own querying language or displayed using visualization tools such as Grafana [29]. Moreover, alerting can be configured to send notifications to the user. In our work, we configure Prometheus to gather metrics from multiple clusters. As shown in Figure 2.11 (b) the Prometheus server in the management cluster pulls metrics from all the geo-distributed clusters and persists them in a persistent storage. These metrics include CPU utilization, memory usage and network traffic at the cluster, nodes and application levels. Our controllers rely on Prometheus as the sensor to provide the information required for making placement, autoscaling and provisioning decisions.

2.5.11 Serf

In geo-distributed environments, it is important to know the location of workload clusters in the network with respect to each other for network-aware placement, autoscaling

28. Prometheus – https://prometheus.io/
29. Grafana – https://grafana.com/

and provisioning decisions. The location may be expressed as the inter-cluster network latency. In this thesis, we use *Serf* [30], which is a lightweight and open-source tool that relies on gossiping protocols to estimate the network latency between nodes running the agent. In our works, serf runs in the management cluster as well as the workload clusters. The management cluster uses the network latency estimate from the serf agent when making decisions about offloading a deployment from one cluster to a neighboring cluster, bursting to a neighboring cluster or provisioning clusters from a nearby cloud provider.

2.6 Towards container orchestration in geo-distributed multi-cluster environments

As discussed in the previous sections, the need to deploy large scale containerized applications in a scalable, highly available and fault-tolerant manner has led to the emergence of cluster managers and container orchestration platforms such as Kubernetes. These platforms automate the deployment, upgrading and scaling of containerized applications across a cluster of nodes. As these orchestrators are limited to nodes in a single DC, there is a need for platforms that bridge separate clusters and aggregate the resources across them to deliver on the non-functional requirements of modern global applications such as proximity, high availability, fault tolerance, compliance with regional regulations and vendor neutrality. Although platforms such as KubeFed exist to address these requirements, there are still several resource management challenges that need to be addressed to deliver on a stable, highly scalable and fault-tolerant platform.

2.6.1 Autoscaling performance

Autoscaling is one of the most important and attractive aspects of cloud computing. By adjusting the number of resources allocated to applications dynamically, autoscaling allows cloud applications to deliver on their performance promises despite fluctuations in user traffic. As discussed in previous sections, autoscaling can be done at various levels of abstractions such as the container level or the VM level and in various directions, i.e., vertical or horizontal.

Several challenges affect the performance of autoscaling systems including synchronization of autoscaling at the different levels and different configuration parameters such

30. Serf – https://www.serf.io/

as threshold values, amount of resources allocated to Pods and VMs, delays and cool-down times. Different combinations of configurations could result in different behaviors that affect the performance of the system. For instance, under-provisioning could lead to performance degradation, whereas over-provisioning leads to unnecessary spending, especially in public cloud environments. Therefore, it is important to characterize and quantify the performance of the autoscaling system to understand how exactly it is affected by different parameters and mitigate these problems.

In Chapter 4 we perform a performance evaluation of the Kubernetes CA in two configuration settings using standard autoscaling metrics established by the Cloud Group of *Standard Performance Evaluation Corporation* (SPEC) [31]. We also highlight the impact of these configurations on the cost of running a scalable Kubernetes cluster in the public cloud.

2.6.2 Automatic configuration tuning

Like many distributed systems, orchestration platforms for geo-distributed multi-cluster environments can have several configuration parameters that affect multiple aspects of the system such as resource utilization, performance, stability and availability. As most distributed systems have several configuration parameters that often result in complex situations when combined, human operators find it overwhelming to configure systems manually. As a result, systems are often configured with static and default setting that may result in suboptimal performance [71], [180]–[183]. However, geo-distributed multi-cluster environments work under uncertain conditions due to ever-changing network conditions, workloads and user traffic [122], [129], [184]. As a result, it is critical to adaptively adjust certain important system parameters during runtime so that the platform runs with high performance, stability and availability, while keeping resource utilization efficient.

In Chapter 5 we demonstrate through experiments conducted on a realistic geo-distributed multi-cluster environment in *Grid'5000* [32] that static configuration parameters in KubeFed coupled with changing network and workload conditions lead to instability, which in turn negatively affects the availability of the applications deployed on the platform. To mitigate this problem, we propose a control-theoretic approach to adjust a con-

31. SPEC Cloud – https://bit.ly/3n3Kza8
32. Grid'5000 – https://www.grid5000.fr/

figuration parameter in the KubeFed control plane at run time to improve the stability of the system, and, in turn, the availability of applications.

2.6.3 Container orchestration for geo-distributed multi-cluster environments

Although KubeFed is proposed to orchestrate containerized applications in multi-cluster environments, in its current form it allows selecting the clusters manually to host the federated resources with limited support for automated policy-based scheduling. Moreover, it simply propagates resources to the target clusters without any prior checks on the availability of resources. Because of this, KubeFed cannot scale to manage hundreds or thousands of clusters that are typical in certain multi-cluster use cases such as fog computing. Furthermore, it fails to manage resources efficiently, causing resource wastage and fragmentation.

As the number of clusters increases, there is a high probability that each cluster may contain only a modest amount of resources. When applications running on such clusters are faced with sudden increases in workloads and user traffic, an autoscaler in the clusters increases the number of resources allocated to the applications. However, as limited resources are available in a single cluster, the application demands may exceed capacity. In these situations, it may be necessary to burst the application replicas transparently to nearby clusters that have sufficient resources. If no resources are available in the nearby clusters, it may be necessary to transparently provision resources from a public cloud provider in the same or nearby region. Once an application's replicas are burst to an additional cluster, network traffic needs to be routed and load-balanced across all the clusters hosting the application.

After the workload or user traffic spike has passed, the resources provisioned from additional clusters would likely be under-utilized. This decreases the overall resource utilization of the system and also incur unnecessary costs, especially in public clouds. Therefore, it is important to continually monitor the resource utilization of the applications deployed in the platform and remove any under-utilized resources including whole clusters in the public cloud.

These challenges have to be solved to build a mature orchestration platform for managing resources in multi-cluster environments. Therefore, we introduce *mck8s* in Chapter 6 – a platform that offers not only automated policy-based scheduling but also multi-cluster

horizontal pod autoscaling and cloud bursting mechanisms. *mck8s* reuses and extends concepts from Kubernetes and KubeFed. It also integrates other open-source tools such as the Cilium cluster mesh for multi-cluster network discovery and global load balancing, Cluster API for declarative resource provisioning from cloud platforms, Prometheus for monitoring, and Serf for measuring inter-cluster network latencies.

To implement *mck8s*, important decisions have to be made with regards to the manageability and usability of the system. Implementing the placement, autoscaling, resource provisioning and cloud bursting functions requires designing custom controllers that follow the *Monitor* - *Analyze* - *Plan* - *Execute* (MAPE)-K methodology widely adopted in Kubernetes. To reduce the number of abstractions, minimize redundancies, and improve the manageability of the system, we opted to design and implement the controllers that perform the desired functions as replacements to the KubeFed controllers available in KubeFed. Moreover, to improve usability and easy adoption, we designed the manifest files used to deploy the new CRs following both the syntax and core design concepts of Kubernetes.

2.7 Conclusion

In this chapter, we presented the technical background leading to the resource management problems in geo-distributed multi-cluster environments that we studied and the contributions we proposed to address these problems. In the next chapter, we will look at the state of the art in the area of research and elaborate on where our contributions fit in the broader research area.

Chapter 3

State of the art

In the previous chapter, we discussed the three major challenges of resource management in containerized geo-distributed computing environments that we study in this thesis, namely (1) understanding and evaluating the performance of autoscaling systems; (2) adaptive configuration tuning, and (3) container orchestration platforms for geo-distributed multi-cluster environments. In this chapter, we review previous works in the academic literature that addressed these challenges on which our thesis builds upon, and position our contributions.

3.1 Evaluation of autoscaling systems

3.1.1 Autoscaling systems

*Cloud Service Provider*s (CSPs) make different autoscaling policies available to their customers to make the task of adding or removing resources easier [185], [186]. These policies allow cloud customers to configure the autoscaling systems with target utilization metrics or schedules so that the autoscaling systems can seamlessly handle traffic increases and reduce costs when resources are no longer needed [187].

Similarly, several *Virtual Machine* (VM) autoscaling systems and policies have been proposed by the academic community [52], [53], [58], [188], [189], as surveyed by [190]. More recently, following the widespread adoption of container technologies, autoscaling of containers has also found interest in the research community [129], [158], [191]–[196].

3.1.2 Autoscaling evaluation methods and metrics

Selecting the most appropriate autoscaling system and policies is essential for an application not only to ensure the performance of the application under ever-changing conditions but also to avoid unnecessary expenses [190], [197]. Therefore, it is important to study different properties of autoscalers such as under-provisioning, over-provisioning and

oscillation, and their impact on the performance of applications in a wide range of operational conditions [197], [198]. To this end, it is important to find a systematic approach to evaluate and compare different autoscalers [197]. Despite the abundance of autoscaling systems and policies from industry and academia, selecting the most appropriate policies is not easy for cloud users due to the lack of systematic approaches to evaluate and compare autoscalers [199]. Much of the work in autoscaling focuses on proposing new autoscaling mechanisms for a specific application with limited extensive comparisons with the existing autoscalers [197]. Most works evaluate their autoscalers under specific conditions and lack extensive testing to account for the large variety of workloads in cloud environments, uncertainties in the workloads, and unexpected behavior of the system [200].

A number of recent works establish methodologies and metrics for systematic theoretical, simulation-based and experimental performance evaluation of autoscalers. In [190], *Lorido-Botran et al.* perform a theoretical evaluation of autoscaling techniques. They classify autoscaling techniques in the state of the art in five major groups based on the underlying theory or technique used to build the autoscalers, which are threshold-based rules, Reinforcement Learning (RL), queuing theory, control theory and time series analysis. In addition to discussing the advantages and disadvantages of each of these techniques, the authors highlight the lack of standardized testing platforms capable of generating a well-defined set of standardized metrics. The lack of such metrics prevents a comparative analysis of the reviewed techniques quantitatively. Similarly, *Al-Dhuraibi et al.* [199] and *Qu et al.* [157] present taxonomies and surveys of autoscalers. Moreover, *Papadopoulos et al.* introduce eight methodological principles for experimental performance evaluation of cloud systems including autoscalers with focus on metrics selection and reproducibility [201].

In [200], the authors propose an autoscaling evaluation framework called *Performance Evaluation Framework for Auto-Scaling* (PEAS). The PEAS evaluation is formulated as a chance-constrained optimization problem, which is solved using scenario theory. PEAS provides probabilistic guarantees about the performance of autoscaling systems. The authors present an extensive evaluation of six representative autoscalers from the state of the art using the PEAS framework on a discrete event simulator using real workloads. They highlight the difficulty of generalizing autoscalers' performance due to performance variation depending on the evaluation criteria.

Herbst et al. propose evaluation metrics for elasticity together with the measurement approaches [202]. These metrics were later endorsed by the Cloud Group of the *Standard*

Performance Evaluation Corporation (SPEC) [203]. These metrics capture the timing and resource provisioning accuracy aspects of elasticity and allow quantifying the elasticity of systems irrespective of the cloud provider, virtualization technology, or application. The under-provisioning and over-provisioning accuracy metrics capture the deviations of the allocated resources from the respective resource demands, normalized by the measurement period. The under-provisioning and over-provisioning time share metrics characterize the total amount of time resources have been under- or over-provisioned, respectively, normalized by the measurement period. Lastly, the jitter metric captures the number of rescaling operations normalized by the measurement period. These standard metrics help cloud providers to communicate the capabilities and characteristics of their autoscaling systems, and help customers to select an autoscaling system which best suits their needs. They also give researchers a toolbox to evaluate new autoscaling algorithms. These evaluation metrics constitute the current standard, and we, therefore, base our evaluations on them in Chapter 4.

3.1.3 Autoscaling evaluation results

In [197], *Ilyushkin et al.* perform an extensive experimental evaluation of seven representative autoscaling policies from the state of the art on a private cloud environment using more than 10 metrics, including the five metrics proposed in [203]. The evaluated autoscalers are general-purpose as well as workflow-specific ones. The major findings of the evaluation are: (1) general-purpose autoscalers' performance is dependent on their configuration; (2) the performance of autoscalers is negatively affected by long VM booting time; and (3) no autoscaler outperforms all others with all configurations and metrics, which implies that an autoscaler should be selected carefully for a particular application. Although this work pioneers the use of standard autoscaling performance metrics for evaluating the performance of autoscalers, it is limited in the variety of cloud platforms, types of applications and scale of the infrastructure.

Similarly, *Versluis et al.* evaluate the same autoscalers from the state of the art using the SPEC metrics [198]. In contrast to [197], *Versluis et al.* present larger-scale trace-based simulations for workflows from three domains: scientific, industrial and engineering. Moreover, they measure the impact of autoscaling across a variety of cloud environments, workloads, allocation policies and utilization levels. The key takeaways from this work are: (1) different application characteristics lead to significant differences in autoscaling performance; and (2) resource allocation and autoscaling policies should be co-designed.

These works provide a better understanding of the performance of autoscaling policies proposed in the past decade. However, [197] focuses on scientific workflows while [198] focuses on scientific, industrial and engineering workflows.

Other works employ the SPEC metrics to report on the performance of newly-proposed autoscalers [59], [204]–[207]. Whereas the previous works study autoscaling based on VMs, *Bauer et al.* [204], *Podolskiy et al.* [208], *Ramirez et al.* [206] and *Arkian et al.* [207] are also concerned about autoscaling performance at the container level. *Bauer et al.* introduce *Chamulteon* – an autoscaler for applications consisting of multiple services and evaluate its performance using the SPEC metrics against other autoscalers from the state of the art [204]. *Chamulteon* addresses the bottleneck propagation and oscillation problems that arise because of the independent scaling of services. The evaluations are performed both on VMs and on Docker containers in Kubernetes. The authors point out the challenge of synchronized autoscaling in nested resource layers such as containers and VMs.

Podolskiy et al. present a comparison of various public cloud providers' autoscalers using their own *Autoscaling Performance Measurement Tool (APMT)* and two metrics, namely amount of QoS violations and fractions of the autoscaling intervals where QoS requirements are violated [208]. Their evaluation involves autoscaling at the container and VM levels, using Kubernetes *Horizontal Pod Autoscaler* (HPA) for the former and autoscaling mechanisms of the public service providers for the latter. The authors point out the coordination problem between the multiple virtualization layers in *Amazon Web Services* (AWS) and Microsoft Azure that leads to the imbalance of Pods distribution on VMs during scale-out and deletion of VM that have Pods still running on them during scale-in. In contrast, *Google Compute Engine* (GCE) and HPA show better performance because of better coordination. The authors also identify the need to scale out to multiple node pools as a potential research direction.

Ramirez et al. design and evaluate five predictive autoscaling policies for containerized microservices [206]. They show that predictive policies improve autoscaling accuracy as compared to reactive ones. *Arkian et al.* propose *Gesscale* – a resource autoscaler for *Data Stream Processing* (DSP) applications in geo-distributed environments based on a performance model to give predictions about future throughput of the application [207]. They compare its performance with other autoscalers from the state of the art using the SPEC metrics among others. Evaluations show that *Gesscale* produces fewer autoscaling actions and uses less resources than the evaluated autoscalers.

In Chapter 4 we evaluate the Kubernetes scheduling, HPA and *Cluster Autoscaler* (CA) mechanisms – which had not been evaluated experimentally before – extensively in a public cloud environment using the autoscaling performance metrics proposed by *Herbst et al.* [202] and endorsed by SPEC [203]. We study the performance of CA under two configuration settings, namely the default CA and *Cluster Autoscaler with Node Auto-Provisioning* (CA-NAP). The results from our experimental evaluation confirm the findings of previous studies that autoscaling performance depends on the type of application and workload used. Similar to [208], we also highlight the need for proactive autoscaling and further configuration tuning of the autoscaling system for better performance.

Table 3.1 summarizes the literature on autoscaling performance by the autoscaling systems they evaluate, the virtualization type, cloud type, application type and evaluation type.

Table 3.1 – Summary of the literature on autoscaling evaluation.

Ref.	**Evaluated Autoscaling System**	**Virtualization Type**	**Cloud Type**	**Application Type**	**Evaluation Type**
[190]	Several autoscalers from the state of the art	VM	N/A	Various	Theoretical
[200]	[58], [52], [56], [57], [53], [209]	VM	N/A	Web application	Simulation
[197]	*Plan*, [58], [52], [56], [57], [53]	VM	Private	Workflows	Experiments
[198]	[58], [52], [56], [57], [53], [210], *Plan* [197]	VM	Public & Private	Scientific, engineering & industrial	Simulation
[59]	*Chameleon*, [58], [52], [56], [57], [53]	VM	Private	Web app	Experiments
[204]	*Chamulteon*, [58], [52], [56], [57]	VMs & containers	Private	Microservices	Experiments
[205]	[52]	VM	Private	Microservices	Experiments
[206]	five reactive autoscaling policies	Containers	Public	Microservices	Experiments
[208]	AWS, GCE, Azure & HPA	VMs & containers	Public	Compute-intensive app	Experiments
[207]	*Gesscale*	Containers	Fog computing	DSP	Experiments
Contribution 1	Kubernetes CA & CA-NAP	VMs & containers	Public	Microservices & jobs	Experiments

3.2 Automatic configuration tuning

Modern distributed systems are increasingly complex and expose several configurable parameters to users of the systems to support a wide variety of use cases [180], [211]. Some examples of the parameters include the configuration of multiple thread pools, queues, cache size, timeouts and retry values and memory [212]. Different combinations of these configuration parameters may affect several aspects of the system such as performance, availability, stability, and failure detection and recovery [180], [212], [213]. Whereas an optimal set of values for the configuration parameters may result in a significant improvement of a desired metric of the system, a suboptimal configuration may adversely affect the performance of the system [180], [183], [214].

An optimal configuration that works in one setting may not give the same results in a different setting with different infrastructure, network conditions, workloads, and user traffic patterns [213]. However, finding the optimal configuration is error-prone, time-consuming and costly, if not impossible, for human operators due to a large number of configuration parameters and complex interactions between them [211], [212], [215]. Moreover, optimal configurations may require deep knowledge of the system [183] and the effects of a set of configurations may not be known without experience [215]. Several approaches have been proposed in the academic literature over the years to make the task of finding the optimal set of configuration parameters easier.

3.2.1 Problem of misconfiguration and negative consequences

Configuration tuning is a difficult and error-prone task due to a large number of configuration parameters and complex interactions between them [212]. This complexity often leads to misconfigurations that may adversely affect the performance of systems.

Zhu et al. show that a configuration parameter which is key to the performance of MySQL for one workload does not have any obvious relation with the performance of the system for another workload [180]. They also show that the default configuration does not necessarily result in the best performance for all workloads.

Yin et al. study 546 real-world misconfigurations of systems such as a commercial storage system deployed at thousands of customers, CentOS, MySQL, Apache HTTP Server and OpenLDAP [214]. They show that partial or full unavailability, or severe performance degradation may be caused by misconfigurations. These findings indicate the need for automatic configuration checking and tuning.

Similarly, *Xu et al.* study 620 real-world cases of configuration issues reported by users for systems such as Apache, MySQL, Hadoop and commercial storage systems [211]. They report that up to 53% of the configuration errors occur due to default values. This highlights the fact that users have difficulty finding and understanding which parameters should be set among a large configuration space.

In one of the few works that study the configuration of Kubernetes, *Vayghan et al.* investigate the impact of configurations on the availability of microservices deployed on Kubernetes [216]. They evaluate how fast Kubernetes handles Pod and worker node failures under its default as well as its most responsive configurations. In particular, the default configuration leads to significant service outages even with service redundancy. This shows the need to adapt the configuration of the platform as per the availability level required by the applications deployed on it. It also highlights the challenges of identifying the right combination of parameter values for best performance.

In Chapter 5, we show that the default configuration of *Kubernetes Federation* (KubeFed) in a geo-distributed deployment environment may lead to instability, which in turn negatively affects the availability and performance of applications deployed on the system. This is because the default configuration parameters have not been optimized for the geo-distributed deployment environment, where the networking is characterized by high latency, low bandwidth, and a high probability of packet loss, as well as highly variable workloads. We, therefore, propose an automatic controller to adjust the concerned parameter dynamically.

3.2.2 Sampling methods in a high-dimensional parameter space

One of the main challenges in automatic configuration tuning is the high dimensional parameter space produced by a large number of configuration parameters [180]. Although it is impossible to get a complete understanding of the relationship between performance and configuration without covering the whole parameter space, it is time-consuming and costly to sample the entire parameter space or to collect too many samples. Therefore, it is important to find methods that can produce acceptable results using only a limited number of samples.

Three sampling methods, namely *random sampling*, *stratified sampling* and *Latin Hypercube Sampling* (LHS) have been proposed by *McKay et al.* [217]. The authors treat the input sample space as random variables. Unlike random sampling, in stratified sampling, the sampling space is further divided into a number of disjoint subspaces from

which the input variable is sampled, making sure that the entire sampling space is covered. By dividing the range of each input variable into a number of sub-regions of equal marginal probability, LHS guarantees that the input values represent all sections of the input variable's distribution.

Similarly, *Zhu et al.* use the *Divide-and-Diverge Sampling* (DDS) method that divides the parameter space into subspaces and allows randomly selecting a point from each subspace [180]. This method ensures complete coverage of the whole parameter space because each subspace is represented by a sample, unlike random sampling without subspace division where it is very likely that some subspaces are not represented especially when the space is high-dimensional. DDS differs from LHS in that it remembers previously sampled subspaces and re-samples towards a wider coverage of the whole parameter space. This difference explains the DDS method's advantage of coverage and scalability over LHS.

In Chapter 5, we use the DDS sampling technique to divide the space of KubeFed parameters and pick representative parameter values to use for experimental measurement of the stability of the system. Then, we use *Principal Component Analysis* (PCA) to reduce the parameter space and identify the most important parameters that affect the stability of the system.

3.2.3 Automatic configuration tuning systems and methods

In the past decades, several works have proposed to leverage the concepts from autonomic computing for self-configuration and self-adaptation of distributed systems. Many of the works focus on optimizing the performance of systems by finding the best combination of configuration settings from all the possible combinations [180], [181], [218].

As automatic configuration tuning often requires deep knowledge of the concerned system and disparate systems have different parameters, methods proposed for one system do not necessarily generalize to other systems [180]. As a result, most works focus on a specific type of distributed system or framework such as Enterprise Java (J2EE) [219], big data management systems such as Apache Hadoop [220] and Apache Spark [221], *Database Management Systems* (DBMSs) [215], distributed message systems such as Apache Kafka [182], web servers such as the Apache web server [222]–[224], or container management systems such as Docker [225] and Kubernetes [183].

Multi-tier web applications are among the systems with the largest body of work in automatic configuration tuning. *Xi et al.* propose a Smart Hill-Climbing algorithm using LHS sampling strategy for finding an optimal configuration for web application

servers [212]. They show the advantage of their approach over traditional heuristic methods through extensive experiments with an online brokerage application running in an IBM WebSphere environment.

Zheng at al. propose a method to simplify service management and eliminate misconfigurations in multi-tier Internet services based on parameter dependency graphs and the Simplex algorithm [213]. They demonstrate the effectiveness of their method with an online auction service and show that their method eliminates 58% of misconfigurations. Similarly, *Osogami et al.* propose the *Quick Optimization via Guessing* (QOG) algorithm for optimizing the configuration of web systems [226]. Other works that study the automatic configuration tuning of web systems include *Diao et al.* using an agent-based feedback control system [224], *Chung and Hollingsworth* using the Simplex algorithm [227], *Stewart and Shen* using profile-based models to predict service throughput and response time [228], and *Bu et al.* using *Reinforcement Learning* (RL) [229].

Another kind of distributed system with several works in automatic configuration tuning is the Hadoop big data analytics framework. *Starfish* is a self-tuning system for improving performance automatically on Hadoop big data analytic systems [230]. *Starfish* hides the complexity of having too many tuning knobs from users and maintains good performance of the system throughout the data analytics lifecycle. Similarly, the *ALOJA-Machine Learning* (ALOJA-ML) framework tunes the performance of Hadoop by using machine learning techniques based on benchmark performance data [231].

DBMSs have also received a great deal of attention in the automatic configuration tuning literature. *iTuned* automates the task of identifying good settings for database configuration parameters using a technique called *Adaptive Sampling* for sampling and *Gaussian Process Regression* (GPR) [232]. *iTuned* reports the performance impact each database configuration parameter has on a database workload. Similarly, *OtterTune* uses supervised and unsupervised machine learning techniques for automatic configuration tuning of DBMSs [215]. *OtterTune* uses a feature selection technique for linear regression called *Lasso* to identify the parameters that have the strongest impact on the system's performance. Then, it uses GPR to recommend the configurations that improve the target metric.

As container-based systems such as Docker and Kubernetes only recently became popular, there are only a few works in the literature that study their automated configuration tuning. *Chiba et al.* propose a configuration tuning framework for containers on Kubernetes called *ConfAdvisor* which uses a heuristic rule-based approach to improve applica-

tion performance by automatically detecting and correcting misconfigurations [183]. *ConfAdvisor* offers a framework to develop performance improvement configuration advice for container images, containerized applications, and Kubernetes deployment specifications. *ConfAdvisor* achieves significant performance improvement in Cassandra, Liberty and MongoDB as compared to the default configuration. Similarly, *k8s-resource-optimizer* is a tool implemented on top of Kubernetes for automatic and cost-effective tuning of Service-Level Objectives (SLOs) [233]. It is based on black-box performance tuning algorithms such as *BestConfig* [180] and Bayesian optimization. In the context of a simple job processing application, the authors show that their approach can find optimal configurations for different deployment settings and different types of resource parameters.

BestConfig is a system that finds the configuration settings that optimize the performance of general systems under specific workloads [180]. It uses DDS as the sampling technique and *Recursive Bound and Search* (RBS) as the performance optimization algorithm. *BestConfig* significantly improves the throughput of Tomcat, Cassandra and MySQL, and the running time of Hive and Spark join jobs. Similarly, *BOAT* is a framework that allows users to build their own auto-tuners leveraging domain knowledge about the structure of their systems using *Structured Bayesian optimization* (SBO) [234]. Evaluations show that it outperforms traditional auto-tuners in complex tuning problems conducted on databases and neural networks.

In Chapter 5, we use PCA to reduce the sample space to identify the most important parameter that affects the stability of the system. We then use a control-theoretic approach to tune the value of the concerned parameter at runtime in response to changes in network conditions, workloads and user traffic. Unlike the optimization techniques proposed in most of the works in the literature, our approach is not to find the best configuration for a given workload, but rather to adapt the configuration at runtime motivated by the fact that a single best configuration does not exist in a volatile environment such as a geo-distributed multi-cluster environment.

3.2.4 Automatic configuration tuning for improving failure detection and recovery

In order to be fault-tolerant, distributed systems make use of failure detection and recovery mechanisms to detect failures quickly and reliably [235], [236]. A formal specifi-

cation of failure detectors shows that failure detectors can be used to address some fundamental problems in distributed systems such as consensus and atomic broadcast [237].

Failure detectors in message-passing distributed systems face two main challenges. First, failure detectors may be slow to detect failures if it takes long to suspect that a process has crashed. Second, they may incur false positives, i.e., they may suspect that a process is faulty when this process actually is not. This might occur, for instance, due to message delays or losses [235]. This is particularly true in geo-distributed computing environments where the networking environment is characterized by low bandwidth, high latency, and a high probability of network packet loss [71]. To detect failures accurately and reasonably fast, *Chen et al.* propose a failure detection algorithm and *Quality of Service* (QoS) metrics to specify failure detectors for systems with probabilistic behaviors [235].

Failure detection algorithms commonly rely on timeouts [235], where the failure detector starts a timer with a fixed timeout value every time it receives a heartbeat from the process it monitors. If the heartbeat is received within the timeout, the failure detector trusts the monitored process, otherwise, it starts suspecting it has failed. However, selecting appropriate values for timeout parameters has proven particularly challenging in a number of situations. Typically, large values result in slow failure detection whereas small values reduce the reliability of the failure detector. To address this challenge, some works have proposed delay predictors that determine timeout detection values during runtime, for fast detection while not reducing the reliability of detection [235], [238]–[240].

In [241], the authors propose an autonomic failure detector based on feedback control theory that re-configures its timeout and monitoring period parameters at runtime in response to changes in the computing environment or application according to user-defined QoS requirements. Similarly, in Chapter 5, we show that small values for the timeout parameter of KubeFed lead to instability whereas large values may lead to slow failure detection.

In Chapter 5 we propose a control-theoretic approach to adaptively tune the timeout parameter of the failure detector in a geo-distributed multi-cluster environment. Our approach helps to mitigate the undesirable instability caused by premature false-positive detections. Our main focus is to improve the stability of application deployments without impairing the responsiveness of failure detection, unlike most of the works in the literature that focus on finding the right trade-off between accuracy and responsiveness of failure detectors. Moreover, unlike these works, our work addresses a problem in a geo-distributed fog computing environment.

Table 3.2.4 summarizes the literature in automatic configuration tuning.

Table 3.2 – Summary of the literature on automatic configuration tuning.

Ref.	**Optimized / Tuned System**	**Network Environment**	**Sampling Method**	**Optimization/ Tuning Method**
[180], [242]	Tomcat, MySQL, Cassandra, Hadoop, Hive, Spark	*Local-Area Network* (LAN)	DDS, LHS	RBS, *Recursive Random Search* (RRS)
[234]	Neural network	LAN	Random sampling	SBO
[212], [213], [226], [224], [227], [228], [229]	Web servers	LAN	Random sampling, simulated annualing, LHS, sequential-stage with checkpoints	Black-box optimization, Simplex algorithm, QOG, control theory, profile-based performance model, RL
[230], [231]	Apache Hadoop	LAN	N/A, random sampling	Just-in-time optimization, *Machine Learning* (ML)
[232], [215]	DBMS	LAN	Adaptive sampling, LHS	*Gaussian process Representation of a response Surface* (GRS), ML
[183], [233]	Kubernetes	LAN	N/A, hypercube sampling	heuristics rule-based, black-box Bayesian optimization
Contribution 2	KubeFed	*Wide-Area Network* (WAN)	DDS	Control theory

3.3 Container orchestration in geo-distributed environments

Ever since the emergence of cloud computing as the dominant computing paradigm, there has been a growing interest in making the best out of the available resources, location, pricing schemes, and offerings of multiple cloud providers. Deploying applications across multiple cloud providers maximizes scale, availability, performance and fault tolerance [25], [243]. By maximizing choice, the multi-cloud approach also improves cost efficiency for cloud users and profitability for CSPs [244]. Other issues such as avoiding vendor lock-in or complying with regional compliance can also be addressed by using multi-cloud deployments [245].

Previous works have introduced cloud federation, hybrid cloud, multi-cloud, and aggregated service by brokers to deal with the challenges of interoperability and standardization in different cloud providers [246]. Also, several architectural frameworks and platforms have been introduced to achieve the goal of federated and multi-cloud computing such as RESERVOIR [164], OPTIMIS [247] and Contrail [248], to name a few. Other works have looked at optimizing the placement of VMs in multi-cloud environments to optimize performance and cost [135]. In particular, in hybrid cloud scenarios, cloud bursting allows offloading applications from private *Data Center*s (DCs) to public cloud DCs to handle workload spikes [117], [249]–[254].

In recent years, fog computing has emerged as a decentralized paradigm that extends cloud computing to where the data is generated and users are located [8], [9]. As resources in a fog computing environment are geographically distributed with heterogeneity in resources, network characteristics and location, the problem of resource management has been revisited by a number of works. Some works focus on optimizing the placement of jobs and services [122], [255], whereas others address the joint problem of placement and autoscaling [129], [184], [256], [257].

