Data-driven Management Solution for Microservice-based Deep Learning Applications

Zeina Houmani

▶ **To cite this version:**

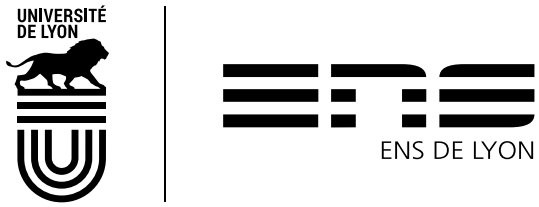
Zeina Houmani. Data-driven Management Solution for Microservice-based Deep Learning Applications. Other [cs.OH]. Université de Lyon, 2021. English. NNT : 2021LYSEN092 . tel-03593003

HAL Id: tel-03593003
https://theses.hal.science/tel-03593003

Submitted on 1 Mar 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Numéro National de Thèse : 2021LYSEN092

THÈSE DE DOCTORAT DE L'UNIVERSITÉ DE LYON

opérée par

l'École Normale Supérieure de Lyon

École Doctorale N° 512

École Doctorale en Informatique et Mathématiques de Lyon

Discipline: Informatique

Soutenue publiquement le 16/12/2021, par :

Zeina HOUMANI

Data-driven Management Solution for Microservice-based Deep Learning Applications

Solution de gestion pilotée par les données pour les applications Deep Learning basées sur des microservices

Devant le jury composé de:

Cedric TEDESCHI	Maître de conférences HDR	IRISA Rennes	Rapporteur
Laurence DUCHIEN	Professeure des universités	Université de Lille	Rapporteure
Michael KRAJECKI	Professeur des universités	Université de Reims	Examinateur
Laure GONNORD	Professeure des universités	Grenoble INP	Examinatrice
Gabriel ANTONIU	Directeur de recherche	Inria Rennes	Examinateur
Eddy CARON	Maître de conférences HDR	ENS de Lyon	Directeur
Daniel BALOUEK-THOMERT	Research associate	Université d'Utah	Co-Directeur

FOREWORD

The research work presented in this thesis is a collaboration between the Laboratoire de l'Informatique du Parallélisme (LIP) at ENS Lyon (Lyon, France) and the Rutgers Discovery Informatics Institute RDI^2 at Rutgers University (New Jersey, USA). During the first and last year, the realization of this thesis was under the Avalon team at the LIP laboratory. In the second year, I moved to the United States to continue the work under RDI^2 at Rutgers University.

This research is supported in part by the NSF under grants numbers OAC 1640834, OAC 1835661, OAC 1835692 and OCE 1745246 and in other part by LIP Laboratory and ENS grants.

Figure 1 – Logos of the collaborating institutions on the research work of this thesis

ACKNOWLEDGMENTS

Getting to the point of writing this acknowledgment section is the result of the help, encouragement, and support from many people around me whom I want to thank.

I would like first to express my sincere gratitude to the rapporteurs in my Ph.D. committee: Laurence Duchien and Cedric Tedeschi for reading my dissertation and writing detailed reports with great feedback. Also, I like to thank my thesis examiners: Laure Gonnord, Michael Krajecki, and Gabriel Antoniu for accepting to be part of the committee and dedicating part of their valuable time to assess my work.

I would like to express my deepest gratitude to my supervisors, Eddy Caron and Daniel Balouek-Thomert for their invaluable efforts, dedicated scientific guidance, and enthusiastic participation during this thesis. I appreciate their fruitful discussions, patience, and confidence in me.

Foremost, I am thankful to Eddy for his consistent support and encouragement throughout my master's internship and thesis. For me, Eddy was more than just an advisor. He quickly became the first scientific role model that I look up to as a growing researcher. He showed great manners, an infectious personality, and outstanding knowledge and expertise in the field of computer science. I thank him for not giving up on me after our awkward first meeting and after knowing my unimpressive knowledge in the film industry.

I'm grateful for Daniel for helping me grow as a researcher. I learned from him to pay close attention to details, to devote enough time for the work planning phase, and to never cut corners. I'm always thankful for his constructive comments which helped me improve my writing skills. Being his first official Ph.D. student made this journey more exciting knowing that we were both learning how to cope with the challenges that came our way during this process.

I'm super happy and fortunate to have had the chance to work closely with Eddy and Daniel. I hope I have lived up to their expectations.

I would like to thank Mr. Christian Perez, the leader of the Avalon team, and Mr. Manish Parashar, the director of the RDI2 team at the time, for their benevolent suggestions and for giving me the opportunity to be part of their research teams during this thesis.

I like to thank all the members of Avalon for the group meetings, work presentations, discussions, and the calming pre-covid19 coffee breaks. I would like to thank the members of the RDI2 team, especially Eduard Renart for the useful discussions and suggestions.

I would like to thank Mr. Yves Caniou, a member of the LIP laboratory staff, for introducing me to the LIP laboratory when I was looking for a research internship. That's when it all started.

I'm grateful to the administrative staff in LIP, especially, Mme Evelyne Blesle, Mme Marie Bozo, Mme Chiraz Benamor, and Mme Sylvie Boyer for helping me with so many things over the past few years and making laboratory life so much easier.

I'm also thankful for Mme Noraida Martinez, administrative staff at Rutgers University, who worked hard for making my visit to Rutgers University well organized.

Thanks to this journey, I met many incredible people in Lyon and New Jersey. I like to thank my friends Sarah, Orsola, Soha, Sally, and Hiba for making ordinary moments in Lyon extraordinary. I like to express my gratitude to Fatima, Ghadir, and Batoul, whom I met in New Jersey, for welcoming me and providing me with the help I needed to adapt quickly to the new environment.

I want to thank my roommates at the Rutgers dorm in New Jersey, Anna and Sunin. Despite their short stay, they made life in the dormitory feel less lonely. I like to thank Anna for the great memories in New York and the long and fun discussions. I'm thankful for Sunin for being a super kind roommate, for the enthusiastic "Momoland" morning alarm, and for introducing me to the delicious Tteokbokki.

I like to thank my childhood friend Bilal for being a brother who has been by my side since forever and has always been a person I can rely on. Huge thanks to my best friends Marwa, Lina, Batoul, Hiba and Maysa for remaining great support despite choosing different career and being thousands of miles apart.

Last but not least, I would like to express my gratitude to my beloved family for their continuous encouragement and support during this special chapter of my life. In particular, I would like to thank first my parents, Khalil Houmani and Amina Cheaib, and my sisters, Zeinab and Zahraa, who have always given me the support, the strength, and the positive energy that I needed. I am grateful to them for always being there for me. Also, I want to thank my uncle Taan Cheaib for the moral support. Many thanks to my cousin Rayan for the numerous walks in Paris, the Boursin, and for being someone I can count on.

In memory of my dearest grandparents, Zamzam cheaib, Zeinab Kanso, and my aunt Khadija Houmani, whom I lost during this journey.

DEDICATION

To my lovely parents and sisters.
Thank you for supporting my dreams and believing in me.
To my late grandmothers and late aunt.
Thank you for caring about me. I will hold you safe in my heart forever.

ABSTRACT

We live in a new era of Big Data, the era of insights. While our capacity to collect real-time data has grown significantly over the past decade, our ability to analyze that data to turn it into knowledge has not kept pace. With new generations of devices and network technologies, the focus of Big Data is shifting toward the design of tools and applications able to extract information from collected data. The majority of emerging applications present expectations of near-real-time processing to maintain the validity of their results. However, guaranteeing their performance requirements is hampered by the traditional Cloud system designs and management strategies. Current systems for Big Data applications rely on heterogeneous resources distributed across the constrained Edge and the powerful Cloud. In addition, the applications are now created as a set of self-contained microservices, developed by independent teams following the DevOps practices. This evolution of systems designs has introduced extreme heterogeneity and uncertainty into emerging applications, highlighting the limitations of traditional management strategies.

In this thesis, we focus on designing a system for Big Data applications that rethinks existing management strategies with a particular emphasis on the heterogeneity of incoming data, applications, and resources. We first study the decoupling of data producers and consumers in emerging microservice-based applications as the entry point to effectively leverage available services, even newly published ones. Accordingly, we propose a data-driven service discovery framework based on data-centric service descriptions and rely on a Peer-to-Peer data-driven architecture. In addition, we present an adaptation scheme that scales deployed microservices to tackle the impact of fluctuating load on real-time performance. Second, we investigate the trade-off between the quality and urgency of the results in Big Data applications as a promising strategy to overcome the limited and heterogeneous capacity of system resources. In particular, we present a data-driven workflow scheduling approach to distribute microservices across the edge of the network, the core, and along the data path. Additionally, we propose a data adaptation strategy that reduces the quality of incoming data when potential quality-latency trade-off optimizations are available. We then apply the proposed approaches in the context of Deep Learning applications.

RÉSUMÉ EN FRANÇAIS

La capacité à collecter des données en temps réel a considérablement augmenté au cours de la dernière décennie. Pourtant, la capacité d'analyser ces données pour les exploiter n'a pas suivi la même progression. Avec le développement des appareils connectés et des technologies réseau, le Big Data a fait émerger des outils et des applications capables d'extraire des informations à partir des données collectées. La majorité des applications issues du Big Data nécessitent un traitement en temps réel pour maintenir la pertinence de leurs résultats. Cependant, garantir la performance en temps réel est entravé par les stratégies de gestion traditionnelles des systèmes Cloud. Les systèmes actuels pour les applications Big Data reposent sur des ressources hétérogènes allant de la périphérie du réseau (faible latence mais puissance limitée) jusqu'au Cloud (latence importante mais disposant d'importante puissance de traitement).

Par ailleurs, les applications sont de plus en plus créées sous la forme d'un ensemble de services autonomes, développés par des équipes indépendantes. Cette évolution des conceptions de systèmes a introduit une hétérogénéité et une incertitude extrêmes dans les applications émergentes, mettant en évidence les limites des stratégies de gestion traditionnelles.

Cette thèse s'intéresse à la conception d'un système pour les applications Big Data (puis plus précisément des applications de Deep Learning) qui repense les stratégies de gestion existantes avec un accent particulier sur l'hétérogénéité des données entrantes, des applications et des ressources. Nous étudions d'abord la problématique d'accès à des services ou des types de services sans être en mesure de connaître leurs identifiants (par exemple leur nom précis). Puis dans un second temps, nous étudions le compromis entre la qualité et l'accesibilité des résultats dans les applications de Big Data pour proposer une stratégie prometteuse pour surmonter la capacité limitée et hétérogène des ressources système. Un aperçu de chaque contribution de cette thèse est présenté ci-dessous :

La découverte de services basée sur les données. La découverte de services est le processus de localisation d'un fournisseur de services approprié aux besoins du client. Les approches actuelles recherchent l'emplacement de services particuliers à l'aide de leurs identifiants. Cependant, les systèmes actuels incluent des services dynamiques, avec des implémentations hétérogènes, et parfois sans identifiants. Dans cette thèse, nous choisissons une approche originale en nous concentrant sur des informations liées

aux données plutôt que sur des identifiants. Cette approche est basée sur une description de microservices centrés sur les données et sur une architecture Peer-to-Peer pour couvrir de vastes zones géographiques.

Un schéma d'adaptation des microservices basé sur les données. Le trafic entrant des applications de Big Data est hétérogène en termes de type, format et qualité des données. De plus, il se caractérise par un taux de génération de données dynamique. Une approche pour controler la latence des applications consiste à mettre à l'échelle les services, ce qui correspond à l'augmentation ou la diminution de leur capacité de calcul. La plupart des approches traditionnelles reposent uniquement sur des métriques liées à l'infrastructure ou aux applications sans tenir compte de l'hétérogénéité du trafic. Dans cette thèse, la mise à l'échelle est effectuée selon des métriques liées aux données et pour des groupes de microservices ayant les mêmes données d'entrée.

Une évaluation expérimentale de l'approche de découverte de services. Le routage du trafic et la supervision des microservices sont realisés par le project *Istio*. Cependant, ce projet ne prend pas en charge la création d'une approche de découverte basée sur les données avec des garanties de qualité de service. Dans cette thèse, un ensemble de services de gestion est développé pour intégrer la découverte de microservice proposée dans *Istio*. L'évaluation des approaches proposées sur *Grid'5000* a montré que la plate-forme peut maintenir une latence et un pourcentage de requêtes traitées acceptables tout en utilisant efficacement les ressources système.

Une approche de compromis latence-précision du traitement des données. La mise à l'échelle des microservices pour garantir une faible latence n'est pas toujours possible en raison des ressources limitées et hétérogènes des systèmes actuels. Par conséquent, il est nécessaire d'adopter des approches de gestion des données et de workflow. Les travaux existants tendent à traiter ces deux aspects indépendamment, traitent rarement l'ensemble du flux de travail de l'application et manquent de formulations générales des approches proposées. Cette thèse présente une nouvelle méthode de compromis latence-précision basée sur la combinaison d'un approche d'adaptation de la qualité des données et d'une approche de placement de workflow.

Une évaluation expérimentale dans le cadre du Deep Learning. Les applications de Deep Learning nécessitent que les décisions soient prises en temps réel tout en s'appuyant sur des ressources hétérogènes et limitées. Dans cette thèse nous avons

évalué sur *Grid'5000* la méthode de compromis proposée sur un cas d'utilisation d'application Deep Learning. Les résultats de l'évaluation ont montré un gain de latence de traitement des données allant jusqu'à 54,4% dans un scénario multi-utilisateurs avec une qualité de traitement supérieure à un seuil.

TABLE OF CONTENTS

Chapter 1

Introduction

Contents

1.1 Context

The exponential growth of digital data sources connected to the network empowers businesses, academia, and the quality of human life. For instance, the applications designed to prevent natural disasters such as early earthquake warning systems [1] analyze geoscience data produced by sensors in real-time to protect human lives. The generated digital data are characterized by several features such as their Variety, Volume, Velocity, and Value [2]. *Variety* comprises various types, resolutions, and data formats, including videos, audios, documents, and others. For example, in agricultural applications, the data collected can be, among others, images from cameras, drones, satellites, and sound data for locating livestock [3, 4]. Data *Velocity* relates to the data creation rate. For instance, after the outbreak of COVID-19, every aspect of life moved online. According to the 2021 DOMO report, every 1 minute, 240k photos are shared on Facebook, and users stream 694k hours of content on YouTube [5]. In addition, the *Value* of the data refers to the information they might hold.

Extracting the values from real-time data requires applications able to process data in a timely manner. Guaranteeing the real-time requirements is one of the main reasons

driving the evolution of infrastructures and applications designs [6, 7]. Traditionally, infrastructures are based on geographically centralized data centers [8]. Due to the volume of generated data and limited network capacity, traditional systems suffer from high latency. Therefore, there has been recent interest in moving resources from data centers located at the network's core to near data producers at the network's edge. The distributed design of current infrastructures triggered an evolution in the way applications are implemented and deployed. In particular, the ongoing race to anticipate the Cloud-Native mindset [9] created a growing emergence of the microservice paradigm. This paradigm [10] is an application design that consists of decomposing traditional large applications into sets of self-contained and loosely coupled services that interact over the network. The 2019 research report from the International Data Corporation (IDC) expected that, by 2022, around 90% of newly developed applications would feature the microservices architectures [11].

The current landscape of infrastructures designs, applications paradigms, and the growth of generated data added new challenges to the management of emerging applications. First, the microservice-based applications are dynamic, created by different entities, and have several microservices implementations. Each entity continuously publishes new microservices or removes existing ones based on their needs [12]. These microservices are not aware of each other's functionality [13] and mostly lack explicit identifiers. Furthermore, several microservices offer the same functionality but with different quality of service and accept specific data characteristics. For example, in a video analytics system, several implementations of the object detection service exist [14, 15]; each provides specific analysis performance, consumes different resources, and accepts particular data type and resolution. Consequently, at a specific time, what exists in a microservice-based environment is not guaranteed, and the selected implementations affect the critical performance of current applications. Second, the number of data sources is dynamic during the application runtime where new data producers can continuously join the system, and existing ones may leave [16]. For example, for data sources that correspond to mobile smartphones, users voluntarily start or stop providing their data for the application [17]. Each data producer generates particular data and aims to utilize available microservices developed to process their specific data. Due to the fluctuating load, the traffic that each microservice receives is not known in advance, which can cause an increase in their response time. Third, in real-time deployments, the available resources are geographically distributed, limited, and provide different computing, memory, storage, and bandwidth capacities. These resources may

range from embedded devices featuring limited capacities to large data centers [18]. Using these resources to deploy time-sensitive Big Data applications requires additional strategies to control the distribution of resources among heterogeneous data sources and microservices.

In this regard, management approaches that overcome these challenges are necessary to maintain the efficiency of time-sensitive applications. Several systems and tools are created to manage Big Data applications in distributed environments and on different system levels (such as resource, data, and application-level). However, their approaches are mostly considered goal-based approaches driven by the system needs to meet a certain quality of service requirements and offer specific functionality to the system's users. These approaches are usually based on explicit assumptions tailored to answer the system's objective. In the context of Big Data applications, adopting goal-driven approaches in real-world deployments is inefficient due to the extreme heterogeneity and uncertainty of emerging applications. Hence, it is crucial to design novel approaches with reasoning built on the characterization of current environments and generated data. Data-driven management approaches are fundamentally different from existing goal-driven ones, but they can complement each other to leverage available services and resources effectively.

This thesis designs a system that relies on data-driven approaches for managing Big Data applications in distributed environments. Among existing Big Data applications, this thesis put an emphasis on Deep Learning applications. The proposed system rethinks current goal-driven management strategies with a particular focus on the characteristics of the generated data, the application, and the infrastructure. To tackle the challenges of the current landscape presented above, we formulate the research questions of this thesis in Section 1.2.

1.2 Research Questions

With the technological advancements, our capacity to collect data has grown significantly [19]. However, the potential of using these data in making efficient management decisions for real-world deployments is not sufficiently addressed in current Big Data systems. Hence, this thesis aims to create a system providing data-driven management on multiple system levels for emerging time-sensitive Big Data applications. This goal was further decomposed into three sets of research questions that address the challenges of current microservice-based applications mentioned in Section 1.1. This section

presents the research questions in the order in which they were addressed in the thesis:

- **RQ1:** Can using data products help select, from even recently published unidentified microservices, the best microservice implementation for a data source?
- **RQ2:** How to minimize the impact of fluctuating and heterogeneous incoming load on the latency of microservice-based applications? How suitable is the adopted strategy in avoiding the misuse of system resources?
- **RQ3:** How to distribute a data processing workflow across a computing continuum with heterogeneous and limited resources at the edge of the network, the core, and along the data path? How the system should respond to an increase in data producers? How it affects the quality and the latency of processing?

This thesis targets these questions while focusing on collected data related to the application, the infrastructure, and the incoming data. The RQ1 focuses on addressing the challenge of dynamic microservices having multiple implementations created by different entities with no explicit identifiers. Addressing this challenge is seen as the entry point to effectively leverage available services, even newly published ones. The RQ2, on the other hand, targets the need for adapting the capacity of specific microservices according to the incoming load. The RQ3 aims to identify a mechanism that can reduce the end-to-end latency of Big Data applications (precisely Deep Learning applications) when processing data on limited and heterogeneous resources.

1.3 Structure of the Manuscript

The manuscript includes seven chapters in total. Each chapter provides the required background for the mentioned concepts and an overview of the related literature. This section presents the outline of the manuscript and the accepted papers.

1.3.1 Outline

The remainder of this thesis is organized as follows:

- Chapter 2 presents the context of my research and an overview of the contributions regarding data-driven management in distributed systems. The context includes a description of the current landscape of system designs and highlights their limitations in dealing with the requirements of emerging applications.

- Chapter 3 presents a data-driven resource management approach able to provide users the best microservices implementations matching their needs without relying on explicit identifiers. This chapter addresses the research question RQ1.

- Chapter 4 presents a service adaptation scheme designed to tackle the challenge of fluctuating discovery load. It shows the implementation details and evaluation results of the proposed system on a real testbed. This chapter addresses the research questions in RQ2.

- Chapter 5 introduces data and workflow management solutions to cope with the limited and heterogeneous resource capacity of current infrastructures. It manages a tradeoff between the critical latency requirement and the quality of processing. This chapter addresses the research questions in RQ3.

- Chapter 6 evaluates the impact of the approaches presented in Chapter 5 on the performance of a specific Deep Learning application use case. The evaluations are conducted on a real testbed and the results provide concrete answers to the research questions in RQ3.

- Chapter 7 summarizes our contributions and discusses the perspectives.

1.3.2 Accepted research publications

Papers in International Conferences

- **Zeina Houmani**, Daniel Balouek-Thomert, Eddy Caron, and Manish Parashar. "Enabling microservices management for Deep Learning applications across the Edge-Cloud Continuum." *In the 33rd IEEE International Symposium on Computer Architecture and High-Performance Computing (**SBAC-PAD'21**).* Belo Horizonte, Brazil, October 2021. (Rank B)

- **Zeina Houmani**, Daniel Balouek-Thomert, Eddy Caron, and Manish Parashar. "Enhancing microservices architectures using data-driven service discovery and QoS guarantees." *In the 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (**CCGrid'20**).* Melbourne, Australia, May 2020, pp. 290-299. (Rank A)

Papers in National Conferences

- **Zeina Houmani**. "Déploiement et validation d'une architecture microservices pour la découverte de services pilotée par les données". In the *Conférence d'informatique en Parallélisme, Architecture et Système (**COMPAS'19**)*. Anglet, France, juin 2019. (Conference without proceedings)

Chapter 2

Enabling Data-driven System Management

Contents

2.1 Introduction

With the development of technologies, wireless networks and devices, new big data applications are being introduced, such as Virtual Reality (VR) [20], autonomous vehicles [21], multimedia processing [22] and others. These applications have several requirements to guarantee their effectiveness and well-functioning. They often demand high computing and bandwidth capacity, as well as real-time processing. A high-quality VR application, for example, needs to send 240 frames per second with a total volume of compressed data larger than 1Gb per second [23].