Earlier works on resource management in geo-distributed environments relied on VMs [8], [18], [135], [258], [259]. However, more recently, containers have been largely adopted due to their lightweight and portable nature, as well as several other benefits such as fast start-up time and scalability. Several container orchestration platforms have been proposed to automate the placement and scaling of containerized applications, and recent works have started using these frameworks for addressing the challenges in geo-distributed computing environments. Therefore, in this section, we first review the container orchestration

frameworks proposed for resource management and application placement. Next, we review the literature focusing on the use of containers for placing, autoscaling and bursting containerized applications. Then, we look at the approaches for virtualized network traffic routing between application components. Lastly, we look at works discussing dynamic provisioning and de-provisioning of VMs for container placement.

3.3.1 Container orchestration frameworks

Currently container orchestration platforms are mostly used for placement, autoscaling and provisioning of containerized applications in a single DC [260]. Large internet companies such as Google developed their own internal container orchestration platforms [160]. Other enterprises use open-source solutions such as Kubernetes [49], Docker Swarm [47] or Hashicorp Nomad [261] in private and public cloud environments. Similarly, public CSPs have managed offerings such as *Amazon Elastic Kubernetes Service* (EKS) [262], *Google Kubernetes Engine* (GKE) [263] and *Azure Kubernetes Service* (AKS) [264]. These solutions are limited because they cannot address the features required in geo-distributed environments such as proximity-aware placement, autoscaling, bursting and dynamic resource provisioning. As a result, they fail to meet non-functional requirements such as proximity, availability and fault tolerance.

Although earlier research had focused on container orchestration in a single DC, few container orchestration frameworks for geo-distributed computing environments have been proposed recently.

Pahl et al. are among the first to propose the use of containers for application deployment in edge and fog computing infrastructures [34]. They propose a container orchestration architecture for the *edge cloud* that is a distributed multi-cloud platform consisting of nodes from cloud data centers as well as small single-board devices such as Raspberry Pis. They also propose an *edge cloud Platform-as-a-Service* (PaaS) which is a container-based PaaS architecture for Raspberry Pi clusters using containers and *Topology and Orchestration Specification for Cloud Applications* (TOSCA)-based service orchestration. Similarly, *Bellavista et al.* propose a container orchestration framework based on Docker Swarm and Kura open-source IoT gateways to run IoT applications on resource-constrained fog nodes such as Raspberry Pis [33].

PiCasso is a platform for deploying containerized services at the edge of the network [265]. Its architecture consists of a *Service Orchestrator* and multiple *edge nodes*, analogous to Kubernetes *master node* and *worker node*, respectively. Similar to our work,

this platform offers automated policy-based placement and network-aware network traffic routing. However, differently from our work, *PiCasso* assumes that all the nodes of the platform are in the same network service provider.

SAVI-IoT is an IoT platform based on microservices models to support big data processing at the edge [266]. This platform leverages both VMs and containers to manage *Internet of Things* (IoT) applications end-to-end. The architecture is organized into edge clouds and a core cloud, where the edge clouds represent cloud DCs closest to an IoT plant, and the core cloud represents a cloud DC with presumably unlimited resources. IoT sensors access the platform via IoT gateways located outside of the edge clouds. *SAVI-IoT* relies on Docker Swarm to manage containerized microservices. It provisions VMs in the core- and edge-clouds as necessary and deploys containers on them. Location-aware placement of containers on VMs is determined using labels. The authors also design and implement an Automatic Management System based on the *Monitor - Analyze - Plan - Execute over a shared Knowledge* (MAPE-K) loop for deploying and scaling applications on the platform. Autoscaling is performed both at the container and VM levels. AMS optimizes the application performance while preventing under-utilization of resources. Similarly, *Foggy* is a proof-of-concept framework for automated IoT application deployment in fog computing environments [267]. Based on Docker containers running on Raspberry Pi devices, *Foggy* provides dynamic resource provisioning and automated application deployment.

C-Ports is an orchestration platform that enables transparent deployment and migration of Docker containers across hybrid and multi-cloud environments taking into consideration user and resource provider objectives and constraints such as availability, capacity, utilization, cost, performance, security, or power consumption [245]. It is built on top of an open-source project called CometCloud, so unlike KubeFed, it is designed to be independent of Kubernetes. *C-Ports* addresses resource discovery, container placement and dynamic adaptation concerns in federated distributed infrastructures. The dynamic resource discovery and selection is based on a constraint-programming model. The show that the constraint-programming model allows faster container deployment than linear or integer programming models which emphasize an optimum solution. The authors demonstrate the effectiveness of *C-Ports* using two use case scenarios, namely cloud bursting and multi-cloud deployment.

Similar to our work, *C-Ports* supports not only the placement and autoscaling of containerized applications but also cloud-bursting and dynamic provisioning of the underlying infrastructure. In contrast to *C-Span*, we provide different heuristics-based placement

policies and threshold-based autoscaling rather than formulating these as optimization problems. Moreover, cloud-bursting in *C-Ports* is based on utilization threshold, unlike our work where cloud-bursting happens when there are unscheduled Pods.

HYDRA is a decentralized location-aware orchestrator for containerized microservices [268]. It aims at addressing scalability and geographical distribution challenges in edge and fog computing environments by building a peer-to-peer overlay network of nodes. In HYDRA, each node acts as both orchestrator and resource. Using simulation, the authors show HYDRA's scalability to 20,000 nodes. Unlike HYDRA, our orchestrator uses a centralized control plane in a hierarchical network architecture. We consider other geo-distributed computing environments such as hybrid cloud and multi-cloud in addition to edge and fog computing. Next, our orchestrator offers autoscaling, network routing and cloud provisioning capabilities in addition to location-aware placement policies. Finally, we evaluate our contributions in a realistic geo-distributed infrastructure.

Unlike many of the works presented, *mck8s* – our container orchestration platform for geo-distributed environments presented in Chapter 6 – addresses several aspects of resource management in geo-distributed computing environments such as placement, joint autoscaling of containers and VMs, bursting, dynamic VM provisioning, and de-provisioning. It also presents a novel multi-cluster application deployment and scaling model with several policies and with the possibility of extension. Significant attention has also been placed on ease of use and streamlining with existing abstractions of Kubernetes and KubeFed, including reuse of the declarative resource specification and request language. The platform is developed to be generic so that different geo-distributed use cases such as hybrid cloud, multi-cloud and fog computing can be supported. Although *mck8s* has been implemented and evaluated around Kubernetes and its ecosystem, we argue that the concepts of the platform are general enough to be applied to other orchestrators such as Docker Swarm.

3.3.2 Container placement

Container placement is a critical part of container orchestration platforms [269], [270]. Other functionalities such as autoscaling, bursting and traffic routing depend on it. Furthermore, placement has a significant impact on the non-functional requirements of containerized applications such as availability, performance and fault-tolerance, especially in geo-distributed computing environments [270]. In this section, we review the academic literature on container placement in hybrid and multi-cloud environments.

Guerrero et al. present an approach to optimize the placement of containerized microservices in multi-cloud environments based on *Non-dominated Sorting Genetic Algorithm II* (NSGA II) [271]. Using this approach, the cloud service cost, network latency among microservices, and microservices repair time are optimized by taking into consideration the scale of the microservices, their allocation in VMs, cloud provider type, VM type, and the number of VMs. The proposed approach makes scheduling decisions at both the container and VM levels. It decides on the allocation of containers in the most suitable VMs and the allocation of VMs in the most suitable cloud provider. Simulation results show a significant improvement over a greedy first-fit algorithm.

Aldwyan and Sinnott address the availability and performance challenges in multi-cloud environments using a genetic algorithm which places containerized web applications by taking proximity to users and inter-DC latencies into consideration [19]. In this approach, application components and data are replicated across DCs from different clouds to improve performance even in the presence of cloud outages. Moreover, this approach makes sure that failover components and data are located in DCs that are near end-users during failures. The proposed solution offers significant improvements in response times under normal and failover situations compared to latency-unaware placement policies.

Similar to these works, in this thesis, we propose greedy best-fit, worst-fit and network-aware placement algorithms as part of our orchestration platform for geo-distributed computing environments. As opposed to these works, our approach allows using the orchestration platforms in different geo-distributed use cases such as hybrid cloud, multi-cloud and fog computing environments. Differently from [271], we evaluate our placement policies on a realistic geo-distributed environment using realistic approaches. Moreover, unlike [19] we rely on continuous monitoring of resource utilization and network conditions instead of using static labels for placement decisions.

In addition to placement in hybrid and multi-cloud environments, other works aim at placement in fog computing environments. *Fard et al.* present a dynamic container placement algorithm called *Minimizing End-to-End Latency* (METEL) that selects the most suitable fog nodes to achieve the minimum response time for a given IoT service [272]. The scheduling algorithms take available computational capacity, proximity of computational resources to data producers and consumers, and dynamic system status into account when making scheduling decisions. The scheduling mechanism is implemented on top of Docker Swarm and evaluated using simulations on the *iFogSim* platform.

Following Kubernetes's success and widespread acceptance in cloud computing, some other works try to adopt Kubernetes in fog computing environments. *Nardelli et al.* approach the problem of container placement in geo-distributed fog computing environments as an *Integer Linear Programming* (ILP) optimization problem [184]. Similarly, *Rossi et al.* use ILP optimization and network-aware heuristics to solve the placement problem [256]. *ge-kube* is a Kubernetes-based container orchestration tool for deployment of containerized applications in geo-distributed environments such as fog and edge computing [257]. It addresses the absence of network and location-aware placement policies in Kubernetes by using an optimization problem formulation and network-aware placement heuristics.

Hona guarantees that fog applications are accessed below a certain tail latency by placing replicas of latency-sensitive applications as close to end-users as possible while making sure that replicas are load balanced [122]. *Hona*'s latency-aware scheduler is integrated into Kubernetes. It addresses the initial placement of replicas, and also updates the placements dynamically as user traffic patterns change. Similarly, *Santos et al.* extend the Kubernetes scheduler to support network-aware scheduling in Smart City deployments [273]. Their scheduling policy offers a significant reduction in network latency as compared to the default Kubernetes scheduling policy.

These works focus on the placement of containerized applications in fog computing environments. In contrast, our work is more generic and can be applied not only in fog computing but also in multi-cloud and hybrid cloud environments. Moreover, unlike these works, our approach follows a multi-cluster model rather than a single cluster whose nodes are geo-distributed. We also provide more placement policies such as greedy best-fit, worst-fit and network-aware to support a wide range of use cases. Unlike [122] we use the amount of network traffic received by a cluster as an indicator for proximity to end-users rather than optimizing for the tail latency. We rely on continuous monitoring of computing and network resources in the geo-distributed infrastructure to make scheduling decisions rather than node labels as done in [273] which may not scale well to larger sizes.

3.3.3 Autoscaling and bursting of containerized applications

Container orchestration platforms ensure acceptable performance and efficient resource utilization by automatically adjusting the resources allocated to containerized applications at run time despite changes in the number of requests served by the applications. Most of the literature focuses on autoscaling of VMs and containers in single-DC scenarios [52], [53], [58], [129], [158], [188], [189], [191]–[196]. However, some recent works address the

autoscaling of containerized applications in geo-distributed computing environments [129], [184], [256], [271].

Guerrero et al. present a horizontal scaling approach for containerized applications along with their placement algorithm based on NSGA II [271]. *Nardelli et al.* propose a model of container deployment and scaling called *Adaptive Container Deployment* (ACD) that optimizes the scale of containers using ILP optimization [184]. Similarly, *Rossi et al.* propose an RL-based approach to control the horizontal and vertical elasticity of containers taking into account the application response time taking into consideration resource demand and network delay between application containers[256].

Voilà scales application replicas in fog computing infrastructures in response to changes in end-user traffic by adjusting the number of replicas and placing them as close to end-users as possible while ensuring that they are not overloaded [129]. The authors rely on their previous work [122] to maintain the tail latency between end-users and application replicas. This solution is built on top of Kubernetes and addresses the lack of location awareness during horizontal pod autoscaling.

Similar to these works, in our work in Chapter 6, we address the autoscaling and placement problems jointly in a geo-distributed computing environment. Differently from these works, our proposed horizontal pod autoscaler uses *Central Processing Unit* (CPU) threshold-based heuristics which scale containers across VMs in multiple clusters. In contrast to some of the works that are evaluated using simulations only, we evaluate our autoscaler on a realistic geo-distributed multi-cluster computing testbed with realistic workloads.

3.3.4 Virtualized network traffic routing

In geo-distributed computing environments, it is important to route network traffic to the nearest application replicas by taking into account the inter-cluster and inter-replica latencies [20], [274]. In container-based environments, network virtualization technologies such as *Software Defined Networking* (SDN) have been used [178], [275]. Most of the literature on the subject focuses on networking services for single cloud providers. However, a few works have attempted to address this issue in geo-distributed computing environments.

Sirius is a network virtualization platform for hybrid and multi-cloud environments [276]. The *Sirius* network hypervisor is implemented as an SDN application on top of Docker with Open vSwitch [277] as software switch. Although *Sirius* allows defining arbitrary vir-

tual network topologies, it is not clear how it may be configured to optimize the latency between clusters and end-users.

In Kubernetes, *kube-proxy* is responsible for routing and load-balancing end-user requests to the application Pods. However, its functionality is based on a simple round-robin fashion without regard to the latency between end-users and application replicas. Another limitation of *kube-proxy* is that it is limited to a single cluster. Open source projects such as Cilium [163], Istio [278] and Linkerd [279] address inter-cluster network communication, routing and load balancing based on container technology. Cilium is an open-source container networking solution based on *Berkeley Packet Filter* (BPF) that offers a highly scalable container network interface, a replacement for *kube-proxy* and multi-cluster connectivity. It allows secure pod-to-pod connectivity in multiple clusters without any gateways or proxies. On the other hand, Istio and Linkerd are service mesh solutions that allow inter-pod communications through proxies. They require attaching a proxy container (a sidecar) to the application pods, which might incur resource overheads. For multi-cluster communication, both Istio and Linkerd provide solutions based on gateways.

Proxy-mity is a proximity-aware traffic routing system for fog computing environments built as an extension of Kubernetes's *kube-proxy* [20]. It addresses the issue that Kubernetes by default does not have a location-aware routing policy when forwarding user requests to application Pods, which is one of the requirements of fog computing. *Proxy-mity* allows adjusting the trade-off between load balancing and proximity using a simple configuration. It significantly reduces the user-to-replica latency compared to Kubernetes's *kube-proxy*.

In Chapter 6, we integrate our container orchestration platform for geo-distributed environments, *mck8s*, with Cilium cluster mesh allowing us to route network traffic between the Pods of a multi-cluster application that is placed in different clusters. We chose Cilium over Istio and Linkerd because it allows direct pod-to-pod secure communications without the need for sidecar proxies or gateways. This is important, especially during horizontal autoscaling and bursting of application Pods from one cluster to multiple clusters.

3.3.5 Dynamic provisioning and de-provisioning of cluster VMs

In certain situations such as cloud bursting, it is important to transparently acquire resources from another cloud provider during high load periods and release those resources when they are no longer needed [280]. This approach is different from VM autoscaling in

that the resources are acquired from multiple cloud providers or DCs. In this section, we look at mechanisms for dynamic provisioning and de-provisioning of VMs for deploying containerized applications.

Nardelli et al. present an elastic VM provisioning approach for container deployments as an ILP problem [281]. Their approach allows acquiring and releasing VMs on demand taking into account the heterogeneity of container requirements and VM resources while optimizing for QoS metrics such as the deployment time of containers and the cost of VMs. A comparison of this approach with greedy first-fit and round-robin heuristics shows that it achieves lower deployment time and cost.

In another work, *Nardelli et al.* incorporate a mechanism to acquire and release multiple geo-distributed VMs on-demand in their ILP-based ACD model [184]. This model optimizes multiple run-time deployment goals by scaling containers vertically and automatically and deciding their allocation on the right VMs. *Aldwyan and Sinnott* also incorporate dynamic provisioning of VMs from geo-distributed DCs in their approach to latency-aware placement of containerized web applications [19]. Their approach aims to minimize SLO violations in normal and failover conditions.

Hoenisch et al. address the joint provisioning and autoscaling of VMs and containers as a multi-objective optimization problem [158]. The authors deal with vertical and horizontal autoscaling of both VMs and containers, resulting in four-dimensional scaling problem. Evaluations of the system show that it provides significant cost savings in cloud environments compared to baseline methods from the state of the art.

Similar to these works, in Chapter 6, we propose a greedy heuristic for dynamic provisioning and de-provisioning of cloud VMs for container deployment. Unlike these works, our approach potentially provisions VMs from multiple private and public cloud providers. We also integrate a VM autoscaling mechanism to deal with changes in workload. Moreover, we de-provision cloud VMs when the resources are not used for an extended period of time. This allows avoiding unnecessary expenses while minimizing over-provisioning.

Table 3.3.5 summarizes the state of the art in placement, autoscaling and bursting of containerized applications as well as dynamic VM provisioning and de-provisioning in geo-distributed computing environments.

Table 3.3 – Summary of the literature on geo-distributed container orchestration.

Ref.	Addressed Problem	Geo-distributed Environment	Platform	Algorithm/ Method	Evaluation Type
[122], [272], [273]	Placement	Fog computing	Kubernetes, Docker Swarm	greedy heuristics	Experiments, simulations
[129], [256], [257], [271]	Placement, autoscaling	Multi-cloud, edge computing, fog computing	Kubernetes, N/A	ILP, RL, NSGA II, greedy heuristics	Experiments, simulations
[19]	Placement, auto-provisioning	Multi-cloud	Docker Swarm	genetic algorithm	Experiments
[184]	Placement, autoscaling, VM provisioning	Fog computing	N/A	ILP	Simulation
[20], [276]	Network traffic routing	Multi-cloud, fog computing	Kubernetes, Docker	Greedy heuristics	Experiments
[33], [34], [245], [265]–[268]	Container orchestration framework	Edge computing, fog computing, IoT	Docker, Docker Swarm	Greedy heuristics, constraint programming	Experiments, simulation
Contribution 3	Container orchestration framework	Multi-cloud, hybrid-cloud, fog computing	KubeFed	greedy heuristics, threshold-based policy	Experiments

Chapter 4

Experimental evaluation of Kubernetes cluster autoscaling

4.1 Introduction

One of the main innovations made possible by dynamic cloud resource provisioning is elasticity, where the set of compute, storage and networking resources allocated to an application can vary over time to accommodate fluctuations in the workload created by end users. The choice of the amount of resources allocated to an application is typically made by an autoscaler which dynamically adjusts the amount of resources according to user demands. Numerous autoscalers have been proposed over the years to react to variations of either measured workloads (e.g., [52]) or short-term predictions of future workloads (e.g., [53]). Other autoscalers combine both reactive and proactive components (e.g., [56]–[59]).

Classical cloud platforms encourage the use of horizontal elasticity where capacity is adjusted by adding or removing identically-configured *Virtual Machine*s (VMs). In the same essence, Kubernetes – the leading open-source container orchestration platform – proposes the *Cluster Autoscaler* (CA) that dynamically adjusts the number and size of VMs on which containers run. Like the other Kubernetes components, CA is highly configurable. In its default configuration, CA adds or removes identical nodes. However, Kubernetes recently introduced the *Cluster Autoscaler with Node Auto-Provisioning* (CA-NAP) capability that adds nodes automatically from multiple node pools [65]. Unlike most autoscalers from the state of the art, CA-NAP allows dynamic provisioning of differently-sized nodes. This is especially useful when some pods have significantly lower or greater resource request than the rest of the pods in the workload. CA-NAP can then provision nodes that specifically match the request of these pods. Moreover, it has the potential for significant cost saving in public clouds by selecting the right-sized VMs to match the workload.

Although CA exposes some configurable parameters including CA-NAP, choosing the best configuration is far from being trivial. The objective of this chapter is to address the following questions. (1) How much cost saving does CA-NAP offer as compared to CA?; (2) How do the two configurations compare with regards to autoscaling performance? (3) How do CA and CA-NAP compare with other autoscalers in the related works?

To address these questions, we conduct extensive experiments on *Google Kubernetes Engine* (GKE) using two representative applications with respective real-world and synthetic workloads. We provide detailed analysis of the performance of CA in the two configurations using standard autoscaling performance metrics (i.e., under- and over-provisioning accuracy, under- and over-provisioning timeshare, instability of elasticity and deviation from the theoretical autoscaler) endorsed by Cloud Group of *Standard Performance Evaluation Corporation* (SPEC) [203].

Our results show that even though CA-NAP outperforms CA in terms of autoscaling performance, it does not offer significant cost saving. Moreover, the autoscaling performance of CA and CA-NAP is influenced by the composition of the applications deployed on the cluster. We show the potential of further performance improvement and cost reduction by tuning additional configuration parameters of CA.

4.2 Experimental setup

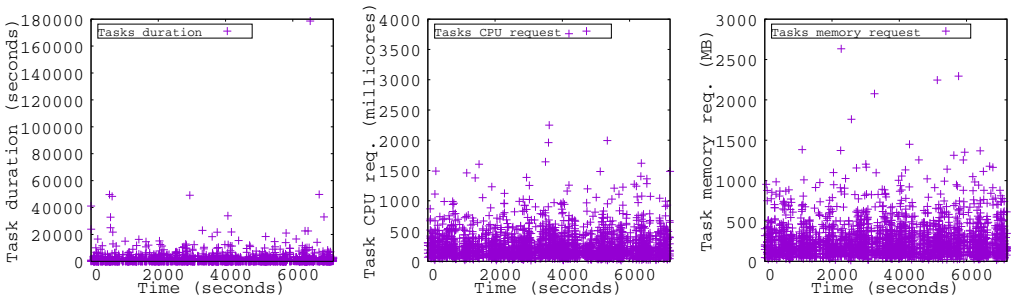
We present our experimental setup used for comparing the two strategies that Kubernetes uses for resizing the cluster, i.e., the default CA where nodes are provisioned only from one node pool and CA-NAP where nodes are provisioned from multiple node pools. In our experiments, we deploy two types of applications on GKE and for comparison, we use elasticity metrics from SPEC Cloud Group and the cost of running the clusters in GKE.

4.2.1 Applications and workloads

We designed two sets of experiments *E1* and *E2* which differ in the deployed applications and the corresponding workloads.

In *E1*, the application is composed of Kubernetes Deployments and Jobs based on a subset of the tasks data set in the widely studied Google cluster traces [1] [282]–[284].

1. Google Cluster Traces – https://github.com/google/cluster-data



(a) Tasks duration. (b) Tasks CPU request. (c) Tasks memory request.

Figure 4.1 – Characteristics of workload used in E1, which is based on Google cluster traces.

Table 4.1 – Workload characteristics for E1.

	min	**max**	**mean**	**std. err.**
Duration (seconds)	0.97	178,600.77	1,451.52	99.77
CPU req. (millicores)	15.00	3,760.00	258.41	4.73
Memory req. (MB)	13.00	2,633.00	256.83	4.46

These traces capture the characteristics of Linux containers used inside Google to execute tasks on the Borg compute clusters. The characteristics of our workload can be seen in Figure 4.1 and a summary of the statistics is presented in Table 4.1. Since the original traces contain 29-days-long data, we analyzed and fitted appropriate statistical distributions on the task duration, *Central Processing Unit* (CPU) request and memory request data sets separately, and generated a 2-hour-long workload. The workload contains a total of 2,467 tasks of different durations. We deploy long-running tasks as *nginx* web server Deployments and short running tasks as Jobs. We set the corresponding CPU and memory requests for the applications as per the CPU and memory request of the tasks in the workload traces.

In *E2*, we use a microservices-based test and reference application – *TeaStore* [205] – composed of six services. We chose this application as a representative because Kubernetes is often preferred to run microservices as it makes it easy to run the loosely coupled, self-contained components using containers and also provides abstractions for service discovery. We use a synthetic workload created on Apache JMeter with increasing and decreasing load intensity with up to 1000 concurrent user threads emulating users browsing the application using a profile which comes with *TeaStore*[2].

2. TeaStore Testing and Benchmarking – https://bit.ly/38OuRHD

4.2.2 Testbed setup

The experiment setup, whose details are shown in Table 4.2, consists of the application, a Kubernetes cluster with cluster autoscaling enabled, and a workload generator. Under each major class of experiment, we perform 6 experiments, each repeated 3 times, by varying the autoscaling configuration (*CA* or *CA-NAP*) and the size of the worker nodes (*small (4* Virtual Central Processing Unit*s (vCPUs), 15 GB* Random Access Memory *(RAM))*, *medium (8 vCPUs, 30 GB RAM)* or *large (16 vCPUs, 60 GB RAM)*). All experiments run on Kubernetes version 1.14.7-gke.14 in GKE in europe-west4-a region. CA is enabled for all clusters (default configuration for CA and with CA-NAP parameter enabled for CA-NAP). In all experiments, we inject the workload from VMs in Google Cloud in the same region but separate from the Kubernetes clusters.

In *E1*, all experiments start with only one worker node and CA adds/removes nodes to/from the cluster in response to workload changes. For each experiment in *E1*, we inject the workload for two hours and wait for an additional 30 minutes to observe scale-in.

For each experiment in *E2*, we start with different numbers of worker nodes to have just enough resources to place all six Deployments of the application. We have four worker nodes in *Scenario 1*, two worker nodes in *Scenario 2* and one worker node in *Scenario 3*. We deploy *TeaStore* services on Kubernetes using the Deployment manifest provided by the developers[3]. To automatically scale the Deployments in response to workload changes, we enable *Horizontal Pod Autoscaler* (HPA) for all six Deployments. The details of the configuration of the Deployments and HPA can be seen in Table 4.3. We access the application using the IP address exposed by the LoadBalancer Service of the front-end Deployment. For each of the six experiments we run the workload for 1 hour and wait an additional 30 minutes to observe scale-in.

3. Run TeaStore on Kubernetes – https://bit.ly/36w4K6M

Table 4.2 – Details of experiments setup.

Exp.	Scen-ario	Auto-scaler Type	Application	Workload	Node type	Starting no. of nodes	Cluster Autoscaler configuration			
							min no. of nodes	max no. of nodes	Max memory GB)	Max CPU
E1	1	CA	nginx Deployments and Jobs based on Google cluster traces	Based on Google cluster traces	small	1	1	100	-	-
		CA-NAP							1500	400
	2	CA			medium	1	1	100	-	-
		CA-NAP							1500	400
	3	CA			large	1	1	100	-	-
		CA-NAP							1500	400
E2	1	CA	Teastore	Synthetic with increasing and decreasing load intensity	small	4	4	100	-	-
		CA-NAP							1500	400
	2	CA			medium	2	2	100	-	-
		CA-NAP							1500	400
	3	CA			large	1	1	100	-	-
		CA-NAP							1500	400

Table 4.3 – Pods and HPA configuration in Experiments E2.

pod configuration				HPA configuration			
Request		Limit		Scaling metric	Threshold	min no. of pods	max no. of pods
CPU (cores)	Memory (MB)	CPU (cores)	Memory (GB)				
0.5	1024	0.5	1024	CPU	50%	1	100

4.2.3 Evaluation metrics

To assess the performance of CA we use some of the autoscaling performance metrics proposed by SPEC Cloud Group [203]. These metrics allow quantifying the autoscaling capabilities of the two Kubernetes autoscaling strategies and help the developer community to select the appropriate strategy for their workload. The provisioning accuracy metrics θ_U and θ_O describe the relative amount of under-provisioned or over-provisioned resources, respectively, during the measurement interval. The wrong-provisioning time-share metrics τ_U and τ_O measure the time in which the autoscaler under-provisions or over-provisions, respectively, during the time of the experiment. The instability of elasticity metric υ measures the fraction of time in which the demand and the supply change in different directions. The autoscaling deviation σ measures the deviation of a given autoscaler compared to the theoretically optimal autoscaler, that does not exist but is assumed to supply exactly the resources demanded by the workload. We calculate these metrics for the total CPU and memory demanded by our workload and supplied by the CA or CA-NAP in the different scenarios of our experiments. For each of these metrics, the smaller the value is the better the autoscaler performs for that metric.

The metrics and the equations used to calculate them are summarized in Table 4.4. Here we define: (i) T as the experiment duration and the current time as $t \in [0, T]$, (ii) s_t as the total amount of CPU cores or memory supplied by the cluster at time t, (iii) d_t as the total amount of CPU cores or memory demanded by the pods of the application at time t, and (iv) sgn is the signum function which is an odd mathematical function that extracts the sign of a real number. Finally, Δt denotes the time interval between the last and the current change either in demand d or supply s.

In addition to the autoscaling performance metrics, we also calculate the hourly cost of running the clusters use it to compare the autoscaling policies.

4.3 Results

The following are the main findings of our extensive experiments on the autoscaling performance of CA and CA-NAP.

1. **Overall, CA-NAP outperforms CA**, as it provisions differently-sized nodes to match the demand of the workload better.

Table 4.4 – Overview of autoscaling performance metrics.

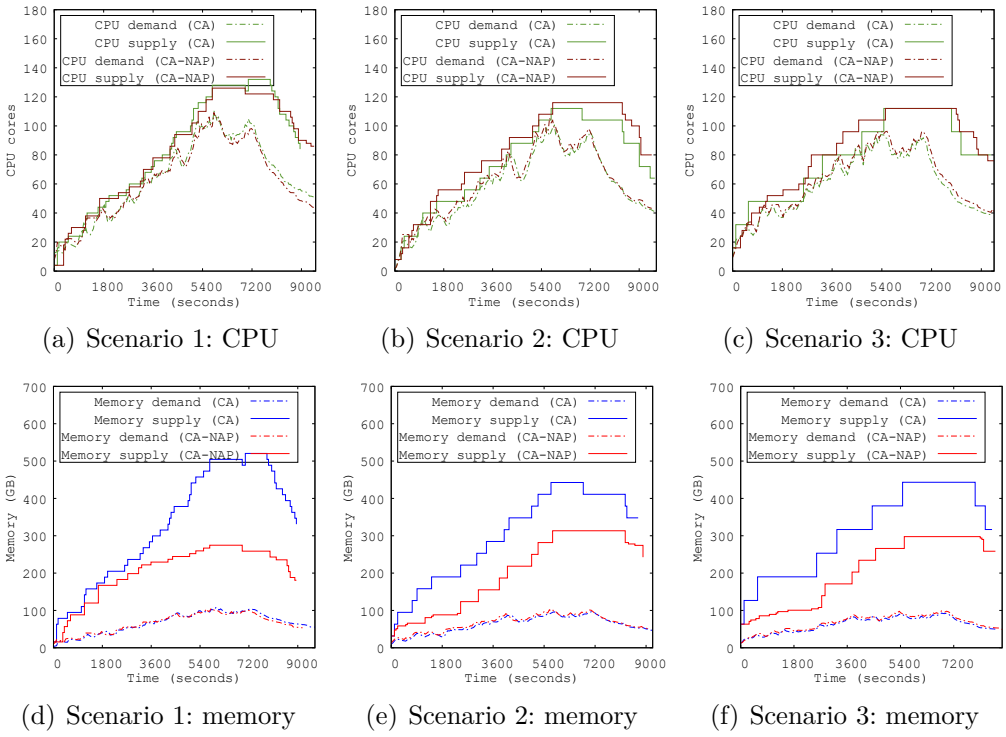
No.	Metric name	Equation
1	Under-provisioning accuracy	$\theta_U[\%] := \frac{100}{T} \cdot \sum_{t=1}^{T} \frac{max(d_t - s_t, 0)}{d_t} \Delta t$
2	Over-provisioning accuracy	$\theta_O[\%] := \frac{100}{T} \cdot \sum_{t=1}^{T} \frac{max(s_t - d_t, 0)}{d_t} \Delta t$
3	Over-provisioning timeshare	$\tau_U[\%] := \frac{100}{T} \cdot \sum_{t=1}^{T} max(sgn(d_t - s_t), 0) \Delta t$
4	Under-provisioning timeshare	$\tau_O[\%] := \frac{100}{T} . max(sgn(s_t - d_t), 0) \Delta t$
5	Elasticity instability	$\upsilon[\%] := \frac{100}{T - t_1} \cdot \sum_{t=2}^{T} min(\lvert sgn(\Delta s_t) - sgn(\Delta d_t) \rvert, 1) \Delta t$
6	Overall provisioning accuracy	$\theta := \frac{1}{2}(\theta_U + \theta_O)$
7	Overall wrong provisioning timeshare	$\tau := \frac{1}{2}(\tau_U + \tau_O)$
8	Autoscaling deviation	$\sigma[\%] := (\theta^3 + \tau^3 + \upsilon^3)^{\frac{1}{3}}$

2. Contrary to our expectations, **CA-NAP does not offer significant cost saving compared to CA.**
3. **The performance of CA-NAP is influenced mainly by the composition of the workload**, performing better for workloads made up of several short- and long-running pods with diverse resource requests.
4. **CA and CA-NAP show worse over-provisioning but better under-provisioning accuracy and under-provisioning timeshare** than the autoscalers studied in the **state of the art** [59], [197], [198], [204].
5. **CA and CA-NAP have the potential to offer even better performance** if the other configuration parameters such as autoscaling interval, scale-in time and expander are **tuned properly.**

The detailed discussion of our results follows.