Guaranteeing these requirements in current systems is challenging due to the escalation in the number of high-quality data sources (such as 4k and 360° cameras) and the increase in the heterogeneity of applications and infrastructures. The heterogeneity of infrastructures refers to resources with different computing and network capacities.

The heterogeneity of applications refers to service implementations providing the same functionality but with specific Quality of Service (QoS) guarantees and data inputs. Thus, there is a need to rethink current management approaches to take this heterogeneity into account while managing emerging applications. In this context, we present a data-driven system that manages applications on data, workflow, and resource levels.

This chapter is organized as follows: Section 2.2 presents a literature review of current Cloud system designs. Section 2.3 highlights the limitations of current systems in dealing with emerging applications. Afterward, in Section 2.4, we put forward a multi-level data-driven platform for managing current applications. Finally, Section 2.5 concludes the chapter.

2.2 Cloud System Design: Current Landscape

The exponential growth of the data generated and the continuous emergence of applications with Quality of Service (QoS) requirements are driving the evolution of how we build systems. This section describes the changes in the infrastructural designs of current systems (Section 2.2.1), their software architectures (Section 2.2.2), as well as the management of the interactions between software components (Section 2.2.3).

2.2.1 The Evolution of Cloud-based Infrastructural Designs

Cloud computing [6,7,24] is defined as a model for enabling convenient and on-demand access, via the internet, to configurable computing resources, including physical and virtual servers, data storage, applications, development tools, and others. The resources are deployed in clusters and managed by certified providers such as Amazon Web Services (AWS), Google Cloud Platform (GCP), and AlefEdge [25]. Furthermore, clients can access applications and data from a Cloud anywhere and anytime [26].

Traditionally, Cloud computing is based on geographically centralized data centers. Guaranteeing the requirement of emerging data-driven applications by solely using centralized computing was not feasible. This required an evolution of centralized Cloud designs to decentralized models. In this section, we present centralized and some decentralized Cloud computing designs.

2.2.1.1 Centralized Cloud computing

Traditional Cloud computing revolves around a pool of large, geographically centralized, and virtualized resources stored in remote data centers located far from data sources. These resources correspond to computing power, storage, platforms, and services granted to users over the internet for a fee. Centralized Cloud provides benefits for both users and businesses [27]. Demanding infrastructures from Cloud providers decrease the costs of purchasing, installing, configuring, and managing private infrastructures in the organizations. In addition, it increases productivity and infrastructure agility due to the ready-to-use tools, hardware, and services. On the other hand, users will have access from anywhere and anytime to the platform with a price based on the pay-as-you-go cost model. Furthermore, Cloud computing provides elasticity of resources as they can adapt to users' demands, making services more reliable.

Despite bringing numerous advantages, centralized Cloud computing poses some serious limitations after the growth of communication technologies, devices, and real-time applications [24]. With the humongous increase in the number of data sources and the advent of the Internet of Things (IoT) with high-resolution data (such as 4k images and 360° videos), the load increased substantially. At the same time, data sources are usually connected to Cloud data centers via wireless networks, which forward the generated data to the central servers. These networks have a limited bandwidth, which means a finite limit to the amount of data that can be transferred across the network to the centralized Cloud. Even with the development of network technologies (such as next-generation 5G wireless technology [28]), the improvement of network capacity to accommodate a large volume of data remains limited, and the cost of integrating these technologies can be high. Hence, sending a large volume of data from devices to the Cloud will certainly create serious delay bottlenecks and unstable network connectivity, which can be disastrous for time-critical applications such as autonomous vehicles and health-related applications (e.g., [29], [30] and [31]). In addition, other limitations exist for the centralized Cloud model, such as the high energy consumption of large data centers [32], the single-point of failure of centralized processing, less mobility support, and the security aspects of exchanging sensitive or personal data between devices and data centers [33]. Thus, an evolution of centralized Cloud infrastructural design to a decentralized model is inevitable to deal with the QoS requirements of emerging applications and distribute computing load over geographically diverse resources.

2.2.1.2 Decentralized Cloud computing

The basic principle of a decentralized Cloud is locating resources closer to data producers, and processing collected data at the edge of the network. Compared to centralized computing, these resources are distributed, heterogeneous in terms of computing and storage capacity, and have lower processing and storage capabilities than Cloud data centers [6]. The proximity of computing resources to data can deliver substantial benefits to businesses and users. Companies can integrate technologies near data sources to collect timely insights from data and, if applicable, take early actions. In addition, they can filter unnecessary or redundant information within the raw data to reduce bandwidth usage and storage cost, which also promotes energy savings and both quality of service and experience improvements. Another benefit of decentralized computing is enabling geospatial analysis for location-aware applications [34] and improving the security of exchanged data by applying security policies at the edge of the network [35]. Users, on the other hand, benefit from faster and more reliable services with better user experiences.

Different types of decentralized models for Cloud computing exist [6]. In this section, we focus on Edge computing, Fog computing, and Edge-to-Cloud computing.

Edge computing. It is a network layer providing local computing capabilities by taking advantage of nearby Edge devices. Edge devices correspond to physical hardware with memory, storage, and computing capacity. These devices are located at or near the physical location of data sources. Edge computing exploits the computing and storage capabilities of Edge devices which limit the resource waste [24]. In addition, it allows executing data processing operations at the edge of the network, which can reduce the volume of data and identify trends. However, even with the growth of these devices and the increase in their computing capacity, the available local resources continue to be constrained, making them unable to cope with the full computing requirements of these applications. Consequently, Edge devices frequently rely on the powerful resources at the Cloud data centers to execute the more complex processing tasks [36]. In the literature, it is referred to as *task offloading* [37]. Relying on Cloud resources requires transferring a large volume of data back to the Cloud. This re-exposes the inherent shortcomings of the centralized Cloud.

Fog computing. It is an approach for expanding traditional Cloud computing capabilities at the edge of the network by providing a large amount of storage, communica-

tion, and computing resources [7]. It introduces an intermediate physical infrastructure layer between data producers and Cloud data centers. This layer is tightly linked to the existence of a Cloud and cannot operate in a standalone mode [8]. Fog computing supports features such as mobility and location awareness, low latency, and virtualization. Fog nodes correspond to any device with computing, storage, and network connectivity. The computing capacity of Fog nodes is heterogeneous [7, 38]. Examples of Fog devices include switches, routers, embedded servers, and Cloudlets. The latter is a small-scale Cloud data center located near data producers [39]. It benefits from Virtual Machines (VM), which permit the available resources to scale based on the load. Unlike other Fog nodes, it can operate disconnectedly from Cloud services. Fog nodes usually have higher computing capacity than Edge devices. This increases the possibility of carrying out more intensive data processing operations near the edge of the network, reducing the need to transfer large data volume to the Cloud.

Edge-to-Cloud computing. It is a computing paradigm where computational, storage, and network resources are integrated into a multi-level hierarchy along the data path between the Edge and the network's core. Hence, unlike previous models, this computing paradigm does not focus on a particular dimension (Edge, Fog, or Cloud). Still, it allows leveraging the computing on Edge devices, Fog nodes (referred to as In-transit nodes), and Cloud data centers. In the literature, this paradigm is referred to as a "*Computing Continuum*" [18, 36, 38]. Each infrastructure level within the continuum consists of a set of nodes offering a specific storage and computing capacity. Each level is connected to its peer infrastructure level via a network link with a particular bandwidth capacity. As we move in the hierarchy closer to data producers, the capacity of the nodes and bandwidth link decrease.

Leveraging the computing continuum to support time-critical and/or location-aware applications was recently explored in [40] with an Earthquake Early Warning use case. It consists of mapping the application workflow to resources along the data path to get early insights and reduce the volume of data to be transferred across the continuum. As Fog nodes have a higher computing capacity than Edge devices, they perform preprocessing operations on the collected data and take actions if applicable. If heavy processing operations are needed, the data are sent toward Cloud data centers to leverage the high computing capacities of deployed servers. Other use cases exist, such as in vehicular industry [41] and healthcare [42].

2.2.2 Shift from Monolithic to Microservices Paradigm

The evolution of infrastructure designs and the rising tendency of organizations to adopt Cloud-based systems forced an evolution in software architectural styles to leverage current infrastructures. The ongoing trend of Cloud-Native computing [43] focuses on building applications that exploit the flexibility, scalability, and resilience of Cloud computing. Traditional software designs failed to fit the current trend as they were not designed for Cloud-based systems or due to their anti-patterns in practice. This section presents the evolution of architectural designs from the traditional monolithic pattern to the Cloud-Native microservices paradigm.

2.2.2.1 Monolithic design and drawbacks

The traditional application architecture is *Monolithic*. It consists of creating an application as a single binary artifact and then decomposing it internally (with code refactoring) into horizontal layers [44]. The layered design consists of 3 main tiers, each performing a specific role within the application [45]: Presentation layer, Business logic layer, and Data access layer. The data access layer is the first layer to be built in practice. It enforces access rules to data stored in the application's data store. The business logic layer is the middle layer of the design. It contains the functional logic driving the application's core services. They are usually implemented using frameworks that provide libraries and tools to build applications such as Microsoft .NET Framework and Java 2 Platform, Enterprise Edition (J2EE) framework [46]. The presentation layer is the front-end user interface that the client uses to interact with the application.

As monoliths continue to rule the environment, several issues were detected:

1. Strong coupling of code within and between the layers. This makes it hard to add changes to the applications and causes code issues that require considerable time to fix, test and deploy;

2. Reliability issue. A bug in any functionality can potentially bring down the entire application;

3. Strong coupling between the development teams. Each code change must be carefully coordinated among teams due to the tight coupling of functionalities. This hinders the application agility and slows down the development process;

4. The application can only support one programming language. This reduces the flexibility of the development team in using specific technologies or libraries, which will affect their productivity;

5. "Scale-everything" issue. Even if only a single functionality requires adaptation to the load, engineers must scale the entire artifact multiple times. This leads to a misuse of the system resources.

The list of issues of this application design goes on. These problems have led to an architectural refactoring of applications [47] and the emergence of Service-Oriented Architectures (SOA).

2.2.2.2 Service-Oriented Architecture (SOA) and anti-patterns

The Service-Oriented Architecture (SOA) consists of decomposing large applications into sets of "*services*" [48–50]. A service is a self-contained module that embodies the code and the data required to perform certain operations (such as business functions, authentication, etc.). The service boundaries are based on business capabilities without a specific size.

SOA introduces a service provider, a service consumer, and a service definition [51]. A service provider is an entity that provides services to another party. A service consumer is any type of software component that demands the service. To access it, the consumer sends a request according to the provider's invocation interface. This standard interface is defined in the service definition of the provider. A service definition specifies, among others, one or more operations that comprise the request format and, optionally, the response messages [52–54]. The exchange of requests and response messages between service consumers and providers is done via an Enterprise Service Bus ESB [55]. It is a centralized software component that performs a set of operations such as message routing, conversion of communication protocols, conversion of message formats, and monitoring. It contains a component called service registry that stocks the service definition of all the available service providers in the system and makes them available to the ESB and to service consumers [56]

The SOA design provides software engineers with several benefits [57]:

1. Simultaneous development. Every service can be developed, tested, and deployed by a separate team which significantly reduce development time;

2. Using different languages and technologies. Developers can choose the language that best suits the services and their purpose;

3. Reusability of services. As services are self-contained and use standard interfaces, they can be incorporated into new applications without wasting time and effort on duplicating existing services.

In addition, for organizations, SOA affects positively the businesses [57]:

1. Faster time to market. This is due to the faster development and upgrading of functionalities in comparison to monolithic design;

2. Scalability. Unlike a monolithic application, there is no need to scale the entire application to deal with a load of specific functionalities. Services can be scaled independently based on the load, which leads to better use of resources;

3. Increase the reliability of the application. For example, if one service is down, only the operations provided by this service will be affected and not the entire application.

After years of using SOA in practice, some anti-patterns were identified, such as: 1) many SOA applications implements in the ESB a set of operations for communication management which turn the Enterprise Bus into a single monolithic artifact [56]. As the service providers and consumers use this component to interact, once it is down, the entire application will be affected; 2) there is no guidance about the service granularity in SOA. So, when a service integrates multiple operations, it turns to a monolithic application; 3) sharing databases between services. Managing the data of multiple services in the same database couple development teams, complexify database replication, force the use of the same database technologies, and cause security risks.

Traditional SOA did not fit the Cloud-Native approach. So, software engineers created from the SOA a new architectural design to deal with these anti-patterns and increase the agility, resiliency, and scalability of service-based applications. This design is the *microservices* paradigm.

2.2.2.3 MicroServices-based Architecture MSA

The microservice architectural style was officially introduced in 2012 as "*an approach to developing a single application as a suite of small services, each running in its*

own process and communicating with lightweight mechanisms, often an HTTP resource API" [10]. In more detail, the characteristics of microservices are as follow:

- *Fine-grained services.* Each service is responsible for a well-defined functionality [58–61]. In practice, deciding the size of the functionality in the microservices paradigm is not an easy task, and it depends on the use cases of the system [62–65]. Making microservices not granular enough will turn them into monolithic, and making them too granular might increase the application's latency due to extra remote calls over the network [44].

- *Database-per-microservice.* Each service is self-contained and controls its own private data store [66–68]. This increases the scalability of services as the database can be easily scaled in a database cluster when the microservice needs to scale with the load. In addition, it helps to enhance the security of the data and the agility of the development process [69]. In the literature, there are two other mainstream approaches for using database systems in microservice architectures. However, Database-per-microservice is the most commonly used [70].

- *Lightweight communication via APIs.* Service providers and consumers may communicate directly with each other [58] or through a proxy (referred to as API Gateway) [56]. The communication is either synchronously via HTTP requests, asynchronously via messages, or mixed by using a synchronous request with an asynchronous response [71].

- *Lightweight deployment with containers.* A container is a form of system virtualization. It consists of packaging the code and all the dependencies in a single unit of software [72]. Containers are quickly evolving to become a standard implementation for microservices even with their non-negligible impact on the performance [73–75]. This is because containers are lightweight, provide fast start-up times, and have a low overhead which increases the agility of the microservice-based applications [75].

These features create self-contained and loosely coupled services that increase the applications' scalability, flexibility, resiliency, and availability. In addition, it allows continuous development and delivery with more independent development teams [76]. This paradigm is adopted by several well-known platforms such as Netflix [77], Amazon [78] and LinkedIn [79]. Also, it has been widely used in Internet of Things IoT systems [80–84].

Several concepts and techniques in the area of microservices were borrowed from SOA, such as service definition, service registry, service discovery (finding where the service is located), heterogeneity of technology stack, scalability, the organization of development teams, etc. So, as there are some differences in design and architecture, there is also an overlap that makes microservices a design that emerged from the years of experience with SOA and the real-world use cases and not an entirely new paradigm [60].

With Cloud-Native computing, microservices-based applications are pushed to a very large scale. As the scale rises, the effort required to manage the massive incoming traffic increases. Cloud-Native *Service Mesh* is a technology that emerged to manage external and internal traffic. Details about Service Mesh are presented in Section 2.2.3

2.2.3 Service Mesh: concept, features and projects

Splitting applications into microservices and supporting the architectures with technologies such as docker [85] and Kubernetes [86] improved the efficiency and productivity of the development and deployment of the applications. However, the operational complexity during service runtime has not been mitigated.

The first attempt for managing incoming traffic was to add libraries to deployed services. These libraries aim to control the security between services, latency, monitoring, and load balancing. Examples of these libraries are Finagle of Twitter [87] and Hystrix of Netflix [88]. However, with the microservices paradigm, the integration of libraries into the technological architecture stack became complex and time-consuming: ① microservices-based applications have a large number of microservices, so adding a library to each service is complex; ② managing traffic at the application level do not fit the "single functionality" constraint of the microservices paradigm; ③ microservices are developed in various programming languages and implementing the libraries in all supported languages is inefficient; ④ microservices used in the same application can belong to different entities and might not be using compatible libraries. These challenges led to the emergence of *Service Mesh*.

Definition. Service Mesh is a dedicated infrastructure layer that controls communication between services and decouples the inter-service communication management from the application layer. This additional layer forms a distributed and scalable network of interconnected proxies within the microservices network. In practice, communications between microservices pass through the proxy servers deployed alongside the services without the need for any code modification.

Service Mesh is composed of two key architectural components, a data plane, and a control plane. The *data plane* includes a set of intelligent proxies. Instead of calling services directly over the network, these proxies intercept the network packet and encapsulate the complexities of the service-to-service exchange. This component is managed as a whole by a control plane. The *control plane* enforces the authentication policies specified by the system admins, collects metrics, and configures the proxies for traffic routing. These two components communicate via a predefined API. The architectural design of a Service Mesh is presented in Figure 2.1.

Fundamental Features. The main responsibilities of Service Mesh include, among others, service discovery, failure recovery, routing, load balancing, authentication/authorization, and monitoring.

- Traffic routing: through configuring routing rules, Service Mesh can dynamically hide and expose specific services or create testing and versioning deployments.
- Failure recovery: system's reliability is handled by setting timeouts, retries, circuit breaking, and health checks. Circuit breaking rules back off the requests of overloaded services and health checks verify whether services are still available.
- Load balancing: it provides the capability of routing the traffic across the network using modern routing mechanisms.

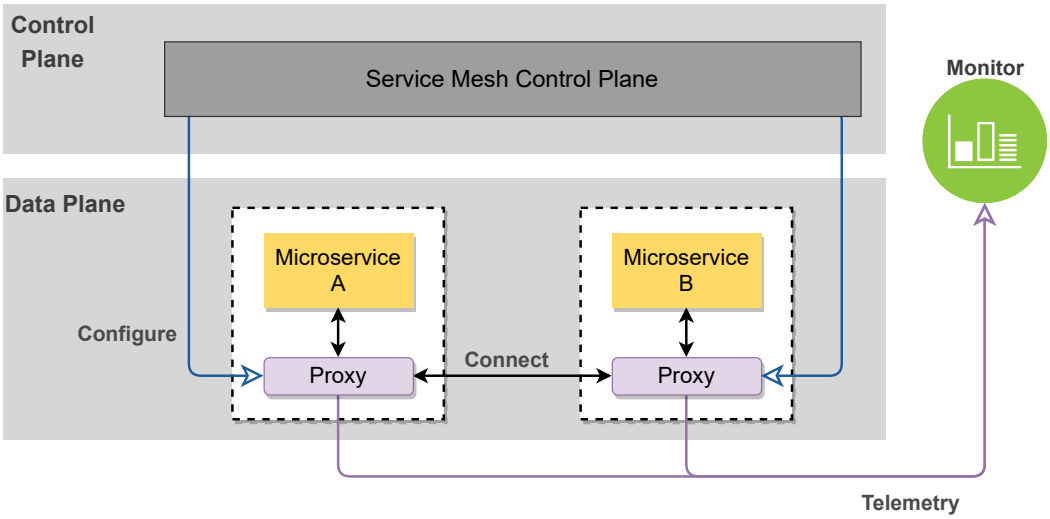


Figure 2.1 – The architecture of a Service Mesh containing a control plane and a data plane [89].

- Authentication and Authorization: Service Mesh has the ability to ensure the security of traffic by forcing a service-to-service and an end-user authentication and policies from the control plane.
- Monitoring: it provides a set of network performance metrics like latency, bandwidth, and uptime for every level of the architecture stack. In addition, it offers detailed logging for events and distributed tracing of requests.

The features provided are not fundamentally new. Instead, the Service Mesh is ultimately a shift in *where* the traffic management is handled, not *what* can be done.

Service Mesh projects. In 2016, the first Service Mesh project was released (Linkerd [90]). Since then, dozens of new open-source Service Mesh platforms have been created, including Istio [91], Consul Connect [92], and AWS App Mesh [93]. In this work, we chose the Istio open-source project. Even though it is considered the most complex Service Mesh to install, configure and operate, it quickly becomes the standard for Service Mesh [94]. This is due to its maturity and extensible features in comparison to the other mainstream implementations of Service Mesh.

Istio is an ongoing collaboration between IBM, Google and Lyft. Its data plane consists of a set of Envoy proxies that mediate all inbound and outbound traffic for all services in the Service Mesh. *Envoy* [95] is a high-performance distributed proxy able to support small applications as well as large-scale microservices architectures. If needed, developers can use other proxies in the data plane.

Istio's control plane is itself a modern Cloud-Native application. Its core components are all written and deployed as separate microservices. ❶ *Pilot* is a component responsible for traffic management and resiliency; ❷ *Mixer* enforces access control and usage policies across the Service Mesh and collects telemetry data from the Envoy proxy and other services; ❸ *Citadel* enables strong service-to-service and end-user authentication; ❹ *Galley* is a component mainly responsible for the validation of configuration data and translating them into the common format of Istio.

2.3 Shortcomings of Current Cloud Systems

The emerging applications have the following characteristics: 1) The distributed data sources provide a large volume of data concurrently with heterogeneous data characteristics (type, format, resolution, rate, etc.). Furthermore, these data sources are dynamic where new data producers can join the system to take advantage of the provided

services; 2) the applications' workflows consist of a set of functionalities demanding different computing and bandwidth requirements. Therefore, the intensity of resource usage within a workflow can vary from low to high intensive based on the functionality provided, adopted technologies, and accepted input data quality; 3) the microservice paradigm has become the leading design for Cloud-Native systems [96]. However, despite its benefits, this paradigm adds challenging characteristics to current applications: i) *tasks are dynamic.* New functionalities or new versions of existing functionalities can be added or removed by the application developers or due to the underlying environment. As a consequence, the available functionalities in emerging applications are not constant. ii) *tasks are heterogeneous.* Besides the heterogeneity in terms of programming languages and technologies used, microservice-based tasks have multiple instances providing the same functionalities but offer particular QoS and accept data of specific characteristics. Thus, the selection of instances during runtime can't happen randomly to guarantee the requirements of users. iii) *tasks are highly decoupled.* The workflow tasks are developed, deployed, and managed by different teams within the same organization or from different entities [13]. As a result, the microservices composing the application workflow are not designed to work together automatically.

These characteristics highlight the limitations of current system designs in guaranteeing the real-time requirements of emerging applications and maintaining proper functioning of their workflows:

1. Edge-only and Fog-based systems provide limited support for emerging applications as they cannot fulfill the computing needs of intensive tasks within the application's workflow. Furthermore, when relying on the Cloud to offload tasks to powerful resources, the network latency will prevent processing data in real-time, which is in some application use cases can be critical [97]. The Edge-to-Cloud continuum is a promising design for emerging applications. However, it did not establish concrete maturity yet [40].

2. Systems managing microservice-based workflows do not take into account the extreme heterogeneity, dynamicity, and decoupling of this paradigm in real-life deployments. For example, in production, two microservices are unaware of each other's functionality and have separate lifecycles [13].

3. Current Cloud-Native landscape provides a set of scheduling tools such as Kubernetes and Docker Swarm to deploy microservice-based applications on available

resources [98]. These centralized tools provide scheduling strategies within a centralized multi-node computing environment which does not match the current decentralized environment for emerging applications. The scheduling approaches proposed in the literature make goal-driven scheduling decisions motivated by the system objective of meeting particular QoS requirements with limited emphasis on characterizing current environments [99].

4. Several systems assume that the computing and network capacity is infinite or ignores the possibility of multiple data sources joining the system with high-resolution data [100]. In production, keeping these assumptions will negatively affect the processing quality for all users.

5. Existing system designs target specific application use cases such as video analytics applications [15, 101] and Augmented Reality AR [102]. The absence of general formulations of emerging applications and their needs makes it difficult for developers to integrate these approaches into other application use cases.

6. Management strategies rarely handle the entire workflow of emerging applications. Instead, they focus on specific services within the workflow [15]. In infrastructures with limited and heterogeneous resources, not taking into account the entire workflow is inefficient because they share the available computing and network resources, affecting the system performance.

These shortcomings of current systems limit the ability to support emerging applications and their QoS requirements. Exploiting the Edge-to-Cloud continuum capabilities is hindered by the lack of understanding of application requirements in real-life deployments. Propelled by the need to overcome these limitations, we propose a novel system that rethinks current management strategies at multiple system levels with an emphasis on the characteristics of applications and infrastructures, particularly their extreme heterogeneity. More details about this ecosystem are presented in Section 2.4.

2.4 Data-Driven Ecosystem: Definition and Design

2.4.1 Data-driven decision-making

In the context of this thesis, data-driven decision-making refers to making management decisions based on the examination of collected data. Creating a data-driven manage-

ment approach consists of the following four steps: ① specify the management objective that needs to be accomplished and the reason it is required; ② among generated data, determine the metrics essential to accomplish the desired goal. ③ identify the components that can provide the desired data; ④ analyze collected data and turn analysis results into management actions. The logic of a data-driven decision-making approach when receiving data from data sources is illustrated in Figure 2.2.

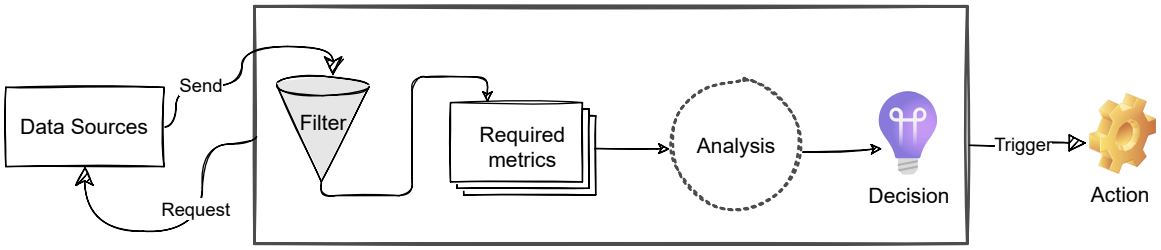


Figure 2.2 – The logic of a data-driven decision-making approach when receiving data from data sources.

In the literature, several data-driven approaches are designed to manage specific system components. The management decisions are based on different data metrics. Mahmud *et al.* [103] propose a context-aware application placement policy in Fog computing environments. The scheduling approach considers the characteristics of data producers (data size and data rate) due to their direct impact on Fog node functionalities and application characteristics. Furthermore, they jointly consider during scheduling the heterogeneous computation and networking capacity of Fog nodes and the QoS requirements (service delivery deadline) of applications.

Renart *et al.* [12] present a framework that enables applications to specify data-driven, location-aware, and resource-aware processing of data streams. It provides a content-driven programming model that enables users to specify functional rules triggered by data content and determine which topologies are executed and where [104]. In addition, authors adopt a data-driven discovery process to associate data producers with consumers [16]. The discovery is based on a matching mechanism between the data-input-related and location-related metrics provided by the producers and consumers. A match between the profiles triggers a predefined management action.

Wang *et al.* [15] propose an online algorithm to adapt the configuration of incoming load (frame sampling rate and frame resolution) and the allocation of bandwidth for Edge-based real-time video analytics. Data and resource management decisions are driven by data related to video content and network capacity. Due to the dynamicity of

incoming load and limited resource capacity, the configuration is continuously optimized while taking into account the energy consumption, system latency, and analytics quality.

2.4.2 System design: an overview

The proposed ecosystem adopts data-driven decision-making approaches on multiple system levels: workflow, resources, and data. This section briefly presents each management layer and its components. For each component, we present the steps mentioned in Section 2.4.1 to create data-driven decision-making. Figure 2.3 presents the full overview of the proposed system.

Resource management layer. This layer is dedicated to discovering deployed microservices and adapting them to the incoming load. The challenge of this layer is to discover microservices based on the characteristics of clients' data with regard to guaranteed quality of service in terms of response time. This layer consists of two data-driven management components: microservice discovery and microservice adaptation. Details about the discovery approach are presented in Chapter 3 and the adaptation approach in Chapter 4.

1. *Microservice discovery:*

    - **Goal:** this component is responsible for the discovery of available functionalities and the different microservices providing them.
    - **Data metrics:** the discovery is based on matching the characteristics of incoming data, such as data type, format, and resolution, with those of the deployed services.
    - **Data source:** the workload-related data are received from the users, and the microservices-related data is extracted from the microservices descriptions provided by the application developers.
    - **Action:** the component decides what functionalities and microservices suit the needs of the users.

2. *Microservice adaptation:*

    - **Goal:** this component continuously adapts the capacity of microservices to prevent the performance degradation or the misuse of resources.

- **Data metrics:** the adaptation decision is based on resource- and workload-related metrics, such as free resources, number of active users, and data type.
- **Data source:** the resource-related data collected from the underlying infrastructure and the workload characteristics collected from the users.
- **Action:** the analysis of the collected data might trigger an increase or decrease in the number of microservices following the variation of the load.

Workflow management Layer. This layer represents the system level the developers use to interact with the ecosystem. It is dedicated to scheduling submitted microservice-based workflows on heterogeneous system resources. The main challenge of this layer is mapping the tasks to the available resources with regard to the heterogeneity of both workflows and infrastructure. This layer consists of two management components responsible for making data-driven decisions: tasks categorization and workflow scheduling. Details about the approaches of this layer are presented in Chapter 5 and a demonstration of a specific use case is presented in Chapter 6.

1. *Tasks categorization:*
   - **Goal:** this component aims to group submitted tasks having similar resource usage characteristics into multiple categories. Each category has a scheduling policy and priority.
   - **Data metrics:** the categorization decision is based on the characteristics of each submitted task, such as the given functionality and accepted input data.
   - **Data source:** the required metrics are given by the developers of the application to be deployed in the system.
   - **Action:** after analyzing the collected metrics, the component tags the task with the selected category.

2. *Workflow scheduling:*
   - **Goal:** this component is responsible for distributing the tasks in the application's workflow on system resources.

- **Data metrics:** the placement decision is based on the category labels and characteristics of available resources such as CPU, RAM, storage.
- **Data source:** the data metrics are collected from the categorization component and the underlying infrastructure.
- **Action:** after analyzing the collected metrics, the component allocates resources and map the tasks to them.

Data management Layer. This layer is responsible for selecting appropriate data quality for each incoming load providing a system performance that meets the application requirement. The challenge of this layer is to optimize the trade-off between the latency of processing and its quality during runtime. This layer consists of two components: Performance estimation and data quality adaptation. Details about the approaches of this layer are presented in Chapter 5 and validated with a specific use case in Chapter 6.

1. *Performance models:*

   - **Goal:** this component formulate the general analytical models of the performance of emerging applications.
   - **Data metrics:** the models depend on resource-, workload- and workflow-related data, such as bandwidth, data quality, and throughput.
   - **Data source:** the required data are collected from the underlying infrastructure and system components.
   - **Action:** the collected metrics help this component estimate the performance of the application with the current system state.

2. *Data quality adaptation:*

   - **Goal:** this component decides whether the system can handle the original data qualities of all data sources or a quality reduction is required. It manages trade-offs to optimize performance.
   - **Data metrics:** the adaptation decision is based on the estimations of performance models and the supported data qualities.
   - **Data source:** the required data are collected from the performance models and system components.

- **Action:** this component selects the optimal data qualities for existing data sources and triggers the adaptation.

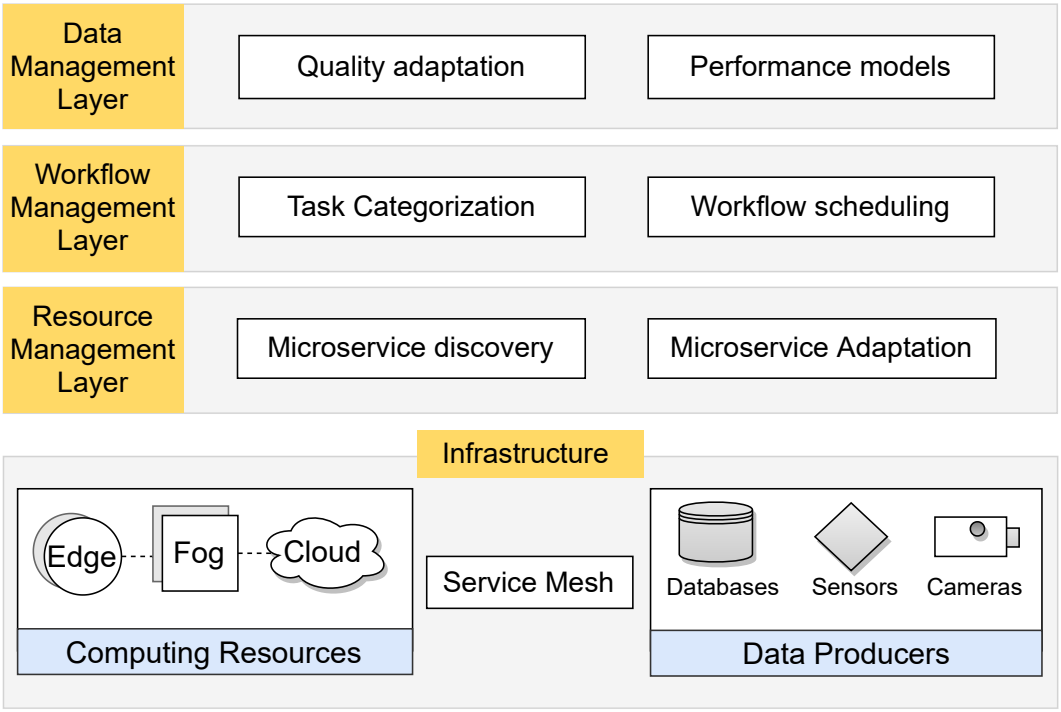


Figure 2.3 – A global overview of the ecosystem design consisting of three management layers, each of them with its respective components. The infrastructure has heterogeneous computing resources and data producers. In addition, it has Service Mesh for managing the submitted microservice-based applications.

2.5 Conclusion

The characteristics of emerging microservice-based applications highlight the shortcomings of current system designs in guaranteeing real-time data processing. The simultaneous processing of incoming workloads on limited, distributed, and heterogeneous resources requires the development of new management approaches driven by the characteristics of user-generated data and resources rather than on the general goal of the system. This chapter presents a literature review of the evolution of traditional designs and their limitations in managing emerging applications. In addition, it presents an overview of a system dedicated to managing emerging applications at different levels using data-driven management approaches.

Chapter 3 elaborates the data-driven microservices discovery of the resource management layer.

Chapter 3

Data-driven Service Discovery approach

Contents

3.1 Introduction

Producers and consumers of data in current computing systems are decoupled. Unlike traditional applications, the microservice paradigm supports Conway's law [105] which means aligning microservices ownership to the structure of teams [58]. Thus, implementing and managing microservices by autonomous teams from the same or different organizations creates applications with several functionalities not designed to work together. Several implementations of these functionalities usually exist, creating applications with heterogeneous microservices. These microservices use different technologies, data formats, data resolutions and expect particular Quality of Service (QoS).

Building high cohesion in these complex systems relies on managing decoupled microservices to work together without the need for redevelopment. Service discovery mechanisms help achieve that. However, in current practice, service discovery implementations are goal-based, designed to fulfill the general system goal depending on the client's required functionalities. They often let the client discover the location of a provider for the requested service using its identifier. Using goal-based service discovery approaches built on services' identifiers in current highly decoupled systems with heterogeneous microservices implementations is inefficient; Current mechanisms prevent the discovery of newly created microservices published by the different teams and those designed without explicit identifiers. In addition, they prevent the use of microservices' implementations that fit the client's data and QoS needs. Hence, there is a need to rethink traditional discovery approaches to overcome these challenges.

In this context, this work addresses the challenges by integrating information related to the client's data and QoS needs into the discovery process. As a result, this data-driven service discovery approach helps service consumers and providers. On the service consumer side, this approach allows the discovery of services according to data products. On the service provider side, it allows for the integration of third-party services with context-aware features. Our approach is built on a data-centric microservice description and a Peer-to-Peer (P2P) data-driven architecture to ensure the system's resiliency and cover wide geographical areas.

This chapter is organized as follows: Section 3.2 presents a literature review of service description models, context-aware discovery mechanisms, and architectural designs for discovery approaches. Section 3.3 describes our data-centric microservice description model. The architectural design of the proposed approach is presented in Section 3.5. Section 3.6 presents an illustrative example of the discovery approach. Finally, Section 4.7 concludes the chapter.

3.2 Literature Review: Service Discovery

Service discovery is the process of identifying then locating a service provider according to the client's needs specified in the discovery request. It provides the ability to search for desired services while reducing the need for services configuration. Service discovery mechanisms have been widely investigated in the literature of web services and the Internet of Things (IoT). This section provides a comprehensive literature review of some concepts in service discovery and networking techniques used in this work.

3.2.1 Service description models

Service providers prepare descriptions of their services before they are published. A service description model describes deployed service with a set of keywords and syntax following a specific description language and format. These descriptions allow consumers to discover available services and invoke them without the necessity of knowing how they are implemented. Existing approaches to formally describe web services for discovery are mainly based on the Ontology Web Language OWL/RDF [106], the Extensible Markup Language XML, or the JavaScript Object Notation JSON/YAML.

Description languages depend on the type of web services. Two main types of services exist, SOAP-based and REST-based web services. SOAP services represent the traditional web services that use the SOAP messaging protocol to share data [107]. SOAP is a lightweight protocol for exchanging information over HTTP. Regardless of the service type, service description languages can be categorized into syntactic-based or semantic-based service descriptions. Syntactic descriptions have a textual structure and rely on markup languages. Whereas semantic descriptions explore the meaning of functionalities and attributes using ontological approaches. An ontology differs from a markup-based schema in that it is a knowledge representation, not a message format.

Syntactic-based descriptions. SOAP web services are described using the standard Web Service Description Language WSDL [52]. It is an XML-based machine-readable language that describes the functional characteristics of a web service with a focus on its communications aspect. It presents the interfaces, binding protocols, the operations with their inputs and outputs, and the network endpoint addresses (URIs) at which these operations can be invoked. In the literature of web services, many authors worked on WSDL and service discovery. Paliwal *et al.* [108], for example, proposed service categorization and selection strategies that depend on the WSDL service description files. They extract from the WSDL descriptions the characteristics of services such as input and output of supported operations to associate services to concepts. Chen *et al.* proposed in [109] a web service clustering approach that depends on a set of features extracted from the WSDL service description files such as port, type, service name, message, and content.

For our system, we spot two major problems with the use of WSDL. First, microservices are considered loosely coupled services that operate independently without being coupled to server endpoints. As WSDL description language describes endpoints, it can't fit the RESTful design of microservices. Secondly, in practice, the WSDL de-

scription automatically generates a source code which is then compiled into a service. If the description changes, the service no longer works. Due to that, any change in the service's WSDL description file requires a recompilation of the code. This makes WSDL not flexible enough to suit dynamic environments.

Unlike SOAP services, RESTful services lack a widely accepted service description. Each service provider uses its own service description template with a set of attributes that helps fulfill its system's objectives. RESTful web services support XML and JSON formats for service descriptions. The Web Application Description Language (WADL) [53] is a syntactic XML-based description for RESTful services. It describes services in terms of resources, URI patterns, supported media types, and HTTP methods. Several metadata formats for describing REST APIs penetrated the industry market with rising popularity, such as Swagger [110], RESTful API Modeling Language (RAML) [111], and API Blueprint [112]. These languages aim to provide JSON/YAML human-readable and machine-processable documentation for RESTful web services which are not dependent on specific programming languages. In [92, 113], developers created their own JSON-based service description models. Some of the information can be removed, but no additional properties can be added. For our system, the common issue with all these service description languages is that they are considered operation-centric and do not fully support data-related attributes.

Semantic-based descriptions. In the literature, many efforts exist to add semantics to the discovery of SOAP and RESTful services. The key to semantic discovery is having semantics in the descriptions of the services [114]. In [115–117], authors adopt an OWL-based description model to describe available SOAP services semantically. A service description contains a *ServiceProfile*, *ServiceModel*, and a *ServiceGrounding*. A *ServiceProfile* describes a service in terms of its data inputs, outputs, preconditions that should be satisfied before the service is executed, and effects after the execution. A *ServiceModel* describes the set of functionalities/operations provided by a service. A *ServiceGrounding* describes how a service is invoked, including the protocol, message format, and port number. In [115], Gomes *et al.* add to the description models the geographic location metrics such as latitude, longitude, and altitude to locate service providers. Rohallah *et al.* extended the service descriptions in [116] with QoS profiles to describe QoS parameters such as Response Time and Execution Price.

OWL-based service descriptions provide a rich description of functionalities with possible extensions to add new attributes and facilitate the discovery process. However, they are considered heavyweight with a bulky format. These descriptions do not

suit systems with constrained resources and network capacities [118]. In addition, expressing service descriptions in semantic description languages requires clients to have knowledge of Semantic Web Services and description details which can make their usage difficult [119].

Several semantic descriptions exist for RESTful services such as hRESTS [54] and RESTdoc [120]. hRESTS (HTML for RESTful Services) consists of inserting HTML tags within the HTML documentation of RESTful web services to enable the annotation of operations, inputs and outputs, HTTP methods, and labels. It has a set of extensions to add an extra semantic annotation for RESTful service descriptions such as MicroWSMO [54] and SA-REST [121]. RESTdoc consists as well of inserting annotations in the HTML documentation of restful services. Moreover, it provides adapters to automatically transform the data in service descriptions to RDFs (Resource Description Framework). RDF is a standard for interlinking between services. The major limitation of these description languages is that the HTML documentation can be unavailable or hard to get in most public web services.

In this thesis, we create our own description for RESTful services. It is a syntactic XML-based description that is not operation-centric as existing languages but is data-centric with QoS metrics. Details about this description are presented in Section 3.3.

3.2.2 Context-aware service discovery

Service discovery mechanisms usually involve three steps: ① retrieve the functional and non-functional information about available services in the system; ② execute a matchmaking strategy between the information in the descriptions of the services and the client's request; ③ finally, select the service that matches the request and return it to the client. The first step depends on the discovery scope of the designed mechanism. Service discovery scope refers to the range method operates. It can be of two types, local and remote discovery. Local discovery allows users to interact with their immediate environment. However, remote discovery provides remote access to distant devices and services. The information used in the matchmaking and selection processes helps improve the quality of service selection. Based on the type of this information, service discovery approaches can be classified into several categories, such as context-based, protocol-based, semantic-based, and QoS-based approaches [122]. In this work, we focus on context-based service discovery approaches. This category of discovery uses context information to select the appropriate service for the clients. Context is formally defined

as "any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves" [123]. For web services discovery, the context is any information able to support the client's requests. Contextual information can be divided into three main categories:

- Environment context: it includes information about the physical environment such as location, temperature, and time.
- Service context: it includes information about the type of the service, data input, data output, network, and resources (such as battery and memory level).
- Social context: it is user-related information such as user preferences, age, gender, and social behavior.

Pattaret *et al.* [124] present a distributed three-phase service discovery mechanism that provides personalized search results based on client's requirements. These requirements are categorized into two types, optional and essential. Essential requirements represent the services needed. Optional requirements refer to Environmental information (such as time, region, and temperature) and social information (such as individual preferences). First, essential requirements are used to extract a matching set of services only containing the requested services. This matchmaking process helps reduce the search space. Further, the optional requirements of the user are matched with the set of filtered services. A semantic-based similarity score is computed between the client's requirements and the characteristics of filtered services. Finally, based on this similarity score, the services are ranked and returned to the client.