Autoscaling dynamics

In Figures 4.2 and 4.3, we present the total CPU and memory demand of the application pods and the total CPU and memory of the worker nodes supplied by the autoscaling strategies in each experiment scenario. We present the plots from only one of the three runs of each experiment. Unlike some of the related works [59], [197], [204], the number



(a) Scenario 1: CPU (b) Scenario 2: CPU (c) Scenario 3: CPU

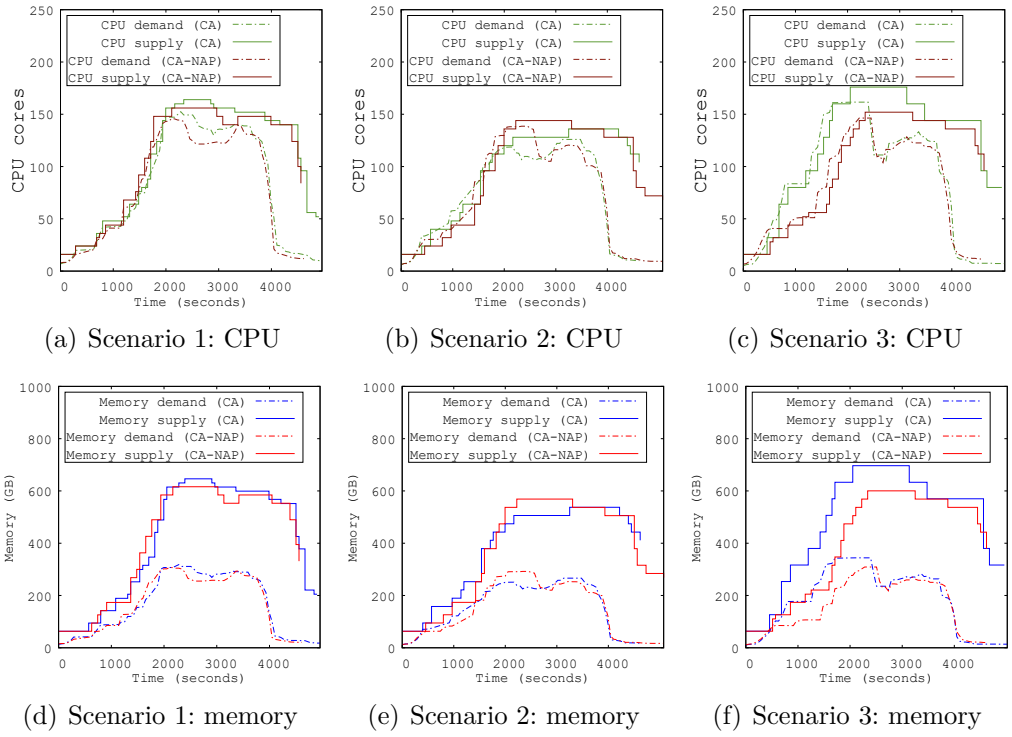
(d) Scenario 1: memory (e) Scenario 2: memory (f) Scenario 3: memory

Figure 4.2 – Resource demand vs. supply for experiments in *E1*.

of VMs could not be used for comparing CA and CA-NAP as the latter supplies nodes of different sizes. Instead, we report the total CPU and memory demand and supply.

The slight CPU over-provisioning by CA-NAP that can be seen in Figures 4.2(b) and 4.2(c) can be explained by the fact that CA-NAP supplies more smaller nodes during scale-out than CA as shown in Table 4.5. This reflects the resource overhead of CA-NAP due to the fixed amount resources reserved for the *Operating System* (OS) and system daemons on all worker nodes. Since more nodes are provisioned by CA-NAP than CA, the overhead becomes significant as it is multiplied by the number of nodes. As a result, we see more overhead in Scenario 3 than Scenario 2 as fewer nodes of larger size are supplied by CA in Scenario 3. We conclude that the overhead becomes larger as more nodes of smaller size are provisioned.

Another interesting observation is the memory over-provisioning by CA as compared to CA-NAP seen in Figures 4.2(d) – 4.2(f), reaching up to 100.73%, 30.8% and 48.30% for the three scenarios, respectively. This is because CA-NAP supplies nodes with higher CPU-to-memory ratio than CA, as can be seen in Table 4.5 reflecting the nature of the workload.



(a) Scenario 1: CPU (b) Scenario 2: CPU (c) Scenario 3: CPU

(d) Scenario 1: memory (e) Scenario 2: memory (f) Scenario 3: memory

Figure 4.3 – Resource demand vs. supply for experiments in *E2*.

We, therefore, conclude that for workloads made up of several short- and long-running tasks with high diversity in resource demand, CA-NAP matches the demand better than CA. Moreover, CA scales-in faster than CA-NAP in all 3 scenarios, but more so in Scenarios 2 and 3 as larger nodes are removed at a time than CA-NAP. Furthermore, in all plots for *E1* we see the impact of the fast autoscaling interval (10s) on CA-NAP in that CA-NAP performs more autoscaling actions than CA.

In the plots for Experiments *E2*, we do not see significant differences in the way CA and CA-NAP supply resources – except for the case of Scenario 3 (Figs. 4.2(c) and 4.2(f)), in which case CA provisions more CPU and memory than CA-NAP. This is because in *E2* at the application level the HPA scales-out the pods as the traffic intensity increases, creating multiple pods having the same resource request (0.5 CPU cores and 1024 MB of memory each). Unlike *E1* the pods in *E2* do not have diverse resource requests, and thus as shown in Table 4.5 the nodes supplied by CA-NAP do not have diversity in their CPU-to-memory ratio compared to those in *E1*.

Unlike most of the general-purpose autoscalers studied in [197] or the Chameleon [59] and Chamulteon [204] autoscalers, we see in all the plots for Experiments *E1* and *E1* that both CA and CA-NAP scale-in slowly. This is because of the scale-in cool-down period of

Table 4.5 – Total number of nodes and CPU and memory supply at the peak of Exp. *E1* and *E2*.

Exp.	Scenario	GKE machine type	Memory (GB)	No. of CPU cores	Qty
E1	CA Scenario 1	n1-standard-4	15	4	33
	CA-NAP Scenario 1	n1-standard-4	15	4	13
		n1-highcpu-2	1.80	2	5
		n1-highcpu-8	7.20	8	8
	CA Scenario 2	n1-standard-8	30	8	14
	CA-NAP Scenario 2	n1-standard-8	30	8	8
		n1-highcpu-4	3.60	4	12
		n1-standard-2	7.5	2	2
	CA Scenario 3	n1-standard-16	60	16	7
	CA-NAP Scenario 3	n1-standard-16	60	16	1
		n1-highcpu-4	3.6	4	12
		n1-standard-8	30	8	6
E2	CA Scenario 1	n1-standard-4	15	4	41
	CA-NAP Scenario 1	n1-standard-4	15	4	11
		n1-standard-8	30	8	14
	CA Scenario 2	n1-standard-8	30	8	20
	CA-NAP Scenario 2	n1-standard-8	30	8	16
		n1-standard-4	15	4	9
	CA Scenario 3	n1-standard-16	60	16	11
	CA-NAP Scenario 3	n1-standard-16	60	16	1
		n1-standard-4	15	4	10
		n1-standard-8	30	8	12

10 minutes configured in the autoscalers, which, unfortunately, is not currently modifiable in GKE.

Analysis of autoscaling performance metrics

We present the results of the autoscaling performance metrics in Tables 4.6 – 4.8. In these tables, the rows show the experiment sets and scenarios being compared whereas each column shows a metric. The metric values are reported as the mean from three runs of each experiment. To complement the results in the tables, we present the spider charts shown in Figure 4.4. Each spider chart contains six points on the circumference of a circle, one for each autoscaling metric. The closer to zero and the thinner the web, the better the autoscaling configuration performs.

The autoscaling performance metrics are the average of the respective metrics for CPU and memory provisioning calculated using the equations given in Table 4.4. *It is*

Table 4.6 – Autoscaling performance metrics for all scenarios in Experiments *E1* and *E2*.

Scenario	θ_U[%]	θ_O[%]	τ_U[%]	τ_O[%]	υ[%]	σ[%]	Cost($)
E1 CA 1	1.52	183.88	12.76	**87.66**	**83.33**	114.62	4.18
E1 CA 2	0.39	204.32	5.08	94.92	91.71	125.35	4.10
E1 CA 3	**0.32**	224.48	**3.16**	96.84	95.96	134.43	4.16
E1 CA-NAP 1	1.09	126.10	5.80	94.20	87.46	**101.72**	**3.91**
E1 CA-NAP 2	1.38	169.34	7.62	92.38	92.11	117.00	4.24
E1 CA-NAP 3	0.46	**124.14**	3.80	96.20	92.44	104.99	4.00
E2 CA 1	3.66	**85.42**	19.59	80.41	60.27	75.83	5.70
E2 CA 2	**2.67**	115.36	18.24	81.76	62.79	83.46	**5.47**
E2 CA 3	4.12	197.89	18.74	81.26	68.32	114.82	5.86
E2 CA-NAP 1	2.73	95.06	**15.37**	83.73	58.03	**75.81**	5.76
E2 CA-NAP 2	5.88	105.35	23.65	**76.35**	**55.94**	78.13	5.94
E2 CA-NAP 3	3.23	184.57	17.04	82.96	61.58	107.22	5.65

important to notice that, the smaller a value is, the better. The best values for each metric in the respective experiment set are presented as bold. The cost metric is calculated by aggregating the CPU and memory provisioned by the clusters for the duration of the experiment and multiplying by Google Cloud's per-hour pricing for CPU and memory. Here, we report the cost of running the clusters for one hour.

As shown in Table 4.6, almost all cases exhibit large values of over-provisioning accuracy θ_O. This can be explained by the over-provisioning of memory in all cases and scale-in delay of 10 minutes. This also explains the large values of the wrong over-provisioning time τ_O as the clusters are over-provisioned for the largest part of the experiment duration. This is because both CA and CA-NAP do not scale-out base on CPU utilization threshold but rather do so whenever there are pods that could not be scheduled due to a shortage of resources. The large values for θ_O are similar to those of some of the general-purpose autoscalers studied in [197] but different from those reported by [198], Chameleon [59] and Chamulteon [204], whereas almost all the above works except [198] report large values for τ_O similar to ours.

Again, unlike the findings in [59], [197], [198], [204], we show that CA and CA-NAP in all cases have better performance in terms of under-provisioning accuracy θ_U and under-provisioning timeshare τ_U. This is because of the small autoscaling interval of 10s and the fast VM booting times in Google Cloud, which, unlike the autoscalers in other works, allow CA and CA-NAP to provision VMs faster and minimize under-provisioning.

CA vs. CA-NAP overall comparison

First, to help compare CA and CA-NAP per experiment set, we present the metrics aggregated by auto-scaling configuration (i.e., CA and CA-NAP) for Experiments E1 and E2 in Table 4.7 as well as in Figures 4.4(a) and 4.4(b). The presented metrics in the table are the mean of nine measurements per autoscaling configuration. For the comparison in E1, CA-NAP shows the best θ_O (139.86%), τ_U (5.74%) and σ (107.90%) whereas CA performs better in the remaining three metrics. However, CA-NAP shows the best σ (107.90%) because its θ_O (139.86%) is significantly lower than that of CA (204.23%). In the case of E2, CA-NAP outperforms CA on all metrics except θ_U.

Next, we look at the comparison of CA and CA-NAP across all scenarios in the two experiment sets as presented in Table 4.8. The presented metrics are the mean from 18 measurements per each autoscaling configuration. The same results are plotted using a spider chart in Figure 4.4(c). In this comparison, CA-NAP outperforms CA in four out of six metrics: θ_O (134.09%), τ_U(12.22%), υ(74.59%) and σ(97.48%).

Cost comparison

In Tables 4.6 and 4.7, we also present the hourly cost of running the clusters in each scenario. Although the cluster under CA-NAP is cheaper than CA for the case of E1, and CA is cheaper in E2, we observe no significant cost savings by one over another. In the case of *E1*, although CA provisions far more memory than CA-NAP, it is not significantly more expensive than CA-NAP because CA-NAP slightly over-provisions CPU and the unit cost of memory is much less than that of CPU. For the case of *E2*, we do not see a significant difference in cost since both CA and CA-NAP are close to each other in resource supply.

Influence of workloads

Taking the deviation from the theoretically optimal autoscaler σ as the single most important metric, since it captures all the other metrics, we conclude that CA-NAP outperforms CA in autoscaling performance, more so in *E1* than *E2*. The composition of the workload influences the performance of CA-NAP in that it performs better for workloads like the one in *E1* that are composed of several short- and long-running tasks with diverse resource requests, thus allow it to provision differently-sized nodes to match the demand. The nature of the workload influences the cost of the cluster as well as discussed in Section 4.3. In *E1*, CA-NAP would have been significantly cheaper than CA

Table 4.7 – Autoscaling performance aggregated per experiment and autoscaling policy.

Exp.	AS Policy	θ_U[%]	θ_O[%]	τ_U[%]	τ_O[%]	υ[%]	σ[%]	Cost($)
E1	CA	**0.75**	204.23	7.00	**93.14**	**90.33**	124.80	4.14
	CA-NAP	0.98	**139.86**	**5.74**	94.26	90.67	**107.90**	**4.05**
E2	CA	**3.48**	132.89	18.86	81.14	63.79	91.37	**5.68**
	CA-NAP	3.95	**128.33**	**18.69**	**81.01**	**58.52**	**87.05**	5.78

Table 4.8 – Overall Autoscaling performance metrics for *CA* and *CA-NAP*.

AS Policy	θ_U[%]	θ_O[%]	τ_U[%]	τ_O[%]	υ[%]	σ[%]
CA	**2.11**	168.56	12.93	**87.14**	77.06	108.09
CA-NAP	2.46	**134.09**	**12.22**	87.63	**74.59**	**97.48**

if the workload resource request had high memory-to-CPU ratio as opposed to the high CPU-to-memory of our workload, in which case CA would have supplied far more CPU and would have been more expensive since the unit cost of CPU cores is far more than that of memory ($0.0347721 / vCPU hour vs. $0.0046607 / GB hour).

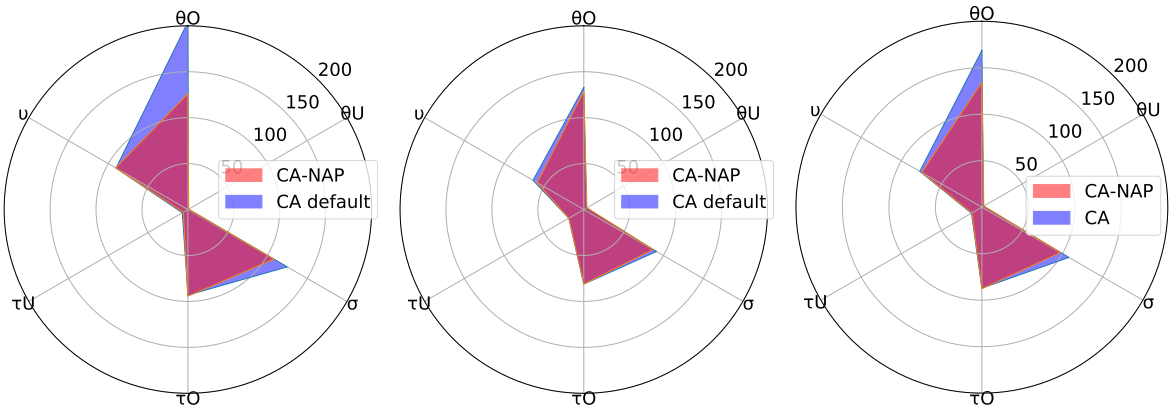
Influence of configuration parameters

The Kubernetes CA has several configuration parameters that influence its performance in addition to the strategy for node provisioning (CA and CA-NAP) we have studied in this chapter. The three most important parameters that would influence the performance of CA and CA-NAP are autoscaling interval, scale-in cool down time and extender. We have used the default values for these parameters in this work.

The autoscaling interval has a default value of 10 seconds and impacts the speed of scale-out, the number of scaling actions, the over- and under-provisioning timeshare metrics, and in the case of CA-NAP the size of nodes to be provisioned. The smaller the autoscaling interval, we observe faster scale-out, more autoscaling actions, better under-provisioning timeshare, worse over-provisioning timeshare, and in the case of CA-NAP smaller nodes are provisioned, and vice-versa.

The scale-in time has a default value of 10 minutes and influences the over-provisioning accuracy metric, speed of scale-in, and cost of the cluster. The smaller this parameter is the better over-provisioning accuracy, the faster scale-in and the lower the cost of the cluster, and vice-versa.

The extender parameter has 5 possible values (*random*(default), *most-pods, least-waste, price* and *priority*), and specifies the strategy used by CA-NAP to decide from



(a) CA vs. CA-NAP in E1 (b) CA vs. CA-NAP in E2 (c) CA vs. CA-NAP overall

Figure 4.4 – Scaling behavior overview using spider charts.

which of the multiple node pools to provision a node during scale-out. Because of the default *random* value used in our experiments, we observe that CA-NAP does not provision the same types of nodes at every run of the experiments. Therefore, it is important to study the behavior of CA-NAP with the other possible node pool selection strategies.

4.4 Conclusion

In the last few years, Kubernetes has emerged as the de-facto container orchestration platform in the cloud. However, although autoscaling is a widely-studied research topic in the community, the autoscaling mechanisms offered by Kubernetes remain largely unexplored.

In this chapter, we report results from our extensive experimental evaluation of the Kubernetes CA under two configurations. In its default configuration (CA) the autoscaler provisions nodes at the time of scale-out from only one node pool, whereas when configured with CA-NAP it provisions nodes from multiple node pools. We compare these two configurations using SPEC's autoscaling performance metrics and monetary cost of running the clusters. We show that CA-NAP outperforms CA overall because it provisions nodes of different sizes to match the workload demand better. CA-NAP shows better performance for applications that are composed of several short-running tasks and long-running services with diverse resource requests. The composition of the applications also influences the cost of running the clusters, as it determines the size and number of nodes to be provisioned. Moreover, CA and CA-NAP perform better in terms of under-provisioning

timeshare and worse in terms of over-provisioning accuracy compared to other autoscalers from the related work [59], [197], [198], [204].

In this work, we showed the impact of different configurations and applications on the autoscaling performance and cost of running of a Kubernetes cluster. As the Kubernetes CA is highly configurable, it would be interesting to study the impact of tuning additional parameters on autoscaling performance and cost saving. Furthermore, the complex interaction of the container-level autoscaling mechanisms (horizontal and vertical pod autoscalers) and VM-level autoscaling (cluster autoscaler) remains unexplored.

Chapter 5

Improving stability in geo-distributed multi-cluster environments

5.1 Introduction

Geo-distributed computing environments such as hybrid cloud, multi-cloud and fog computing extend cloud computing by harnessing geographically-distributed computing resources for moving computation closer to where data are generated (e.g., IoT devices). One of the main challenges in geo-distributed computing environments is the autonomous management of tens of thousands of remote nodes and clusters found in diverse locations. Several approaches based on modified container orchestration frameworks such as Kubernetes have been proposed [127], [257], [273]. More recently, Kubernetes introduced the notion of *Kubernetes Federation* (KubeFed) which provides abstractions to manage multiple geo-distributed Kubernetes clusters from a single control plane.

Since Kubernetes was designed for managing local clusters in public and private cloud settings, it assumes reliable network connectivity between the nodes with low latency, high bandwidth and low packet loss. KubeFed makes a similar assumption: while it is designed to manage Kubernetes clusters located in different regions of the same cloud provider or multiple cloud providers, KubeFed assumes high reliability of the network connectivity between the control plane and the managed clusters. However, such assumptions are not met in many geo-distributed computing environments [13]. As a result, static configurations, including the default values of the configuration parameters for both Kubernetes and KubeFed are not necessarily well-suited to the case of geo-distributed computing computing infrastructures.

It is well known that configuration parameters may have a strong influence on the performance and availability of systems [180]. However, finding the optimal configuration

settings that result in the best performance of the system is not easy because of the large parameter space and the complex interaction of multiple parameters. This is the case of Kubernetes and KubeFed which are composed of several embedded control loops [160] with numerous configuration parameters.

In this chapter, we demonstrate that federated applications deployed on a geo-distributed KubeFed infrastructure with static configuration, including the default settings, may suffer from important instability where containers get repeatedly created and deleted before being able to provide a useful service. To our best knowledge, we are the first to report this undesirable behavior of KubeFed.

Our contribution is two-fold. First, we demonstrate the existence of the instability problem in a realistic geo-distributed computing infrastructure based on KubeFed and identify one configuration parameter, *Cluster Health Check Timeout* (CHCT), whose value influences stability the most. We show that, to obtain the best system behavior, the value of this parameter should be adjusted according to the characteristics of the execution environment. Second, we propose, implement, and experimentally evaluate a feedback controller that improves the stability of the system by dynamically adjusting this configuration parameter at run time. We show that this controller is very effective for improving the system's stability across a wide range of inter-cluster network latencies and packet loss rates, without losing the ability to detect actual cluster failures. In our evaluations, the system stability improves from 83–92% with no controller to 99.5–100% using the controller. By enabling self-configuration and self-adaptation, our solution helps make KubeFed more autonomous and reliable in geo-distributed computing environments.

5.2 Analysis of instability in geo-distributed Kubernetes federations

When a replicated application is deployed in a Kubernetes federation, in certain settings, the application incurs significant instability where containers are repeatedly created and deleted, which in turn causes application unavailability and unacceptably long application response times. In this section, we first experimentally demonstrate the existence of this undesirable behavior, and then analyze its causes.

5.2.1 Experimental setup

To highlight the unstable deployment problem, we set up an experimental testbed as close as possible to a realistic geo-distributed computing environment, depicted in Figure 5.1. We deploy six Kubernetes 1.14 clusters in five sites of the Grid'5000 experimental testbed [285]: two in Rennes, and the other four clusters in Nantes, Lille, Grenoble and Luxembourg. Every cluster has one master node and five worker nodes. KubeFed v0.1.0-rc6 is deployed on the first cluster as the host cluster, and the remaining five member clusters are then joined to the federation. Each node in the host cluster has 4 *Central Processing Unit* (CPU) cores and 16 GB of *Random Access Memory* (RAM) allocated to it, whereas each node in the member clusters has 4 CPU cores and 4 GB of RAM. We control the network characteristics inside each cluster and between the host and member clusters using the "traffic control" (`tc`) tool available in Linux systems. The internal network of each cluster has 1 Gbps bandwidth, whereas the network characteristics between the host cluster and the member clusters are defined in Table 5.1. These values are based on a recent study [13] which highlights the characteristics of today's networking technologies used in edge computing settings.

The application used for our tests is a simple federated deployment of nginx web server that scales progressively. We scale the total number of replicas from 75, 100, 500, 1500, 2500, to 3500 to be distributed equally among the five member clusters of the federation. The task of automatically balancing the pod replicas across the member clusters is handled by KubeFed's *Replica Scheduling Preference* (RSP) controller.

We define three scenarios for our experiments:

— **Stationary scenario**: a federation with Network Setting 1, with no variation in the networking environment and no cluster failure;

— **Network variability scenario**: a federation where the networking environment varies between Network Setting 1 and Network Setting 2, with no cluster failure;

— **Cluster failure scenario**: a federation with Network Setting 1, with no change in the networking environment but with a failure and a recovery of one member cluster.

5.2.2 The instability problem

As the total number of replicas of the pods of the federated deployment increase, RSP calculates the number of replicas to be distributed to each member cluster. Unless

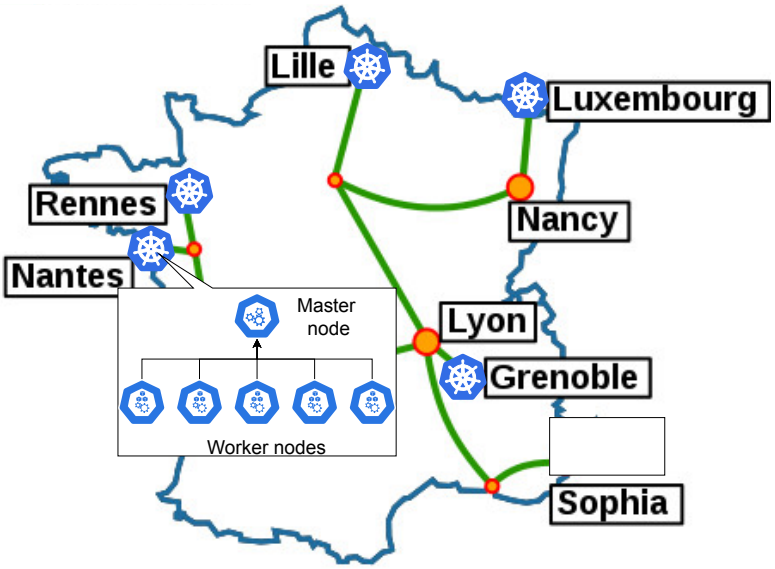


Figure 5.1 – Experimental setup in Grid'5000 consisting of one host cluster in Rennes, and five member clusters in Rennes, Nantes, Lille, Grenoble (France), and Luxembourg. Distances between sites range from 100 km to 850 km. Each cluster has a master node and five worker nodes. Image adapted from the Grid'5000 website.

Table 5.1 – Parameters of the network environment between the host cluster and the member clusters.

Network setting	Bandwidth (Mbps)	Packet loss (%)	RTT (msec) from host to member clusters				
			Rennes	Nantes	Lille	Grenoble	Luxembourg
1	15	5	100	102	123	117	127
2	10	10	200	202	223	217	227

other requirements such as the minimum number of replicas or weights per cluster are specified by the user, RSP opts for an even distribution across all member clusters. As per the source code of KubeFed, $clusterCount \times log_{10}(replicas)$ iterations are required to distribute all replicas among the member clusters, where $clusterCount$ is the number of member clusters and $replicas$ the requested number of containers. The time complexity of this algorithm is $O(clusterCount^2 \times log_{10}(replicas))$. After determining the number of replicas per cluster, RSP updates the Federated Deployment Object's *override* field. The changes are then automatically pushed to each cluster by the Sync Controller as depicted in Figure 2.8.

In a geo-distributed federation setup, there are two ways in which instability may arise due to network delays or transient network failures:

1. **Reconciliation failure**: Push reconciliation requests to one or more member clusters may time out prematurely at the time of scheduling by RSP, in which case the *sync*

controller tries to re-sync the resources until the desired state is achieved. If transient network and cluster failures continue to happen, it may take a long time for the reconciliation to terminate.

2. **Health check failure**: One or more member clusters may be declared *unhealthy* by the *kube-controller-manager* if health check requests time out because of transient disconnection of member clusters, in which case RSP re-calculates the distribution of replicas and rebalances them by moving replicas away from the now-unhealthy clusters to healthy ones. RSP re-syncs the resources to the unhealthy clusters, if they become healthy again. These actions may repeat over and over in network environments with a large number of transient failures.

The unstable behavior is manifested by the number of replicas on the affected member clusters being significantly fewer than the desired numbers, sometimes even reaching zero. Figure 5.2 shows the number of deployment replicas, which is the number of replicas pushed by the *kubefed-controller-manager*, and the actual number of running pods in one of the member clusters in our setup during a period of instability. As shown in the figure, the number of replicas that the *kubefed-controller-manager* pushes to the *member cluster* fluctuates widely over time, in turn affecting the number of pods actually running on the cluster.

To quantify the unstable behavior we introduce the *stability* metric υ as follows:

$$\upsilon[\%] := \frac{1}{n} \cdot \sum_{i=1}^{n} \left(100 - \frac{100}{T} \cdot \sum_{t=1}^{T} \frac{d_i - p_{i,t}}{d_i} \right) \tag{5.1}$$

where n is the total number of *member clusters* and $i \in [1, n]$; T is the full experiment duration and time $t \in [0, T]$; d_i is the desired number of pods in cluster i; $p_{i,t}$ is the number of running pods in cluster i at time t. Stability is a measure of how much and for how long the number of replicas in the member clusters is close to the desired number of replicas: a system which fails to deploy any replica during the entire experiment will have $\upsilon = 0\%$ whereas a perfectly working and stable system will have $\upsilon = 100\%$.

5.2.3 The influence of configuration parameters

Among the several configuration parameters in KubeFed, we identified eight which might influence the behavior and stability of the system such as timeout durations, health check periods and numbers of retries. We then measured the stability derived from 705

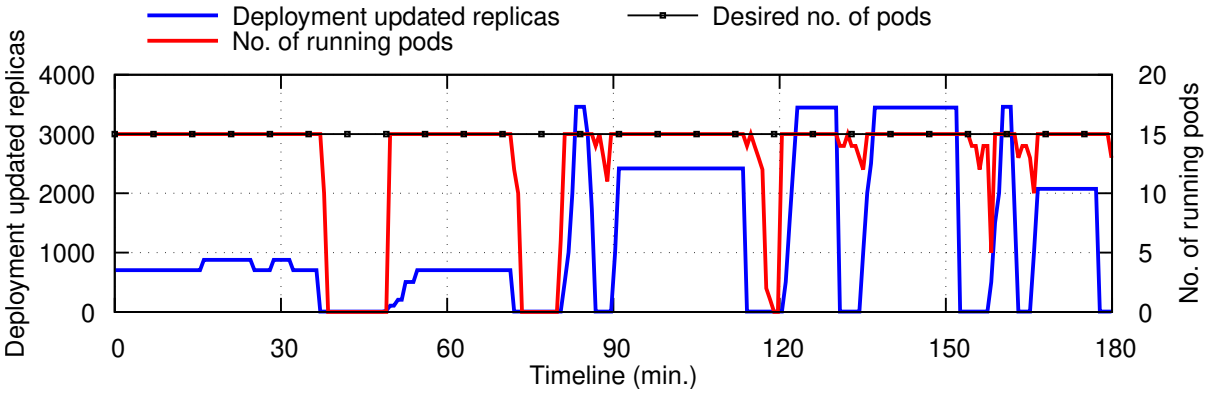


Figure 5.2 – The number of updated replicas of the deployment and the number of actual running pods on one of the member clusters of the federation.

sets of parameter values chosen according to the *Divide-and-Diverge Sampling* (DDS) strategy. Next, to identify which of KubeFed's configuration parameters has the greatest influence on the stability of the deployments in the member clusters, we conduct *Principal Component Analysis* (PCA) on the data obtained from the measurement of stability by varying the values of these parameters.

Our results show that the first source of stability variations between different configurations can be attributed to a single parameter. Specifically, we observe instability mainly when the *Cluster Health Check Timeout* (CHCT) parameter has too low values. Even the default value of 3 s for this parameter leads to significant instability. We also notice that increasing the value of the CHCT parameter significantly improves the stability of the system.

5.2.4 Trade-off between instability and failure detection delay

Although the stability of the system improves when the value of the CHCT parameter is increased, setting very large values to the CHCT parameter leads to slower failure detection as the system needs to wait until the CHCT timeout expiration before it updates the status of the failed cluster as "Offline." As shown in Figure 5.3, increasing CHCT leads to greater system stability; however, it also increases the failure detection delay. The goal of our controller is to identify a sweet spot which implements the necessary trade-off between these two effects.

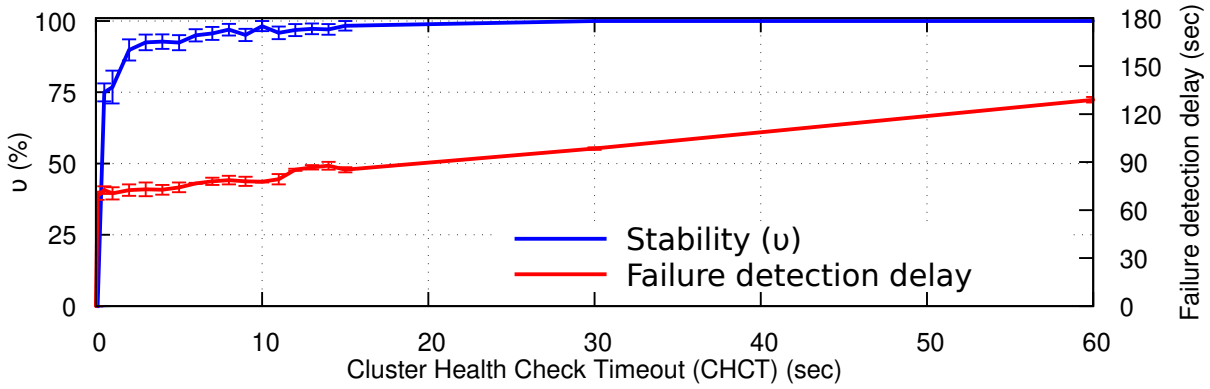


Figure 5.3 – Stability and failure detection delay as CHCT varies.

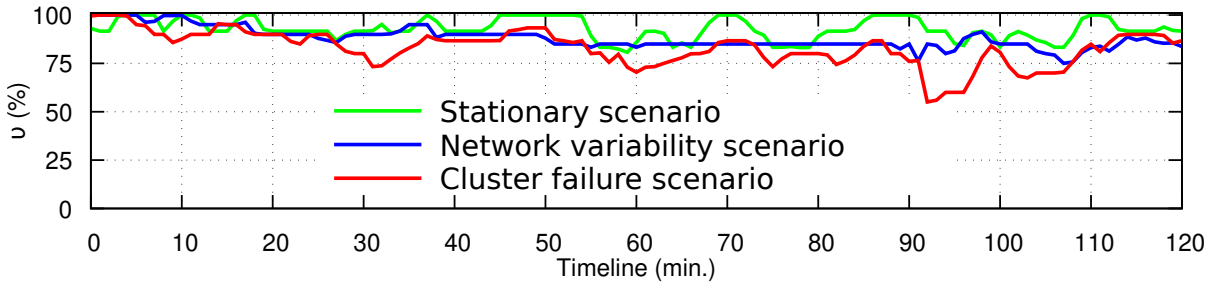


Figure 5.4 – Instability of the uncontrolled system under the three scenarios.

5.2.5 The influence of the networking environment

The last important factor which influences the occurrence of instability is the network performance between the clusters. Figure 5.4 depicts the stability of the system for the three scenarios. We see that the system is very unstable for the *Stationary scenario*. Moreover, stability gets worse as the network latency and packet loss is increased or one of the clusters fails in the *Network variability* and *Cluster failure* scenarios, respectively. Table 5.2 shows the system stability measures of the uncontrolled system under the three evaluation scenarios.

5.3 A control-based tuning approach to improve stability

Since the CHCT parameter has the highest impact on the stability of the system, a natural solution would consist of finding an optimal value for this parameter. Generally, the CHCT value should be set as low as possible to reduce the delay in detecting actual

Table 5.2 – Average no. of timeout errors per minute (N) and stability (υ) of the uncontrolled system for the three evaluation scenarios.

Experiment Scenario	**Avg.** N	**Avg.** υ **(%)**
Stationary	4	92
Network variability	3	87
Cluster failure	3	83

cluster failures, but not too low either because this would generate instability. However, no single "best" value can be found, as the choice of a good value largely depends on the operational conditions such as the inter-cluster network characteristics and the application workload.

Instead, we propose to dynamically adjust the CHCT value at run time using a feedback controller which reacts to changes in operational conditions. Unlike other configuration tuning methods, this adaptive approach does not require prior knowledge of the infrastructure and it is simple to implement. In this section, we present the details of our solution including our design decisions, controller design, and tuning of the controller parameters.

5.3.1 Feedback controller design

Feedback controllers are widely used in mechanical and electrical systems, and they have also gained widespread use in computer systems [286], [287]. A controller implements a feedback loop which monitors the system to be controlled, and implements automatic changes and then manipulates the input as needed to drive the system's variable toward the desired setpoint.

Figure 5.5 shows the design of our proposed solution. The controller continuously monitors the measured output, called Process Variable, PV, in control theory terminology, of the *kubefed-controller-manager* to detect indications that the CHCT value is either too high or too low. It then produces a signal called control output (CO) that reduces the error (e) that indicates the deviation of the measured output from the reference value SP. Finally, the controller decides whether CHCT must be adjusted, and the actuator implements the change.

Choosing the process variable A naïve approach would consist of periodically evaluating the KubeFed stability metric, and of incrementing the CHCT value whenever the measured stability differs from the desired setpoint of 100% ($e > 0$). However, this would

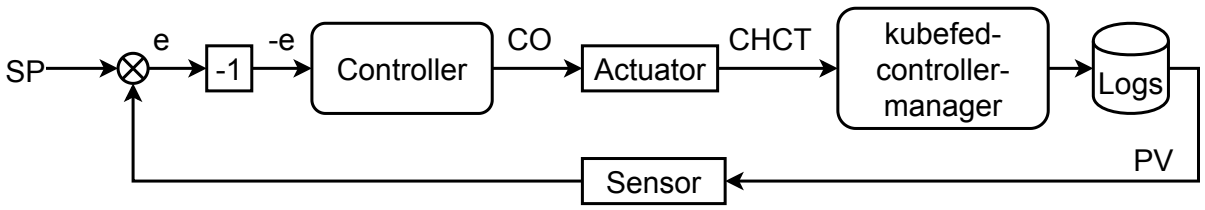


Figure 5.5 – System design of our controller.

mean that the system must enter a period of instability before the CHCT value is updated. This reaction would be too late for our purpose which is precisely to prevent instabilities from occurring.

It is, therefore, necessary to base the controller reactions on other metrics which show early indications that instability is about to occur. For this we use the timeout errors written by the *kubefed-controller-manager* in its log file whenever it fails to reach one of its member clusters. KubeFed starts deleting containers in the failed cluster and restarting them in other clusters when these timeout errors accumulate, thereby potentially triggering instability. We can thus use the occurrence of the first timeout errors in the log file as early indications that instability may soon take place.

Another motivation for selecting the number of timeout errors as the process variable PV is because this metric is readily available in the host cluster where we deploy our controller, unlike stability which needs to be computed after collecting metrics from each individual member cluster. Frequently collecting metrics from the member clusters may be very difficult, especially in periods of bad network performance when the CHCT value needs to be quickly adjusted.

Controlling the CHCT value If no timeout errors are reported, then we know that the system should achieve 100% stability. We, therefore, define the setpoint SP to 0 timeout error. As a result, the controller increments the CHCT parameter value until the number of timeout errors found in the log files during the control interval reaches zero.

However, setting SP to zero creates a new problem. In the standard feedback control theory, one should allow both positive and negative errors so that the controller can automatically increase or decrease CO proportional to the error. In our case, since we define SP as zero, it is impossible to observe a number of timeout errors lower than the setpoint, and the controller cannot decrease the CHCT value as a result of such negative errors. As a result, even though we can increase CO proportionally to the error, for the decreasing part we need to deviate from the standard approach of feedback control design

and come up with a different approach. For simplicity, we decided to decrease CHCT periodically if no timeout error has been identified, independently from any indication that the CHCT value may be too high. The controller ensures the trade-off between improving stability and fast detection of failures by preventing CHCT from reaching very large values that could lead to increase in the failure detection delay.

We choose a sampling interval of 1 minute for practical reasons. To change the CHCT value of a running KubeFed, it is necessary to stop and restart the containers which execute the *kubefed-controller-manager*. This operation takes a few dozen seconds. A sampling interval of 1 minute, therefore, gives enough time for the system to change the CHCT value before starting the next iteration of the control algorithm.

Control algorithm Our control algorithm is presented in Algorithm 1. The controller periodically measures the number of timeout errors which occurred in the previous period, and compares it to the setpoint $SP = 0$. If timeout errors have occurred, then the controller increments CHCT by a value proportional to the number of timeout errors and to the positive gain K_p, which is in line with the standard design of a proportional feedback controller. On the other hand, if no timeout errors have been found during three consecutive periods, the controller decreases CHCT proportionally to the negative gain K_n.

The two gain parameters K_p and K_n respectively define how aggressive the controller should be in increasing and in decreasing the CHCT value.

5.3.2 Tuning the controller parameters

Defining the controller parameters requires one to find a trade-off between a system which would react too slow to environment changes to provide an appropriate reaction and one which may potentially over-react to any such changes.

To get an initial estimate for K_p we use the Ziegler-Nichols rules, which are a set of simple heuristics that perform well in a wide variety of situations [288]. The Ziegler-Nichols tuning method does not require detailed knowledge of the controlled system, and the rules can be expressed entirely in terms of the system's step-input response (i.e., the system's reaction characteristics upon a change of its parameter value) [286].

Figure 5.6 shows the step response of the system as the CHCT parameter is suddenly increased from the default value of $CHCT = 3\,sec$ to $CHCT = 13\,sec$ at time $t = 10\,min$.

Algorithm 1: Feedback controller algorithm.

```
Data: Positive Gain K_p, Negative Gain K_n
Result: CO
initialization;
SP := 0;
decrement_period := 3;
t := decrement_period;
while true do
    PV = number of timeout errors;
    e = -(SP - PV);
    CHCT = current value of the CHCT parameter;
    if e > 0 then
        CO = CHCT + K_p * e;
        t = decrement_period;
    else
        Do nothing;
    t = t - 1;
    if t == 0 then
        CO = (1 - K_n) * CHCT;
        t = decrement_period;
```

From the system's step response, we estimate three parameters that are used in the Ziegler Nichols tuning rules:

— The process gain K is the ratio of the change in process output ΔPV that results from a change of input ΔCO:

$$K = \frac{\Delta PV}{\Delta CO}$$

— The time constant T is the time it takes for the process to settle to a new steady-state after experiencing a sudden change in input, i.e., the time it takes the process to reach about two-thirds of its final value.

— The dead time τ is the delay until an input change begins to affect the output.

From Figure 5.6, we find $K = 1.5$, $T = 60\,s$, and $\tau = 60\,s$. Based on these step-input response values we can define K_p:

$$K_p = \alpha \times \frac{T}{K \times \tau}$$

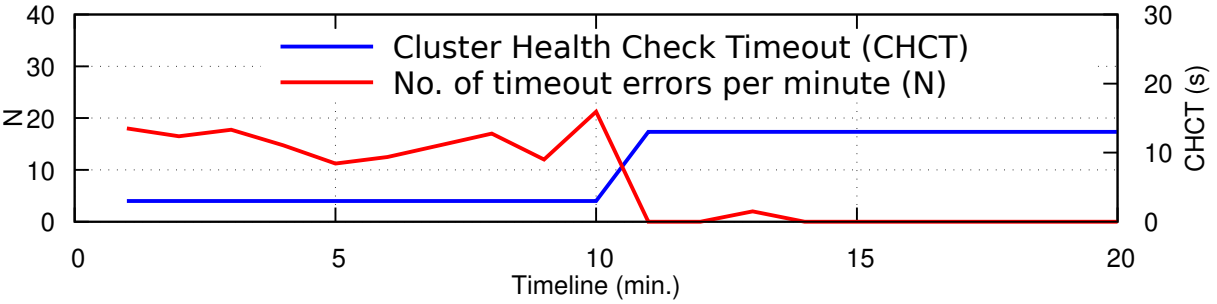


Figure 5.6 – Step response of KubeFed controller manager as the CHCT parameter is increased from the default 3 s to 13 s.

where α is a coefficient which typically falls in the range $[0.3, 1.2]$ [286]. We can therefore estimate that K_p should fall in the range $[0.2, 0.8]$. Based on these estimations, in the next section we experiment the controller with K_p values of 0.1, 0.5 and 1. Similarly, we use K_n values of 0.1, 0.25 and 0.5.

5.4 Evaluation

5.4.1 Experimental setup

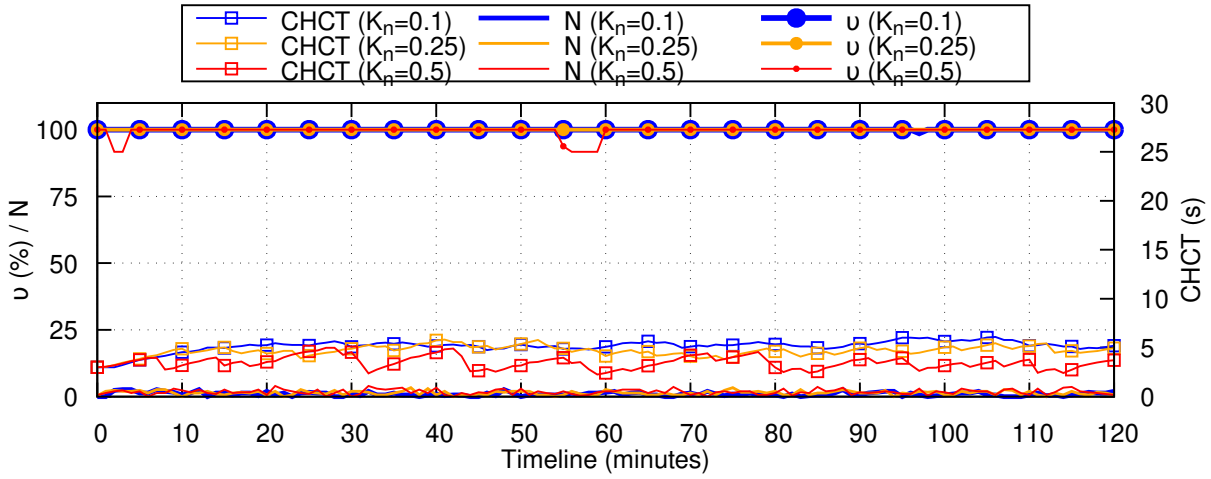
We evaluate our controller using the same experimental setup as described in Section 5.2.1 with the same application and workload for the three scenarios. However, now we also deploy our controller on the master node of the host cluster.

We run a total of nine experiments per scenario, one for each combination of the K_p and K_n parameters of the controller. Each experiment is run for two hours. We repeat each of the 27 experiments three times and report the mean value.

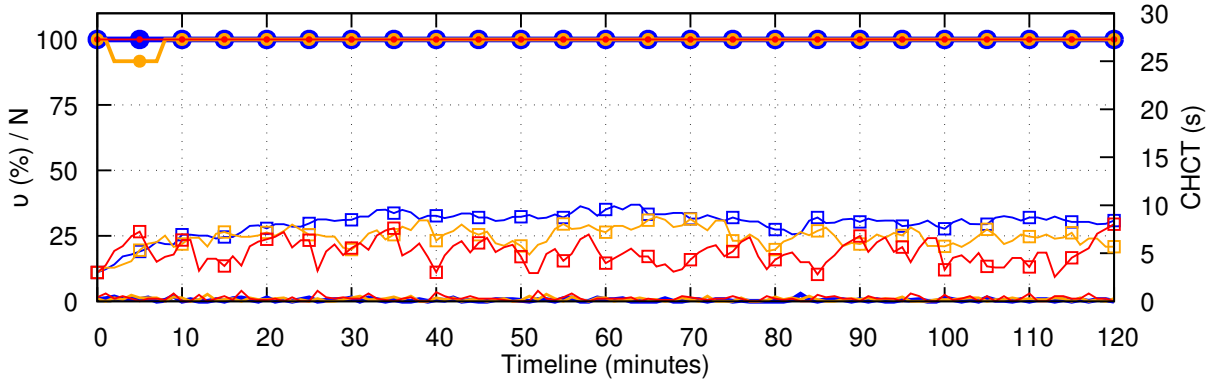
5.4.2 Experimental results

We present the results of the stationary, network variability and cluster failure scenarios in Figures 5.7, 5.8 and 5.9 respectively. In all scenarios, the controller adjusts CHCT according to the conditions, and significantly improves the federation stability compared to the no-controller scenario from Figure 5.4.

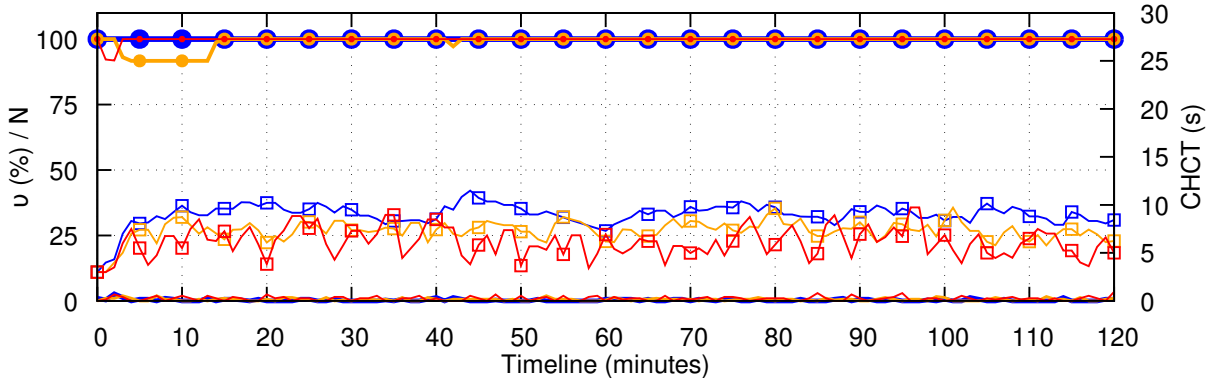
In the *Network variability scenario*, CHCT increases from $t = 30\,min$ as a reaction to the degraded networking performance, and decreases back at $t = 90\,min$ when network performance returns to normal. Similarly, in the *Cluster failure scenario*, CHCT increases after $t = 30\,min$ as a reaction to the detected cluster failure, and decreases from $t = 90\,min$



(a) $K_p = 0.1$

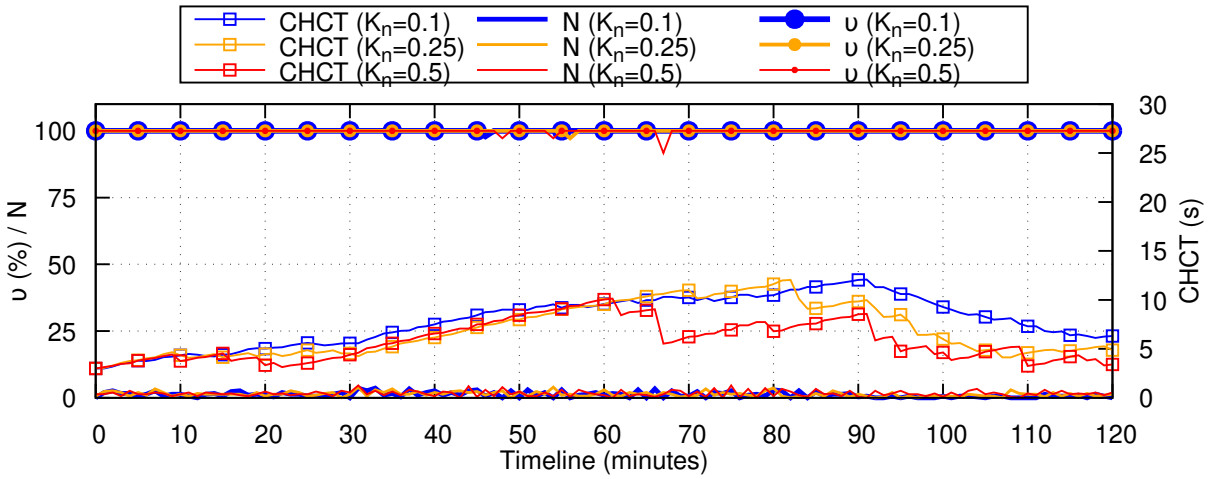


(b) $K_p = 0.5$

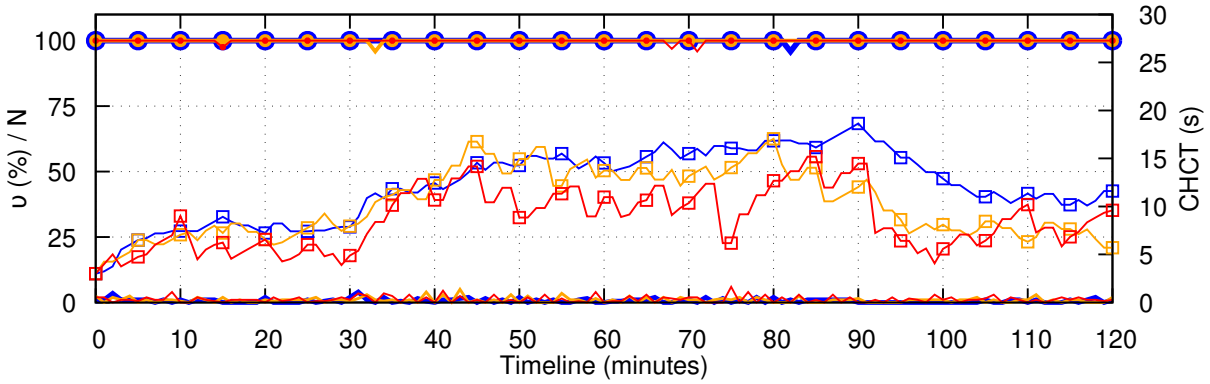


(c) $K_p = 1$

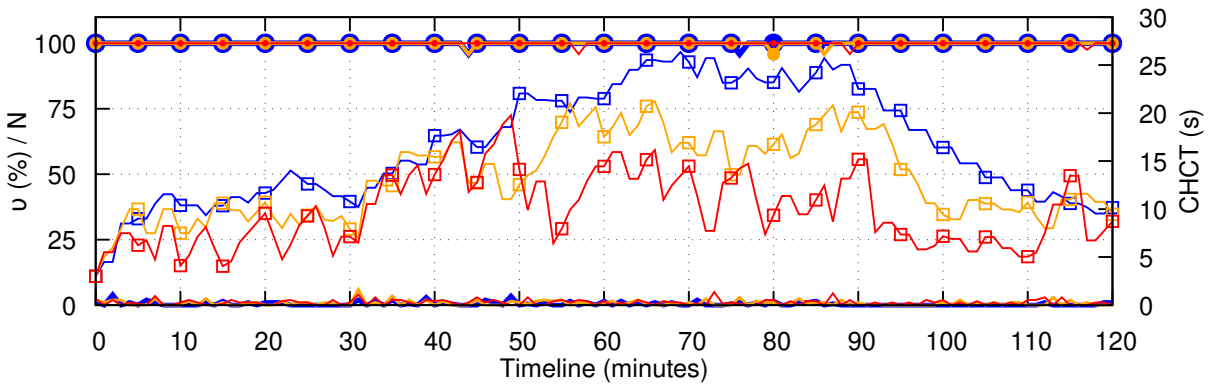
Figure 5.7 – Stationary scenario with different values of K_p.



(a) K_p=0.1

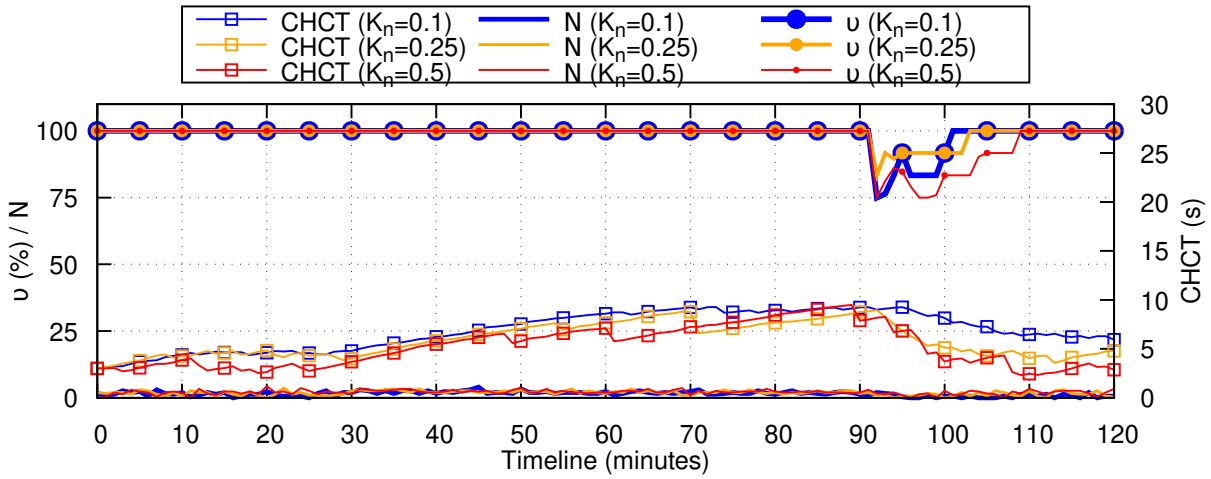


(b) K_p=0.5

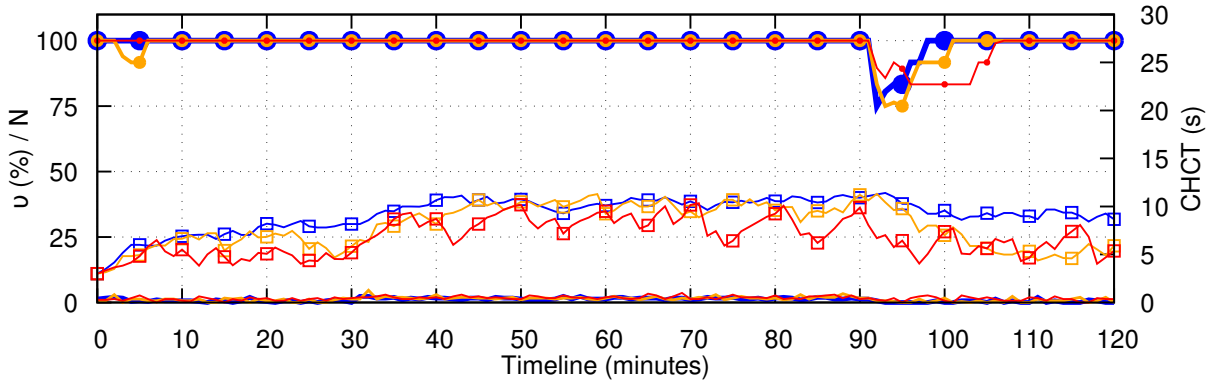


(c) K_p=1

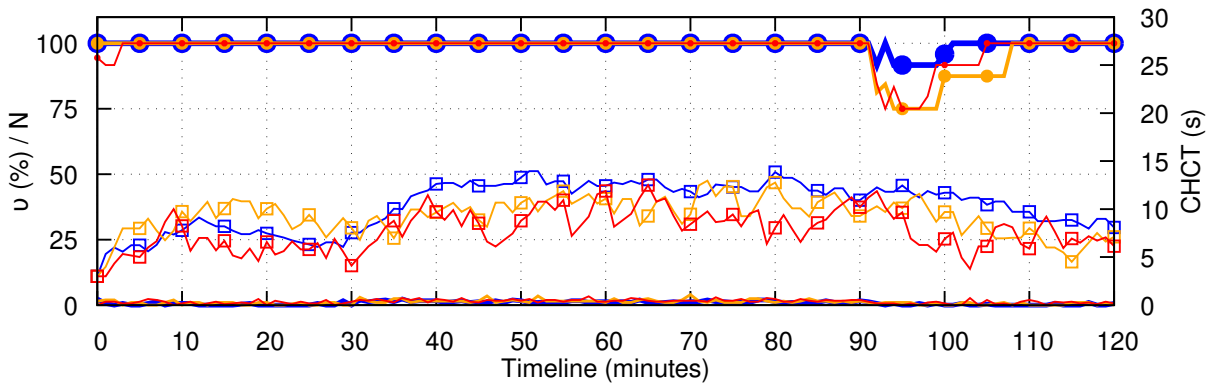
Figure 5.8 – Network variability scenario: network latency increases at $t = 30\,min$ and decreases back at $t = 90\,min$.



(a) K_p=0.1



(b) K_p=0.5



(c) K_p=1

Figure 5.9 – Cluster failure scenario: one cluster fails at $t = 30\,min$ and recovers at $t = 90\,min$.

Table 5.3 – Accuracy of the proposed controller as the values of K_p and K_n vary in the three scenarios. The best values are **Bold**, whereas the worst values are *Italic*.

K_p	K_n	Stationary scenario		Network variability scenario		Cluster failure scenario	
		N	υ (%)	N	υ (%)	N	υ (%)
0.1	0.10	0.98	99.99	1.18	**99.98**	1.55	98.77
	0.25	*1.41*	**100.00**	1.40	99.98	1.85	99.16
	0.50	0.87	99.91	*1.64*	*99.89*	*2.11*	97.71
0.5	0.10	0.48	**100.00**	0.64	99.97	1.05	99.22
	0.25	0.78	99.59	0.90	99.97	1.28	98.66
	0.50	1.12	**100.00**	1.19	99.92	1.63	98.39
1	0.10	**0.31**	**100.00**	**0.46**	99.93	**0.82**	**99.48**
	0.25	0.55	*99.22*	0.75	99.90	1.10	*97.65*
	0.50	0.86	99.87	1.00	99.91	1.36	98.14

after cluster recovery. The stability drop at $t = 90\,min$ is a direct consequence of cluster recovery, as several pods get stopped in other clusters and restarted in the recovered one.

In all scenarios, we see a faster increase of the CHCT parameter as K_p increases, and faster decrease as K_n increases. In some cases, the larger values of K_n lead to a brief instability as the CHCT parameter is aggressively decreased to very low values, leading to timeouts.

To determine values of K_p and K_n which work in all three scenarios, we compare all 27 cases for accuracy. Table 5.3 shows the accuracy of the controller in decreasing the number of timeout errors N, and in improving the stability υ. For each scenario, we show the best values for N and υ in **bold** and the worst values in *italic*. We see that the controller with K_p value of 1 and K_n value of 0.1 has the best values for N in all three scenarios, and the best value of υ in two out of three scenarios. Thus, we conclude that the controller works best in all three scenarios for this combination of values of the parameters K_p and K_n. This configuration improves stability from 83–92% with no controller (see Table 5.2) in stationary situations to 99.5–100% using the controller, even in challenging scenarios with network variability or cluster failures.