Hussein *et al.* [125] propose a dynamic service discovery in a smart space IoT environment. This environment is characterized by its heterogeneity, where things have various resource capabilities. Additionally, the dynamicity in location is a major characteristic where things can move within, join or leave the environment. The service discovery relies on a semantic-based service matching algorithm responsible for matching available services in a smart space with user's real-time requests. The matchmaking process is based on combining two types of contextual data: objective and subjective contexts. The objective context represents the physical aspects of the user's surrounding environment, and the subjective context represents Social factors. After filtering out unrelated services, this algorithm generates a list of services that match the user's needs. The service descriptions are located in a centralized data store.

Ko *et al.* [126] contribute as well in context-based service discovery for IoT environment. They present a user-centric IoT-based service discovery framework in an urban computing environment. It adopts a service discovery mechanism for IoT environments based on the available IoT resources and other social and environmental contextual information. The former consists of the temperature, noise, humidity, brightness, the season, day of the week, phase of the day, and location. The social characteristics of the users are their demographic information, such as their age, gender, and occupation. The authors considered the contextual information to filter out the set of tasks that do not contribute to supporting clients' needs. The service descriptions are represented in Web Services Description Language (WSDL) and include information about the operations and interface details of the services. The available smart objects are registered and discovered using a decentralized Domain Name Server (DNS).

Yu *et al.* [127] provide an approach for selecting appropriate services at runtime using users' and services' contexts. The approach includes a user's context modeling based on four aspects: user profile, resources, activities, and physical locations. Service context is based on the category of the service. Each category consists of a set of Metadata that captures the non-functional properties of the service. These Metadata are presented in an OWL-based service description registered in a centralized store. Context-aware criteria represent the data from the service profile as well as data from the user context. The selection process evaluates each criterion for each available service and gets the overall score to select the most suitable service.

Butt *et al.* [128] present a RESTful web service discovery protocol for the IoT environment. The contextual information used for service selection is related to users, services, and the environment. Services and environmental contexts are added to the service descriptions in an attribute-value format with semantic information. The user sends a request containing the desired service and location. The discovery process searches IoT devices providing the given service in the specified location. If multiple services exist, the contextual information is used to select the appropriate match.

Other efforts in the literature focused on extending traditional protocols for service discovery to add contextual information. mDNS/DNS-SD [129] is a combination of two protocols for distributed service discovery where the service selection criterion is the service identifier. Multicast Domain Name System (mDNS) is a communication protocol that resolves hostnames to IP addresses without a local name server. Each device in the system is a client and server in the discovery process. DNS Service Discovery (DNS-SD) is a protocol for describing and resolving services using semantic-

based DNS Resource Records. Stolikj *et al.* [130], propose an extension of mDNS/DNS-SD that enables clients to discover and select services based on contexts. Each service is associated with a set of context properties. The service discovery process consists of sending a query with wanted/unwanted context tags, service type, and the logical domain. Then, it returns the descriptions of services satisfying the client's query.

The first common issue of existing approaches is that they are built on services identifiers. Therefore, clients specify the identifiers of the required services in their discovery requests, and the discovery process will execute the matching and selection strategies for the specified service. Using service identifiers prevents discovering new services or those without explicit identifiers (such as in IoT environment). Additionally, the other issue is that the main design focus is either device-centric or user-centric (profile and preferences). These designs are not sufficient for current data-driven systems.

Besides existing contextual information and functionality-driven strategies, this thesis aims to present a data-centric context-based discovery approach. It focuses on the users' data objects and discovers the available services designed for this data.

3.2.3 Architectural design of discovery approaches

The service discovery process relies on the interaction with system data stores, referred to as *service registries*. When services are deployed in the system, they register their service description models in these registries. During the discovery process, contacting the registries allow the discovery of available services in the system. Based on the architectural design of these registries, service discovery approaches are categorized into centralized, distributed, and hierarchical/hybrid approaches.

Centralized approaches. They consist of adopting a central registration of service descriptions. They use only one global registry where all available services are registered. Albalas *et al.* [131], present an energy-aware service discovery mechanism with a single registry to host the description of each resource in the network. Their goal is to keep the service registry regularly updated with messages that contain the latest status of associated services. The energy consumption has been reduced by varying the updating intervals. Kim *et al.* [132], present a protocol-based service discovery for IoT devices using the Domain Name System (DNS). Each IoT device creates and pushes the DNS name (such as unique ID, location, and device model) and its service list to a centralized DNS server. To retrieve the available services, the client sends two requests. In the first request, the client gets the list of available devices from the DNS server. If

the client selects a device, it sends another request to the DNS server to get the service lists of the chosen IoT device.

The advantage of centralized web service discovery approaches is that the management of the distributed resources is done centrally. However, in large-scale systems, using a fully centralized registry for all the services in the system is inefficient. This design can't handle the large number of services that need to be registered. In addition, parallel accesses to the registry are very limited, and its response time could be high when dealing with a high load. For this purpose, distributed and hybrid designs have been proposed for service discovery approaches to bring more flexibility and resiliency to large-scale systems.

Distributed approaches. They consist of adopting a set of local registries that communicate together to discover the available services. The Peer-to-Peer (P2P) technology is widely used among existing distributed topologies. In a P2P network, all member nodes have equal capacity for sharing information and can establish direct connections with any other member node to download information [133]. A member node in the P2P overlay network is referred to as a peer. Any peer has the ability to join the network, which makes this technology highly scalable. In addition, as each peer acts as a client and a server, the P2P network doesn't have a single point of failure. The nodes' failures don't compromise the overall availability of the services, which makes this technology robust. P2P networks for service discovery allow implementing a distributed and large-scale infrastructure to realize the discovery at the local scale.

Using P2P technology to discover declared services in a large-scale environment using multi-criteria was first introduced by Tedeschi *et al.* in [134] and was later integrated into several projects, such as in [135] and [136]. In [134], the available services are defined using a set of service-related attributes such as service's name, processor type, and operating system. The service description is stored in a set of *Prefix* trees, each for an attribute. Each service's attribute is stored in a *Placement* table located only in the leaf node of its dedicated tree. During the discovery process, the discovery request is routed in the tree based on the prefix of the specified attribute. The leaf node matching the attribute's prefix returns the stored information about the services having the requested attribute.

Li *et al.* [137], present a distributed discovery mechanism based on P2P technology. In the adopted network, every node maintains connections to other nodes and can send messages. Every peer provides an information registry. When new services are deployed in the network, their descriptions are stored in the local information registry. Each peer

exposes a set of discovery APIs that clients can invoke. In addition, they maintain a routing table to enable efficient query routing.

Cirani *et al.* [138], present an automated service and resource QoS-aware discovery mechanisms for large-scale IoT networks using P2P architecture. The proposed mechanisms allow the discovery of RESTful services and resources in a local and global scope. Smart objects residing in the same or different networks can interact. Due to the adoption of P2P architecture, this technique is self-configuring, fault-tolerant, and scalable. Rui *et al.* [139] present as well a QoS-aware discovery process for a service-oriented IoT environment by using the P2P network. The P2P network can be modified in four main steps, Edge adding, Edge removal, Edge updating, and maximum in-degree updating.

Aside from P2P-based architectures, some innovative topologies exist. Rapti *et al.* [140], for example, propose distributed service discovery and selection techniques based on artificial potential fields. Their approach to selecting the requested services is inspired by physics and specifically from the interaction between electrically charged particles. Through the balance of forces applied among service nodes and service requests, the system route the user request in the network to achieve the composition of the services. The charges formed are based on the percentage of requested services provided by service providers and the availability of service nodes.

A challenging aspect of service discovery in dynamic environments is discovering newly published services without human interference. In [141], the author presents a research project that focuses on creating a self-healing system allowing the dynamic adaptation to newly created microservices. They mentioned that in production, the service discovery should work together with management components to control the changes of services versions. Krivic *et al.* [142], target as well this issue by creating a distributed agent-based service discovery mechanism in an IoT environment using the Machine-to-Machine (M2M) technology. M2M enables communications between machines using embedded devices that can capture data and transmit it. The discovery and registration processes are managed by a gateway, server, and device agents. Each IoT device is linked to a previously deployed device agent. The Gateway agent automatically initiates the discovery process of the new device. The device returns data about its services. These data are stored in the database of the Server Agent.

Hierarchical approaches. Other than centralized and distributed designs, some researchers focused on designing Hybrid/Hierarchical architectural designs for service discovery. In hierarchical mode, the involved registries for service descriptions are organized in a hierarchical structure where local and global registries exist. This design

is known for its scalability and ability to reduce network traffic and overhead.

Helal *et al.* [143] propose an energy-efficient technique for service discovery based on a structured P2P multi-tier architecture. The geographical region covered by the system is divided into areas. Each area has an area router and a local store. With the hierarchical model, the area router manages all the stores in its area. Each local store contains the attributes of a few regular sensors within its direct communication range. In addition, area routers store the attributes that can be found at each local store in its area and can communicate due to the P2P network. This system focus on guaranteeing a high success rate while maintaining energy efficiency. Energy efficiency is achieved by limiting the number of hops that a service discovery request must traverse before being satisfied. A high success rate is guaranteed using the hierarchical structure.

Ben Fredj *et al.* [144], present a semantic-based IoT service discovery mechanism using the hierarchical structure for IoT environments. An IoT environment is modeled as a tree hierarchy of smartspaces (e.g., country, region, city). Each smart space is managed by a gateway component that maintains the service descriptions of IoT services in its scope and processes discovery requests. The aggregated information of each semantic gateway is then sent to its parent gateway. The system adopts a dynamic clustering of services to support dynamic environments and reduce the number of service-request matching operations. Within a smart space, services are grouped based on their location. In each gateway, a routing table is built, representing each cluster. During discovery, the parent gateway decides, based on the semantic request description and on the user's geographical position, which gateway should first receive the request to start performing service search.

In this thesis, we adopt a P2P-based distributed architectural design for the proposed data-driven service discovery mechanism. The system adopts a dynamic grouping of services based on data rather than location. Each *Group* of services has its own local registry for service descriptions and gateway for management. The geographical area covered by the system is divided into *Regions* and *Regions* into *Zones* to reduce the network overhead. Unlike the hierarchical design, no global registries are used. The services descriptions are only stored in the local groups' registries within the *Zones* and can be accessed from any geographical area due to the P2P structure and the exposed Discovery APIs. Details about the architecture are presented in Section 3.5.

The Table 3.1 below summarizes the discovery approaches mentioned in the literature review and analyzes them based on their architecture, category, service description, and discovery scope.

Table 3.1 – Service discovery approaches in the literature review.

Architecture	**Research**	**Category**	**Service Description**	**Discovery Scope**
Centralized	[125, 127]	Context-aware	Semantic	Remote
	[131]	Energy-aware	Syntactic	Remote
	[132]	Protocol-aware	Semantic	Remote
Distributed	[124, 126, 128, 130, 137]	Context-aware	Semantic	Remote
	[134]	Context-aware	Syntactic	Remote
	[138, 139]	QoS-aware	Semantic	Remote
	[140]	QoS-aware	Syntactic	Local
	[142]	Energy-aware	Syntactic	Remote
Hierarchical	[143]	Energy-aware	Semantic	Local
	[144]	Semantic-aware	Semantic	Remote

3.2.4 Discovery patterns for microservices architectures

Service discovery for microservices architectures has two possible patterns: client-side service discovery [145] and server-side service discovery [146].

When implementing client-side discovery, the client is responsible for identifying available services and selecting the appropriate one. Thereafter, the client contacts the target services directly. The client is the only entity that implements the logic of service discovery. Netflix OSS [147] provides a great example of the client-side discovery pattern. It provides a client library called Netflix Ribbon, that allows the interaction with Netflix's registry and to load balance requests across the available service instances. The advantage of this pattern is that the client has full control over the discovery process. However, it has a major security issue when dealing with external clients. Allowing a direct interaction between clients and the system's database might affect the availability of the system.

For the service-side discovery pattern, an intermediate component acts as a middle-

man to intercept clients' requests. The client exclusively talks to the intermediate component that runs at a known location. Then, this component will be responsible for identifying available services, selecting the appropriate one, and then forwarding to it the client's request. An AWS Elastic Load Balancer (ELB) [148] is an example of a server-side discovery component. It distributes incoming application traffic across multiple targets. Even though this pattern abstracts the discovery details from the client, it has a significant drawback in large-scale systems. As the intermediate component intercept all client's request and forward their data to the selected services, it can easily become a system bottleneck that affects the system performance.

In this thesis, the proposed discovery approach is based on a hybrid pattern. The client has full control over the discovery, and the client-service interactions are direct such as in the client-side pattern. However, client-registry interactions to identify and select appropriate services are done via an intermediate component like the server-side pattern. Details about the proposed discovery approach are presented in Section 3.4.

3.3 Data-centric Service Description Model

The main goal of service discovery is to show clients the available microservices deployed in the platform. To this end, each microservice must be registered in the platform using a data model to declare its availability for the discovery clients. As shown in Section 3.2.1, the service's data model usually contains basic network and service configurations such as service identifier and location. However, in data-driven service discovery, additional information related to the functionality of the service and its data should be specified.

In this thesis, we propose a data-centric service description. It is an XML-based syntactic profile that consists of the following groups of keywords: *Identification* keywords, *QoS* keywords, *Service Access* keywords, and *Data* keywords. Listing 3.1 presents an example of a microservice description. Its Identification keywords correspond to the service ID, description, and version (lines 2-4). The QoS keywords show the performance of the microservice via the two metrics, Request-Per-Second RPS and service up-time (lines 20, 21). The Service Access keywords contain the information required to access the microservice when it is discovered. It includes the hostname, port, secure port, protocol, REST interfaces, service status, health check, and request parameters (lines 5-10, 22-34, respectively). Finally, the Data keywords group provides details about the data the microservice works on and generates as a result. It contains the

Listing 3.1 – An XML-based service description model of a microservice with identification, performance, access and data related keywords.

```
<?xml version="1.0" encoding="utf-8"?>
<ApplicationID>Crop</ApplicationID>
<Description>Remove unwanted areas of an image</Description>
<InstanceVersion>v1.2</InstanceVersion>
<Hostname>service.com</Hostname>
<Port>9500</Port>
<SecurePort>443</SecurePort>
<Protocol>http</Protocol>
<Interface>/crop</Interface>
<Status>ON</Status>
<InputType>image</InputType>
<OutputType>image</OutputType>
<MaxInput>12</MaxInput>
<MinInput>1</MinInput>
<MaxInputSize>50MB</MaxInputSize>
<MaxPixelDimension>2000x15000</MaxPixelDimension>
<InputFormat>
  <format>PNG</format>
</InputFormat>
<RPS>100</RPS>
<Uptime>1567889</Uptime>
<HealthCheck>/health</HealthCheck>
<Parameters>
  <Parameter id="areaWidth">
    <ValueType>int</ValueType>
    <Description>Width of the area to extract</Description>
    <required>true</required>
  </Parameter>
  <Parameter id="file">
    <ValueType>string</ValueType>
    <Description>Location of the local/distant input object</
        Description>
    <required>true</required>
  </Parameter>
</Parameters>
```

input and output data type, min/max number of input data in a request, max input size, max input resolution, and input format (lines 11-19).

Data and *QoS*-related keywords are considered as the primary contextual information in the microservices profiles. They are examined during the data-driven service discovery process to find the appropriate service for the client. This process, explained in detail in Section 3.4, is a two-steps mechanism. Data keywords are examined in the two steps of the discovery and the QoS keywords only during the second step.

Contextual information in the discovery requests needs to be specified in order to apply the matchmaking with the available description models. During the discovery process, the client program sends information concerning its data properties (such as its data type, data format, and size) and the required service performance. If these properties match exactly the data and QoS keywords in the microservice description, the microservice is considered a matching profile. However, the profile is not a match if a strict matching is considered and one of these contextual information differs.

Three service statuses are supported in the system: *ON*, *OFF*, and *Waiting*. Services with status *ON* correspond to those ready to receive requests. Services with status *OFF* are stopped and can't receive new requests. Status *Waiting* means services are deployed but not yet ready to receive requests. During discovery, only services with status *ON* can be discovered. If a service has multiple instances (discussed in Chapter 4) and at least one instance is *ON*, the service can be discovered.

3.4 Service Discovery Mechanism

As described in Section 3.2.4, the proposed service discovery is based on a hybrid discovery pattern. This pattern aims to maintain the security of the system database, reduce the bottleneck in the intermediate components, and give clients full control over the discovery process. The hybrid pattern in our system involves two components during the discovery process: the *service registry* and the *API Gateway*.

The **service registry** represents a database cluster that contains the data-centric descriptions of available microservices deployed in the platform. This database must be highly available in order to discover existing microservices at any time. In our system, new microservices instances can be created and destroyed dynamically. Due to that, this component must be continuously updated. When a new instance is deployed, its description is registered in the service registry to declare its availability. This service description is removed when the microservice is no longer available.

Two different patterns exist to handle the registration and deregistration of microservices within the service registry. This process can be done directly (self-registration pattern) or via an intermediate component called the "Registrar" (third-party registration pattern). In this platform, we use the third-party registration pattern since it decouples existing microservices from the registration process. This helps us deploy platform-independent microservices that do not need to implement any registration logic to participate in our platform. During service discovery, the service registry is queried by the discovery clients to find a matching profile with their data objects. The interaction between the client and the registry during the hybrid discovery process is intercepted by an API gateway.

The **API gateway** is an API management tool that usually sits between clients and backend services to manage the client-services interactions. In our system, the API gateway is used to manage the discovery API. It provides a customized API to apply the proposed data-driven discovery mechanism. It receives the discovery requests and contacts the dedicated registry to lookup for functionalities and microservices according to the client's objects. Any interaction between the client and the chosen microservice is direct to protect the API Gateway from overuse.

The proposed data-driven discovery mechanism consists of two steps illustrated in Figure 3.1. The mechanism works as follow:

- **Step 1.1:** At first, the client program does not know anything about the available services in the system. The client initiates the service discovery by sending a discovery request to the API Gateway containing information about its data. The API Gateway forwards the request to the service registry.

- **Step 1.2:** The service registry filters the set of stored microservices by matching the contextual information of the client's data with the data-related keywords in the stored service descriptions. Then, it returns to the client a list containing only the names of all the available functionalities in the system that can be applied to this type of data object.

- **Step 1.3:** After the client program receives the list of available functionalities, it becomes in charge of selecting the most suited functionality according to its own objectives.

- **Step 2.1:** In the second step, the client sends another request to the registry via the API Gateway, specifying the chosen functionality as well as more details

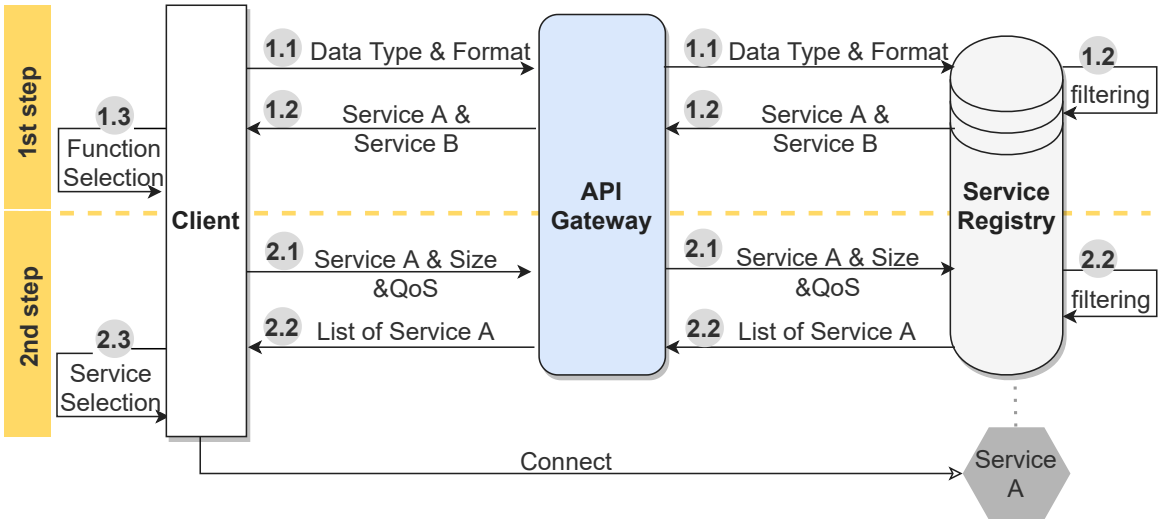


Figure 3.1 – Workflow of the data-driven service discovery initiated by the client to the API Gateway and service registry.

about its data object and preferred service performance. These parameters help reduce the search space.

- **Step 2.2:** The registry applies another matchmaking to create a new list of available services. This list contains the full descriptions of the existing microservices in the platform that can offer the desired functionality in the preferred performance and support the client's data object. Then, the registry returns the new list to the client.

- **Step 2.3:** At this point, when the client program receives the new list, it has discovered all the existing microservices that match its requirements. The client selects the preferred service in terms of machine performance and requested parameters.

After the discovery process, the client contacts the selected service instance directly. The entire architecture design of this data-driven service discovery process is presented in the Section 3.5.

3.5 Data-Driven Architectural Design

The service discovery process and its integration in a Service Mesh rely on the interaction of several system components. As the complexity of service and infrastructure

grows, there is a need to reduce the number of components implicated in the discovery process and prevent them from causing potential degradation of system performance.

Among the possible architectural designs described in Section 3.2.3, we propose a distributed data-driven architecture with the following design goals: (i) A single-purpose API Gateway specific for each type of data supported by existing microservices. This implementation design allows the management of services based on data. (ii) A Grouping of services based on each data input supported to allow later data-driven resource management. (iii) A Zone management to allow clients to discover services in a specific geographical area and balance loads between areas. (iv) A Peer-to-Peer (P2P) model that creates an overlay network between the Zones. This provides the ability to discover the remote resources deployed on several sites.