5.5 Conclusion

Geo-distributed computing platforms need to operate in difficult and uncertain networking conditions. In particular, it is notoriously difficult to distinguish actual node failures from delays caused by the networking or local node condition. We demonstrated that these effects can create significant instability in Kubernetes Federations. We identified the main configuration parameter which influences this behavior, and proposed a feedback controller which dynamically adapts its value to the operational conditions, and improves the system stability from 83–92% with no controller to 99.5–100% using the controller.

Chapter 6

Container orchestration in geo-distributed multi-cluster and fog environments

6.1 Introduction

Virtualized computing infrastructures are increasingly geo-distributed. For reasons such as high availability, low user-perceived latency, privacy, and compliance with national regulations, many infrastructures are being designed as a set of server clusters located in different regions [11].

Managing large-scale applications in these environments is a difficult challenge. Geographical resource distribution increases fragmentation where the resources in one location may be overloaded while those in another location may remain under-utilized. High resource utilization may result in performance degradation in geo-distributed deployments such as edge computing which are resource-constrained [289]. It is therefore important to provide users with simple yet powerful ways to control the scale and location of their replicated applications. When an application running in its preferred location runs out of resources, it may need to acquire additional resources in the local cluster or, if the local cluster runs out of resources, in another nearby cluster. In extreme cases where no suitable cluster resources may be found, the platform may need to burst to a public cloud where additional resources may be rented for the duration of the overload, then decommissioned when the workload decreases.

We base this work on the popular Kubernetes container orchestration platform. Kubernetes has fully demonstrated its ability to efficiently orchestrate the resources within a single cluster. However, it does not implement any notion of resource location and therefore does not allow its users to control the location of resources assigned to run their applications [129]. *Kubernetes Federation* (KubeFed) extends Kubernetes with an explicit notion

of multi-cluster environment. However, in its present form, KubeFed's main focus is on the manual placement of resources on selected clusters and implements only one generic automated scheduling mechanism which distributes replicated pods evenly between all available workload clusters. The lack of automation limits KubeFed's ability to manage a large number of clusters, whereas the absence of policy-based scheduling prevents users of multi-cluster deployments from specifying their desired scheduling preferences, and limits the efficient use of resources.

To address these challenges, we propose *mck8s*, a comprehensive orchestration platform for multi-cluster computing environments that offers multi-cluster scheduling with various placement policies, multi-cluster horizontal pod autoscaling, and dynamic cloud cluster provisioning capabilities to automate the deployment, resource provisioning, and scaling of multi-cluster applications. Our platform builds upon Kubernetes, KubeFed, and other leading cloud-native tools. Our work has the following objectives: (1) Maximize resource utilization in multi-cluster environments; (2) Guarantee that all applications submitted to a multi-cluster environment find a place to run by making use of all existing resources and provisioning additional resources from the cloud if necessary; (3) Maintain the performance of applications by adjusting the number of replicas in response to changing user traffic; (4) Guarantee that user requests are routed between multi-cluster application pods in multiple workload clusters; and (5) Guarantee that resources are not over-provisioned and wasted unnecessarily when they are not being used. Our experimental results in a geo-distributed multi-cluster environment complemented with resources from the cloud show that *mck8s* reduces resource fragmentation by balancing resource allocation for multi-cluster applications, and keeping the percentage of pending pods below 6% as compared to 65% in the case of vanilla KubeFed for the same workload.

6.2 Background

KubeFed allows application deployment and resource management on multiple clusters, called workload clusters, from a host cluster that hosts the KubeFed control plane. KubeFed builds upon Kubernetes using *Custom Resource Definition*s (CRDs) and introduces new abstractions such as *Federated Namespaces, Federated Deployments, Federated Services* and *Federated Jobs* that help to conceptualize multi-cluster applications. In fact, KubeFed allows federating any Kubernetes resources to be used in a multi-cluster environment.

In KubeFed, users can create and deploy federated resources using declarative manifest files in the usual Kubernetes fashion. A manifest file for a federated resource contains three parts, namely, *template*, *placement* and *override*. The *template* defines the aspects of the federated resource that are common across all selected clusters. The *placement* specifies the clusters selected for hosting the federated resource. The *override* defines aspects of the federated resource that are specific to certain clusters.

In its current form, KubeFed allows selecting the clusters manually to host the federated resources with limited support for automated policy-based scheduling. Because of this, KubeFed cannot scale to manage hundreds and thousands of clusters that are typical in certain geo-distributed computing use cases such as fog computing. Moreover, KubeFed simply deploys resources to the target workload clusters without any prior checks on the availability of resources. As a result, it fails to manage resources efficiently, causing resource wastage and fragmentation.

To illustrate these problems, we deployed a total of 1,126 federated deployments and jobs replaying the Google cluster traces [290] for one hour on a geo-distributed KubeFed environment containing five Kubernetes clusters having different capacities. In the setup, each cluster has one master node and five worker nodes: the nodes of Clusters 1 and 5 have 4 *Central Processing Unit* (CPU) cores and 16 GB *Random Access Memory* (RAM), whereas those of Clusters 2–4 have 2 CPU cores and 4 GB of RAM. We distribute deployments and jobs to different clusters according to a binomial distribution. This is reflected in the stacked plot in Figure 6.1 that shows a wide variation in the number of resources requested from the different clusters. Figure 6.2 shows the CPU allocation in percent, which is calculated as the ratio of the total CPU request of pods in each cluster to the total cluster CPU. We see that Clusters 2, 3 and 4 are over-allocated up to six, seven and two times their total CPU capacity, respectively, whereas Clusters 1 and 5 are under-allocated. This is because KubeFed simply deploys the pods on the preferred clusters without checking the availability of resources and does not try to balance the deployment when the preferred clusters run out of resources. As a result, we see that up to 65% of the deployed pods remain in a "pending" state. Moreover, overall across all the five clusters, we see that the CPU allocation reaches up to twice the total CPU capacity offered by the clusters, suggesting the need to provide additional resources.

To address these challenges that have to be solved in order to build a mature orchestration platform for managing resources in multi-cluster environments, we introduce *mck8s* – a platform that offers not only automated policy-based scheduling but also multi-cluster

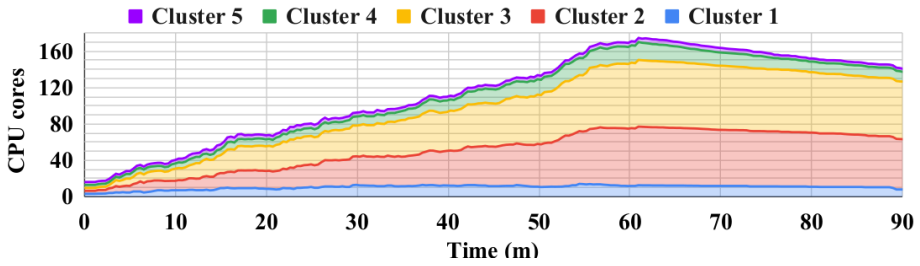


Figure 6.1 – KubeFed: Total CPU request of pods per cluster.

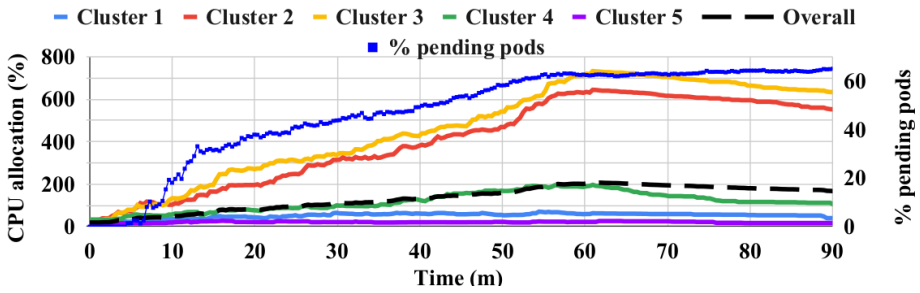


Figure 6.2 – KubeFed: CPU allocation ratio and percentage of pending pods.

horizontal pod autoscaling, network traffic routing and cloud bursting mechanisms. *mck8s* builds upon and borrows concepts from Kubernetes and KubeFed. It also integrates other open-source tools such as the Cilium cluster mesh for multi-cluster network discovery and global load balancing, Cluster API for declarative resource provisioning from cloud platforms, Prometheus for monitoring, and Serf for measuring inter-cluster network latencies.

6.3 Application deployment model

In this chapter, we propose *mck8s* – a comprehensive container orchestration platform for geo-distributed multi-cluster environments to address the challenges of resource fragmentation. Our platform builds upon Kubernetes and KubeFed and offers policy-based multi-cluster scheduling, multi-cluster horizontal pod autoscaling, cloud cluster provisioning and autoscaling and multi-cluster re-scheduling.

In this section, we introduce the abstractions and *Custom Resource*s (CRs) that make up *mck8s*. The abstractions and CRs introduced here provide the necessary control mechanisms for creating, updating and deleting multi-cluster applications. Moreover, different placement policies are proposed to allow flexibility for the orchestration platform to be used under different use cases. Lastly, we have focused on ease of use by making sure

that the manifest files used to create these resources as similar as possible to those in Kubernetes.

6.3.1 Multi-Cluster Deployment (MCD)

An *Multi-Cluster Deployment* (MCD) consists of a set of Kubernetes Deployments on one or more clusters that have the same Deployment name. For simplicity, we assume that all the pods of all the Deployments in an MCD have the same container image, CPU request and memory request. However, Deployments in different clusters may have a different number of replicas. MCDs hold status information about resource requests, number of replicas and locations.

Example manifest files of two MCDs are shown in Listings 6.1 and 6.2. Similar to KubeFed's *Federated Deployment*, *mck8s*' MCD allows the user to specify the preferred clusters on which the multi-cluster application will be deployed. In contrast, our MCD introduces placement, substitution and bursting policies to give the user more control, automation and flexibility in deciding how they want their applications deployed in the multi-cluster environment.

The multi-cluster placement policies are either resource-based (worst-fit and best-fit) or network traffic based (traffic-aware). If substitution is enabled in the case of cluster-affinity placement, substitute clusters are selected if the preferred clusters are incapable of placing the MCD. Similarly, if busting is enabled, an MCD deployed on a single cluster may be transformed into an MCD on multiple clusters if, for example, its replicas grow in response to user traffic. Unlike the scheduling controller in KubeFed, these policies are integrated into the MCD and are not part of yet another overarching controller, making it easier to use. Moreover, unlike KubeFed, our manifest files are designed to be very much similar to those of vanilla Kubernetes, thus allowing existing manifest files for single Kubernetes clusters to be easily used on *mck8s*.

6.3.2 Multi-Cluster Job (MCJ)

Similarly, a *Multi-Cluster Job* (MCJ) consists of a set of Kubernetes Jobs that run on one or more clusters. MCJ supports the placement and substitution policies as described in Section 6.3.1.

Listing 6.1 – An example MCD manifest file for traffic-aware placement.

```
apiVersion: fogguru.eu/v1
kind: MultiClusterDeployment
metadata:
  name: mcd-app-1
spec:
  placementPolicy: traffic-aware
  enableBursting: true
  burstingPolicy: nearest-first
  numberOfLocations: 2
  selector:
    matchLabels:
      app: mcd-app-1
      tier: backend
  replicas: 5
  template:
    metadata:
      labels:
        app: mcd-app-1
        tier: backend
    spec:
      containers:
        - name: mcd-app-1
          image: "k8s.gcr.io/hpa-example"
          resources:
            requests:
              memory: 512Mi
              cpu: 500m
            limits:
              memory: 512Mi
              cpu: 500m
          ports:
            - name: http
              containerPort: 80
```

Listing 6.2 – An example MCD manifest file for cluster-affinity placement.

```
apiVersion: fogguru.eu/v1
kind: MultiClusterDeployment
metadata:
  name: mcd-app-2
spec:
  locations: cluster2, cluster5
  enableSubstitution: true
  substitutionPolicy: nearest-first
  enableBursting: true
  burstingPolicy: nearest-first
... (redacted)
```

Listing 6.3 – An example MCS manifest file.

```
apiVersion: fogguru.eu/v1
kind: MultiClusterService
metadata:
  name: mcs-app-1
  annotations:
    io.cilium/global-service: "true"
spec:
  selector:
    app: mcd-app-1
    tier: backend
  ports:
  - protocol: TCP
    port: 80
    targetPort: http
```

6.3.3 Multi-Cluster Service (MCS)

A *Multi-Cluster Service* (MCS) resource manages the lifecycle Kubernetes Services corresponding to the Deployments under an MCD in the clusters that host them. Unlike KubeFed's *Federated Service*, it is not required to specify the location(s) for the MCS, as *mck8s* finds the corresponding MCD automatically. Moreover, we present a simpler manifest file similar to that of vanilla Kubernetes, as shown in Listing 6.3. Our MCS is also integrated with Cilium global load balancing system to enable routing of user requests to multiple clusters, which is required for bursting.

6.3.4 Multi-Cluster Horizontal Pod Autoscaler (MCHPA)

A *Multi-Cluster Horizontal Pod Autoscaler* (MCHPA) aims to adjust the number of deployment replicas of MCDs in response to changing traffic so that the quality of service provided by the MCDs is maintained. Therefore, it periodically monitors the number of deployment replicas of an MCD in all the clusters they are deployed on, computes the number of desired replicas based on the average resource utilization of the pods of the deployments, and adjusts the number of replicas of the MCD to the desired number of replicas. Unlike Kubernetes and KubeFed, MCHPA does not require Kubernetes' *Horizontal Pod Autoscaler* (HPA) to run inside each cluster, rather MCHPA runs inside the management cluster, requiring to define only one resource to manage the scalability of each MCD. As shown in Listing 6.4, an MCHPA resource can be defined very easily in the same way as an HPA.

Listing 6.4 – An example MCHPA manifest file.

```
apiVersion: fogguru.eu/v1
kind: MultiClusterHorizontalPodAutoscaler
metadata:
  name: mchpa1
spec:
  scaleTargetRef:
    apiVersion: fogguru.eu/v1
    kind: MultiClusterDeployment
    name: mcd-app-1
  minReplicas: 2
  maxReplicas: 100
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 75
```

Listing 6.5 – An example CPCA manifest file.

```
apiVersion: fogguru.eu/v1
kind: CloudProvisioner
metadata:
  name: cp1
spec:
  cloudClusterName: cloud1
  cloudProvider: OpenStack
  floatingIP: 1.2.3.4
  credentials: xxxxxxxxxxxx
```

6.3.5 Cluster Provisioner and Cluster Autoscaler (CPCA)

Whenever additional resources are required to augment the capacity of the fixed clusters, the *Cluster Provisioner and Cluster Autoscaler* (CPCA) resource interfaces with public cloud services to provision a Kubernetes cluster(s) on demand. CPCA is also responsible for adjusting the number of worker nodes of the cloud clusters and eventually decommission the cloud clusters altogether if not needed for a certain amount of time. The CPCA resource needs to be created only once by using a manifest file show in Listing 6.5.

Listing 6.6 – An example MCR manifest file.

```
apiVersion: fogguru.eu/v1
kind: MultiClusterRescheduler
metadata:
  name: mcr1
```

6.3.6 Multi-Cluster Re-scheduler (MCR)

To make sure that cloud clusters are not overprovisioned, the *Multi-Cluster Re-Scheduler* (MCR) resource periodically checks for deployments on the cloud clusters that were deployed because of a shortage of resource on their preferred clusters. The MCR attempts to place these deployments back on the preferred clusters once again. The manifest file needs to be applied only once and requires only the name of the resource as shown in Listing 6.6.

6.4 System design

6.4.1 System model

A multi-cluster environment is defined as a management cluster and a set of n workload clusters $\Gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_n\}$, where each cluster γ_i has one master node and q_i worker nodes. The architecture of the multi-cluster environment is shown in Figure 6.3. Although multiple layers can be supported, for the sake of simplicity, we present a two-layered architecture. At the *management layer*, we find the controllers and tools, whereas at the *workload clusters level* we find the workload clusters that are controlled by the management cluster and on which applications are executed.

We assume that each workload cluster is homogeneous in terms of resource capacity, i.e., each node m_{i_j} in workload cluster γ_i has $m_i.cpu$ CPU cores and $m_i.memory$ RAM. However, the nodes of different workload clusters may have different capacities. The workload clusters are geographically distributed, and the inter-cluster latency is defined by the matrix $\mathrm{L} = [l_{i_j}]$ as measured by *Serf*.

The management cluster is responsible for monitoring and configuring all workload clusters, accepting application deployment requests from users of the system, selecting the right clusters to host the applications, adjust the deployments in response to changes in user traffic, and provision, scale and deprovision cloud clusters. The workload clusters are

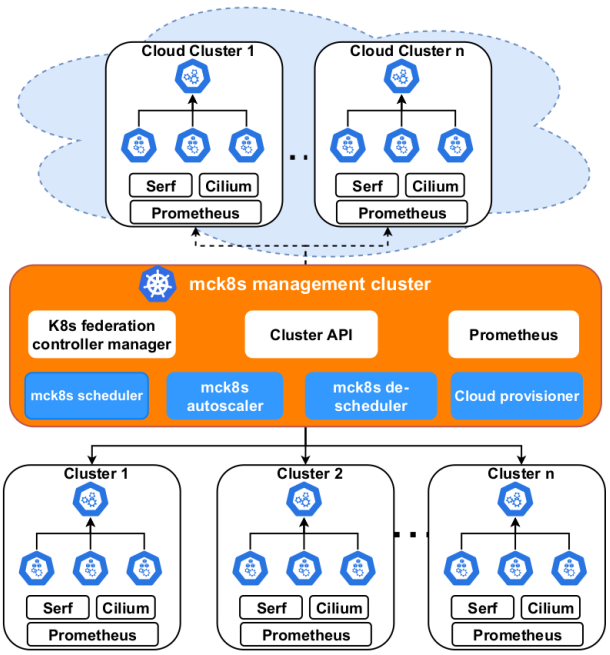


Figure 6.3 – mck8s architecture.

responsible for accepting application deployment requests from the management cluster and select the worker nodes that will host the pods of the application and executing them. Moreover, the workload clusters are responsible for local monitoring and estimating their distance in terms of network latency from other clusters.

6.4.2 Problem formulation

The multi-cluster scheduling problem is to map the Deployments of the MCD $\Delta = \{\delta_1, \delta_2, \ldots, \delta_l\}$ to l or more clusters from the set of workload clusters $\Gamma = \{\gamma_1, \gamma_2, \ldots, \gamma_n\}$ that have enough available capacity to match the Deployment's resource request. The choice of clusters depends on the placement policy specified by the user, the capacity of the cluster nodes and the resource availability.

The multi-cluster horizontal pod autoscaling problem aims to estimate and allocate at run time the number of Deployment replicas per cluster $\delta_i.replicas$ in response to the changing traffic received by the application. This depends on the resource request of the deployments $\delta_i.cpu\_req$ and $\delta_i.mem\_req$ and the target resource usage specified by the user θ_i.

Table 6.1 – Variables used in system modeling and algorithms.

Variable	Definition
Γ	$= \{\gamma_1, \gamma_2, \ldots, \gamma_n\}$, set of all clusters
γ_i	$\in \Gamma$, cluster of index i
n	$= \mid \Gamma \mid$, number of clusters
q_i	$= \mid \gamma_i \mid$,number of nodes in cluster γ_i
$\gamma_i.cpu.avail$	Total number of available CPU cores in cluster γ_i
$\gamma_i.memory.avail$	Total amount of available RAM in cluster γ_i
M_i	$= \{m_{i_1}, m_{i_2}, \ldots, m_{i_q}\}$, set of nodes in cluster γ_i
$m_i.cpu$	CPU cores of nodes of cluster γ_i
$m_i.memory$	Memory of nodes of cluster γ_i
B	$= \{\beta_1, \beta_2, \ldots, \beta_m\}$, set of all cloud locations
β_j	$\in$ B, cloud location of index j
Δ	$= \{\delta_1, \delta_2, \ldots, \delta_l\}$, l $\leq$ n, multi-cluster deployment as a set of deployments
$\Delta.clusters$	Clusters on which deployments of Δ are running on
$\delta_i.name$	Name of deployment δ_i
$\delta_i.cpu\_req$	CPU cores allocated to replicas of deployment δ_i
$\delta_i.mem\_req$	Memory allocated to replicas of deployment δ_i
$\delta_i.cpu\_util$	Avg. CPU utilization (%) of replicas of deployment δ_i
$\delta_i.mem\_util$	Avg. memory utilization (%) of replicas of deployment δ_i
$\delta_i.replicas$	Number of replicas of deployment δ_i
$\delta_i.ntk$	Total network traffic received by replicas of deployment δ_i
$P^i_{pending}$	Total number of pending pods in deployment δ_i
$P^i_{running}$	Total number of running pods in deployment δ_i
L	$= [l_{i_j}]$, symmetric n+m x n+m matrix of inter-cluster latencies
$\delta_i.current\_replicas$	Current number of replicas of deployment δ_i
$\delta_i.desired\_replicas$	Desired number of replicas of deployment δ_i
$\delta_i.min\_replicas$	Minimum number of replicas for deployment δ_i
$\delta_i.max\_replicas$	Maximum number of replicas for deployment δ_i
$\Delta.desired\_replicas$	Set of desired replicas for the deployments in Δ
θ_i	Target average CPU utilization (%) for replicas of deployment δ_i

The cloud provisioning/bursting problem addresses resource limitations by dynamically provisioning resources from the closest cloud *Data Center* (DC) β_j among a set B of cloud DCs such that the ratio of unscheduled (pending) pods is minimized. The closeness of the cloud DC is estimated using the network latency as measured by *Serf*.

The cloud cluster autoscaler problem is to adjust the number of worker nodes q_i in cloud cluster β_j so that all pods have a place to run while the cluster is not over-provisioned. As a result, the cluster autoscaler adds worker nodes when there are pending pods, and removes a worker node if the cluster is over-provisioned.

The multi-cluster re-scheduler periodically monitors applications currently deployed on cloud cluster β_j for reasons of shortage of resources on the preferred clusters, submits them to the scheduler so that they can be re-scheduled on their preferred clusters. This is also beneficial in terms of cost savings that would result in using the fixed fog results as much as possible and minimize over-provisioning the cloud clusters.

6.4.3 Design and implementation

mck8s addresses the geo-distributed resource management problems using four main controllers that are deployed on the management cluster: *multi-cluster scheduler*, *multi-cluster horizontal pod autoscaler*, *cloud cluster provisioner and autoscaler* and *multi-cluster re-scheduler*. It builds upon Kubernetes, parts of KubeFed, Cluster-API, Cilium, Serf and Prometheus. Each controller's design follows the MAPE loop [38] and Kubernetes design principles. As a result, the controllers are implemented as Kubernetes operators using the Kopf operator framework [1].

Multi-cluster scheduler

The multi-cluster scheduler is responsible for the full lifecycle (i.e., creation, updating and deletion) of the MCD, MCS and MCJ resources. When a user submits the specifications for MCD and MCJ resources to the management cluster, the scheduler checks the number of requested resources and placement constraints. Similar to the Kubernetes scheduler's approach to select the right worker nodes to place a pod, *mck8s*' scheduler goes through two major steps for selecting the right workload clusters to host the Deployments or Jobs. In the first step, the scheduler filters out those clusters whose nodes do not

1. https://kopf.readthedocs.io/en/stable/

have the capacity to place the pods of the Deployment. In the second step, the scheduler prioritizes the clusters based on the placement policy specified by the user.

In the case of a *cluster-affinity* policy shown in Algorithm 2, the user specifies a preferred cluster that will have the highest priority if it has already passed the filtering step and has sufficient available resources. Otherwise, a substitution cluster is selected based on the *substitutionPolicy* specified by the user if the *enableSubstituion* field is set to *true*. Similarly, if busting is enabled, an MCD deployed on a single cluster may be transformed into an MCD on multiple clusters if, for example, its replicas grow in response to user traffic. The substitution policy is *nearest-first*, which selects the nearest workload cluster in terms of network latency to the preferred one having sufficient resources.

If no preferred clusters are specified, the placement is done automatically by the scheduler based on the resource-based or traffic-aware placement policies specified. As detailed in Algorithm 3 the filtering step is the same as that of the cluster-affinity case, whereas in the prioritizing step the clusters are prioritized based on the *worst-fit*, *best-fit* or *traffic-aware* placement policies. In *worst-fit*, the clusters are sorted in descending order based on the available resources, and the clusters with the largest available resources are selected. On the contrary, with *best-fit* placement policy, the clusters are sorted in an ascending order based on their available resources and the clusters with the least resources are selected. The last policy implemented by *mck8s* is *traffic-aware* in which case the clusters are sorted in a descending order based on the amount of network traffic they receive and the clusters that receive the highest amount of network traffic are selected.

If no cluster is found to place the applications, the scheduler updates the status of the concerned MCD and MCJ resource for the cloud cluster provisioner to be notified and provision a Kubernetes cluster in the selected cloud data center.

The scheduler is also responsible for the lifecycle of the MCS resources that manage Service entries for the deployments of the target MCD when a specification for MCS is submitted to the management platform.

Multi-cluster horizontal pod autoscaler

For the applications deployed in the platform to maintain their performance requirements despite changes in traffic, *mck8s* provides a reactive threshold-based horizontal pod autoscaler. Unlike Kubernetes and KubeFed, the controller runs in the management cluster and monitors the resource utilization of the deployment pods. Currently, *mck8s* supports scaling based on CPU utilization.

Algorithm 2: Cluster-affinity placement

Input: Definition for Δ with $\delta.cpu\_req$, $\delta.mem\_req$, $\delta.replicas$ and $\Delta.preferred\_clusters$

for γ in $\Delta.preferred\_clusters$ **do**
  if $\delta.cpu\_req < m.cpu$ and $\delta.memory\_req < m.memory$ **then**
    append γ to $\Delta.eligible\_clusters$
  else
    Get replacement cluster
  if len($\Delta.eligible\_clusters$) == 0 **then**
    if $\exists$ cloud cluster β **then**
      append β to $\Delta.eligible\_clusters$
    else
      Provision cloud cluster
  else
    for γ in $\Delta.eligible\_clusters$ **do**
      if $\delta.replicas \times \delta.cpu\_req < \gamma.cpu.avail$ and $\delta.replicas \times \delta.memory\_req < \gamma.memory.avail$ **then**
        Append γ to $\Delta.selected\_clusters$
      else
        Get replacement cluster
  if len($\Delta.selected\_clusters$) == 0 **then**
    if $\exists$ cloud cluster β **then**
      append β to $\Delta.selected\_clusters$
    else
      Provision cloud cluster
Place Δ on $\Delta.selected\_clusters$

Algorithm 3: Policy based placement

Input: Definition for Δ with $\delta.cpu\_req$, $\delta.mem\_req$, $\delta.replicas$ and $\Delta.placement\_policy$

for γ in Γ **do**
  if $\delta.cpu\_req < m.cpu$ and $\delta.memory\_req < m.memory$ **then**
    append γ to $\Delta.eligible\_clusters$
  if len($\Delta.eligible\_clusters$) == 0 **then**
    if $\exists$ cloud cluster β **then**
      append β to $\Delta.eligible\_clusters$
    else
      Provision cloud cluster
  else
    for γ in $\Delta.eligible\_clusters$ **do**
      if $\delta.replicas \times \delta.cpu\_req < \gamma.cpu.avail$ and $\delta.replicas \times \delta.memory\_req < \gamma.memory.avail$ **then**
        Append γ to $\Delta.selected\_clusters$
  if len($\Delta.selected\_clusters$) == 0 **then**
    if $\exists$ cloud cluster β **then**
      append β to $\Delta.selected\_clusters$
    else
      Provision cloud cluster
switch $(placement - policy)$
case $traffic - aware$**:**
  Descending sort $\Delta.selected\_clusters$ by $\delta.ntk$
case $worst - fit$**:**
  Descending sort $\Delta.selected\_clusters$ by $\gamma.cpu.avail$ and $\gamma.memory.avail$
case $best - fit$**:**
  Ascending sort $\Delta.selected\_clusters$ by $\gamma.cpu.avail$ and $\gamma.memory.avail$
end switch
Place Δ on $\Delta.selected\_clusters$

The controller manages the MCHPA resource created for each MCD. The controller periodically computes the desired number of replicas based on the target utilization threshold, the current number of replicas, and the average CPU utilization of the pods of the target deployment. If the desired number of replicas is greater than that of the current replicas, the controller updates the number of replicas of the MCD and the multi-cluster scheduler adjusts the number of replicas. If, on the other hand, the number of desired replicas is less than that of the current replicas, the controller waits for a configurable cool-down period (10 minutes by default) to avoid fluctuations before it updates the MCD. The details of the controller are as shown in Algorithm 4.

Algorithm 4: Multi-cluster horizontal pod autoscaling (MCHPA)

Input: Definition for MCHPA with target MCD Δ, θ, $\delta.min\_replicas$ and $\delta.max\_replicas$
Output: $\Delta.desired\_replicas$

while not exited **do**
  For the given MCD Δ, get MCD $\Delta.clusters$, $\delta.cpu\_req$
  for γ in $\Delta.clusters$ **do**
    Get $\delta.current\_replicas$, $\delta.cpu\_usage$
    Compute $\delta.desired\_replicas = (\delta.current\_replicas \times \delta.cpu\_util)\ /\ \theta$
    if $\delta.desired\_replicas < \delta.min\_replicas$ **then**
      $\delta.desired\_replicas \leftarrow \delta.min\_replicas$
    else if $\delta.desired\_replicas > \delta.max\_replicas$ **then**
      $\delta.desired\_replicas \leftarrow \delta.max\_replicas$
    if $\delta.desired\_replicas < \delta.current\_replicas$ **then**
      Wait for cool down period
    Append $\delta.desired\_replicas$ to $\Delta.desired\_replicas$
  return $\Delta.desired\_replicas$

Cloud provisioner and cluster autoscaler

One of the objectives of *mck8s* is to dynamically provision additional workload clusters from private or public cloud DCs to complement a multi-cluster environment when its clusters run out of resources. This avoids under-provisioning while avoiding the high cost of running clusters in the cloud even when they are underutilized.

The cloud provisioner and autoscaler manages the entire lifecycle of a Kubernetes cluster in the cloud including provisioning, cluster autoscaling and deprovisioning. The controller manages an object that is defined and deployed once with information about

the cloud provider, region and authentication credentials. The controller runs periodically and monitors the status of all MCDs deployed in the platform. When the controller finds one or more MCDs that could not be deployed because of a shortage of resources, it computes the number and size of nodes needed to host the MCDs. Then, it provisions a Kubernetes cluster with the computed size and number of *Virtual Machine*s (VMs) using Cluster API [2] tool that allows creating Kubernetes clusters declaratively from cloud providers. The cluster autoscaler adjusts the number of nodes of the Kubernetes cluster in the cloud as the number of requested resources fluctuates. Finally, the controller removes the Kubernetes cluster from the cloud altogether if it is under-utilized for a pre-defined amount of time.