Two possible patterns exist to integrate an API gateway in a system: General purpose API backend and Backend For Frontend (BFF). The general-purpose pattern provides a single entry point to all backend services, while the BFF pattern introduces several entry points for each type of client. With the latter pattern, the incoming load is shared among multiple customized gateways tailored to the needs of each type of client. This also reduces the possibility of a bottleneck within these entry points. Our architecture adopts a customized BFF pattern where the type of clients is defined based on the supported data types in the system. Thus, if the system supports *Images* and *Videos*, two BFFs will be created: "BFF API Gateway Image" and "BFF API Gateway Video". Each implemented BFF Gateway is linked to a cluster of service registries. This cluster is responsible for storing the descriptions of microservices managed by this BFF Gateway. The stored descriptions in the cluster are not replicated to the other clusters in the system. So, the "BFF API Gateway Image" will be linked to a cluster of "service registry Image" independent of the registries of the other data types.

In this work, deployed microservices are managed by groups referred to as Data-driven Microservices Groups (DMG). Grouping microservices is based on their supported data objects. Thus, if the data input of type *Image* is supported, *DMG Image* will represent all the microservices in the Zone supporting this data type. Each DMG is managed by the BFF Gateway of the same data type. So, the *DMG Image* will be managed by the *BFF API Gateway Image*.

The wide distribution of data consumers and producers in architectures such as IoT systems can increase system latency, affecting the user experience. For this reason, we use in our system the concept of *Regions and Availability Zones* adopted by Amazon EC2 [148]. The Regions are designed to be completely isolated to ensure the stability

of our system, but the Availability Zones within a Region are connected. The resources that belong to the same geographical area are linked to the same *Availability Zone* within a *Region*. Each Zone has its own BFF gateways and service registries. It contains a component called Zone Manager (ZM), responsible for managing incoming requests. This component represents the entry point of the architecture in each Zone. It receives requests from clients located in its Zone and determines to which BFF Gateway these requests should be forwarded in order to achieve the discovery of local services. In addition to local services, Zone Manager supports the global discovery of services via a P2P network. This network is formed by the Zone Managers within the same Region. The system can discover microservices from different geographical areas by forwarding the client's discovery requests to other peers. Once the discovery process is complete, and the chosen BFF Gateway has received the list of available microservices from its dedicated registry, it sends the results back to the ZM, which in turn passes them to the clients. An overview of the data-driven architecture is presented in Figure 3.2.

The proposed architecture allows data-driven management of deployed microservices. It reduces the number of transmitted messages in the system by forwarding the clients' requests directly to the appropriate BFF Gateway and service registry cluster. Moreover, the BFF pattern adopted in this architecture aims at reducing the number of requests to be processed by each API Gateway that has become responsible for a single data category instead of all deployed microservices. This reduces the bottlenecks within these components and, as a result, improves the system performance. In addi-

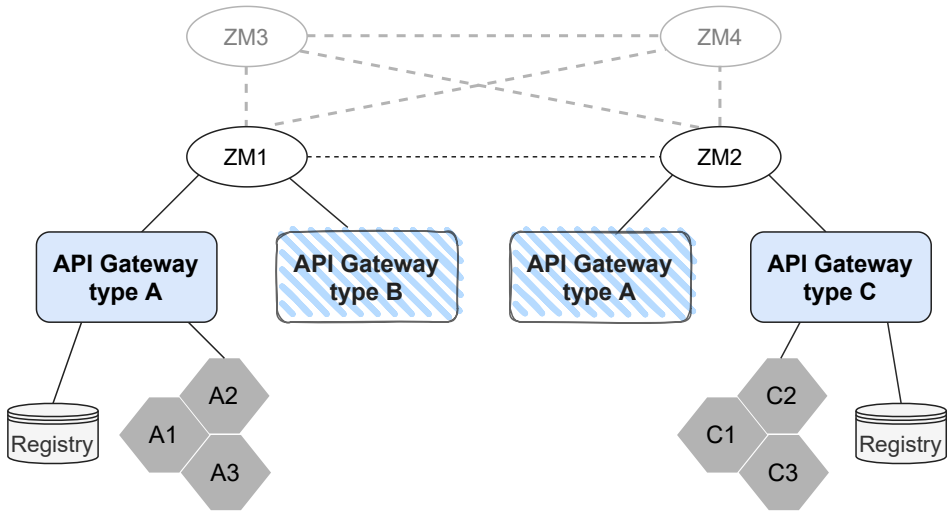


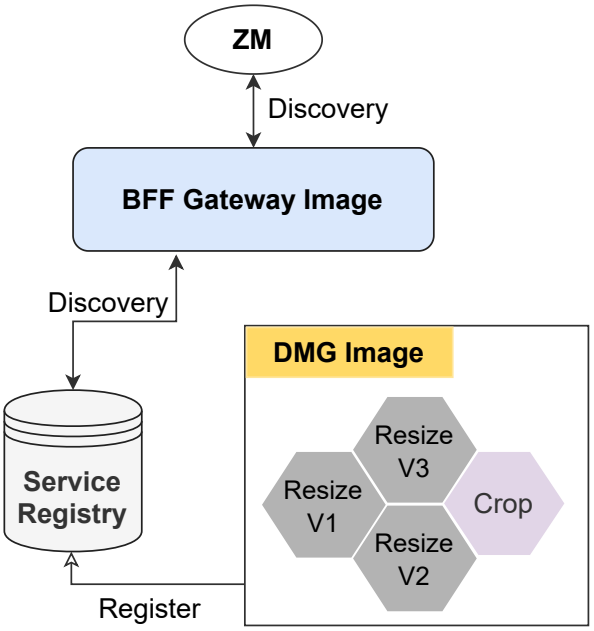
Figure 3.2 – Data-driven architectural design for service discovery with a Peer-to-Peer network between the Zone Managers of the same Region for inter-zone connections.

tion, this architecture avoids the continuous replication of microservices descriptions in the entire system by creating separate service registry clusters dedicated to each data type in every Zone of the Regions. Besides, this P2P architecture between the different Zone Managers creates an evolutionary system that allows connecting the entry points of all the Zones by forming a robust system for query processing.

3.6 Service Discovery Illustrative Example

This section aims to clarify the data-driven discovery approach with a simplified example. Figure 3.3 represents the implemented architecture and the details of deployed microservices. This system supports only data inputs of type *Image* in a specific geographical area. It exposes four microservices to external clients, each accepts data inputs of different characteristics, as shown in Figure 3.3b. These microservices are grouped in a DMG Image and managed by a BFF API Gateway Image. The data-centric descriptions of all these microservices are registered in a service registry.

The implementation of the discovery process relies on REST APIs to allow HTTP access of web clients. In this example, the client program has one input of type *Image*



(a) Architectural design with the main discovery components.

Crop service	Resize service:v2
InputFormat=PNG MaxInputSize=50MB MaxImageWidth=1600 MaxIMageHeight=1200 RPS=20	InputFormat=JPG MaxInputSize=40MB MaxImageWidth=3264 MaxIMageHeight=2448 RPS=30
Resize service:v1	**Resize service:v3**
InputFormat=JPG MaxInputSize=40MB MaxImageWidth=4080 MaxIMageHeight=3072 RPS=50	InputFormat=PNG MaxInputSize=40MB MaxImageWidth=3264 MaxIMageHeight=2448 RPS=60

(b) Data and quality-related metrics of deployed microservices.

Figure 3.3 – System supporting data objects of type *Image* with four microservices.

and `PNG` format. Once the client initiates the service discovery, the interactions in the system are as follows:

- ❏ The client program sends a discovery request describing the object's characteristics within the query parameters. In this example It sends the discovery request using l'URL `http://IP:PORT/lookup?inputType=image&inputFormat=png&quantity=1`.

  i. This system contains a match for the client's request. The service registry filters existing descriptions and returns, via the BFF Image, an HTTP response of status code `200 OK` to indicate that the request has been successfully completed. This response is presented in the Listing 3.2 below. It contains a list of existing functionalities, their names, and descriptions.

  Listing 3.2 – Functionalities that can be applied to a `PNG` image.

```
{
  "crop-service":"Remove unwanted areas of an image",
  "resize-service":"Change the pixels information of an image"
}
```

  ii. In case there were no matching functionalities present in the system, an HTTP response of status code `204` containing an empty response payload body is returned, and the discovery process stops.

- ❏ Since the discovery approach is data-driven, different data provides different discovery results. For example, if the client had an Image of `JPG` format instead of `PNG`, the registry's response will be as presented in Listing 3.3. As the microservice `crop` only supports `PNG` images, it won't match the client's request.

  Listing 3.3 – Functionalities that can be applied to a `JPG` image.

```
{
 "resize-service":"Change the pixels information of an image"
}
```

- ❏ When the client program receives the list of functionalities, it discovers what kind of actions can be applied to its `PNG` image in the existing system. The selection process of the desired functionality is made on the client-side.

❑ In the next step of the discovery process, another request is sent for the service registry to discover the available microservices offering the chosen functionality. Within the query parameters, additional information concerning the client's data and its quality requirements are specified. If the client program in this example wants to use the `Resize` functionality on its `PNG` image, it sends the following request `http://IP:PORT/services?service=resize-service&inputSize=20&inputFormat=png&Dimension=1096x1540&RPS=50`.

i. Based on the available microservices, this system contains a resizing microservice matching the client's request. So, the registry returns the full registered description of these microservices. The format of the response is presented in the Listing 3.4. Additional information might be added to the response, depending on the data type of the client's object. Among the details of the instance, the client receives the self-linking URL to access these microservices.

Listing 3.4 – Response message associated to the discovery of services matching the client's requirements.

```
{
  "service": "resize-service",
  "description": "Change the pixels information of an
     image",
  "version": "v3",
  "inputType": "image",
  "inputFormat": "png",
  "maxInput": 2,
  "maxInputSize": 40MB,
  "maxImageWidth": 4044,
  "maxImageHeight": 3805,
  "port": 9700,
  "links":
  [ {
    "rel": "self",
    "href": http://ip_address:9700/image/show/
    [original-thumbnail-poster]?file=image.png
  } ]
  "ip_address": "172.17.0.1"
}
```

ii. In case none of the available microservices offers the desired functionality and support the client's requirements, an HTTP response with status code `204` containing an empty list is returned.

- ❑ On the client-side, the discovery client selects the appropriate microservice based on the returned metrics.
- ❑ Finally, using the returned self-linking URL, the client interacts directly with the chosen microservice instance.

3.7 Conclusion

With the microservices paradigm, producers and consumers of data are growing continuously with different Quality of Service (QoS) requirements and data supported. Therefore, keep using goal-based service discovery approaches built on identifiers to discover services locations, prevent the discovery of newly published services. This chapter introduced a standalone data-driven service discovery framework that allows client programs to discover the available functionalities and microservices depending on their data objects. It is built on a data-centric model to allow the matching between clients' requirements and services descriptions. In addition, it uses a data-driven microservices architecture with a Peer-to-Peer (P2P) network that enables remote discovery in different geographical areas.

Dealing with fluctuating load affects the system's performance. Therefore, designing resource management strategies is needed to guarantee a QoS during discovery. The next chapter discusses the resource management approaches adopted.

Chapter 4

Data-driven Resource Adaptation Approach

Contents

4.1 Introduction

The data-driven architecture covers several geographical areas due to the integration of Peer-to-Peer technology. During runtime, the number of data producers in the system is dynamic where new users can continuously join the system, and existing ones may leave. When several data producers join, the system becomes a target to resource overhead and performance degradation due to the increase in incoming load.

Dealing with fluctuating load requires designing resource adaptation solutions to guarantee the Quality of Service (QoS) requirements of deployed microservices. A pri-

mary challenge of managing the resources in the proposed data-driven system is the microservices heterogeneity in terms of data supported. Each data producer generates particular data and aims to utilize available microservices developed to process their specific data. Managing heterogeneous microservices lead a data type to take over the available system resources when dealing with high load. Thus, to overcome this challenge, we propose a data-driven adaptation scheme that controls deployed microservices by data type to reduce system response time. It groups existing microservices in Data-driven Microservices Groups (DMG) and scales them dynamically based on the incoming load. The integration of this adaptation scheme in the data-driven architecture requires the implementation of management services. This chapter presents the adaptation scheme, the management components, and their implementation in an existing Service Mesh, Istio.

The chapter is organized as follows: Section 4.2 presents a literature review on existing resource scaling approaches. Section 4.3 presents significant limitations of Istio Service Mesh in creating a data-driven service discovery system. The architectural design for improving the QoS and the different system components are described in Section 4.4. Section 4.5 presents the adaptation algorithms. Section 4.6 shows in detail the evaluation of the system performance with stable and dynamic incoming load. Finally, Section 4.7 concludes this chapter.

4.2 Literature Review: Scaling Approaches

Scalability describes the elasticity of a system. It is the ability to adapt system resources to changing demands [149]. Rather than a manual scaling that requires human intervention, Cloud computing systems adopt dynamic scaling techniques for adapting the resources automatically. These techniques are known as *auto-scaling* techniques.

System resources can refer to physical machines, Virtual Machines (VM), containers, or storage. In this work, the focus is on container-based microservices. Application containerization is a lightweight virtualization technology [72, 150]. It consists of organizing all the runtime components necessary to execute an application in an isolated environment, including configuration files, libraries, and environment variables. An example of virtual containers is Docker [85]. In comparison to traditional VMs, containers have higher portability. They demand lower compute resources than VMs and have a better performance. Due to that, containers fit to serve the high elasticity needs of microservices in Cloud platforms.

There is a variety of scaling approaches in the literature with diverse characteristics. This section defines the scaling actions and types. In addition, it presents a set of production scaling solutions and experience studies targeting containerized systems.

4.2.1 Scaling actions

Two scaling actions exist to adapt the system resources efficiently: 1) increasing the system capacity to prevent resources from being under-provisioned when dealing with incoming load. A resource is considered under-provisioned when its capacity does not meet the performance requirements of the workload [151]; 2) decreasing the system capacity to efficiently using the reserved resources while preventing them from being over-provisioned. A resource is over-provisioned when its capacity can be sized down while still meeting the performance requirements of the workload [151]. The latter action is also referred to as *Scale In* or *Scale Down*. Approaches providing the first scaling action are generally categorized into horizontal, vertical, and hybrid scaling approaches. Horizontal scaling is usually referred to as *Scale Out*. It consists of adding additional instances of resources without updating the specification of existing ones. When scaling out containers, new instances are created, and the amount of CPU, RAM, and disk allocated are also copied over. The instances share the processing power of the machine, and incoming load is balanced between them. Kubernetes autoscaler [152] is an example of this scaling category. Vertical scaling or *Scale Up* consists of adding additional CPU, RAM, and disk to existing resources to increase their capacity. The limit of scaling up a container is the total available capacity of the machine. An example of exclusive vertical scaling for containers is the "Elastic Docker engine" [153]. Hybrid scaling approaches use a combination of horizontal and vertical scaling. Studies in [154–157], for example, present scaling techniques of this category.

In this work, the focus is on container-based *Scale Down/Out* approaches.

4.2.2 Scaling types: proactive, predictive, and reactive

Scaling techniques are divided into three groups [151, 158]: proactive, predictive, and reactive approaches. ***Proactive scaling*** is also referred to as scheduled scaling. It allows to schedule for increasing and decreasing the system capacity ahead of the variation in incoming demands. The scheduled scaling is triggered at a specific date and time with a predefined desired number of instances. This scaling type is usually used

in use cases where the load that will appear in the future is already known. Cloud providers such as Amazon AWS [159] and Google Cloud [160] provide this feature.

When the incoming load is not known in advance, rule-based auto-scaling techniques are used. They consist of generating rules that define a condition and an action to be executed if the condition is met. Those rules are defined on specific performance metrics. For example, a scaling rule can be *if CPU_usage>60%, then add one instance*. The rule-based approaches are further categorized as predictive and reactive scaling approaches. ***Predictive scaling*** techniques consist of creating prediction models that analyze historical data to predict when the workload will change. These models require to be trained on real workloads before being used in production. This scaling type is widely investigated in the literature. Ye *et al.* [161], for example, propose an autoscaler for containerized elastic application based on resource demand prediction model. Its goal is to minimize the violation of Service Level Agreements (SLA) when dealing with fluctuating demands to guarantee applications' performance and save cost. Abdullah *et al.* [162] present a novel predictive auto-scaling approach for microservices running on a Fog environment with limited computing capacity. It aims to minimize the violation of response time specified in the SLA. The authors target the time and resources challenges of existing proactive systems in building predictive models. They present a simple and computationally inexpensive machine learning model able to be trained on small datasets. ***Reactive scaling***, on the other hand, means that the system resources are scaled in reaction to changing demands. A threshold-based scaling strategy is a reactive approach based on two scaling rules that define when to increase and decrease the system capacity. Those rules are defined on specific performance metrics and predefined thresholds. If the current value of the chosen metric exceeds or falls below the threshold, the system triggers the scaling decisions.

In this work, the proposed system deals with a dynamic load that is unknown in advance. Furthermore, due to the simplicity of reactive scaling approaches in comparison to proactive approaches and the fact that they are easy to understand for clients [151], this work targets threshold-based scaling approaches to address under- and over-provisioning resources.

4.2.3 Production threshold-based auto-scaling solutions

Several Cloud providers and frameworks adopt the threshold-based scaling approach, such as Google's Kubernetes [152], Amazon EC2 [159], and Google Cloud Platform [160].

Kubernetes (k8s) is a production-ready open-source container management system [86]. It runs a set of pods that encapsulate the application's containers. These pods are accessible via the network using an abstraction called *service*. k8s adopts a Horizontal Pod Auto-scaling (HPA) approach principally based on CPU utilization [152]. HPA is responsible for adapting the number of pods **of a single service** to maintain an average CPU utilization across all pods close to the desired value.

The Amazon EC2 AWS platform offers auto-scaling approaches for managing *Auto Scaling groups* [159]. Each group is a set of EC2 instances that need to be treated as a logical unit for the purposes of management. EC2 instances are similar to VMs in their concept, but they manage their resources differently. The auto-scaling approaches apply scaling rules on each EC2 group based on infrastructure-level metrics such as "CPU utilization" or application-level metrics such as "Request Count Per Target". These approaches increase the EC2 group's capacity by a dynamic number of EC2 instances when the specified metric is above a threshold or decrease the capacity when the metric is below the threshold for a time interval. The adaptation of EC2 instances happens **within the same group** while respecting a predefined minimum and maximum number of instances in a group.

The Google Cloud Platform provides an auto-scaling approach for managing groups of VM instances that need to be controlled as a single entity [160]. These groups are called *Managed Instance Groups* (MIGs). MIGs let the system operates applications on multiple identical VMs. The auto-scaling approach uses multiple auto-scaling policies where each is a single-level rule based on one metric. For example, one policy can be based on the average CPU utilization as an infrastructure-level metric. Another policy can be based on application throughput as an application-level metric. The chosen scaling policy is the one recommending the largest number of instances. The auto-scaling of VM instances happens **within the same group**.

For this work, these auto-scalers are inefficient. The proposed data-driven system aims to manage microservices as Data-driven Microservices Groups (DMGs) and apply auto-scaling strategies on the groups themselves, not the microservices within the group.

4.2.4 Microservices scaling approaches

Microservice scaling is considered a relatively new research topic [163]. Nowadays, researchers have taken some initiative for rule-based auto-scaling strategies for containerized microservices [164–166]. Gotin *et al.* [164] present a threshold-based auto-scaling

system to perform horizontal scaling for I/O-intensive and compute-intensive microservices. Authors investigate which performance metrics to be used by the auto-scaler in order to prevent overloaded message queues and avoid SLA violations. Their evaluation showed that the traditional CPU utilization metric is suitable for scaling all classes of microservices if they have constant characteristics. However, it is vulnerable if microservices are responsible for different computational operations.

Abdel Khaleq *et al.* [165] propose an auto-scaling system that takes into account the heterogeneity of microservices in terms of QoS and resource requirements. The QoS metric is the Response Time, and it is extracted from the Service Level Agreements (SLA). Authors added intelligence to their auto-scaling approach via machine learning and reinforcement learning models. These models help enhance the threshold values of the auto-scaling. The evaluated performance metrics for auto-scaling are the queue size for microservices high on input traffic, CPU usage for CPU-intensive microservices, and memory usage for memory-intensive microservices.

Taherizadeh *et al.* [166] present a multi-level auto-scaling system with dynamically changing thresholds. Authors argue that using infrastructure-level metrics such as CPU usage might be helpful for some basic applications, but their effectiveness drops when dealing with heterogeneous services. The proposed system uses not only infrastructure but also application-level monitoring data such as application throughput. Authors define a fine-grained auto-scaling for containerized applications as an approach able to satisfy application performance requirements (e.g., response time) while optimizing the resource utilization in terms of the number of container instances.

In this work, the proposed microservices auto-scaling approach considers the heterogeneity of deployed microservices in terms of their data characteristics. Each supported data type has an auto-scaling threshold that is dynamically adjusted based on available instances. The auto-scaling rules are based on an application-level metric (total number of active clients), and the QoS metric we tend to satisfy is the system response time. The proposed system tends to optimize resource usage by continuously adapting the number of instances using *Scale Out* and *Scale Down* algorithms.

4.3 Limitations of the Istio Service Mesh

The proposed data-driven system is implemented in the Istio Service Mesh to manage the services interactions. It offers a set of key features that allows to secure, observe, connect and control microservices across the Service Mesh network. Istio works na-

tively with Kubernetes (k8s) [86]. All microservices deployed in k8s are distributed across one or multiple virtual clusters, called *Namespaces*, that create logically isolated environments.

Istio presents three significant limitations in creating a data-driven service discovery architecture with QoS guarantees:

1. Microservices deployed in Istio are not immediately visible to the clients outside the platform. To expose them, there is a need to manually create a set of routing rules linked to a component called *Ingress Gateway*. However, in our dynamic architecture, where new microservices come and go, this is inefficient. Therefore, exposing available microservices to external clients and hiding them when they are deleted must be done automatically without any external intervention.

2. When microservices have many instances, Istio is responsible for balancing the incoming traffic between them within the same *Namespace*. Our architecture manages microservices by data type as DMG representing *Namespaces* dedicated to each type. When multiple instances of a DMG exist, the system needs to balance requests between the microservices instances replicated in different *Namespaces* to prevent inconsistent workload distribution.

3. Istio maintains an internal service registry containing the descriptions of microservices running in the Service Mesh. In addition, Istio offers a Consul adapter as an integrated registry. However, Consul [167] provides its own service description format that does not support any additional properties. These two *service registries* use the Domain Name System (DNS) [168] for service discovery. This creates the need to modify Istio and integrate an alternative service registry that supports our proposed data model without affecting the data-driven discovery process.

Overcoming the limitations of Istio requires the integration of new components: *IngressController*, designed to automate the creation and deletion of routing rules ①. *LoadBalancer* to control sharing incoming requests among DMG instances ②. Finally, the *ServiceRegistry* that accepts the proposed data model and allows profiles matching for the data-driven discovery process ③.

4.4 Architecture Design for Improving QoS

Providing the missing functionalities in the Service Mesh and integrating resource adaptation strategies in the data-driven architecture requires implementing and deploying new management components. This section presents the set of management components used and their roles in the resource adaptation process. Three categories of management services exist in the system: API management services, operational services, and adaptation services. The full architectural design of the QoS-aware system is presented in Figure 4.1.

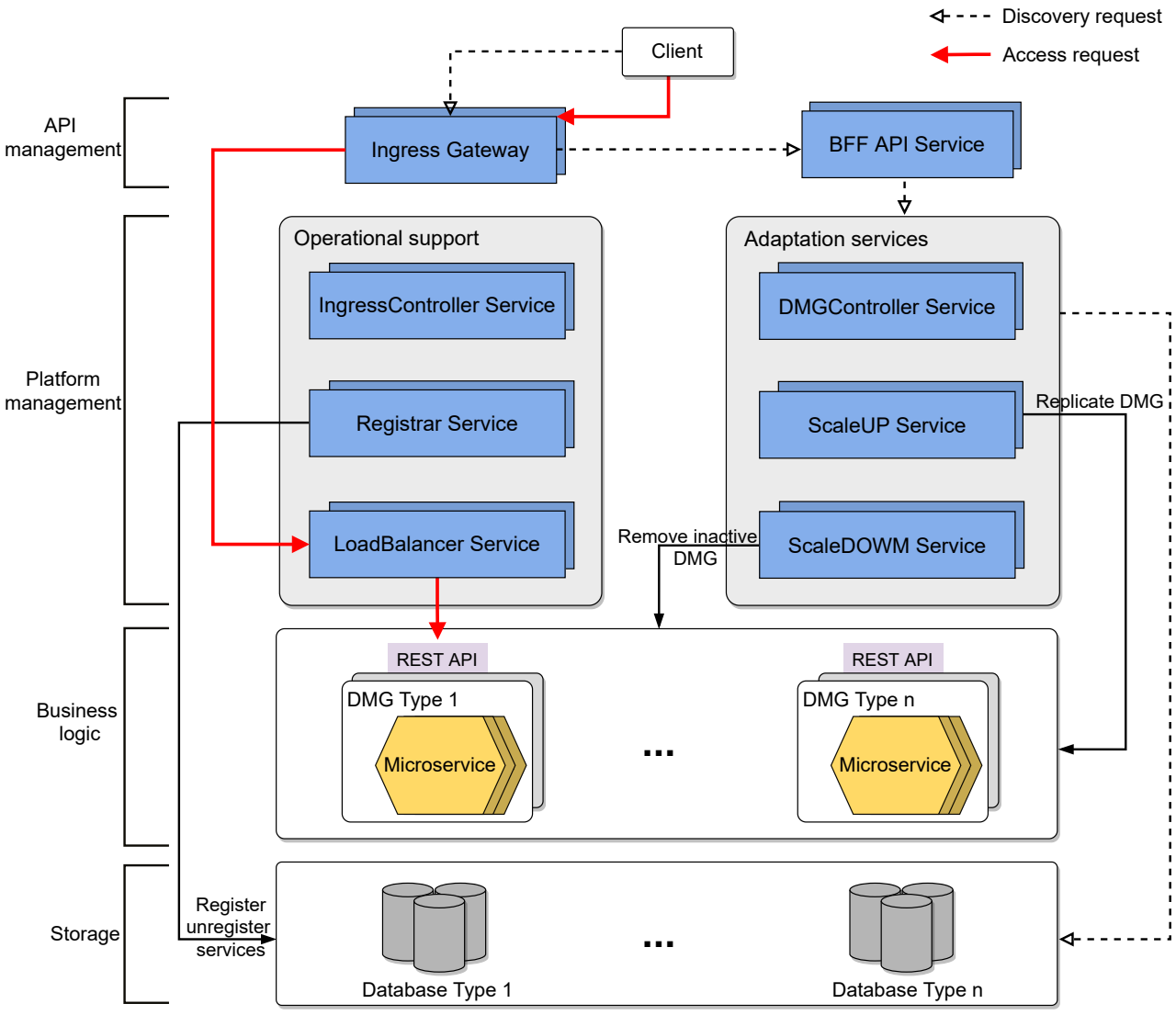


Figure 4.1 – Overview of the data-driven QoS architecture. It provides operational and adaptation support to control the discovery and access requests initiated by the clients.

4.4.1 API management services

The system adopts a data-driven discovery approach with a new communication strategy between clients and the internal system components. Integrating this approach in the system requires exposing a discovery API and control incoming traffic. The management services in this category are presented below:

1) Ingress Gateway Service: It is a given service by the Service Mesh Istio. It is required for exposing services to external clients and managing routing rules. The routing rule is a powerful tool for traffic management in Istio. Based on a matching condition, a routing rule forward the incoming requests to particular destinations. The matching conditions can be on traffic ports, header fields, URIs, and more. In this work, the matching conditions are on services' URIs. The *Ingress Gateway* is the entry point of the Service Mesh. For the discovery requests, this component forwards the load to the *BFF API Gateway Service*. For accessing the services after discovery, the system does not use the default Load Balancer given by the Service Mesh. The *Ingress Gateway* is designed to forward the load to a customized *LoadBalancer Service*.

2) BFF API Gateway Service: it is a service created to manage the customized discovery API. It is created for each data type supported in the platform. As described in Chapter 3 on page 42, this component receives the discovery requests and contacts the dedicated registry to lookup for functionalities and microservices according to the client's data object. Before this component proceeds to the discovery mechanism, it is responsible for checking whether the discovery requests must be rejected or not. To do so, the *BFF API Gateway service* sends the client's data type, specified in the incoming requests, to another management service, called the *DMGContoller Service*, presented later in Section 4.4.3.

4.4.2 Operational services

Grouping deployed microservices by data type as DMG is not supported in the literature. Thus, the system needs to implement services for managing the microservices registration, exposing them to external clients, and controlling their traffic routing. These management services control the deployed microservices and help the resource management process. The services in this category are presented below:

1) Registar Service: it is a customized third-party registration component. It

receives the microservices description models at startup, registers them in the local service registry, and then un-registers the microservices at shutdown. This Registrar is the only component in the platform able to notify the management services when a new microservice is deployed or removed. Many third-party registration components exist in the literature to automate the registration process in containerized platforms such as Registrator [169] and Joyent [170]. However, existing components are not helpful in our architecture for two reasons: first, they create services description based on the environment and does not support predefined data-centric service description models; second, they are designed to work with specific registries such as Consul and etcd [171] and do not support customized databases. For those two reasons, we need to create our own Registrar to manage the registration of DMGs.

2) IngressController Service: it is a standalone service able to update Istio's routing rules automatically. This component is needed in order to dynamically expose deployed microservices to external clients in the *Ingress Gateway* without any human intervention. It makes Istio more suitable for dynamic environments. The *IngressController service* exposes two `http` endpoints to receive notifications from the *Registrar Service* when microservices are added or removed. Based on the notifications, it updates the routing rules. If a new microservice is deployed in the system, the *IngressController service* adds a new rule containing the URI of the service as a matching condition. If the microservice is removed, the rule is deleted. The destination of the added routing rules is a customized *Load Balancer Service*.

3) LoadBalancer Service: it is a standalone service designed to apply load balancing algorithms. As Service Mesh does not support routing traffic across DMG instances, this component was needed. It receives incoming requests, and based on their URI, it distributes them among the instances of the requested microservices located in different DMGs. The *Load Balancer* uses a Round Robin algorithm by default, and additional algorithms can be added as well. It exposes an `http` endpoint to receive incoming traffic from the *Ingress Gateway service* and another to receive notifications from the *Registrar Service* when new microservices are added or deleted.

4.4.3 Adaptation services

The core of the resource adaptation scheme is implemented in three management services. With these adaptation services, the data-driven architecture is able to *Scale Out*

and *Scale Down* groups of microservices for specific data types and shed load to reduce system overhead. The services in this category are the following:

1) DMGController Service: it represents the trigger component of the adaptation strategies deployed in the system. For each discovery request, it receives from the *BFF Gateway service* the type of the client's data. Based on the total number of active clients in the DMGs of the specified data type, the *DMGController Service* decides whether the request for this data type can proceed with the discovery process. The active clients of a DMG correspond to those currently using the microservices within the group. If the targeted DMG is overloaded, it triggers the *ScaleOUT Service* to increase the system capacity. If the *ScaleOUT Service* is unable to scale the microservices due to the lack of free resources, the *DMGController Service* starts shedding the load. If the targeted DMG is not overloaded, the discovery process proceeds normally.

2) ScaleOUT Service: it is responsible for the "*ScaleOut*" algorithm, presented later in Section 4.5. It allows the system to adapt to the incoming requests by creating new DMG instances for overloaded data types while considering available resources. It exposes an `http` endpoint for receiving overload notifications and interacts with the *service registries* to add newly created DMGs.

3) ScaleDOWN Service: it is responsible for the "*ScaleDown*" algorithm, presented in Section 4.5. This algorithm is triggered periodically by the *ScaleDOWN Service* to check the resource capacity of each supported data type. If a data type has a small number of active clients in comparison to a fixed threshold and has more than one DMG instance, the algorithm scales down the DMGs of this data type and free allocated resources.

4.5 Management Algorithms

The proposed data-driven microservices architecture is dynamic where new microservices can be added, and others removed. This affects the number of incoming requests, which will be continuously changing. The fickleness of load leads to a misuse of system resources: Excessive load causes an overuse of resources, which significantly slows down the processing system. Besides, deficient load gives rise to an underuse of reserved resources.

One way to avoid the overuse of resources is to set a limit to the maximum number

of concurrent requests that microservices can process. However, our system contains microservices of different data types that share the system's resources. Thus, microservices of a specific DMG can sometimes receive a considerable number of requests. This can lead the system to reach its fixed threshold (referred to as `maxreq`) without allowing other microservices groups of different data types to process additional requests. In this data-driven architecture, avoiding a specific data type from taking over Zone's resources is done by defining a rate limiter and a resource quota (CPU, RAM) for each data type supported. Thus, when deploying a DMG, the system can only reserve resources from those assigned to its data type without violating its resource quota.

On the other hand, the number and size of data in each incoming request are unknown in advance. So, using only a rate limiter to control the resource usage can sometimes lead to a misuse of system resources. For example, with a limit rate equal to 10 concurrent requests, an application that accepts only images with dimensions less than 2560x1600 will need fewer resources to processes 10 HD grayscale images with dimensions 1280x720 than 10 Widescreen RGB images with the maximum dimensions.

As a consequence, we designed an adaptation scheme based on three QoS algorithms: ScaleOut, ScaleDown, and Load Shedding. This scheme helps create a system able to handle the rise and fall of incoming load with minimal performance degradation. In addition, it prevents rejecting requests when the rate limiter is reached, but there are still enough resources in the platform. The details of the algorithms are presented in this section.

4.5.1 ScaleOut algorithm

This algorithm is responsible for increasing the capacity of the system when an overload is detected. As described in Section 4.4, the component responsible for detecting the overload is the *DMGController Service*. The *BFF Gateway Service* notifies the controller for each discovery request by sending the client's data type. The controller checks whether an action should be done to prevent an overload from the microservices of this data type, or the discovery request can enter the system directly. The ScaleOut algorithm is presented in Algorithm 1. The logic of function `Discovery` is implemented in the *BFF Gateway Service*, and that of function `NotifyDMG` is implemented in the *DMGConroller Service*.

When a notification is received, the *DMGController Service* examines the total number of running requests for the active users (referred to as `RunningReq`) in all the

Algorithm 1: Scale out overloaded data-driven microservices groups

```
Data: dataType, dataFormat
Result: Discover functionalities depending on the data
1  Microservice Discovery(dataType, dataFormat)
2      begin
3          HandleREQ ← NotifyDMG(dataType);
4          if HandleREQ == TRUE then
5              MSlist ← FindServices(dataType, dataFormat);
6              return (MSlist);
7          else
8              return(NULL);

Data: dataType
Result: Modifying system capacity based on the load
9  boolean NotifyDMG(dataType)
10     begin
11         RunningReq ← countREQ(dataType);
12         maxreq ← countMAX(dataType);
13         if RunningReq > 85%× maxreq then
14             if resourceavailable()==TRUE then
15                 ScaleUP(dataType);
16             else
17                 nbDMGtype ← countDMG(dataType);
18                 if nbDMGtype > 1 then
19                     DMGname ← FindOveloadedDMG(dataType);
20                     UpdateDMGstate(DMGname, "OFF")
21             return(FALSE);
22         else
23             return(TRUE);
```

DMG that belongs to the client's data type. It verifies whether `RunningReq` exceeds 85 percent of the total rate limiter `maxreq` specified for these DMGs. If this threshold is not yet reached, the controller demands the *BFF Gateway Service* to accept the discovery request. The discovery request will be processed normally, and the client will discover the existing functionalities. However, if the threshold is reached, two options exist: ① if the resource quota specified for this data type allows the reservation of additional system resources, the *DMGController Service* triggers the *ScaleOUT Service* presented in Section 4.4 to create a new DMG instance. At each creation of a new

instance, the `maxreq` threshold will be automatically incremented. That increases the capacity of the data type to process more requests; ② However, if there are no more free resources sufficient to replicate a DMG for this data type, the algorithm hides first the overloaded DMG while ensuring that one instance remains. Hiding a DMG instance consist of changing the status of its microservices from *ON* to *OFF*. Changing the status of overloaded microservices allows them to finish processing their currently active requests while preventing them from appearing to upcoming discovery clients. Also, it decreases the allowed `maxreq` for the targeted data type. Later, the controller triggers the *LoadShedding* algorithm and notifies the *BFF Gateway Service* of the overload.

4.5.2 ScaleDown algorithm

Hiding DMG instances during the discovery process simply means changing their status but without actually removing them and releasing their allocated resources. Thus, when the system hides multiple DMGs due to system overload, the allowed resources by the specified quota become all reserved but not efficiently used. This prevents the system from using these resources to deploy new DMGs and serve additional clients. Algorithm 2 is implemented to prevent the underuse of system resources.

It is applied by the *ScaleDOWN Service* presented in Section 4.4. This service triggers the ScaleDown algorithm periodically to check the system DMGs. It has two responsibilities: ① releasing resources reserved by the hidden DMG (with status *OFF*) after they finish processing the requests of currently active clients; ② hiding the deployed DMG instances that are no longer needed. The ScaleDown algorithm considers a DMG

Algorithm 2: Scale down underutilized data-driven microservices groups

```
   Data: maxreq, DMGtype
   Result: Remove the useless DMG
1  void ScaleDownDMG(DMGtype)
2      begin
3          nbDMGtype ← countDMG(DMGtype);
4          RunningReq ← countREQ(DMGtype);
5          if RunningReq ≤ 50%× maxreq then
6              if nbDMGtype > 1 then
7                  DMGmin ← selectDMGmin();
8                  UpdateDMGstate(DMGmin,"OFF");
9          RemoveInactiveDMG(DMGtype);
```

instance of a specific data type as unessential if the capacity `maxreq` of this type is underused. If the total number of running requests `RunningReq` in all the DMG of this type is less than 50 percent of the total data type capacity `maxreq`, the algorithm will demand to change the status of one instance to *OFF*. The DMG that will be considered unessential for the system and that can be avoided is the one with the fewest number of requests at the time when the algorithm is triggered. The `maxreq` of the data type will decrease, and the algorithm will later remove the hidden DMG.

4.5.3 Load Shedding algorithm

This algorithm represents a rate limiting technique. It aims to shed some of the incoming load so that the system can continue operating and providing services for a subset of requests rather than crashing completely. In our QoS approach, we shed load in two cases: ① when the total number of running requests for a specific data type exceeds the predefined `maxreq` but there are not enough free resources (CPU, RAM) in the platform to deploy new DMG instances; ② when a new request arrives at the system and the new deployed DMG is not yet ready to receive requests (its status is still *Waiting*). In the first case, the Load Shedding algorithm will continue to reject the incoming requests for a specific data type until the already existing DMGs are no longer overloaded. For the second case, as long as the status of the newly deployed DMGs are still "*Waiting*" the algorithm will continue to shed the load forwarded to them by the *LoadBalancer Service*. When the microservices of the new DMGs are deployed and their status switch to *ON*, the Load Shedding algorithm will stop. The component responsible for applying this algorithm is the *DMGController Service*.

4.6 Evaluation of System Adaptation

The integration of the adaptation scheme in the Service Mesh offers our system the ability to adapt itself to dynamic and heterogeneous load while effectively using the physical resources of the platform. This section presents the methodology and the results of a set of experiments realized on the proposed platform. These experiments aim to evaluate the behavior and the scalability of the proposed data-driven system when dealing with unexpected load.

4.6.1 Methodology overview

4.6.1.1 Platform

Our platform is integrated into Service Mesh Istio 1.16 and built on top of Kubernetes 1.15. We implemented and deployed the architecture presented in Figure 4.1 that includes a service registry cluster, seven customized management services, and the required components of Istio and Kubernetes. The registry cluster is implemented as a distributed MySQL Cluster [172] with 6 nodes members. MySQL Cluster is a multi-master database where each data node member can accept write and read operations. It ensures that updates made by any application are instantly available to all other members. The architecture contains one DMG *Image* grouping a set of microservices dedicated to graphics and image processing. The DMG *Image* is managed in Kubernetes as a *Namespace*. The microservices that belong to it are deployed within the defined *Namespace* and are isolated from the microservices of any other DMG instance.

This platform is evaluated in two scenarios: when the rate of the incoming load is stable and when it is dynamically changing. It is important to note that the evaluation results showed in this section are not specific to this use case. This simplified use case is to show the behavior of the system in order to guarantee a QoS. The work we present here is neither limited to particular services deployed in the DMGs nor specific to certain data types.

4.6.1.2 Testbed

The platform evaluation is performed on the French large-scale platform *Grid'5000* [173]. It represents a distributed testbed designed to support experimental-driven research in parallel and distributed systems. The experimental setup of this work contains 27 compute nodes on the `dahu` cluster at the site of Grenoble. Each node is equipped with two Intel Xeon Gold 6130 processors, 16 cores per CPU, 192 GiB memory, and a primary disk drive SDD with a capacity of 240 GB. The nodes are connected by 10 Gbps Ethernet network and run 64-bit Debian stretch Linux with Java 8 installed.

Kubernetes follows master-slave architecture. So, in all the experiments, one reserved node is dedicated to be the master node, while all the remaining nodes are slaves nodes. Each experiment has been conducted with a new reservation of the testbed and deployment of the platform. We ensured that the experiments were isolated with no interference originated from other users.

4.6.1.3 Platform configuration

The resource quota specified for the data type *Image* is 13 CPUs and 23GB memory. This quota means that the total number of resources reserved by the DMGs for data type Image cannot exceed the limit. This prevents specific data types from taking over the system resources.

The initial rate limiter `maxreq` of the data type *Image* is set to 80 concurrent requests. This threshold dynamically changes based on the number of available DMGs.

4.6.1.4 Benchmarks

Two test files are created for the evaluation of the proposed platform. They represent two clients generating a different type of load for 20 minutes. The definition of load in these experiments is a set of discovery requests for the existing services.

The first test file is a data source responsible for generating a stable load of 600 requests per second RPS. The second test file is responsible for generating a dynamic load with a distribution of requests of 50 RPS, then 100 RPS, and lastly, another 50 RPS. Each rate last for 5 min.

4.6.1.5 Metrics

The evaluation metrics of these experiments are the following: system Response Time (RT), percentage of accepted requests, number of DMG replicas, and the number of incoming requests.

The *number of incoming requests* refers to the number of discovery requests generated by the test files and sent to the system. The *system response Time* corresponds to the average time needed to complete the discovery mechanism and access the selected microservices. The *percentage of accepted requests* presents the percentage of discovery requests allowed to enter the platform after the system verifies its ability to process them. The *number of DMG replicas* corresponds to the number of DMG instances for a specific data type. As in these experiments only a DMG *Image* is deployed, the number of DMG replicas corresponds to the number of DMG *Image* instances available.

During the experiments, these metrics are collected using the open-source visualization layer Grafana [174] and the storage back-end Prometheus [175]. In addition to monitoring the services deployed in the cluster, we need to monitor the Kubernetes cluster itself (such as namespaces/DMG, pods, etc.). We deployed the kube-state-metrics

v2.0.0+ endpoint [176] to monitor the Kubernetes API server. This endpoint is then linked to Prometheus to expose the collected metrics.