Algorithm 5: Cloud cluster provisioner and autoscaler

Input: Definition for cloud cluster β, cloud provider information (region, credentials, etc.)
Output: β

```
1:  while not exited do
2:    if ∃ β then
3:      if | P^i_pending | > 0 then
4:        Scale-out
5:        Calculate number and size of additional nodes
6:      else
7:        if number of nodes of cloud cluster == 1 then
8:          if number of deployment on cloud cluster == 0 then
9:            Remove cloud cluster
10:         else
11:           if ∃ nodes where sum of resource requests < node allocatable then
12:             Scale-in
13:             Remove nodes
14:   else
15:     for Δ_i in {Δ_1, Δ_2, ..., Δ_n} do
16:       if Δ_i.status.message == 'to_cloud' then
17:         total_cpu_req += δ_i.replicas × δ_i.cpu_req
18:         total_memory_req += δ_i.replicas × δ_i.memory_req
19:     Compute number and size of nodes for the cloud cluster
20:     Provision cloud cluster β
21:     return β
```

2. https://github.com/kubernetes-sigs/cluster-api

Multi-cluster re-scheduler

The Multi-Cluster Re-scheduler manages the custom resource of the same name. The controller is deployed on the management cluster once and periodically checks the cloud cluster for MCDs deployed on it because of a shortage of resources on their preferred clusters. When the controller finds such MCDs, it passes these MCDs to the Multi-Cluster Scheduler so that it attempts to schedule them once again, in which case they will be deployed on their preferred clusters if enough resources are available. In so doing, the re-scheduler contributes to minimizing overprovisioning in the cloud cluster.

6.5 Evaluation

We evaluate *mck8s* using four sets of experiments: (1) Scheduling of several short-running jobs and long-running services with a wide range of resource requests modeled after the Google cluster traces; (2) Autoscaling and placement policies as user traffic moves from one cluster to another; (3) Bursting of a multi-cluster deployment during autoscaling from a source cluster to other clusters and eventually to the cloud, along with service routing; and (4) Performance of *mck8s* in terms of deployment times for multi-cluster scheduling. We ran all experiments three times and the results from one of the runs is presented, except in the case of Experiment 2 where the results are the average of the three runs.

6.5.1 Experimental setup

We perform our experiments in the Grid'5000 experimental testbed (see Figure 6.4). The management cluster that runs the KubeFed controllers is deployed in Rennes, and five Kubernetes clusters are located in five different sites. Each cluster has one master node and five worker nodes. Clusters 1 and 5 use nodes with 4 CPU cores and 16 GB RAM, whereas clusters 2–4 use nodes with 2 CPU cores and 4 GB of RAM. Moreover, an OpenStack cluster in Nancy acts as the cloud platform where we provision a Kubernetes cluster during cloud bursting.

We use Kubernetes v1.18.0, Kubernetes Federation v0.1.0-rc6, Cluster API v0.3.10 with OpenStack provider v0.3.1, Cilium v1.9.3, Serf 0.8.2, and Prometheus Operator 0.45.0.

Table 6.2 – Inter-site network latency (RTT) in milliseconds in Grid'5000.

	Rennes	Nantes	Lille	Luxembourg	Nancy	Grenoble
Rennes	-	2.16	23.26	27.41	25.18	17.45
Nantes	2.16	-	22.21	26.29	24.16	16.38
Lille	23.26	22.21	-	11.88	9.70	12.06
Luxembourg	27.41	26.29	11.88	-	2.90	15.33
Nancy	25.18	24.16	9.70	2.90	-	13.14
Grenoble	17.45	16.38	12.06	15.33	13.14	-

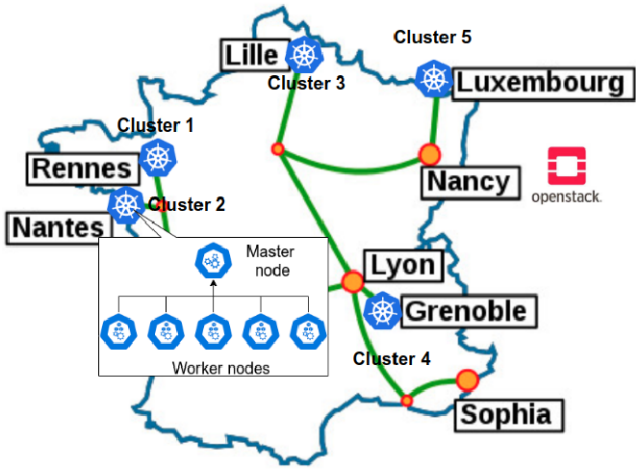
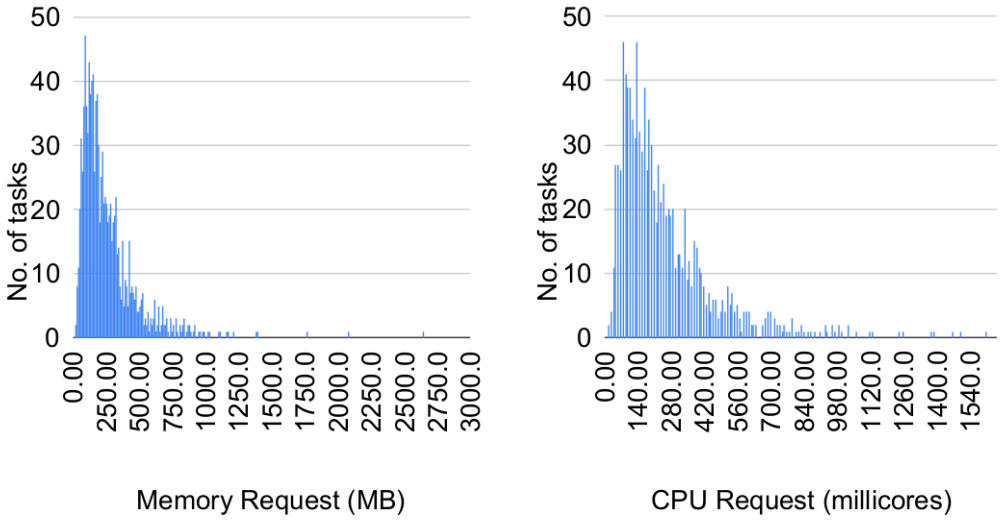


Figure 6.4 – Experimental setup in Grid'5000 consisting of a management cluster in Rennes, and five workload clusters in Rennes, Nantes, Lille, Grenoble (France), and Luxembourg, and an OpenStack cluster in Nancy. Distances between sites range from 100 km to 850 km. Each Kubernetes cluster has a master node and five worker nodes. Image adapted from the Grid'5000 website.

6.5.2 Multi-cluster scheduling

To evaluate the capability of *mck8s*' scheduler to place a variety of deployments and jobs, we deploy a workload based on the Google cluster traces. The Google cluster traces capture the characteristics of thousands of containers with diverse resource requirements, duration and inter-arrival rates that were executed in Google's Borg compute clusters [291]. In particular they exhibit heterogeneity in CPU and RAM request, inter-arrival rates and job duration, and has been extensively used to evaluate resource scheduling in the cloud [292].

We created a synthetic workload that matches the statistical distribution of the Google cluster trace, and augmented it with location information generated using a binomial distribution. Properties of the workload are shown in Figure 6.5 and Table 6.3. We ran the workload for 60 minutes during which 1,126 tasks were created. We wait 30 more minutes to observe tasks finish running and free up resources. Tasks longer than 60 minutes are



(a) Tasks memory request.

(b) Tasks CPU request.

Figure 6.5 – Characteristics of workload based on Google cluster traces.

Table 6.3 – Distribution of tasks across locations (clusters).

Location	Cluster 1 (Rennes)	Cluster 2 (Nantes)	Cluster 3 (Lille)	Cluster 4 (Grenoble)	Cluster 5 (Luxembourg)
No. of tasks	125	362	391	163	85

treated as long-running services (MCD) whereas shorter ones are treated as short-running jobs (MCJ). As new tasks are submitted to the management cluster, *mck8s*' scheduler goes through the filtering and prioritizing phases of scheduling and places a replica of the task on the preferred cluster as specified in the workload. In this experiment, substitution is enabled with substitutionPolicy 'nearest-first', meaning that if the preferred cluster is out of resources during placement, the scheduler places the task on the closest substitution cluster using network latency measured by Serf.

The stacked-area plot in Figure 6.6 shows the allocation of application pods on the workload clusters of the multi-cluster environment. As MCDs and MCJs are submitted to the management cluster, the multi-cluster scheduler places the deployments and jobs preferably on their preferred clusters as specified in the manifest files if the latter have sufficient resources available. However, if a cluster does not have sufficient resources to place a deployment and since substitution is enabled, some deployments are placed on clusters that are close to the original preferred cluster instead. When all clusters run out

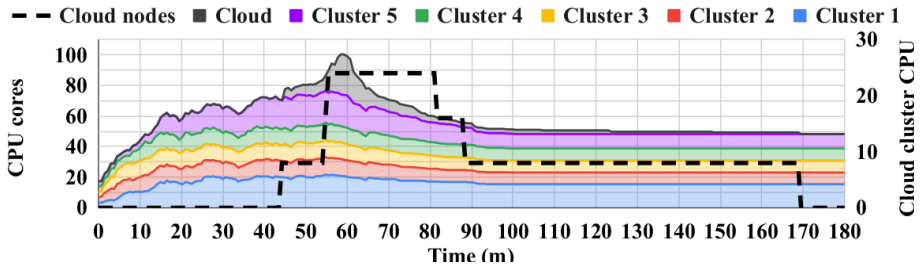


Figure 6.6 – Multi-cluster scheduling pods CPU request and cloud cluster lifecycle. Dashed line represents CPU cores of cloud nodes, whereas the stacked area represents the total CPU request of the pods running in the clusters.

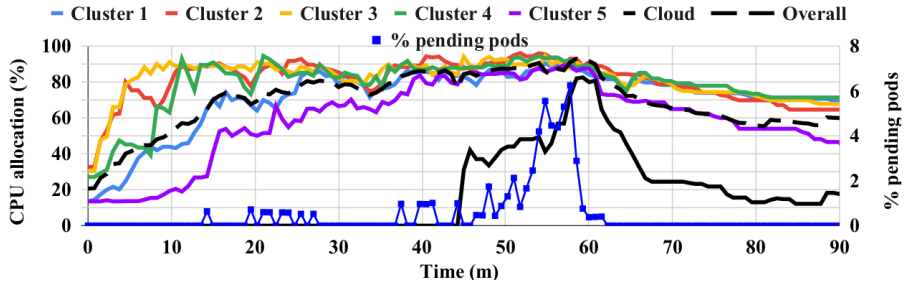


Figure 6.7 – Multi-cluster scheduling: Per-cluster and overall CPU allocation and percentage of pending pods.

of resources at $t = 43\,min$, the cloud provisioner provisions a Kubernetes cluster in the cloud where the multi-cluster scheduler places the deployments that could not be placed in the multi-cluster environment because of a shortage of resources. The broken lines show the total amount of CPU cores provisioned in the cloud, then scaled down and eventually decommissioned when no longer needed.

Figure 6.7 shows the per-cluster and overall CPU allocation, which is measured as the ratio of the total CPU request of pods to the total cluster CPU. We see that *mck8s*' substitution and policy results in a balanced CPU allocation per-cluster and overall, as opposed to that of KubeFed discussed in Section 6.2. Again, in Figure 6.7 we see that *mck8s*' scheduling policy results in only a maximum of 6% of the submitted pods are pending as opposed to 65% in the case of KubeFed (Figure 6.1).

6.5.3 Autoscaling and policy-based placement

In this experiment, we create a scenario that shows how an application deployed on one of the clusters responds to user traffic as the source of traffic moves. For this experiment, we deployed the Cilium cluster mesh on the workload clusters to enable cross-cluster service routing and load balancing. The application used in this evaluation is a two-tier MCD consisting of an nginx front-end and a simple PHP web application as a backend.

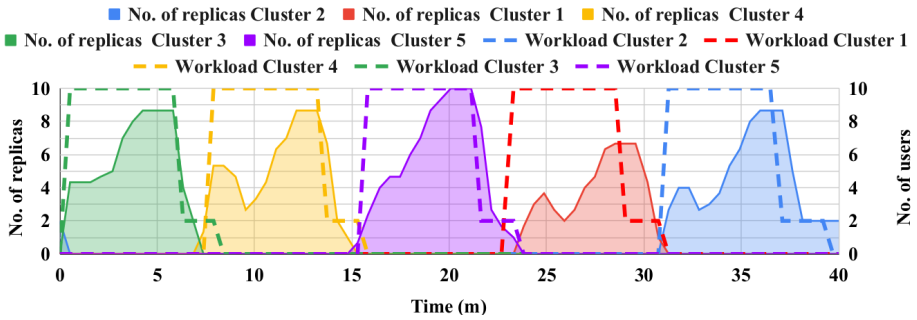


Figure 6.8 – Multi-cluster horizontal pod autoscaling and traffic aware placement. Dashed lines represent no. of users, whereas solid areas represent no. of replicas.

The frontend is deployed on all five clusters, whereas the backend is initially deployed on Cluster 2 with two replicas. We allocated 0.5 CPU cores [3] and 512 MB of memory to the pods of both the backend and frontend applications. To enable autoscaling, the corresponding MCHPA object is also applied for the backend, with CPU as the metric and 75% CPU utilization threshold. Then, constant user traffic with 10 concurrent users is applied to the application for 7 minutes from one source of traffic starting at Cluster 3 and then consecutively moving to Clusters 4, 5, 1 and 2.

In Figure 6.8, we show the results from the scheduling using the *traffic-aware* placement policy. We see that initially the backend was deployed on Cluster 2 and even though the source of traffic has moved to Cluster 3 it still receives the traffic thanks to the load balancing by Cilium. When the scheduler tries to select the appropriate cluster for the application following the update from the multi-cluster horizontal pod autoscaler during the next cycle, it selects Cluster 3 as the frontend on this cluster is receiving the most traffic. As a result, the backend is moved to Cluster 3. Moreover, the number of replicas is increased to accommodate the traffic from 10 concurrent users. In the same manner, the backend follows the user traffic to clusters 4, 5, 1 and 2. By doing so, *mck8s* makes sure that the application is deployed closer to end-users and the number of replicas is adjusted to make sure that the application meets its performance requirements.

6.5.4 Multi-cluster horizontal pod autoscaling and bursting

We now evaluate the autoscaling, bursting and cloud provisioning features of *mck8s* working together that allow an MCD faced with a sudden increase in user traffic to make

3. In k8s, one CPU is equivalent to 1 *Virtual Central Processing Unit* (vCPU)/Core for cloud providers and 1 hyperthread on bare-metal Intel processors. A Container with request of 0.5 CPU cores is guaranteed half as much CPU as one that asks for 1 CPU core. Read more here. `https://bit.ly/3sueJ7E`

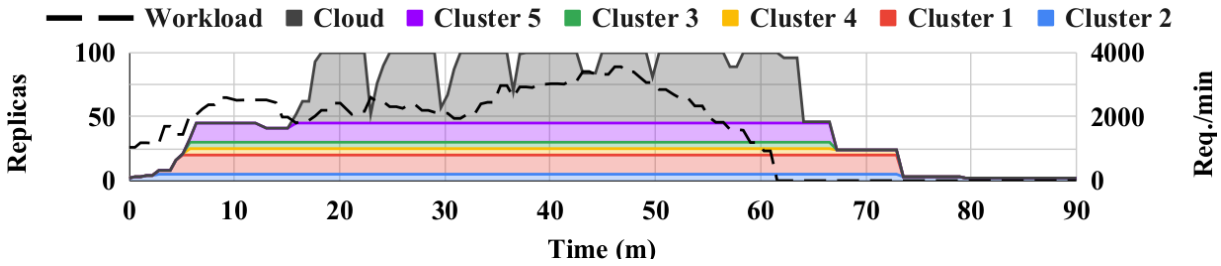


Figure 6.9 – No. replicas of the MCD during multi-cluster horizontal pod autoscaling and cloud bursting. Dashed lines represent requests per minute generated by the workload.

use of neighboring clusters as well as resources in the cloud. The results are shown in Figures 6.9–6.10.

Similar to the previous experiment, we deployed Cilium cluster mesh on the clusters to enable inter-cluster service routing and used the same two-tier application for evaluation. We allocated 0.5 CPU and 512 MB of memory to the pods of both the frontend application, whereas 1 core of CPU and 1024 MB memory were allocated to the backend. Initially, we deploy five replicas of the front end and two replicas of the backend on Cluster 2. An MCHPA instance is created for the backend, with 2 minimum number of replicas, 100 maximum number of replicas, CPU as the scaling metric, and 50% CPU utilization threshold. MCHPA is configured with a cool-down period of 10 minutes, whereas the cluster autoscaler is configured with a scale-down delay and deprovisioning delay of 10 minutes and 20 minutes, respectively.

The workload used for this evaluation and shown using broken lines in Figure 6.9 is based on the San Francisco taxi traces [293] and it is applied to the frontend service on Cluster 2. The autoscaler adjusts the number of replicas in response to the changing workload and the scheduler configured with the traffic-aware placement policy, bursting enabled and the nearest-first bursting policy places the replicas in the right clusters. As can be seen in Figures 6.9 the application bursts from Cluster 2 successively to clusters 1, 4, 3, 5. When Cluster 5 runs out of resources the cloud provisioner provisions a Kubernetes cluster on the OpenStack cloud and joins it to the multi-cluster federation so that more replicas are deployed on it. As the user traffic decreases after 60 minutes, the application is "pulled back" to the original Cluster 2. Figure 6.10 shows the full lifecycle of the cloud cluster as it is created, autoscaled, and finally deprovisioned. After the workload starts decreasing at around 47 minutes, we notice a slow scale down of replicas as well as a scale-down and deprovisioning of the cloud cluster due to the cool-down periods and the relatively low CPU utilization threshold.

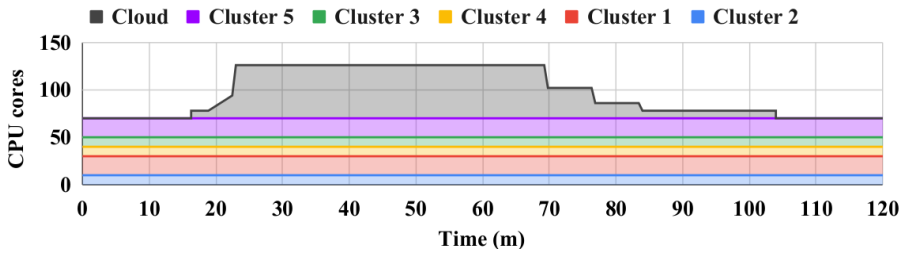


Figure 6.10 – Total CPU cores provided by clusters including cloud cluster's full life cycle during multi-cluster horizontal pod autoscaling and bursting.

6.5.5 Deployment times

To demonstrate the performance of scheduling in *mck8s* we perform a few measurements of deployment time by varying the number of replicas of an MCD from 1 to 100 and the placement policies (cluster-affinity and traffic-aware). In this experiment, we allocate 0.5 CPU cores and 512 MB of memory to the pods of the MCD. The results are shown in Table 6.4. We see that, in general, deployment time increases as the number of replicas to be deployed increases because the number of clusters that the scheduler has to filter and prioritize increases as well, especially if the resource request of the deployment is relatively high or the clusters have relatively smaller capacity or available resources. In the case of the traffic-aware placement policy, the deployment times are higher overall than those of the cluster-affinity policy, because the scheduler has to sort all clusters based on the received traffic before selecting the appropriate clusters whereas in the cluster-affinity placement policy the scheduler already knows the preferred cluster to be selected if it has enough resources available, otherwise, the scheduler looks for a substitute which is common as the number of replicas grows. We see that the deployment time varies between 1.88s for 1 replica to 11.52 s for 100 replicas for the cluster-affinity placement policy. On the other hand, it takes 8.62 s for deploying 1 replica and 9.71 s to place 100 replicas with the traffic-aware placement policy. This illustrates that the *mck8s* scheduler scales well as the number of replicas increases. And since the scheduler is the core component of *mck8s*, this shows that *mck8s* is well suited to serve as an orchestration platform in a geo-distributed multi-cluster environment that is expected to handle thousands of applications within a short amount of time.

Table 6.4 – Multi-cluster scheduling deployment times as no. of replicas change.

No. of replicas	Deployment time (seconds)	
	Traffic aware with bursting	Cluster Affinity with bursting
1	8.63	1.88
10	8.67	1.78
100	9.71	11.53

6.6 Conclusions

In this work, we address the gap in integrated resource management of geo-distributed clusters. We propose *mck8s*, a generic and integrated orchestration platform for multi-cloud deployments with policy-based scheduling, autoscaling and cloud bursting capabilities. Although *mck8s* introduces new controllers and interfaces, we argue that it can be easily adopted because of its simple integration with vanilla Kubernetes. Using realistic experiments, we show that *mck8s* balances the resource allocation across multiple geo-distributed clusters and reduces the fraction of pending pods from 65% in the case of Kubernetes Federation to 6% for the same workload. The *mck8s* implementation is freely available under a liberal open-source license [4].

In addition to the simple heuristics presented in this chapter, *mck8s* may be further extended to include more sophisticated and proactive placement and autoscaling algorithms to address different multi-cluster use cases.

4. mck8s – https://github.com/moule3053/mck8s

Chapter 7

Conclusion and future directions

7.1 Conclusion

In the last two decades, cloud computing has emerged as the dominant computing paradigm for enterprise computing. Cloud computing presents a seemingly infinite pool of virtualized compute, network and storage resources in very large *Data Center*s (DCs) that are located in different regions of the world. However, the traditional application deployment model on a single cloud DC fails to fulfil non-functional requirements such as proximity, high availability, fault tolerance, compliance with privacy and regional regulations and vendor neutrality that numerous applications must fulfill. These requirements can only be fulfilled if applications are deployed on computing resources that are sufficiently geographically distributed at strategic locations in various regions of the world. Some of the applications require even more distribution at the level of cities, neighborhoods and facilities. As a result, geo-distributed deployment models such as hybrid cloud, multi-cloud and fog computing have emerged.

Geo-distributed deployments take advantage of the increasing geographical distribution of resources in public and private cloud DCs as well as the edge of the network. Hybrid cloud deployments make use of resources from private and public clouds. Multi-cloud deployments exploit resources from multiple regions of the same or multiple public cloud providers. Fog computing deployments are often larger in scale than the previous two deployments, and span resources in private clouds, public clouds and resources at the edge of the network.

Despite their benefits, geo-distributed computing paradigms such as hybrid cloud, multi-cloud and fog computing pose difficult resource management challenges. These challenges emerge from: (a) large number of resources; (b) large geographical distribution of resources; (c) heterogeneous characteristics of networking outside of cloud DCs, often with low bandwidth, high latency and high probability of packet loss; (d) heterogeneity in the type and capacity of resources; and (e) resource constraints, especially in fog computing.

Earlier works address the resource management challenges in geo-distributed computing environments using hardware virtualization technology. However, these efforts are limited by several challenges such as portability and interoperability issues.

Recent works leverage the characteristics of container technology such as portability, lightweightness, minimal performance overhead and fast bootup times. Several container orchestrations platforms have been proposed for automating application deployment and resource management. State-of-the-art container orchestration platforms are often designed to manage a single cluster, and do not take into account the location of resources, resource heterogeneity and the poor networking condition between nodes that is common in geo-distributed environments. As a result, the placement, autoscaling, network routing and resource provisioning policies offered by state-of-the-art container orchestrators are not optimized for geo-distributed deployments.

In this thesis, we propose three contributions that address various challenges concerning application deployment and resource management in geo-distributed computing environments using containers. The ultimate goal of this thesis is to propose a comprehensive, scalable and reliable container orchestration platform that supports various application deployment use cases in geo-distributed environments and encompasses different types of clusters in private clouds, public clouds and the edge of the network. We also aim at allowing application developers to use the platform efficiently by presenting easy-to-use application model and interfaces.

7.1.1 Contribution 1: Performance evaluation of Kubernetes cluster autoscaler

In the last few years, containers have emerged as the preferred virtualization technique for packaging, deploying and scaling applications in the cloud. However, *Virtual Machine*s (VMs) are still largely used for resource provisioning in *Infrastructure-as-a-Service* (IaaS) clouds and to provide strong isolation between tenants. Similarly, Kubernetes has emerged as the most popular orchestration platform for automating application deployment using containers.

In order to maintain acceptable performance despite workload changes, autoscaling is used in the cloud to dynamically adjust the amount of resources allocated to applications. To this end, Kubernetes offers horizontal and vertical autoscaling of containers. Similarly, it offers the *Cluster Autoscaler* (CA) that scales worker nodes horizontally. CA

is highly configurable and offers several parameters such as choice of differently-sized multiple worker node pools, various criteria to choose from multiple worker node pools and a configurable cool-down period. Different combinations of these parameters affect the performance of different applications disparately. Moreover, they impact the cost of running the infrastructure in the cloud. Therefore, it is important to quantitatively evaluate the impact of these parameters on different applications. This understanding does not only help operators to select the right parameters for their applications but can also be used as input for designing better autoscaling mechanisms.

We perform an in-depth analysis of the performance of the Kubernetes CA using monetary cost and standard autoscaling performance metrics. We evaluate CA under two configurations that determine the size of the worker nodes to be added during scale-out. We evaluate the impact of different applications and workloads on cost and autoscaling performance and show quantitatively that these metrics are affected by the load characteristics of the applications. Moreover, we demonstrate the potential for performance gains from tuning additional autoscaling configuration parameters.

7.1.2 Contribution 2: Improving stability in a geo-distributed multi-cluster environments

Geo-distributed computing environments are characterized by heterogeneous inter-cluster connectivity. More often than not, these networks have properties such as high latency, low bandwidth and high probability of packet loss. These properties may result in delayed and inaccurate failure detection that affects the stability, availability and performance of systems.

We demonstrate experimentally that poor network conditions between the control plane and workload clusters lead to instability in a geo-distributed computing environments based on *Kubernetes Federation* (KubeFed). We also show that, among other things, the degree of instability is impacted by configuration parameters, variations in network conditions and different failure scenarios.

To improve the stability of the system, we first identify the configuration parameter that has the largest impact. Then, we design, implement and evaluate a proportional controller that improves the stability of the system by adjusting the concerned configuration parameter at run time despite variations in network conditions and across a range

of failure scenarios. Our controller improves the stability of the system from 83 – 92% in the uncontrolled case to 99.5 – 100% using the controller.

7.1.3 Contribution 3: Container orchestration for geo-distributed multi-cluster environments

Geo-distributed compute environments are characterized by large scale and heterogeneity of resources. We demonstrate experimentally that this could lead to resource fragmentation where some workload clusters are over-provisioned while others are idle. High resource utilization may be detrimental to the performance of resource-constrained geo-distributed computing environments such as fog computing.

State-of-the-art container orchestration frameworks are limited to single clusters, and as a result lack the necessary placement, autoscaling and dynamic provisioning mechanisms to address the resource fragmentation problem in geo-distributed computing environments. In these environments, it is important to use topologically nearby clusters when one cluster runs out of resources. Moreover, it may be necessary to dynamically provision resources from a nearby public cloud DC temporarily if applications need to handle sudden workload spikes.

To address these challenges, our third contribution is a container orchestration platform for geo-distributed multi-cluster deployments. Our contribution builds on KubeFed, and extends it to support network- and resource-aware placement policies, multi-cluster autoscaling, inter-cluster network routing, and transparent provisioning, de-provisioning and autoscaling of cloud clusters. Our proposed system aims to be a comprehensive container orchestration platform that supports applications from multiple use cases in the fog-to-cloud continuum. Our platform improves the overall resource utilization of the geo-distributed system by reducing the percentage of pending pods to 6% as opposed to 65% in the case of KubeFed for the same workload.

7.2 Future directions

In this thesis, we presented three contributions that address various aspects of resource management in geo-distributed computing environments. Although we believe our contributions have filled some gaps in the state of the art, additional work is required to fulfill the vision of a comprehensive, scalable and resilient orchestration platform for

geo-distributed computing environments. Therefore, in this section, we highlight some future research directions that address the limitations of our work. We also explore broader opportunities that provide wider perspectives.

7.2.1 Mechanisms for improving autoscaling in container orchestration platforms

In Chapter 4, we identified few configuration tuning opportunities for Kubernetes CA, which has several configuration parameters. In addition to the parameter that decides whether to use differently-sized worker nodes during scale-out, there are other parameters such as the criteria to decide the exact size of worker nodes (VMs) during scale-out, cool-down period during scale-in, and autoscaling interval. These parameters offer different performance-cost trade-offs for different types of applications. Moreover, different types of traffic patterns may have different impacts. As CA configurations apply to the entire cluster, they affect all applications deployed on the cluster. It would be difficult for operators of the cluster to identify the optimal configuration parameters in this complex configuration space. On the other hand, leaving the configurations at their default values may result in sub-optimal performance and unnecessary expenses. Therefore, one possible research direction is investigating automatic configuration tuning approaches to navigate the complex configuration space and maintain the desired performance-cost trade-off for all applications under different workloads. Furthermore, this begs the broader question of which applications to run in the same cluster, and which to separate in different clusters.

Another research direction concerns the multiple levels and directions of autoscaling in container orchestration platforms. For instance, Kubernetes supports autoscaling at the container and VM levels. It also supports horizontal and vertical autoscaling of containers and horizontal autoscaling of VMs. Although Kubernetes at the moment does not support vertical autoscaling of worker nodes (VMs), there is a possibility to choose differently-sized VMs during horizontal scaling. A combination of the levels and directions of autoscaling creates a four-fold autoscaling problem [158]. The horizontal and vertical scaling of containers is application-specific whereas the horizontal and vertical scaling of VMs affects all applications on the cluster [294]. An optimal choice may improve resource utilization and reduce costs while ensuring acceptable performance. As finding the optimal setting could be difficult, error-prone and expensive if done manually, it is important to devise automatic approaches to address it.

The third possible research direction deals with the timing of horizontal autoscaling decisions and the amount of resources allocated during vertical autoscaling. Currently, horizontal pod autoscaling relies on a reactive threshold-based heuristic. Horizontal autoscaling of VMs also uses a reactive approach based on the presence of unscheduled containers. On the other hand, vertical autoscaling of containers relies on recommendations based on historical resource use. Reactive horizontal and vertical autoscaling decisions may result in delays that affect the performance of the application, especially during workload spikes. Therefore, it would be beneficial to investigate predictive approaches to proactively estimate the timing of horizontal autoscaling and the amount of resource allocation during vertical autoscaling. As predictive approaches may suffer from inaccuracy and over-provisioning, it is important to optimize them depending on the applications and workloads. As autoscaling and scheduling decisions cannot be taken in isolation, there is also a need to study these two problems together. Hybrid approaches that incorporate reactive and proactive approaches have been proposed for VM autoscaling [59]. It would be interesting to investigate hybrid autoscaling approaches for joint autoscaling of containers and VM too.

7.2.2 Mechanisms for improving the resilience of geo-distributed computing environments

In the coming years, it is expected that computing is going to be even more geo-distributed to the point where it becomes pervasive and ubiquitous like the other utilities we depend on [295]. In the future, we will see new classes of applications that span resources in cloud DCs as well as fog nodes distributed throughout the network. Applications are expected to be highly available and resilient to failure.

In Chapter 5, we focused on transient failure scenarios that could arise because of the poor network conditions in geo-distributed computing environments. We also proposed a control-theoretic approach for configuration tuning to address the instability that arises. However, in large-scale geo-distributed deployments, other kinds of failures could arise because of hardware faults, DC failures, and absence of redundancy in applications and data.

To address these problems, a possible research direction would be to investigate failure scenarios at a large scale and understand their impact on applications' availability and performance [296]. Then, it is important to devise intelligent mechanisms to proactively

detect failures, and recover from them fast. However, this may require collecting a vast amount of data about the health of applications and the infrastructure. To this end, efficient ways of doing health checks need to be devised.

Another approach to ensure the resilience of geo-distributed deployments is by making applications and data more redundant. However, carefully choosing the right degree of replication is important to avoid unnecessary resource usage and expenses at a large scale.