4.6.2 Evaluation results

Figure 4.2 shows the variation of the average RT and the percentage of accepted requests for a stable incoming load. We can observe three different phases. The first phase shows the baseline of 1 request per second. It presents a low average response time and a percentage of accepted requests equal to 100%. We aim to saturate the existing DMG by increasing the incoming rate to 600 RPS in order to show the system's behavior. When we switch from the baseline to this new rate, the second phase of the graph

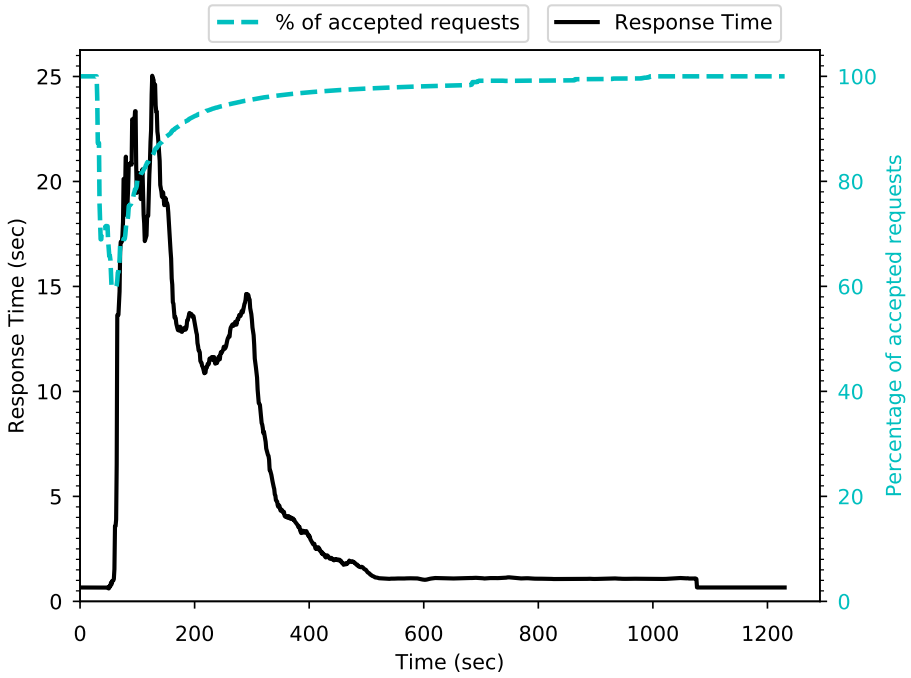


Figure 4.2 – With a stable incoming rate, the system's response time and the percentage of accepted requests stabilize around values close to the baseline due to DMG scaling.

shows that the average RT begins to increase respectively until it reaches a peak of 25 seconds. In parallel, as the existing DMG alone cannot process this incoming rate, the system starts shedding load. The graph shows that the percentage of accepted requests falls off rapidly until only 60% of incoming requests are accepted.

The drop of system performance triggers the *DMGController Service* that detects a system overload and notifies the *ScaleOUT Service*, which runs our ScaleOut algorithm.

This algorithm replicates the targeted DMG to increase the system capacity and improve the performance. In parallel, the *DMGController Service* runs the LoadShedding algorithm to prevent requests from blocking the saturated DMG.

As requests continue to arrive, the final phase in the graph shows a decrease in the average RT. This decrease continues until it stabilizes around a value close to our baseline. At the same time, due to the creation of new DMG instances, the number of accepted requests rises respectively until it maintains its peak of 100%.

In the second experiment, we switch to a dynamic distribution of load rather than a stable incoming rate to focus on the number of DMG *replicas*. Figure 4.3 shows the variation in the number of DMG *replicas* while the incoming rate varies over time following a specific distribution of the load. We aim with this distribution to show the system capacity to adjust the number of DMG *replicas* according to a load that goes up and down respectively.

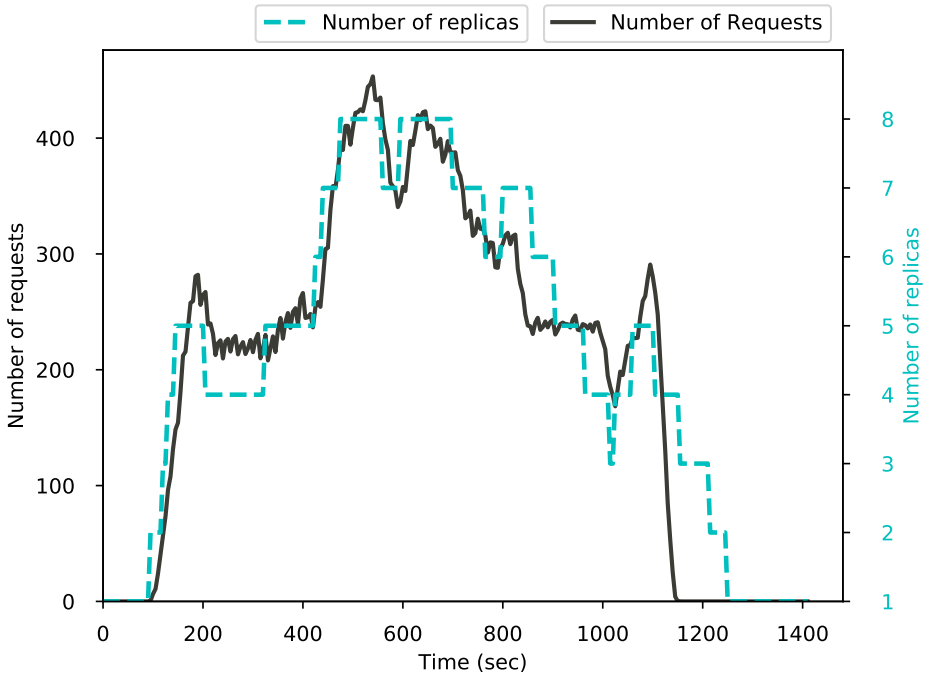


Figure 4.3 – With a dynamic incoming rate, the system tunes the number of replicas according to the load.

At first, since the unique DMG *replicas* targeted in this experiment cannot process more than 80 concurrent requests, sending 50 requests each second leads to a system overload. This triggers the *DMGController Service* and then the *ScaleOUT Service*, responsible for the ScaleOUT algorithm, to create multiple new DMG *replicas*. As

the graph shows, the number of *replicas* increased rapidly from 1 to 5 DMG *replicas*. Similarly, when the number of requests goes from 50 to 100 RPS, the number of *replicas* continues to increase slowly while following the variation of the incoming rate.

Later, to show how the system reacts if the incoming rate suddenly falls off, we switched back the number of incoming requests from 100 to 50 RPS. As the incoming rate is lower than the capacity of the total number of *replicas*, the *ScaleDOWN Service* triggers the ScaleDown algorithm in order to free unnecessary DMG. As the graph shows, following this diminution, the number of *replicas* decreases respectively with a variation similar to the incoming rate.

4.7 Conclusion

Adopting Peer-to-Peer technology and supporting multiple data types require integrating dynamic data-driven management strategies to guarantee a quality of service. The current Istio Service Mesh is not designed to support that. Thus, this work introduces new components to overcome the Service Mesh limitations.

We have deployed this Service Mesh data-driven architecture on a real-life testbed. Results showed that the platform can adapt and maintain a QoS in terms of response time and percentage of accepted requests when receiving incoming rates that exceed system capacity. In addition, the system is effectively adapting itself to the incoming load by replicating a sufficient number of DMG, enough to process the incoming requests, and removing allocated resources when they are no more needed.

In chapters 3 and 4, we targeted the resource management layer of the data-driven system. In the next chapter, we present the data and workflow management layers with the experiments conducted.

Chapter 5

Data Quality Management and Workflow Scheduling Strategies

Contents

5.1 Introduction

Ecosystems supporting emerging applications have become highly heterogeneous and geographically distributed, bringing significant challenges in delivering processing results in a timely manner with respect to other Quality of Service (QoS) constraints [36]. Many emerging applications require real-time data processing with a specific quality of the processing results, referred to as accuracy. Meeting the application's objectives when dealing with multiple data sources and resources of heterogeneous capabilities highlights the need for resource and data management solutions that enables trade-offs between the time and the quality of the application.

Resource management aims at task allocation strategies and collaborative infrastructure designs [177, 178]. On the other hand, data management often consists of customizing tasks to suit the resource-constrained systems while addressing the trade-off between QoS metrics [101, 179]. Existing work tends to approach these two aspects independently and rarely manages the entire application workflow, resulting in inefficiencies between the design and the deployment of emerging applications. This work aims to combine a data quality adaptation approach and a workflow scheduling strategy for time-sensitive workflows that rely on constrained resources located at the network's Edge, powerful core, and along the data path. The system manages trade-offs between the end-to-end workflow latency and the accuracy of results to meet the performance requirements of emerging applications. It offers means to developers to automatically distribute workflows across the Edge-to-Cloud computing continuum.

The chapter is organized as follows: Section 5.2 presents a literature review on existing data quality adaptation and workflow scheduling approaches. The system architecture, modeling, and the management objective formulation are described in Section 5.3. Section 5.4 presents the data adaptation strategy. Section 5.5 shows in detail the workflow scheduling strategy on the heterogeneous resources. Finally, Section 5.6 concludes this chapter.

5.2 Literature: Data and Workflow Management

5.2.1 Configuration adaptation for Edge-based systems

Analyzing incoming data in environments with limited resources while guaranteeing QoS constraints is challenging. Several data processing systems and frameworks either allocate predetermined resources for data processing in a way to achieve fairness among users or assume that the available resource capacity is infinite [100]. Keeping these assumptions when dealing concurrently with multiple data sources on limited resources will negatively affect the processing quality for all users. In practice, when processing data concurrently, the resources available to each data source are often unknown. Online and offline profiling-based configurations adaptation with trade-offs between QoS requirements is currently a promising solution to address the issue of limited resources [180]. Online profiling corresponds to periodically checking the possible configuration adaptations and selecting the optimal one [101, 180]. However, offline profiling consists of checking the possible configurations only once offline, and then during

runtime they choose and adjust the optimal configuration [15, 100, 102, 181–184]. The latter is inefficient in emerging dynamic applications since new services can be added or removed, new data sources can join the system with different data resolutions, and the complexity of the data content can vary during runtime.

In [184], Zamani *et al.* tackle the latency accumulation problem caused by the constrained bandwidth capacity between data producers and consumers. The proposed system decides between low-resolution and high-resolution configuration based on the outcome of the detection algorithm. If at least one object is detected, a high-resolution image is delivered to the consumer. Otherwise, a low-resolution is given. However, the authors do not consider the limited computing capacities of the streaming platform.

Wang *et al.* [15] address the issue of limited computing and network resources between IoT devices and Edge nodes. They adopt a configuration adaptation and a bandwidth allocation for a multi-user video analytics system. A configuration refers to a particular combination of video resolution and frame rate. Their objective is to find the optimal trade-off between analytics accuracy and energy consumption, with a long-term latency constraint. Unlike our system, the authors do not manage workflows in a computing continuum but single services distributed on the Edge nodes.

Kim *et al.* [180] present an online configuration adaptation algorithm in a GPU-enable Edge server for video analytics systems. Its objective is to minimize the total latency and maximize the total accuracy of analyzing multiple video streams while utilizing limited computing resources. It optimizes the resource-accuracy trade-off with respect to frame rate and frame resolution of video streams. Unlike our system, the authors target services and not application workflows. In addition, the infrastructure consists of a single Edge server and not a heterogeneous computing continuum.

The configuration adaptation is adopted by Mobile Augmented Reality (MAR) systems deployed at the network Edge [102, 183]. Lui *et al.* [183] present a multi-user MAR system that enables mobile users to dynamically change their AR configurations according to wireless conditions and computation workloads in edge servers. The dynamic configuration adaptations strategy targets the frames resolution and rate in order to reduce the service latency of MAR users and maximize the quality of augmentation.

5.2.2 Workflow scheduling strategies

In computing infrastructures, *scheduling* refers to the allocation of computing resources and the mapping of the application to those resources in a way to meet QoS con-

straints (such as latency and energy consumption) or achieve an optimization system goal (such as minimizing application makespan and maximizing throughput) [185]. Based on the application, scheduling approaches can be classified into three main categories. First, *job scheduling* techniques are designed to schedule processing jobs on the cluster workers [186–188]. Second, *single task scheduling* techniques for scheduling provided services with a guarantee of specific performance targets [189–191]. Lastly, *workflow scheduling* techniques are designed to distribute the entire application on the available resources [14,99,192]. This category is not yet well discussed in the literature. The contribution of this work belongs to the last category.

The workflows of emerging applications include tasks with diverse data quality, performance, and resource requirements. Cloud-based infrastructures consist of heterogeneous resources offering various computing and network capacities. The diversity of both applications and resources must be considered during scheduling to reduce performance degradation and achieve efficient scheduling [193,194].

Krishnapriya *et al.* [193] stressed the need to take into account the extreme heterogeneity of applications, infrastructures, and incoming data in the context of the Internet of Things (IoT) applications. Otherwise, scheduling approaches can lead to sub-optimal results.

Zhang *et al.* [194] present a task classification approach based on user-defined resource requirement (CPU, RAM) and task priority. The goal of task classification is to divide workflows into multiple task classes with similar resource demand and performance characteristics. The class of each task is used during scheduling to determine the allocation policy that should be applied to the task. Task priorities ensure that high-priority tasks are scheduled earlier than low-priority tasks. Similarly, in our work, we adopt a task classification to characterize the heterogeneous workflow for scheduling. Some classes have a higher scheduling priority than others, and each follows a scheduling policy.

The functional partitioning of a data-driven workflow and its distribution across Edge-to-Cloud resources has been considered in [14]. Ali *et al.* [14] present a system for large-scale video stream analytics. The analysis pipeline is decomposed into stages; each consists of a set of processing tasks. The basic processing tasks are considered non-intensive tasks and are placed on the Edge resources. For intensive tasks, the authors propose three strategies to distribute the stages across Edge, in-transit, and Cloud resources. The first consists of observing the inference time of the task when deployed on each layer; the second strategy relies on estimating the content of the frames; the

third strategy consists of observing historical data to decide the best placement of tasks. Among the proposed strategies, only the first strategy was further explored by the authors in [99] under deadline constraints. This scheduling strategy is goal-driven, where the scheduling decisions made are motivated by the author's goal of satisfying the deadline constraints of the user's submitted job. Unlike our data-driven scheduling strategy proposed in this work, these approaches do not consider the difference in the tasks' computing and network requirements nor the impact of incoming data on the application performance.

Renart *et al.* [192] tackle the problem of dynamically decomposing and placing IoT data-flows on heterogeneous Edge and Cloud resources. They explore the heterogeneity of operators, their interactions and resources capacity to orchestrate the IoT data-flows while optimizing the end-to-end application latency. They present a programming model based on R-Pulsar system [195] allowing developers to define dataflow splitting across the Edge and the Cloud.

5.3 System Architecture and Modeling

Several management microservices are implemented to allow the optimization of the application latency when deployed in distributed, heterogeneous and limited environment. An overview of the system architecture is presented in Figure 5.1. When an application workflow is submitted to the system, the workflow management strategy is executed in order to place the workflow on the infrastructure resources. As it is shown in Figure 5.1, four workflow management microservices are implemented to achieve the workflow management strategy. Details about this strategy are presented in Section 5.5. Later, when a user joins the system, the data management strategy is executed. It aims to specify the data quality that should be adopted by each existing user in order to optimize the application latency. The best quality distribution is selected based on performance estimations. As shown in Figure 5.1, four data management microservices are used to accomplish the data quality adaptation strategy. Details about this strategy are presented in Section 5.4. The data-driven microservice discovery mechanism presented previously in Chapter 3 is used to assign workflows to users based on their data quality.

The remainder of this section presents the infrastructure, workflow, and performance modeling. The latter consists of an end-to-end latency and a processing accuracy modeling. In addition, this section provides a formulation of the system objective.

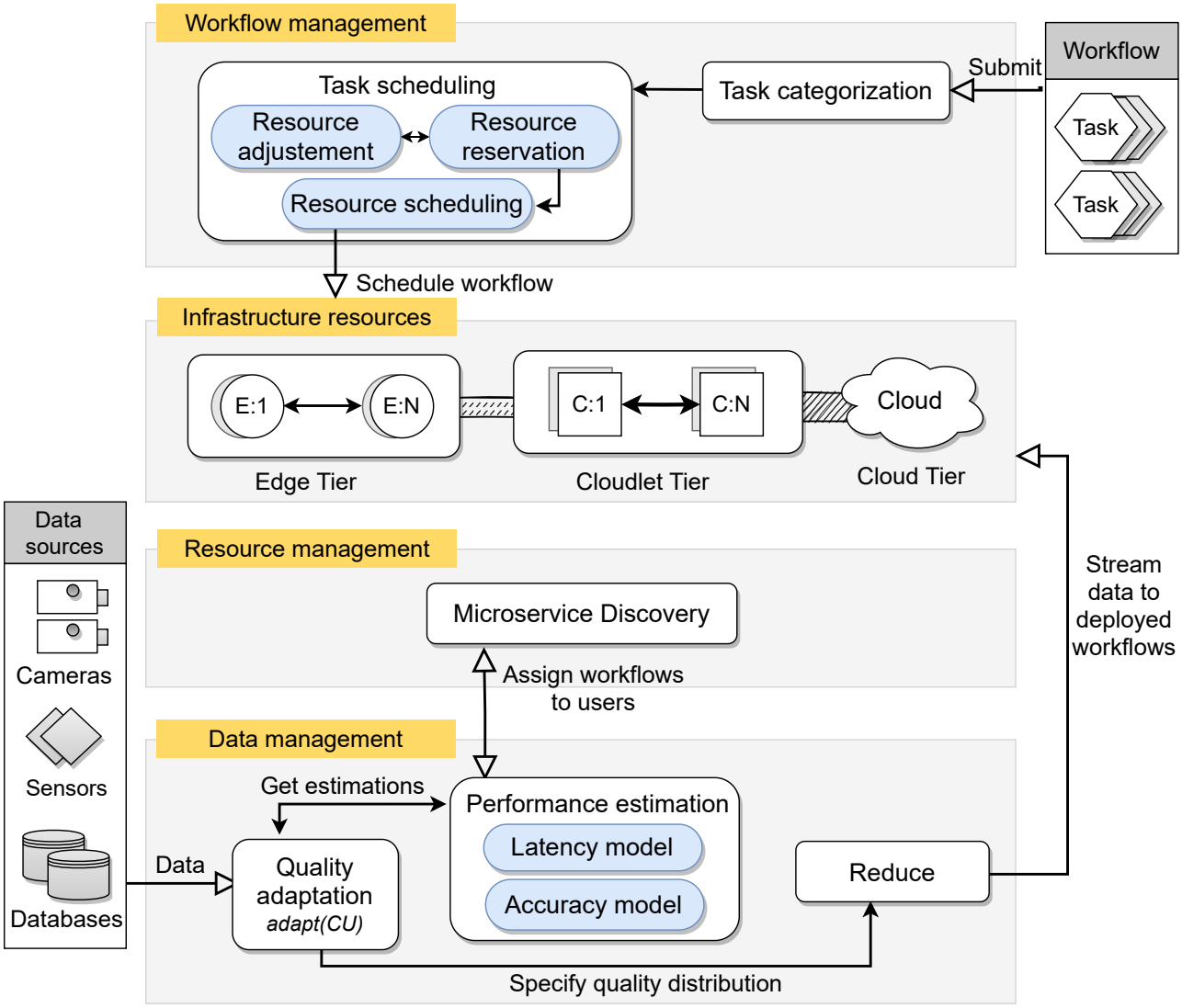


Figure 5.1 – A global overview of the system design illustrating the management microservices and their interactions.

5.3.1 Infrastructure model

The infrastructure design is based on the Edge-to-Cloud computing continuum. The set of computing resources available is given by $R = \{r_k\}$ which represents the set of Edge (E), Fog (F), and Cloud (C) nodes. In this work, the terms Cloudlet and Fog refer to the resources along the data path from the edge of the network to the core. Cloudlet resources provide computational capabilities greater than Edge resources and less than Cloud resources. The capacity of the Cloudlet-Cloud network link is a thousand times higher than the capacity of the Edge-Cloudlet link.

The system supports multiple data sources located at the Edge. The set of data sources in the system are denoted by $U = \{u_1, u_2, ..., u_k\}$. Each user u has a set of configuration metrics such as its data type, format, quality, data rate, QoS, etc. For simplicity, in this chapter, we focus on the user's data type, quality, and size. The QoS constraints are predefined for the application. Let $CU = \{c_{u_i}, 1 \leq i \leq k\}$ the set of configurations for each user $u \in U$. The configuration c_u of a user is defined as $c_u = (dt, dq, ds)$ where dt, dq and ds refer to the original user's data type, quality and size, respectively. Due to the data quality management strategy adopted in this work as a part of the proposed trade-off solution (presented later in Section 5.4), the quality dq and size ds of a user are inconstant during runtime. This adaptation strategy is defined as $adapt(CU) \longrightarrow \{CU' = \{c_{u'}\} \mid \forall c_{u'}, c_{u'} = (dt, dq', ds')\}$. It takes as input the set of users' configurations and generates the output CU'. This output corresponds to the set of users' configurations after applying the adaptation strategy. As it is shown in Figure 5.1, this configuration is obtained via the use of performance models and the data-driven service discovery approach. The metrics (dq', ds') of a user might refer to its original quality and size (dq, ds) or new adapted configuration. For simplicity, in this section, we will refer to the quality and size of user's data respectively as q and s, whether original or adapted.