In large-scale geo-distributed computing environments, compute and data nodes may be found in many locations outside of DCs. This means a vast amount of the infrastructure does not have the multiple layers of physical security in cloud DCs. As a result, geo-distributed computing environments may be more susceptible to attacks and data breaches because of the larger attack surface and their physical locations. Therefore, it is important to investigate approaches for detecting security breaches and attacks. Moreover, it is important to devise ways for strong application and data security that work efficiently at a large scale.

7.2.3 Extending *mck8s* to support additional placement and re-scheduling algorithms

In Chapter 6, we proposed *mck8s* – a container orchestration platform for geo-distributed computing environments. To make *mck8s* more comprehensive so that it can support various use cases and applications, we incorporated cluster-affinity, resource-based and network-aware placement heuristics. *mck8s* can be further extended in various ways to accommodate additional types of applications.

First, a data-locality-aware placement algorithm may be incorporated to address big data analytics applications that run on geo-distributed environments. Such an algorithm could automatically place data processing tasks on the appropriate clusters containing the data to be processed. However, such an algorithm should be supported by a mechanism to schedule and migrate data across the geo-distributed clusters.

Second, a priority-based algorithm could be incorporated to complement our network-aware placement heuristic. A priority-based placement algorithm allows latency-sensitive applications to be deployed at the cluster closest to the application's end-users by displacing less critical applications in the event of insufficient resources. The less critical applications may be placed at a neighboring cluster or in the cloud using our replacement algorithms.

Third, our re-scheduling algorithm can be extended to support re-scheduling applications from neighboring clusters to the original clusters. For instance, this would allow less critical applications displaced due to priority-based placement to be re-scheduled on the original clusters. This requires additional monitoring and keeping track of applications' placement history. Therefore, it is important to investigate efficient ways for monitoring at a large scale. Such monitoring systems may be based on open-source, highly available and scalable monitoring systems such as Thanos [1] and Cortex [2].

7.2.4 Towards decentralized resource management in geo-distributed computing environments

In our third contribution, we showed that application deployment and resource management in geo-distributed computing systems can be done from a centralized control plane that is located at a higher hierarchy in the architecture or the cloud. This approach provides a local and global resource view. However, currently, our model supports deploying geo-distributed applications from the centralized control plane only. Moreover, our proposed system was not evaluated at scale.

In realistic geo-distributed environments such as fog computing, a centralized control plane model may face challenges because of several reasons such as a large number of workload clusters and workload clusters joining and leaving the federation frequently. Following current software development and operation practices, it is expected that applications are deployed and updated repeatedly. Monitoring a large number of clusters may require collecting and transferring a large amount of data. Moreover, keep-alive messages might be lost because of a high probability of packet loss.

In order to address these challenges, it is important to investigate different approaches that ensure a highly available and performant control plane. One approach could be to make the control plane redundant. Redundant control plane nodes are used in Kubernetes to ensure a highly available control plane. Similarly, the control plane nodes of the management and workload clusters could be designed in a distributed and highly-available manner so that these nodes are located in multiple clouds, DCs or regions. In this way, the resiliency of the geo-distributed computing environments could be ensured should one node or DC crashes. Although this approach allows distributing the load of the management cluster, it may require more resources.

1. https://thanos.io/
2. https://github.com/cortexproject/cortex

Another approach could be decentralizing the control plane and distribute it throughout the network [51]. Our model currently supports a local view of each workload clusters and allows deploying applications on each workload cluster. However, those deployments are limited to a single workload cluster as the workload clusters do not have a global view of the entire environment. However, similar to network routing algorithms, one could envision deploying applications on the geo-distributed computing environment from any of the workload clusters. In this setup, each workload cluster could have a limited view of few neighboring clusters and has information about available resources. Therefore, using this approach it would be possible to deploy on multiple clusters from any of the workload clusters. However, care must be taken to avoid consistency issues and increase in the number of control messages.

BIBLIOGRAPHY

[1] M.-G. Avram, « Advantages and challenges of adopting cloud computing from an enterprise perspective », *Procedia Technology*, vol. 12, 2014.

[2] S. Li, Y. Zhang, and W. Sun, « Optimal Resource Allocation Model and Algorithm for Elastic Enterprise Applications Migration to the Cloud », *Mathematics*, vol. 7, *10*, 2019.

[3] S. Li and W. Sun, « Utility maximisation for resource allocation of migrating enterprise applications into the cloud », *Enterprise Information Systems*, vol. 15, *2*, 2021.

[4] Google Cloud. (2021). Machine types, [Online]. Available: https://cloud.google.com/compute/docs/machine-types. (accessed: 21.03.2021).

[5] I. Sfiligoi, F. Wuerthwein, B. Riedel, and D. Schultz, « Running a pre-exascale, geographically distributed, multi-cloud scientific simulation », *in Proceedings of the International Conference on High Performance Computing*, 2020.

[6] C. S. M. Babou, D. Fall, S. Kashihara, I. Niang, and Y. Kadobayashi, « Home edge computing (HEC): design of a new edge computing technology for achieving ultra-low latency », *in International conference on edge computing*, 2018.

[7] N. Mohan, L. Corneo, A. Zavodovski, S. Bayhan, W. Wong, and J. Kangasharju, « Pruning Edge Research with Latency Shears », *in Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, 2020.

[8] S. Yi, Z. Hao, Z. Qin, and Q. Li, « Fog computing: Platform and applications », *in Proceedings of the 3rd IEEE Workshop on Hot Topics in Web Systems and Technologies*, 2015.

[9] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, « Fog computing and its role in the internet of things », *in Proceedings of the 1st edition of the MCC Workshop on Mobile Cloud Computing*, 2012.

[10] F. Bonomi, R. Milito, P. Natarajan, and J. Zhu, « Fog computing: A platform for internet of things and analytics », *in Big Data and Internet of Things: A Roadmap for Smart Environments*, vol. 546, 2014.

[11] A. Ahmed, H. Arkian, D. Battulga, A. J. Fahs, M. Farhadi, D. Giouroukis, A. Gougeon, F. O. Gutierrez, G. Pierre, P. R. S. Jr., M. A. Tamiru, and L. Wu, « Fog Computing Applications: Taxonomy and Requirements », *CoRR*, vol. abs/1907.11621, 2019.

[12] O. Debauche, S. Mahmoudi, S. A. Mahmoudi, P. Manneback, and F. Lebeau, « A new edge architecture for ai-iot services deployment », *Procedia Computer Science*, vol. 175, 2020.

[13] S. A. Noghabi, L. Cox, S. Agarwal, and G. Ananthanarayanan, « The emerging landscape of edge computing », *GetMobile: Mobile Computing and Communications*, vol. 23, *4*, 2020.

[14] D. Yimam and E. B. Fernandez, « A survey of compliance issues in cloud computing », *Journal of Internet Services and Applications*, vol. 7, *1*, 2016.

[15] N. Grozev and R. Buyya, « Multi-cloud provisioning and load distribution for three-tier applications », *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 9, *3*, 2014.

[16] S. B., I. Cunha, Y.-C. Chiu, S. Sundaresan, and K.-B. E., « Internet performance from facebook's edge », *in Proceedings of the Internet Measurement Conference*, 2019.

[17] W.-J. Wang, Y.-S. Chang, W.-T. Lo, and Y.-K. Lee, « Adaptive scheduling for parallel tasks with QoS satisfaction for hybrid cloud environments », *The Journal of Supercomputing*, vol. 66, *2*, 2013.

[18] D. Petcu, « Multi-cloud: expectations and current approaches », *in Proceedings of the International Workshop on Multi-Cloud Applications and Federated Clouds*, 2013.

[19] Y. Aldwyan and R. O. Sinnott, « Latency-aware failover strategies for containerized web applications in distributed clouds », *Future Generation Computer Systems*, vol. 101, 2019.

[20] A. J. Fahs and G. Pierre, « Proximity-aware traffic routing in distributed fog computing platforms », *in Proceedings of the 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2019.

[21] Facebook Engineering. (2020). Fault tolerance through optimal workload placement, [Online]. Available: https://bit.ly/2RQuAAF. (accessed: 22.04.2021).

[22] ——, (2020). Throughput autoscaling: Dynamic sizing for Facebook.com, [Online]. Available: https://bit.ly/3gxAlxA. (accessed: 22.04.2021).

[23] ——, (2018). Location-Aware Distribution: Configuring servers at scale, [Online]. Available: https://bit.ly/3xgTlq4. (accessed: 22.04.2021).

[24] A. Gordon, « The hybrid cloud security professional », *IEEE Cloud Computing*, vol. 3, *1*, 2016.

[25] Z. Wu and H. V. Madhyastha, « Understanding the latency benefits of multi-cloud webservice deployments », *ACM SIGCOMM Computer Communication Review*, vol. 43, *2*, 2013.

[26] L. Jiao, J. Li, T. Xu, W. Du, and X. Fu, « Optimizing cost for online social networks on geo-distributed clouds », *IEEE/ACM Transactions on Networking*, vol. 24, *1*, 2014.

[27] I. Benkacem, T. Taleb, M. Bagaa, and H. Flinck, « Optimal VNFs placement in CDN slicing over multi-cloud environment », *IEEE Journal on Selected Areas in Communications*, vol. 36, *3*, 2018.

[28] E. S. Gama, R. Immich, and L. F. Bittencourt, « Towards a multi-tier fog/cloud architecture for video streaming », *in Proceedings of the IEEE/ACM International Conference on Utility and Cloud Computing Companion*, 2018.

[29] A. Sampaio and N. Mendonça, « Uni4cloud: an approach based on open standards for deployment and management of multi-cloud applications », *in Proceedings of the 2nd International Workshop on Software Engineering for Cloud Computing*, 2011.

[30] Y. Elkhatib, B. Porter, H. B. Ribeiro, M. F. Zhani, J. Qadir, and E. Rivière, « On using micro-clouds to deliver the fog », *IEEE Internet Computing*, vol. 21, *2*, 2017.

[31] M. Aazam and E.-N. Huh, « Fog computing micro datacenter based dynamic resource estimation and pricing model for IoT », *in Proceedings of the 29th IEEE International Conference on Advanced Information Networking and Applications*, 2015.

[32] ——, « Dynamic resource provisioning through fog micro datacenter », *in Proceedings of the IEEE International Conference on Pervasive Computing and Communication Workshops*, 2015.

[33] P. Bellavista and A. Zanni, « Feasibility of fog computing deployment based on docker containerization over raspberrypi », *in Proceedings of the 18th International Conference on Distributed Computing and Networking*, 2017.

[34] C. Pahl, S. Helmer, L. Miori, J. Sanin, and B. Lee, « A container-based edge cloud paas architecture based on raspberry pi clusters », *in Proceedings of the 4th IEEE International Conference on Future Internet of Things and Cloud Workshops*, 2016.

[35] S. T. Selvi, C. Valliyammai, and V. N. Dhatchayani, « Resource allocation issues and challenges in cloud computing », *in Proceedings of the International Conference on Recent Trends in Information Technology*, 2014.

[36] F. Nzanywayingoma and Y. Yang, « Efficient resource management techniques in cloud computing environment: a review and discussion », *International Journal of Computers and Applications*, vol. 41, *3*, 2019.

[37] S. Singh and I. Chana, « A Survey on Resource Scheduling in Cloud Computing: Issues and Challenges », *Journal of Grid Computing*, vol. 14, *2*, 2016.

[38] J. O. Kephart and D. M. Chess, « The vision of autonomic computing », *Computer*, vol. 36, *1*, 2003.

[39] R. Buyya, R. N. Calheiros, and X. Li, « Autonomic cloud computing: Open challenges and architectural elements », *in Proceedings of the 3rd International Conference on Emerging Applications of Information Technology*, 2012.

[40] I. Brandic, « Towards self-manageable cloud services », *in Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference*, 2009.

[41] M. Maurer, I. Breskovic, V. C. Emeakaroha, and I. Brandic, « Revealing the MAPE loop for the autonomic management of cloud infrastructures », *in Proceedings of the IEEE Symposium on Computers and Communications*, 2011.

[42] S. Dobson, R. Sterritt, P. Nixon, and M. Hinchey, « Fulfilling the vision of autonomic computing », *Computer*, vol. 43, *1*, 2010.

[43] M. R. Nami and K. Bertels, « A survey of autonomic computing systems », *in Proceedings of the 3rd International Conference on Autonomic and Autonomous Systems (ICAS)*, 2007.

[44] D. Bernstein, « Containers and cloud: From lxc to docker to kubernetes », *IEEE Cloud Computing*, vol. 1, *3*, 2014.

[45] D. Merkel, « Docker: lightweight linux containers for consistent development and deployment », *Linux Journal*, vol. 2014, *239*, 2014.

[46] R. Peinl, F. Holzschuher, and F. Pfitzer, « Docker cluster management for the cloud-survey results and own solution », *Journal of Grid Computing*, vol. 14, *2*, 2016.

[47] Docker. (2021). Swarm mode overview, [Online]. Available: https://docs.docker.com/engine/swarm/. (accessed: 04.03.2021).

[48] Mesosphere. (2021). Marathon: A container orchestration platform for Mesos and DC/OS, [Online]. Available: https://mesosphere.github.io/marathon/. (accessed: 04.03.2021).

[49] Kubernetes. (2021). Production-Grade Container Orchestration, [Online]. Available: https://kubernetes.io/. (accessed: 04.03.2021).

[50] ——, (2015). Borg: The Predecessor to Kubernetes, [Online]. Available: https://kubernetes.io/blog/2015/04/borg-predecessor-to-kubernetes/. (accessed: 15.03.2021).

[51] L. Larsson, H. Gustafsson, C. Klein, and E. Elmroth, « Decentralized Kubernetes Federation Control Plane », *in Proceedings of the 13th IEEE/ACM International Conference on Utility and Cloud Computing*, 2020.

[52] T. C. Chieu, A. Mohindra, A. A. Karve, and A. Segal, « Dynamic scaling of web applications in a virtualized cloud computing environment », *in Proceedings of the IEEE International Conference on e-Business Engineering*, 2009.

[53] H. Fernandez, G. Pierre, and T. Kielmann, « Autoscaling web applications in heterogeneous cloud infrastructures », *in Proceedings of the IEEE International Conference on Cloud Engineering*, 2014.

[54] N. Roy, A. Dubey, and A. Gokhale, « Efficient autoscaling in the cloud using predictive models for workload forecasting », *in Proceedings of the 4th IEEE International Conference on Cloud Computing*, 2011.

[55] A. Gandhi, P. Dube, A. Karve, A. Kochut, and L. Zhang, « Adaptive, model-driven autoscaling for cloud applications », *in Proceedings of the 11th International Conference on Autonomic Computing*, 2014.

[56] W. Iqbal, M. N. Dailey, D. Carrera, and P. Janecek, « Adaptive resource provisioning for read intensive multi-tier applications in the cloud », *Future Generation Computer Systems*, vol. 27, *6*, 2011.

[57] A. Ali-Eldin, O. Seleznjev, S. Sjöstedt-de Luna, J. Tordsson, and E. Elmroth, « Measuring cloud workload burstiness », *in Proceedings of the 7th IEEE/ACM International Conference on Utility and Cloud Computing*, 2014.

[58] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal, « Dynamic provisioning of multi-tier internet applications », *in Proceedings of the 2nd International Conference on Autonomic Computing*, 2005.

[59] A. Bauer, N. Herbst, S. Spinner, A. Ali-Eldin, and S. Kounev, « Chameleon: A hybrid, proactive auto-scaling mechanism on a level-playing field », *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, *4*, 2018.

[60] S. K. Garg, A. N. Toosi, S. K. Gopalaiyengar, and R. Buyya, « SLA-based virtual machine management for heterogeneous workloads in a cloud datacenter », *Journal of Network and Computer Applications*, vol. 45, 2014.

[61] N. Kherraf, H. A. Alameddine, S. Sharafeddine, C. M. Assi, and A. Ghrayeb, « Optimized provisioning of edge computing resources with heterogeneous workload in IoT networks », *IEEE Transactions on Network and Service Management*, vol. 16, *2*, 2019.

[62] B. S. Kapil and S. S. Kamath, « Resource aware scheduling in Hadoop for heterogeneous workloads based on load estimation », *in Proceedings of the Fourth International Conference on Computing, Communications and Networking Technologies*, 2013.

[63] Z. Li, J. Ge, H. Yang, L. Huang, H. Hu, H. Hu, and B. Luo, « A security and cost aware scheduling algorithm for heterogeneous tasks of scientific workflow in clouds », *Future Generation Computer Systems*, vol. 65, 2016.

[64] V. Podolskiy, A. Jindal, and M. Gerndt, « Multilayered autoscaling performance evaluation: Can virtual machines and containers co-scale? », *International Journal of Applied Mathematics and Computer Science*, vol. 29, *2*, 2019.

[65] S. Lanzman. (2017). Node Auto-provisioning, [Online]. Available: https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/proposals/node_autoprovisioning.md. (accessed: 01.04.2021).

[66] Amazon Web Services. (2021). Auto Scaling groups with multiple instance types and purchase options, [Online]. Available: https://docs.aws.amazon.com/autoscaling/ec2/userguide/asg-purchase-options.html. (accessed: 01.04.2021).

[67] Google Kubernetes Engine. (2021). Using node auto-provisioning, [Online]. Available: https://cloud.google.com/kubernetes-engine/docs/how-to/node-auto-provisioning. (accessed: 01.04.2021).

[68] M. Bertier, O. Marin, and P. Sens, « Implementation and performance evaluation of an adaptable failure detector », *in Proceedings of the International Conference on Dependable Systems and Networks*, 2002.

[69] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish, « Detecting failures in distributed systems with the falcon spy network », *in Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011.

[70] M. A. Tamiru, J. Tordsson, E. Elmroth, and G. Pierre, « An Experimental Evaluation of the Kubernetes Cluster Autoscaler in the Cloud », *in Proceedings of the 12th IEEE International Conference on Cloud Computing Technology and Science*, 2020.

[71] M. A. Tamiru, G. Pierre, J. Tordsson, and E. Elmroth, « Instability in Geo-Distributed Kubernetes Federation: Causes and Mitigation », *in Proceedings of the 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2020.

[72] X. Masip-Bruin, E. Marín-Tordera, G. Tashakor, A. Jukan, and G.-J. Ren, « Foggy clouds and cloudy fogs: a real need for coordinated management of fog-to-cloud computing systems », *IEEE Wireless Communications*, vol. 23, *5*, 2016.

[73] S. Yi, C. Li, and Q. Li, « A survey of fog computing: concepts, applications and issues », *in Proceedings of the 2015 Workshop on Mobile Big Data*, 2015.

[74] I. Martinez, A. S. Hafid, and A. Jarray, « Design, Resource Management and Evaluation of Fog Computing Systems: A Survey », *IEEE Internet of Things Journal*, vol. 8, *4*, 2020.

[75] Esther Shein. (2019). The most important cloud advances of the decade, [Online]. Available: https://tek.io/2RJNzNM. (accessed: 25.05.2021).

[76] Flexera. (2019). RightScale 2019 State of the Cloud Report from Flexera, [Online]. Available: https://bit.ly/3mXQVb4. (accessed: 19.04.2021).

[77] N. Galov. (2021). Cloud Adoption Statistics for 2021, [Online]. Available: https://bit.ly/3gzgNsT. (accessed: 19.04.2021).

[78] E. Commission. (2021). Cloud computing - statistics on the use by enterprises, [Online]. Available: https://bit.ly/3uZ5uOs. (accessed: 19.04.2021).

[79] IBM Cloud Team. (2017). A Brief History of Cloud Computing, [Online]. Available: https://ibm.co/3hRTYkt. (accessed: 25.05.2021).

[80] L. A. Barroso, U. Hölzle, and P. Ranganathan, *The datacenter as a computer: Designing warehouse-scale machines*. 2018.

[81] RedHat. (2019). The state of Linux in the public cloud for enterprises, [Online]. Available: https://red.ht/2Q1AVsK. (accessed: 12.04.2021).

[82] Y. Xing and Y. Zhan, « Virtualization and cloud computing », *in Future Wireless Networks and Information Systems*, 2012.

[83] M. Vaezi and Y. Zhang, « Virtualization and cloud computing », *in Cloud Mobile Networks*, 2017.

[84] P. Padala, X. Zhu, Z. Wang, S. Singhal, K. G. Shin, *et al.*, « Performance evaluation of virtualization technologies for server consolidation », *HP Labs Tec. Report*, vol. 137, 2007.

[85] W. Vogels, « Beyond Server Consolidation: Server consolidation helps companies improve resource utilization, but virtualization can help in other ways, too. », *Queue*, vol. 6, *1*, 2008.

[86] T. C. Ferreto, M. A. Netto, R. N. Calheiros, and C. A. De Rose, « Server consolidation with migration control for virtualized data centers », *Future Generation Computer Systems*, vol. 27, *8*, 2011.

[87] R. L. Grossman, « The case for cloud computing », *IT Professional*, vol. 11, *2*, 2009.

[88] P. Neto, « Demystifying cloud computing », *in Proceeding of the Doctoral Symposium on Informatics Engineering*, 2011.

[89] Flexera. (2021). Flexera 2021 State of the Cloud Report, [Online]. Available: https://info.flexera.com/CM-REPORT-State-of-the-Cloud?lead_source=Website%20Visitor&id=Blog. (accessed: 04.04.2021).

[90] G. Laatikainen, A. Ojala, and O. Mazhelis, « Cloud services pricing models », *in Proceedings of the International Conference of Software Business*, 2013.

[91] S. Chaisiri, B.-S. Lee, and D. Niyato, « Optimization of resource provisioning cost in cloud computing », *IEEE Transactions on Services Computing*, vol. 5, *2*, 2011.

[92] S. Li, Y. Zhou, L. Jiao, X. Yan, X. Wang, and M. R.-T. Lyu, « Towards operational cost minimization in hybrid clouds for dynamic resource provisioning with delay-aware optimization », *IEEE Transactions on Services Computing*, vol. 8, *3*, 2015.

[93] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, *The cost of a cloud: research problems in data center networks*, 2008.

[94] M. Finnegan. (2015). The Guardian goes all-in on AWS public cloud after OpenStack 'disaster', [Online]. Available: https://bit.ly/3afTErj. (accessed: 16.04.2021).

[95] Y.-S. Chang, Y.-K. Lee, T.-Y. Juang, and J.-S. Yen, « Cost evaluation on building and operating cloud platform », *International Journal of Grid and High Performance Computing*, vol. 5, *2*, 2013.

[96] I. Foster, Y. Zhao, I. Raicu, and S. Lu, « Cloud computing and grid computing 360-degree compared », *in Proceedings of the Grid Computing Environments Workshop*, 2008.

[97] M. Gall, A. Schneider, and N. Fallenbeck, « An architecture for community clouds using concepts of the intercloud », *in Proceedings of the 27th IEEE International Conference on Advanced Information Networking and Applications*, 2013.

[98] A. Marinos and G. Briscoe, « Community cloud computing », *in Proceedings of the IEEE International Conference on Cloud Computing*, 2009.

[99] Q. Li, Z.-y. Wang, W.-h. Li, J. Li, C. Wang, and R.-y. Du, « Applications integration in a hybrid cloud computing environment: Modelling and platform », *Enterprise Information Systems*, vol. 7, *3*, 2013.

[100] Y. Wu, C. Wu, B. Li, L. Zhang, Z. Li, and F. C. Lau, « Scaling social media applications into geo-distributed clouds », *IEEE/ACM Transactions On Networking*, vol. 23, *3*, 2014.

[101] C. Gong, J. Liu, Q. Zhang, H. Chen, and Z. Gong, « The characteristics of cloud computing », *in Proceedings of the 39th International Conference on Parallel Processing Workshops*, 2010.

[102] T. Dillon, C. Wu, and E. Chang, « Cloud computing: issues and challenges », *in Proceedings of the 24th IEEE International Conference on Advanced Information Networking and Applications*, 2010.

[103] K. V. Vishwanath and N. Nagappan, « Characterizing cloud computing hardware reliability », *in Proceedings of the 1st ACM Symposium on Cloud Computing*, 2010.

[104] N. Bitar, S. Gringeri, and T. J. Xia, « Technologies and protocols for data center and cloud networking », *IEEE Communications Magazine*, vol. 51, *9*, 2013.

[105] CLAudit. (2017). CLAudit – Planetary-scale cloud latency auditing, [Online]. Available: https://bit.ly/3ghwt3t. (accessed: 14.04.2021).

[106] A. Vakali and G. Pallis, « Content delivery networks: Status and trends », *IEEE Internet Computing*, vol. 7, *6*, 2003.

[107] M. Rosemain and R. Satter. (2021). Millions of websites offline after fire at French cloud services firm, [Online]. Available: https://reut.rs/3dyByD0. (accessed: 21.04.2021).

[108] D. Cronin. (2021). Give OVH a Break. And Use the Data Center Fire as a Teachable Moment, [Online]. Available: https://bit.ly/3aqgTim. (accessed: 21.04.2021).

[109] OpenAI. (2021). Scaling Kubernetes to 7,500 Nodes, [Online]. Available: https://bit.ly/2RHVxXl. (accessed: 12.04.2021).

[110] B. Tolson. (2019). Data Sovereignty and the GDPR; Do You Know Where Your Data Is?, [Online]. Available: https://bit.ly/3iKc1K1. (accessed: 17.06.2021).

[111] B. Satzger, W. Hummer, C. Inzinger, P. Leitner, and S. Dustdar, « Winds of change: From vendor lock-in to the meta cloud », *IEEE internet computing*, vol. 17, *1*, 2013.

[112] J. Opara-Martins, R. Sahandi, and F. Tian, « Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective », *Journal of Cloud Computing*, vol. 5, *1*, 2016.

[113] C. Metz. (2014). How Facebook Moved 20 Billion Instagram Photos Without You Noticing, [Online]. Available: https://bit.ly/3e6LWkl. (accessed: 12.04.2021).

[114] Google Cloud. (2020). Hybrid and multi-cloud patterns and practices, [Online]. Available: https://cloud.google.com/solutions/hybrid-and-multi-cloud-patterns-and-practices. (accessed: 13.04.2021).

[115] Y. Mansouri, V. Prokhorenko, and M. A. Babar, « An automated implementation of hybrid cloud for performance evaluation of distributed databases », *Journal of Network and Computer Applications*, vol. 167, 2020.

[116] P. Trakadas, N. Nomikos, E. T. Michailidis, T. Zahariadis, F. M. Facca, D. Breitgand, S. Rizou, X. Masip, and P. Gkonis, « Hybrid clouds for data-intensive, 5G-enabled IoT applications: An overview, key issues and relevant architecture », *Sensors*, vol. 19, *16*, 2019.

[117] T. Guo, U. Sharma, P. Shenoy, T. Wood, and S. Sahu, « Cost-aware cloud bursting for enterprise applications », *ACM Transactions on Internet Technology*, vol. 13, *3*, 2014.

[118] TechRepublic Staff. (2020). Multicloud: A cheat sheet, [Online]. Available: https://www.techrepublic.com/article/multicloud-the-smart-persons-guide/. (accessed: 25.05.2021).

[119] Microsoft Azure. (2021). Hybrid and multicloud solutions, [Online]. Available: https://bit.ly/3tvh3wE. (accessed: 13.04.2021).

[120] Amazon Web Services. (2021). Global Network, [Online]. Available: https://amzn.to/3e6pcB8. (accessed: 13.04.2021).

[121] B. T. Sloss. (2018). Expanding our global infrastructure with new regions and subsea cables, [Online]. Available: https://bit.ly/3mSrOGu. (accessed: 13.04.2021).

[122] A. J. Fahs and G. Pierre, « Tail-latency-aware fog application replica placement », *in Proceedings of the International Conference on Service-Oriented Computing*, 2020.

[123] Amazon Web Services. (2021). AWS for the Edge, [Online]. Available: https://amzn.to/3spJURi. (accessed: 14.04.2021).

[124] Google Cloud. (2021). Anthos at the Edge, [Online]. Available: https://cloud.google.com/solutions/anthos-edge. (accessed: 14.04.2021).

[125] S. Tripathi. (2022). MobiledgeX Edge-Cloud R2.0 is Released, [Online]. Available: https://bit.ly/3adhsvJ. (accessed: 14.04.2021).

[126] K. Manaouil and A. Lebre, « Kubernetes and the Edge? », PhD thesis, Inria Rennes-Bretagne Atlantique, 2020.

[127] C. Wöbker, A. Seitz, H. Mueller, and B. Bruegge, « Fogernetes: Deployment and management of fog computing applications », *in Proceedings of the IEEE/IFIP Network Operations and Management Symposium*, 2018.

[128] D. Santoro, D. Zozin, D. Pizzolli, F. De Pellegrini, and S. Cretti, « Foggy: a platform for workload orchestration in a fog computing environment », *in Proceedings of the IEEE International Conference on Cloud Computing Technology and Science*, 2017.

[129] A. J. Fahs, G. Pierre, and E. Elmroth, « Voilà: Tail-latency-aware fog application replicas autoscaler », *in Proceedings of the 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2020.

[130] P. Kayal, « Kubernetes in Fog Computing: Feasibility Demonstration, Limitations and Improvement Scope », *in Proceedings of the 6th IEEE World Forum on Internet of Things*, 2020.

[131] O. Laadan and J. Nieh, « Operating system virtualization: practice and experience », *in Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, 2010.

[132] A. Honig and N. Porter. (2017). 7 ways we harden our KVM hypervisor at Google Cloud: security in plaintext, [Online]. Available: https://cloud.google.com/blog/products/gcp/7-ways-we-harden-our-kvm-hypervisor-at-google-cloud-security-in-plaintext. (accessed: 05.04.2021).

[133] Amazon Web Services. (2021). Amazon EC2 FAQs, [Online]. Available: https://aws.amazon.com/ec2/faqs/#compute-optimized. (accessed: 05.04.2021).

[134] Microsoft. (2021). Hypervisor security on the Azure fleet, [Online]. Available: https://docs.microsoft.com/en-us/azure/security/fundamentals/hypervisor. (accessed: 05.04.2021).

[135] J. Tordsson, R. S. Montero, R. Moreno-Vozmediano, and I. M. Llorente, « Cloud brokering mechanisms for optimized placement of virtual machines across multiple providers », *Future Generation Computer Systems*, vol. 28, *2*, 2012.

[136] R. Dua, A. R. Raja, and D. Kakadia, « Virtualization vs containerization to support PaaS », *in Proceedings of the IEEE International Conference on Cloud Engineering*, 2014.

[137] C. Arango, R. Dernat, and J. Sanabria, « Performance evaluation of container-based virtualization for high performance computing environments », *CoRR*, vol. abs/1709.101 2017.

[138] Q. Zhang, L. Liu, C. Pu, Q. Dou, L. Wu, and W. Zhou, « A comparative study of containers and virtual machines in big data environment », *in Proceedings of the 11th IEEE International Conference on Cloud Computing*, 2018.

[139] A. Ahmed and G. Pierre, « Docker container deployment in fog computing infrastructures », *in Proceedings of the IEEE International Conference on Edge Computing*, 2018.

[140] A. Ahmed, A. Mohan, G. Cooperman, and G. Pierre, « Docker container deployment in distributed fog infrastructures with checkpoint/restart », *in Proceedings of the 8th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, 2020.

[141] A. Ahmed and G. Pierre, « Docker-pi: Docker container deployment in fog computing infrastructures », *International Journal of Cloud Computing*, vol. 9, *1*, 2020.

[142] T. Lorido-Botran, S. Huerta, L. Tomás, J. Tordsson, and B. Sanz, « An unsupervised approach to online noisy-neighbor detection in cloud data centers », *Expert Systems with Applications*, vol. 89, 2017.

[143] P. Sharma, L. Chaufournier, P. Shenoy, and Y. Tay, « Containers and virtual machines at scale: A comparative study », *in Proceedings of the 17th International Middleware Conference*, 2016.

[144] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, « An updated performance comparison of virtual machines and linux containers », *in Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, 2015.

[145] K. Razavi, A. Ion, G. Tato, K. Jeong, R. Figueiredo, G. Pierre, and T. Kielmann, « Kangaroo: A tenant-centric software-defined cloud infrastructure », *in Proceedings of the IEEE International Conference on Cloud Engineering*, 2015.

[146] I. Mavridis and H. Karatza, « Performance and overhead study of containers running on top of virtual machines », *in Proceedings of the 9th IEEE Conference on Business Informatics*, 2017.

[147] ——, « Combining containers and virtual machines to enhance isolation and extend functionality on cloud computing », *Future Generation Computer Systems*, vol. 94, 2019.

[148] Docker. (2021). Docker Registry, [Online]. Available: https://dockr.ly/3dHPCu0. (accessed: 22.04.2021).

[149] A. Aral and I. Brandić, « Learning Spatiotemporal Failure Dependencies for Resilient Edge Computing Services », *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, *7*, 2020.

[150] U. Nachmany. (2018). Kubernetes: Evolution Of An IT Revolution, [Online]. Available: https://bit.ly/3fX763x. (accessed: 15.03.2021).

[151] Datadog. (2020). 11 facts about real-world container use, [Online]. Available: https://bit.ly/3tBR073. (accessed: 16.04.2021).

[152] S. S. Manvi and G. K. Shyam, « Resource management for Infrastructure as a Service (IaaS) in cloud computing: A survey », *Journal of Network and Computer Applications*, vol. 41, 2014.

[153] M. Liaqat, V. Chang, A. Gani, S. H. Ab Hamid, M. Toseef, U. Shoaib, and R. L. Ali, « Federated cloud resource management: Review and discussion », *Journal of Network and Computer Applications*, vol. 77, 2017.

[154] K. Toczé and S. Nadjm-Tehrani, « A taxonomy for management and optimization of multiple resources in edge computing », *Wireless Communications and Mobile Computing*, vol. 2018, 2018.

[155] B. Jennings and R. Stadler, « Resource management in clouds: Survey and research challenges », *Journal of Network and Systems Management*, vol. 23, *3*, 2015.

[156] A. Aral, I. Brandic, R. B. Uriarte, R. De Nicola, and V. Scoca, « Addressing application latency requirements through edge scheduling », *Journal of Grid Computing*, vol. 17, *4*, 2019.

[157] C. Qu, R. N. Calheiros, and R. Buyya, « Auto-scaling web applications in clouds: A taxonomy and survey », *ACM Computing Surveys*, vol. 51, *4*, 2018.

[158] P. Hoenisch, I. Weber, S. Schulte, L. Zhu, and A. Fekete, « Four-fold auto-scaling on a contemporary deployment platform using docker containers », *in Proceedings of the International Conference on Service-Oriented Computing*, 2015.

[159] Google Cloud Platform. (2014). An update on container support on Google Cloud Platform, [Online]. Available: https://cloudplatform.googleblog.com/2014/06/an-update-on-container-support-on-google-cloud-platform.html. (accessed: 15.03.2021).

[160] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, « Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade », *Queue*, vol. 14, *1*, 2016.

[161] E. A. Brewer, « Kubernetes and the path to cloud native », *in Proceedings of the 6th ACM Symposium on Cloud Computing*, 2015.

[162] Kubernetes. (2021). Cluster Networking, [Online]. Available: https://bit.ly/3gopSEE. (accessed: 21.04.2021).

[163] Cilium. (2021). Deep Dive into Cilium Multi-cluster, [Online]. Available: https://bit.ly/3aI27E3. (accessed: 20.04.2021).

[164] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I. M. Llorente, R. Montero, Y. Wolfsthal, E. Elmroth, J. Caceres, *et al.*, « The Reservoir model and architecture for open federated cloud computing », *IBM Journal of Research and Development*, vol. 53, *4*, 2009.

[165] Kubernetes. (2021). Custom Resources, [Online]. Available: https://bit.ly/32DDXWD. (accessed: 22.04.2021).

[166] ——, (2021). Operator pattern, [Online]. Available: https://bit.ly/3gx7Q2Y. (accessed: 22.04.2021).

[167] ——, (2017). Node Auto-provisioning, [Online]. Available: https://bit.ly/3ncm0I5. (accessed: 22.04.2021).

[168] Google Kubernetes Engine. (2021). Using node auto-provisioning, [Online]. Available: https://bit.ly/3tIlsMx. (accessed: 22.04.2021).

[169] Kubernetes. (2021). What are Expanders?, [Online]. Available: https://bit.ly/3aAlfDA. (accessed: 22.04.2021).

[170] Kubernetes SIG Multicluster. (2020). Kubernetes Cluster Federation, [Online]. Available: https://github.com/kubernetes-sigs/kubefed.

[171] F. Faticanti, D. Santoro, S. Cretti, and D. Siracusa, « An Application of Kubernetes Cluster Federation in Fog Computing », *in Proceedings of the 24th Conference on Innovation in Clouds, Internet and Networks and Workshops*, 2021.

[172] SIG (Special Interest Group) Multi-Cluster. (2021). Kubernetes Cluster Federation, [Online]. Available: https://bit.ly/3n3HerM. (accessed: 21.04.2021).

[173] Kubernetes. (2021). Controllers, [Online]. Available: https://bit.ly/3appAtq. (accessed: 20.04.2021).

[174] J. Dobies and J. Wood, *Kubernetes Operators: Automating the Container Orchestration Platform*. O'Reilly Media, Inc., 2020.

[175] D. Petcu, « Portability and interoperability between clouds: challenges and case study », *in European Conference on a Service-Based Internet*, vol. 6994, 2011.

[176] R. Ranjan, « The cloud interoperability challenge », *IEEE Cloud Computing*, vol. 1, *2*, 2014.

[177] G. Iuhasz, P. Jamshidi, W. Wang, and G. Casale, « Load balancing for multi-cloud », *in Model-Driven Development and Operation of Multi-Cloud Applications*, 2017.

[178] S. Paul, R. Jain, M. Samaka, and J. Pan, « Application delivery in multi-cloud environments using software defined networking », *Computer Networks*, vol. 68, 2014.

[179] eBPF. (2021). What is eBPF?, [Online]. Available: https://bit.ly/3xcpqzp. (accessed: 20.04.2021).

[180] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang, « Bestconfig: tapping the performance potential of systems via automatic configuration tuning », *in Proceedings of the Symposium on Cloud Computing*, 2017.

[181] W. Zheng, R. Bianchini, and T. D. Nguyen, « Massconf: automatic configuration tuning by leveraging user community information », *in Proceedings of the 2nd ACM/SPEC International Conference on Performance engineering*, 2011.

[182] L. Bao, X. Liu, Z. Xu, and B. Fang, « Autoconfig: Automatic configuration tuning for distributed message systems », *in Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018.

[183] T. Chiba, R. Nakazawa, H. Horii, S. Suneja, and S. Seelam, « Confadvisor: A performance-centric configuration tuning framework for containers on kubernetes », *in Proceedings of the IEEE International Conference on Cloud Engineering (IC2E)*, 2019.

[184] M. Nardelli, V. Cardellini, and E. Casalicchio, « Multi-level elastic deployment of containerized applications in geo-distributed environments », *in Proceedings of the 6th IEEE International Conference on Future Internet of Things and Cloud*, 2018.

[185] Amazon Web Services. (2021). What is Amazon EC2 Auto Scaling?, [Online]. Available: https://amzn.to/32SyP12. (accessed: 27.04.2021).

[186] Google Compute Engine. (2021). Autoscaling groups of instances, [Online]. Available: https://bit.ly/32Sz7oE. (accessed: 27.04.2021).

[187] F. Al-Haidari, M. Sqalli, and K. Salah, « Impact of CPU utilization thresholds and scaling size on autoscaling cloud resources », *in Proceedings of the 5th IEEE International Conference on Cloud Computing Technology and Science*, 2013.

[188] A. Ali-Eldin, J. Tordsson, and E. Elmroth, « An adaptive hybrid elasticity controller for cloud infrastructures », *in Proceedings of the IEEE Network Operations and Management Symposium*, 2012.

[189] A. Naskos, E. Stachtiari, A. Gounaris, P. Katsaros, D. Tsoumakos, I. Konstantinou, and S. Sioutas, « Dependable horizontal scaling based on probabilistic model checking », *in Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2015.

[190] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, « A review of auto-scaling techniques for elastic applications in cloud environments », *Journal of Grid Computing*, vol. 12, *4*, 2014.

[191] E. Casalicchio and V. Perciballi, « Auto-scaling of containers: The impact of relative and absolute metrics », *in Proceedings of the 2nd IEEE International Workshops on Foundations and Applications of Self* Systems*, 2017.

[192] W. Zhang, Z. Zhang, and H.-C. Chao, « Cooperative fog computing for dealing with big data in the internet of vehicles: Architecture and hierarchical resource management », *IEEE Communications Magazine*, vol. 55, *12*, 2017.

[193] J. Kovács, « Supporting programmable autoscaling rules for containers and virtual machines on clouds », *Journal of Grid Computing*, vol. 17, *4*, 2019.

[194] S. N. Srirama, M. Adhikari, and S. Paul, « Application deployment using containers with auto-scaling for microservices in cloud environment », *Journal of Network and Computer Applications*, vol. 160, 2020.

[195] M. Imdoukh, I. Ahmad, and M. G. Alfailakawi, « Machine learning-based auto-scaling for containerized applications », *Neural Computing and Applications*, vol. 32, *13*, 2019.

[196] Y. Li and Y. Xia, « Auto-scaling web applications in hybrid cloud based on docker », *in Proceedings of the 5th International Conference on Computer Science and Network Technology*, 2016.

[197] A. Ilyushkin, A. Ali-Eldin, N. Herbst, A. Bauer, A. V. Papadopoulos, D. Epema, and A. Iosup, « An experimental performance evaluation of autoscalers for complex workflows », *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, vol. 3, *2*, 2018.

[198] L. Versluis, M. Neacsu, and A. Iosup, « A trace-based performance study of autoscaling workloads of workflows in datacenters », *in Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2018.

[199] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, « Elasticity in cloud computing: state of the art and research challenges », *IEEE Transactions on Services Computing*, vol. 11, *2*, 2017.

[200] A. V. Papadopoulos, A. Ali-Eldin, K.-E. Årzén, J. Tordsson, and E. Elmroth, « PEAS: A performance evaluation framework for auto-scaling strategies in cloud applications », *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, vol. 1, *4*, 2016.

[201] A. V. Papadopoulos, L. Versluis, A. Bauer, N. Herbst, J. von Kistowski, A. Ali-Eldin, C. L. Abad, J. N. Amaral, P. Tuma, and A. Iosup, « Methodological Principles for Reproducible Performance Evaluation in Cloud Computing », *in IEEE Transactions on Software Engineering*, vol. P-300, 2020.

[202] N. R. Herbst, S. Kounev, A. Weber, and H. Groenda, « BUNGEE: an elasticity benchmark for self-adaptive IaaS cloud environments », *in Proceedings of the 10th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2015.

[203] N. Herbst, R. Krebs, G. Oikonomou, G. Kousiouris, A. Evangelinou, A. Iosup, and S. Kounev, « Ready for rain? A view from SPEC research on the future of cloud metrics », *CoRR*, vol. abs/1604.03470, 2016.

[204] A. Bauer, V. Lesch, L. Versluis, A. Ilyushkin, N. Herbst, and S. Kounev, « Chamulteon: Coordinated auto-scaling of micro-services », *in Proceedings of the 39th IEEE International Conference on Distributed Computing Systems*, 2019.

[205] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev, « Teastore: A micro-service reference application for benchmarking, modeling and resource management research », *in Proceedings of the 26th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2018.

[206] Y. M. Ramirez, V. Podolskiy, and M. Gerndt, « Capacity-Driven Scaling Schedules Derivation for Coordinated Elasticity of Containers and Virtual Machines », *in Proceedings of the IEEE International Conference on Autonomic Computing*, 2019.

[207] H. Arkian, G. Pierre, J. Tordsson, and E. Elmroth, « Model-based stream processing auto-scaling in geo-distributed environments », *in Proceedings of the 30th International Conference on Computer Communications and Networks*, 2021.

[208] V. Podolskiy, A. Jindal, and M. Gerndt, « Iaas reactive autoscaling performance challenges », *in Proceedings of the 11th International Conference on Cloud Computing*, 2018.

[209] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes, « AGILE: Elastic distributed resource scaling for Infrastructure-as-a-Service », *in Proceedings of the 10th International Conference on Autonomic Computing*, 2013.

[210] A. Ilyushkin, B. Ghit, and D. Epema, « Scheduling workloads of workflows with unknown task runtimes », *in Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2015.

[211] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker, « Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software », *in Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, 2015.

[212] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia, and L. Zhang, « A smart hill-climbing algorithm for application server configuration », *in Proceedings of the 13th International Conference on World Wide Web*, 2004.

[213] W. Zheng, R. Bianchini, and T. D. Nguyen, « Automatic configuration of internet services », *in Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2007.

[214] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, « An empirical study on configuration errors in commercial and open source systems », *in Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, 2011.

[215] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang, « Automatic database management system tuning through large-scale machine learning », *in Proceedings of the ACM International Conference on Management of Data*, 2017.

[216] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, « Kubernetes as an Availability Manager for Microservice Applications », *CoRR*, vol. abs/1901.04946, 2019.

[217] M. D. McKay, R. J. Beckman, and W. J. Conover, « A comparison of three methods for selecting values of input variables in the analysis of output from a computer code », *Technometrics*, vol. 42, *1*, 2000.

[218] H. Chen, G. Jiang, H. Zhang, and K. Yoshihira, « Boosting the performance of computing systems through adaptive configuration tuning », *in Proceedings of the 2009 ACM Symposium on Applied Computing*, 2009.

[219] S. Bouchenak, N. De Palma, D. Hagimont, and C. Taton, « Autonomic management of clustered applications », *in Proceedings of the IEEE International Conference on Cluster Computing*, 2006.

[220] M. Li, L. Zeng, S. Meng, J. Tan, L. Zhang, A. R. Butt, and N. Fuller, « MRONLINE: MapReduce online performance tuning », *in Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, 2014.

[221] G. Wang, J. Xu, and B. He, « A novel method for tuning configuration parameters of Spark based on machine learning », *in Proceedings of the 18th IEEE International Conference on High Performance Computing and Communications; 14th IEEE International Conference on Smart City; 2nd IEEE International Conference on Data Science and Systems*, 2016.

[222] D. A. Menascé, D. Barbará, and R. Dodge, « Preserving QoS of e-commerce sites through self-tuning: a performance model approach », *in Proceedings of the 3rd ACM Conference on Electronic Commerce*, 2001.

[223] N. Gandhi, D. M. Tilbury, Y. Diao, J. Hellerstein, and S. Parekh, « MIMO control of an Apache web server: Modeling and controller design », *in Proceedings of the American Control Conference*, 2002.

[224] Y. Diao, N. Gandhi, J. L. Hellerstein, S. Parekh, and D. M. Tilbury, « Using MIMO feedback control to enforce policies for interrelated metrics with application to the Apache Web server », *in Proceedings of the IEEE/IFIP Network Operations and Management Symposium*, 2002.

[225] K. Ye and Y. Ji, « Performance tuning and modeling for big data applications in Docker containers », *in Proceedings of the International Conference on Networking, Architecture, and Storage*, 2017.

[226] T. Osogami and S. Kato, « Optimizing system configurations quickly by guessing at the performance », *in Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2007.

[227] I.-H. Chung and J. K. Hollingsworth, « Automated cluster-based web service performance tuning », *in Proceedings of the 13th IEEE International Symposium on High performance Distributed Computing*, 2004.

[228] C. Stewart and K. Shen, « Performance modeling and system management for multi-component online services », *in Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation*, 2005.

[229] X. Bu, J. Rao, and C.-Z. Xu, « A reinforcement learning approach to online web systems auto-configuration », *in Proceedings of the 29th IEEE International Conference on Distributed Computing Systems*, 2009.

[230] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu, « Starfish: A Self-tuning System for Big Data Analytics », *in Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*, 2011.

[231] J. L. Berral, N. Poggi, D. Carrera, A. Call, R. Reinauer, and D. Green, « Aloja-ml: A framework for automating characterization and knowledge discovery in Hadoop deployments », *in Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2015.

[232] S. Duan, V. Thummala, and S. Babu, « Tuning database configuration parameters with iTuned », *Proceedings of the VLDB Endowment*, vol. 2, *1*, 2009.

[233] M. Kaminski, E. Truyen, E. H. Beni, B. Lagaisse, and W. Joosen, « A framework for black-box SLO tuning of multi-tenant applications in Kubernetes », *in Proceedings of the 5th International Workshop on Container Technologies and Container Clouds*, 2019.

[234] V. Dalibard, M. Schaarschmidt, and E. Yoneki, « BOAT: Building auto-tuners with structured Bayesian optimization », *in Proceedings of the 26th International Conference on World Wide Web*, 2017.

[235] W. Chen, S. Toueg, and M. K. Aguilera, « On the quality of service of failure detectors », *IEEE Transactions on computers*, vol. 51, *5*, 2002.

[236] M. Pasin, S. Fontaine, and S. Bouchenak, « Failure detection in large scale systems: a survey », *in Proceedings of the IEEE Network Operations and Management Symposium Workshops*, 2008.

[237] T. D. Chandra and S. Toueg, « Unreliable failure detectors for reliable distributed systems », *Journal of the ACM*, vol. 43, *2*, 1996.

[238] F. Lima and R. Macêdo, « Adapting failure detectors to communication network load fluctuations using SNMP and artificial neural nets », *in Proceedings of the Latin-American Symposium on Dependable Computing*, 2005.

[239] R. C. Nunes and I. Jansch-Porto, « QoS of timeout-based self-tuned failure detectors: the effects of the communication delay predictor and the safety margin », *in Proceedings of the International Conference on Dependable Systems and Networks*, 2004.

[240] L. Falai and A. Bondavalli, « Experimental evaluation of the QoS of failure detectors on wide area network », *in Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2005.

[241] A. S. de Sá and R. J. de Araújo Macêdo, « QoS self-configuring failure detectors for distributed systems », *in Proceedings of the IFIP International Conference on Distributed Applications and Interoperable Systems*, 2010.

[242] Y. Zhu, J. Liu, M. Guo, W. Ma, and Y. Bao, « Acts in need: automatic configuration tuning with scalability guarantees », *in Proceedings of the 8th Asia-Pacific Workshop on Systems*, 2017.

[243] R. Moreno-Vozmediano, R. S. Montero, E. Huedo, and I. M. Llorente, « Orchestrating the deployment of high availability services on multi-zone and multi-cloud scenarios », *Journal of Grid Computing*, vol. 16, *1*, 2018.

[244] J. L. Lucas-Simarro, R. Moreno-Vozmediano, R. S. Montero, and I. M. Llorente, « Scheduling strategies for optimal service deployment across multiple clouds », *Future Generation Computer Systems*, vol. 29, *6*, 2013.

[245] M. Abdelbaky, J. Diaz-Montes, M. Parashar, M. Unuvar, and M. Steinder, « Docker containers across multiple clouds and data centers », *in Proceedings of the 8th IEEE/ACM International Conference on Utility and Cloud Computing*, 2015.

[246] A. N. Toosi, R. N. Calheiros, and R. Buyya, « Interconnected cloud computing environments: Challenges, taxonomy, and survey », *ACM Computing Surveys*, vol. 47, *1*, 2014.

[247] A. J. Ferrer, F. Hernández, J. Tordsson, E. Elmroth, A. Ali-Eldin, C. Zsigri, R. Sirvent, J. Guitart, R. M. Badia, K. Djemame, *et al.*, « OPTIMIS: A holistic approach to cloud service provisioning », *Future Generation Computer Systems*, vol. 28, *1*, 2012.

[248] E. Carlini, M. Coppola, P. Dazzi, L. Ricci, and G. Righetti, « Cloud federations in contrail », *in Proceedings of the European Conference on Parallel Processing*, 2011.

[249] T. Guo, U. Sharma, T. Wood, S. Sahu, and P. Shenoy, « Seagull: intelligent cloud bursting for enterprise applications », *in Proceedings of the USENIX Annual Technical Conference*, 2012.

[250] S. K. Nair, S. Porwal, T. Dimitrakos, A. J. Ferrer, J. Tordsson, T. Sharif, C. Sheridan, M. Rajarajan, and A. U. Khan, « Towards secure cloud bursting, brokerage and aggregation », *in Proceedings of the 8th IEEE European Conference on Web Services*, 2010.

[251] M. R. H. Farahabady, Y. C. Lee, and A. Y. Zomaya, « Pareto-optimal cloud bursting », *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, *10*, 2013.

[252] Y. C. Lee and B. Lian, « Cloud bursting scheduler for cost efficiency », *in Proceedings of the 10th IEEE International Conference on Cloud Computing (CLOUD)*, 2017.

[253] G. L. Stavrinides and H. D. Karatza, « Cost-aware cloud bursting in a fog-cloud environment with real-time workflow applications », *Concurrency and Computation: Practice and Experience*, 2020.

[254] F. Faticanti, J. Zormpas, S. Drozdov, K. Rausch, O. A. García, F. Sardis, S. Cretti, M. Amiribesheli, and D. Siracusa, « Distributed Cloud Intelligence: Implementing an ETSI MANO-Compliant Predictive Cloud Bursting Solution Using Openstack and Kubernetes », *in Proceedings of the International Conference on the Economics of Grids, Clouds, Systems, and Services*, 2020.

[255] D. Kim, H. Muhammad, E. Kim, S. Helal, and C. Lee, « TOSCA-based and federation-aware cloud orchestration for Kubernetes container platform », *Applied Sciences*, vol. 9, *1*, 2019.

[256] F. Rossi, V. Cardellini, and F. L. Presti, « Elastic deployment of software containers in geo-distributed computing environments », *in Proceedings of the IEEE Symposium on Computers and Communications*, 2019.

[257] F. Rossi, V. Cardellini, F. L. Presti, and M. Nardelli, « Geo-distributed efficient deployment of containers with Kubernetes », *Computer Communications*, vol. 159, pp. 161–174, 2020.

[258] N. Ferry, F. Chauvel, A. Rossini, B. Morin, and A. Solberg, « Managing multi-cloud systems with CloudMF », *in Proceedings of the Second Nordic Symposium on Cloud Computing & Internet Technologies*, 2013.

[259] J. L. L. Simarro, R. Moreno-Vozmediano, R. S. Montero, and I. M. Llorente, « Dynamic placement of virtual machines for cost optimization in multi-cloud environments », *in Proceedings of the International Conference on High Performance Computing & Simulation*, 2011.

[260] E. Casalicchio, « Container orchestration: a survey », *Systems Modeling: Methodologies and Tools*, 2019.

[261] Hashicorp Nomad. (2021). Workload Orchestration Made Easy, [Online]. Available: https://www.nomadproject.io/. (accessed: 01.04.2021).

[262] Amazon Web Services. (2021). Amazon Elastic Kubernetes Service, [Online]. Available: https://amzn.to/3byQLCK. (accessed: 18.05.2021).

[263] Google Cloud. (2021). Google Kubernetes Engine, [Online]. Available: https://cloud.google.com/kubernetes-engine. (accessed: 18.05.2021).

[264] Microsoft Azure. (2021). Service Azure Kubernetes (AKS), [Online]. Available: https://azure.microsoft.com/en-us/services/kubernetes-service/. (accessed: 18.05.2021).

[265] A. Lertsinsrubtavee, A. Ali, C. Molina-Jimenez, A. Sathiaseelan, and J. Crowcroft, « PiCasso: A lightweight edge computing platform », *in Proceedings of the 6th IEEE International Conference on Cloud Networking*, 2017.

[266] H. Khazaei, H. Bannazadeh, and A. Leon-Garcia, « Savi-iot: A self-managing containerized iot platform », *in Proceedings of the 5th IEEE International Conference on Future Internet of Things and Cloud*, 2017.

[267] E. Yigitoglu, M. Mohamed, L. Liu, and H. Ludwig, « Foggy: A framework for continuous automated iot application deployment in fog computing », *in Proceedings of the IEEE International Conference on AI & Mobile Services*, 2017.

[268] L. L. Jimenez and O. Schelen, « HYDRA: Decentralized Location-aware Orchestration of Containerized Applications », *IEEE Transactions on Cloud Computing*, 2020.

[269] F. Neves, R. Vilaça, and J. Pereira, « Black-box inter-application traffic monitoring for adaptive container placement », *in Proceedings of the 35th Annual ACM Symposium on Applied Computing*, 2020.

[270] V. Cardellini, F. L. Presti, M. Nardelli, and F. Rossi, « Self-adaptive container deployment in the fog: A survey », *in Proceedings of the International Symposium on Algorithmic Aspects of Cloud Computing*, 2019.

[271] C. Guerrero, I. Lera, and C. Juiz, « Resource optimization of container orchestration: a case study in multi-cloud microservices-based applications », *The Journal of Supercomputing*, vol. 74, *7*, 2018.

[272] H. M. Fard, R. Prodan, and F. Wolf, « A container-driven approach for resource provisioning in edge-fog cloud », *in Proceedings of the International Symposium on Algorithmic Aspects of Cloud Computing*, 2019.

[273] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, « Towards Network-Aware Resource Provisioning in Kubernetes for Fog Computing applications », *in Proceedings of the IEEE Conference on Network Softwarization*, 2019.

[274] Q. Fan, Y. Jiang, H. Yin, Y. Lyu, H. Huang, and X. Zhang, « Resource reservation and request routing for a cloud-based content delivery network », *in Proceedings of the IEEE International Conference on Service-Oriented System Engineering*, 2019.

[275] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, « Software-defined networking: A comprehensive survey », *Proceedings of the IEEE*, vol. 103, *1*, 2014.

[276] M. Alaluna, E. Vial, N. Neves, and F. M. Ramos, « Secure and dependable multi-cloud network virtualization », *in Proceedings of the 1st International Workshop on Security and Dependability of Multi-Domain Infrastructures*, 2017.

[277] Open vSwitch. (2021). Production Quality, Multilayer Open Virtual Switch, [Online]. Available: https://www.openvswitch.org/. (accessed: 18.05.2021).

[278] Istio. (2021). Multicluster Deployments, [Online]. Available: https://bit.ly/3weEHOy. (accessed: 17.05.2021).

[279] Linkerd. (2021). Multi-cluster communication, [Online]. Available: https://bit.ly/3tMZFTi. (accessed: 17.05.2021).

[280] A. S. Fadel and A. G. Fayoumi, « Cloud resource provisioning and bursting approaches », *in Proceedings of the 14th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, 2013.

[281] M. Nardelli, C. Hochreiner, and S. Schulte, « Elastic provisioning of virtual machines for container deployment », *in Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, 2017.

[282] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, « Towards understanding heterogeneous clouds at scale: Google trace analysis », *Intel Science and Technology Center for Cloud Computing, Tech. Rep*, vol. 84, 2012.

[283] Y. Chen, A. S. Ganapathi, R. Griffith, and R. H. Katz, « Analysis and lessons from a publicly available google cluster trace », *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-95*, vol. 94, 2010.

[284] Z. Liu and S. Cho, « Characterizing machines and workloads on a Google cluster », *in Proceedings of the 41st International Conference on Parallel Processing Workshops*, 2012.

[285] D. Balouek, A. C. Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, *et al.*, « Adding Virtualization Capabilities to the Grid'5000 Testbed », *in Proceedings of the International Conference on Cloud Computing and Services Science*, 2013.

[286] P. K. Janert, *Feedback control for computer systems: introducing control theory to enterprise programmers*. O'Reilly Media, Inc., 2013.

[287] T. Abdelzaher, Y. Diao, J. L. Hellerstein, C. Lu, and X. Zhu, « Introduction to control theory and its application to computing systems », *in Performance Modeling and Engineering*, 2008.

[288] J. G. Ziegler and N. B. Nichols, « Optimum settings for automatic controllers », *Transactions of the American Society of Mechanical Engineers*, vol. 64, *11*, 1942.

[289] A. Ali-Eldin, B. Wang, and P. J. Shenoy, « The Hidden cost of the Edge: A Performance Comparison of Edge and Cloud Latencies », *CoRR*, vol. abs/2104.14050, 2021.

[290] C. Reiss, J. Wilkes, and J. L. Hellerstein, « Google cluster-usage traces: format+ schema », *Google Inc., White Paper*, 2011.

[291] C. Reiss *et al.*, *Google cluster-usage traces: format+ schema*, Google White Paper, 2011.

[292] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, « Heterogeneity and dynamicity of clouds at scale: Google trace analysis », *in Proceedings of the 3rd ACM Symposium on Cloud Computing*, 2012.

[293] M. A. Hoque, X. Hong, and B. Dixon, « Analysis of mobility patterns for urban taxi cabs », *in Proceedings of the International Conference on Computing, Networking and Communications*, 2012.

[294] M. Sedaghat, F. Hernandez-Rodriguez, and E. Elmroth, « A virtual machine repacking approach to the horizontal vs. vertical elasticity trade-off for cloud autoscaling », *in Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, 2013.

[295] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, « Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility », *Future Generation Computer Systems*, vol. 25, *6*, 2009.

[296] B. Schroeder and G. A. Gibson, « A large-scale study of failures in high-performance computing systems », *IEEE Transactions on Dependable and Secure Computing*, vol. 7, *4*, 2009.

DOCTORAT BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre : Gestion automatique des ressources dans les environnements multi-clusters géo-distribués

Mot clés : Cloud hybride, multi-cloud, Fog Computing, orchestration de conteneurs, Kubernetes.

Résumé : Les environnements informatiques géo-distribués tels que le cloud hybride, le multi-cloud et le Fog Computing doivent être gérés de manière autonome à grande échelle pour améliorer l'utilisation des ressources, maximiser les performances et réduire les coûts. Cependant, la gestion des ressources dans ces environnements informatiques géo-distribués est difficile en raison de leur large distribution géographique, de mauvaises conditions de réseau, de l'hétérogénéité des ressources et de la capacité limitée. Dans cette thèse, nous abordons certains des défis de la gestion des ressources en utilisant la technologie des conteneurs. Tout d'abord, nous présentons une analyse expérimentale de l'autoscaling dans les clusters Kubernetes au niveau des conteneurs et des machines virtuelles. Deuxièmement, nous proposons un contrôleur proportionnel pour améliorer dynamiquement la stabilité des déploiements géo-distribués dans les fédérations Kubernetes. Enfin, nous proposons un système d'orchestration de conteneurs pour les environnements géo-distribués qui offre des capacités de placement, d'autoscaling, d'éclatement, de routage réseau et de provisionnement dynamique des ressources riches en politiques.

Title: Automatic Resource Management in Geo-Distributed Multi-Cluster Environments

Keywords: Hybrid cloud, multi-cloud, Fog Computing, Container orchestration, Kubernetes.

Abstract: Geo-distributed computing environments such as hybrid cloud, multi-cloud and Fog Computing need to be managed autonomously at large scales to improve resource utilization, maximize performance, and save costs. However, resource management in these geo-distributed computing environments is difficult due to wide geographical distributions, poor network conditions, heterogeneity of resources, and limited capacity. In this thesis, we address some of the resource management challenges using container technology. First, we present an experimental analysis of autoscaling in Kubernetes clusters at the container and Virtual Machine levels. Second, we propose a proportional controller to dynamically improve the stability of geo-distributed deployments at runtime in Kubernetes Federations. Finally, we develop a container orchestration framework for geo-distributed environments that offers policy-rich placement, autoscaling, bursting, network routing, and dynamic resource provisioning capabilities.