5.3.2 Workflow model

An application workflow consists of a sequence of functionalities. Let $W = \{f_1, ..., f_X\}$ be the application workflow with X corresponds to the total number of functionalities provided. Different microservices can be implemented to provide the same functionality f, but each accepts specific data characteristics and has its own configuration. Thus, for each functionality f there is a set of microservices $M = \{m'_1, m'_2, ..., m'_Z\}$ that provide it. Details about the metrics used to describe the configuration of a microservice are given in Chapter 3 page 38. Let $CM = \{c_{m'_y}, 1 \leq y \leq Z\}$ be the set of configurations of each implementation $m' \in M$. For simplicity, in this chapter the microservice's configuration metrics used are defined as $c_{m'} = (t', q', s', RPS')$ where t', q' and s' corresponds to the type, quality and size of the microservice's input data, respectively. RPS' refers to the number of requests the microservice can process per second. Among the different implementations of a functionality f, one microservice at a time is selected to be a part of the workflow assigned to user. Let $m = \sum_{y=1}^{Z} \phi_y m'_y$ represents the selected microservice for a functionality f. The variable ϕ is a binary that indicates

whether a microservice m' is selected from M to provide a functionality f. Based on the data-driven service discovery mechanism presented in Chapter 3 page 40, the selected microservice is the one matching the characteristics of the user's data. Thus, the binary ϕ is presented as follows:

$$\phi = \begin{cases} 1 & \text{if } q'_{m'} = q_u,\ t'_{m'} = dt_u \text{ and } s'_{m'} \geq s_u \\ 0 & \text{otherwise.} \end{cases}$$

The application workflow corresponds to the set of microservices providing the sequence of functionalities needed. Hence, the workflow assigned to a user u is formulated as $W_u = \{m_i, 1 \leq i \leq X\}$. Due to the data quality adaptation strategy adopted in this work, the configuration of the user c_u might changes during runtime. Thus, the workflow assigned to a user u varies over time.

5.3.3 Performance models

This thesis focuses on the application's end-to-end latency, and the quality of data processing referred to as accuracy. This section presents the average end-to-end latency and accuracy of K data sources in a time slot t. A data generated by a data source u is considered as a job J to be processed by the application. Let N be the total number of jobs generated by a data source in a time slot t.

5.3.3.1 End-to-end latency model

The configuration of the job J depends on its user's $c_{u'}$ configuration generated by the adaptation strategy. If $c_{u'}$ is different than the original (or previous) user's configuration c_u, the job is forwarded to a management microservice defined as $reduce(J, c_u, c_{u'})$. It is responsible for applying the adapted configuration $c_{u'} = (dt, dq', ds')$ on job J having a current configuration c_u before starting the data processing. Hence, the end-to-end latency of a job J from a data source u corresponds to the time taken to adapt the job's configuration, if needed, and then to complete its assigned data processing workflow. The end-to-end latency is formulated as follows:

$$T_J^u = \mu \cdot T_{reduce} + T_W \tag{5.1}$$

T_{reduce} is a constant value that refers to the time required to reduce the job's quality via the management microservice *Reduce* (see Figure 5.1). The value μ is a binary

that indicates whether a data quality reduction was needed for this data source or not. Details about data adaptation will be presented in Section 5.4. T_W corresponds to the total time spent in the application workflow assigned to the user based on its data entry. It includes the sum of the response time of all the microservices in the workflow. It is presented as:

$$\begin{aligned} T_W &= \sum_{i=1}^{n} RT(m_i) \quad \text{where} \\ RT(m_i) &= \alpha \cdot T_D + Trans(m_j, m_i) + T_E \end{aligned} \tag{5.2}$$

$RT(m_i)$ represents the response time of the selected microservice m_i to provide a functionality in the workflow. The variable α is a binary that indicates whether a microservice discovery happened or not while processing a job. T_D represents the discovery time of the microservice. $Trans(m_j, m_i)$ represents the time needed to transfer the data input of microservice m_i from its predecessor m_j. A predecessor of a microservice corresponds to its preceding task in the workflow. The microservices m_j and m_i are the chosen microservices to provide two different functionalities. Let r_j and r_i be the resources where m_j and m_i are deployed respectively. For clarity, $Trans(m_j, m_i)$ is replaced by $Trans(r_j, r_i)$. T_E corresponds to the execution time of the microservice.

The transfer time of the data is formulated as:

$$Trans(r_j, r_i) = \begin{cases} \dfrac{s}{w(r_j, F)} + \dfrac{s}{w(F, r_i)} & \text{if } r_j \in E,\ r_i \in C \\ 0 & \text{if } r_i = r_j \\ \dfrac{s}{w(r_j, r_i)} & \text{otherwise,} \end{cases} \tag{5.3}$$

s refers to the data size (in Mbits) to be sent over the network. It might correspond to the original data size or to the reduced size if the function *reduce* was applied ($\mu = 1$ in Equation 5.1). The $w(producer, consumer)$ refers to the bandwidth (Mbits/s) of the network link between the data producer and consumer. If they were deployed on resources $\{r_i, r_j\} \in \{E, C\}$, the transferred data must pass through the Cloudlet tier.

The execution time of a microservice is presented as $T_E = Load/RPS$ where *Load* refers to the data to be processed by m_i. RPS is the throughput capacity of the selected microservice specified in its configuration $c_{m_i} \in CM_i$. It refers to the number of requests that can be processed per second. The value of the throughput differs based on the characteristics of the microservice selected to provide the needed functionality such as technologies used, type and resolution of input data, etc.

Thus, the average end-to-end latency of K data sources in a time slot t is presented as follows:

$$T_t = \frac{1}{K} \sum_{i=1}^{K} \left(\frac{1}{N} \sum_{j=1}^{N} T_j^i \right) \tag{5.4}$$

5.3.3.2 Accuracy model

Due to the data-driven discovery mechanism, changing the quality of the incoming data during a time slot by the data management strategy will cause a change in the selected microservice to provide the functionality needed. The microservice chosen can impact the accuracy of the application since they have different configuration and data characteristics. Let $Q = \{q'1, q'2, ..., q'L\}$ be the set of data quality supported by the deployed microservices providing the same functionality. Let ϕ_J^i and β_J^x be two binary variables that indicate whether microservice m'_i and data quality q'_x are selected for data source u_k to apply a functionality on job J. So, $m_J^k = \sum_{y=1}^{Z} \phi_{J,k}^y m'_y$ is the microservice chosen for job J from data source u_k and $q_J^k = \sum_{x=1}^{L} \beta_{J,k}^x q'_x$ is its data quality with $\sum_{y=1}^{Z} \phi_{J,k}^y = 1$ and $\sum_{x=1}^{L} \beta_{J,k}^x = 1$. Let $\mathcal{E}(q_J^k, m_J^k)$ be the accuracy of the application when processing the job J of data source u_k. Hence, the average accuracy of a data source u_k in time slot t is $\frac{1}{N} \sum_{j=1}^{N} \mathcal{E}(q_j^k, m_j^k)$. During a time slot, $\sum_{y=1}^{Z} \phi_k^y$ and $\sum_{x=1}^{L} \beta_k^x$ can be greater than 1, which indicates that a data source can have multiple data qualities and so can use multiple microservices providing the same functionality. The average accuracy of K data sources in a time slot t is presented as follows:

$$a_t = \frac{1}{K} \sum_{i=1}^{K} \left(\frac{1}{N} \sum_{j=1}^{N} \mathcal{E}(q_j^i, m_j^i) \right) \tag{5.5}$$

5.3.4 System objective

This work aims to reduce the end-to-end latency of applications under a long-term accuracy constraint. To achieve this objective, a latency-accuracy trade-off utility function is needed. It is formulated as $U_{u,t} = a_{u,t} - \Theta T_{u,t}$ where Θ trades off between the latency cost and accuracy. The total utility for K data sources in time slot t is presented as:

$$U_t = \frac{1}{K} \sum_{i=1}^{K} U_{i,t} \tag{5.6}$$

Table 5.1 – An overview of the mathematical notations.

Meaning	Notation
Infrastructure	
Set of Edge nodes	E
Set of Fog nodes	F
Set of Cloud nodes	C
Set of computing resources	$R = \{r_k\} = \{E \cup F \cup C\}$
Network Bandwidth between resources r_j and r_i	$w(r_j, r_i)$
Users	
Set of data sources/ users	$U = \{u_1, u_2, ..., u_k\}$
Set of users' configurations	$CU = \{c_{u_1}, c_{u_2}, ..., c_{u_k}\}$
Original user's data type	dt_u
Original user's data quality	dq_u
Original user's data size	ds_u
Set of users' adapted configurations	$CU' = \{c_{u'_1}, c_{u'_2}, ..., c_{u'_k}\}$
Adapted data quality	dq'
Adapted data size	ds'
Job generated by a user	J
Total number of jobs generated by a user in time slot t	N
Workflow	
Workflow assigned to user u	W_u
Implementation/microservice of a given functionality	m'_i
Set of configuration metrics for a microservice	$c_{m'_i}$
Accepted input data type by a microservice	t'
Accepted input data quality by a microservice	q'
Accepted input data size by a microservice	s'
Microservice's throughput	RPS'
Set of implementations of a particular functionality	$M = \{m'_1, m'_2, ..., m'_Z\}$
Selected implementation from M	m
Binary variable for selecting an implementation	ϕ_y
Set of data quality supported by M	Q
Binary variable for selecting a data quality from Q	β^x
Performance	
Total Latency of a workflow assigned to a user	T_W
Reduce the configuration c_u of job J to $c_{u'}$	$reduce(J, c_u, c_{u'})$
Latency of data quality reduction	T_{reduce}
Binary variable for the reduction	μ
Response time of a selected microservice m_i	$RT(m_i)$
Binary variable for the discovery process	α
Latency of service discovery	T_D
Data transfer time from microservice m_j to m_i	$Trans(m_j, m_i)$
Execution time of a job in a microservice	T_E
Total End-to-end Latency of a job J from user u	T_J^u
Average end-to-end latency of K users in time t	T_t
Processing accuracy of job J with quality q	$\mathcal{E}(q_J, m_J)$
Average accuracy of K users in time slot t	a_t
Objective Formulation	
Latency-accuracy trade-off weight	Θ
Threshold of minimum processing accuracy	a_{min}
Total trade-off utility for K data sources in time t	U_t

Maximizing this utility during runtime, with respect to the accuracy constraint, will increase the system efficiency. As in [15], the long-term utility augmentation during runtime will become limited. So, the system goal is formulated as follows:

$$\begin{aligned} Goal: \quad & \max_{\beta,\phi} \quad \lim_{t\to+\infty} \frac{1}{T}\sum_{t=1}^{T}(U_t) \\ & \text{subject to } \lim_{t\to+\infty} \frac{1}{T}\sum_{t=1}^{T} a_t \geq a_{min} \end{aligned} \tag{5.7}$$

The accuracy constraint in the formula ensures that the trade-off is only possible if the long-term average accuracy exceeds the minimum threshold a_{min}.

To optimize this utility, the system aims to adapt the quality of incoming data. This adaptation is presented in Section 5.4.

5.4 Data Quality Adaptation Strategy

The system contains a set of distributed and limited resources of different computing and network capacity. Analyzing data generated by multiple data sources on these resources while guaranteeing the latency constraint requires adopting a latency-accuracy trade-off solution. This section presents a data quality adaptation strategy for time-critical applications. This strategy is responsible for specifying the distribution of data qualities on the existing data sources. It estimates whether the system can handle the original data qualities of all data sources or a quality reduction is required. Reducing data quality can negatively affect the application's accuracy. So, this strategy controls the quality of generated data while guaranteeing a system accuracy higher than a fixed threshold a_{min}.

Let $B = \{b_1, ..., b_m\}$ be the set of possible quality distributions among data sources. Each distribution b_i is formulated as $\{\beta \cdot q_l \mid 1 \leq \beta \leq k \text{ and } q_l \in Q\}$, where β represents the number of data sources having the data quality q_l and k is the total number of data sources in the system. q_l might corresponds to their original or reduced data qualities. The proposed strategy selects from B the distribution with the fastest analysis latency and an accuracy that does not fall below the a_{min} threshold. However, in specific use cases, the selected data quality distribution may not be the fastest: The system compromises between latency and accuracy in case there is another distribution that

provides a latency gain less than $10ms$ but an accuracy loss greater than or equal to 20%. The strategy is presented in the Algorithm 3 below.

Algorithm 3: Select the distribution of data qualities across system data sources. The selection compromises between latency and accuracy.

Result: quality distribution with the optimal trade-off.

begin

```
1  initialization;
2  for b in B do
3      L ← getEstimatedLatency(b) ;          // model (5.4) in 5.3.3.2
4      A ← getEstimatedAccuracy(b) ;         // model (5.5) in 5.3.3.2
5      if A ≥ a_min then
6          add({b, L, A}, list);
7  if isEmpty(list) == TRUE then
8      return ∅
9  best ← getMinLatency(list);
10 for x in list do
11     if △(x[L], best[L]) < 10 and
12      △(x[A], best[A]) ≥ 20% then
13         best ← x;
14 return(best);
```

In steps 2-6, it starts by calculating the estimated latency and accuracy of each possible data quality distribution. Then, in step 5, it filters those with unacceptable accuracy. If no configuration provides acceptable accuracy, the data source can't join the system at that time period (steps 7 and 8). Otherwise, the preferred data distribution among the acceptable configurations is the one that provides the fastest processing (step 9). In steps 10-12, it checks if there is another distribution that matches the use case presented above. If so, it will be selected as the preferred data quality distribution in the system.

Targeted applications have dynamic data content. Therefore, adapting data quality once during analysis is inefficient as the performance varies depending on the content. To handle the dynamism of data, the adaptation strategy is triggered periodically and when new data sources join the system. The microservices responsible for applying this strategy are deployed on the Edge.

After selecting data qualities and the number of data sources that should adopt each quality, the system randomly maps qualities to data sources. This mapping switches periodically between data sources until the next adaptation period.

5.5 Data-driven Workflow Scheduling Approach

The proposed system has multiple challenges that need to be overcome in order to schedule the workflow effectively on the continuum:

1. Microservices have several implementations and have different computing and network requirements.
2. System has a limited number of resources.
3. Available system resources have different computing and network capacity.
4. In a continuum, resources can become unavailable.

For challenges ① and ③, this work adopts resource reservation and scheduling algorithms based on a task categorization approach. This approach identifies the different categories of tasks in a workflow and their requirements. The reservation algorithm aims to allocate in advance the resources for tasks requiring high computing capacity in the submitted application. Its purpose is to prevent other tasks from allocating powerful resources and forcing intensive tasks to run with insufficient computing power. The scheduling algorithm consists of distributing the tasks on the continuum based on their categories and reserved resources. These two algorithms are presented in Section 5.5.2. For challenge ②, the system adopts a requirement adjustment algorithm. It aims to adjust the resources reserved for the intensive tasks to use the system's limited computing resources efficiently. This adjustment algorithm is presented in Section 5.5.3. The 4th scheduling challenge is beyond the scope of this paper. The categorization approach and each scheduling algorithm are deployed as microservices and located on the Edge tier. This data-driven scheduling approach is triggered when a new or an updated workflow is submitted to the system. When triggered, the interactions between microservices are as follows: the categorization microservice first receives the workflow description. Then, after examining each task, it sends the workflow description with the categorization results to the resource reservation microservice. During the scheduling process, the reservation microservice allocates the resources and triggers the scheduling microservice. If the available system resources are not enough to handle the intensive tasks of the workflow, It triggers the resource adjustment microservice. This section presents the details of each microservice participating in the scheduling process.

5.5.1 Tasks categorization

The tasks categorization strategy consists of differentiating the application's tasks based on their CPU, memory, bandwidth, and storage requirements. The categorization results depend on the functionality given by the tasks, their configuration, and their data input characteristics. In this work, categorizing tasks offering the same functionality is based on the quality of their data input. The task that deals with high-quality data requires more resources than tasks with low-quality data. Identifying the categories of the tasks properly allows serving their requirements better during scheduling on resources with limited capacity.

Tasks are classified into two main categories: Non-Intensive tasks and Intensive tasks. **Non-Intensive tasks (NI)** refer to those which do not demand frequent access to the system's computational resources. In contrast, **Intensive tasks (I)** require a lot of resources to process incoming data. Among the intensive tasks, two main sub-categories can be identified: **High-Intensive tasks (HI)** and **Low-Intensive tasks (LI)**. As the name implies, High-Intensive tasks require more resources than Low-Intensive tasks. The ratios between the required resources of these categories as well as their sub-categories depend on each application use case. An example of categorizing a data-driven application use case is presented in Chapter 6.

5.5.2 Heterogeneity-aware workflow scheduling algorithms

A naive approach to assign tasks to resources is to explore all the possibilities in a brute-force manner which creates a search space of exponential complexity. This work adopts the following pruning techniques to minimize the search space. First, all tasks within the same category have the same resource assignment. Second, intensive tasks can only be deployed on the Fog and Cloud resources. HI tasks have a priority to be assigned to the Cloud and LI tasks to the Fog.

The scheduling approach consists of two parts. The first part aims to reserve resources for intensive tasks. The second part of this scheduling approach is responsible for distributing the workflow on the continuum.

5.5.2.1 Resource reservation algorithm

During resource reservation, the system gives HI tasks a higher placement priority than LI tasks. The resource reservation for HI tasks is presented in Algorithm 4. This

algorithm takes as input the number of HI tasks in the submitted workflow. In step 1 and 2, it retrieves the available resources in Cloud and Cloudlet tiers, respectively. In step 3, it counts the number of HI tasks that can be placed on the Cloud. This depends on the capacity of the Cloud tier and the fixed requirements of the HI tasks. Steps 4-16 check whether the computing requirements of HI tasks can be fully guaranteed at the Cloud or they must be distributed across Cloudlet-Cloud tiers. In addition, they decide whether the available resources can handle the entire computing requirements of HI tasks or a "requirements adjustment solution" is needed. For the first case (steps 4-6), it checks if the capacity of the Cloud tier is greater than the requirements of HI tasks. If so, it reserves all required resources on the Cloud. If not (steps 7-16), it checks whether the Fog resources can handle the requirements of the remaining tasks. If so, the reserved resources for HI tasks will be distributed across Cloudlet-Cloud tiers. However, if the resources available in the Cloudlet are also not enough, it triggers Algorithm 6, which will be presented in Section 5.5.3.

Algorithm 4: Resource reservation for high intensive tasks.

Data: countHI
Result: List of resources reserved for HI microservices

```
List Reserve_HI(countHI)
    begin
1       capCloud ← getFreeResources('cloud');
2       capFog ← getFreeResources('fog');
3       cloudHI ← countCloudHI(capCloud);
4       if cloudHI ≥ countHI then
5           entry ← reserve(countHI, 'cloud');
6           updateReservedList(entry);
7       else
8           remainHI ← countHI - cloudHI;
9           fogHI ← countFogHI(capFog);
10          if fogHI ≥ remainHI then
11              entry ← reserve(cloudHI, 'cloud');
12              updateReservedList(entry);
13              entry ← reserve(remainHI, 'fog');
14              updateReservedList(entry);
15          else
16              Adjust_res('HI');
17      return(getReservedList());
```

After reserving the resources for HI tasks, the system will repeat the same logic to reserve the resources for LI tasks. As they have a priority to be assigned to the Fog, the system checks first whether their computing requirements can be fully guaranteed at the Cloudlet level before checking the Cloud.

5.5.2.2 Workflow scheduling algorithm

The next step in the scheduling approach is to distribute the workflow on the continuum. It is presented in Algorithm 5. This algorithm takes as input the list of resources reserved for intensive tasks and the workflow. For each task in the workflow, it checks whether they are intensive or not. If so, it searches for the resources reserved for its category (steps 4 and 5). However, if not, it looks for the remaining free resources (steps 6 and 7). As NI tasks do not require a lot of computing power, they can be easily placed on system resources without prior reservation. Among discovered resources for each task, it selects those located near its predecessor (step 8). A predecessor of a task corresponds to its preceding task in the analysis pipeline or a data source in case it is the entry task of the workflow. In case several resources are located on the same infrastructure level, the selection of the resource is random. After selecting the resource, it deploys the task and makes it ready for production (step 9).

Algorithm 5: Scheduling application's workflow on reserved resources.

Data: listsIntensive, workflow
Result: Mapping DL pipeline to system resources

```
Void Scheduling(listsIntensive, workflow)
    begin
1       for task in workflow do
2           predecessor ← getPredecessor(task);
3           category ← getCategory(task);
4           if category = 'LI' OR category = 'HI' then
5               list ← getReserved(category, listsIntensive);
6           else
7               list ← getFreeResources(category);
8           res ← selectResource(list, predecessor);
9           deployTask(task, res);
```

5.5.3 Requirements adjustment algorithm

As Algorithm 4 has shown, it is possible that the system cannot handle the computing requirements of intensive tasks. Algorithm 6 is used to handle this issue. It aims to adjust the resource requirements of intensive tasks to place them on the continuum when a full guarantee of required resources is not possible. It attempts to reduce the computing requirements of intensive tasks while ensuring a minimum threshold equal to 50% of their fixed requirements. This allows maximum use of the system's limited resources. This algorithm is only triggered by Algorithm 4 if a new workflow submission occurs.

Algorithm 6 takes as input the category of the task to be placed and the current list of resources reserved for intensive tasks. In steps 2 and 3, it gets the list of remaining free resources on the Cloudlet-Cloud tiers and selects the one with the maximum remaining capacity. If the remaining capacity of the selected resource is greater than or equal to the minimum threshold, the resource is reserved (steps 3-5). However, if not, the algorithm attempts to reach the minimum threshold by adjusting the computing capacity of the other reservations on the same selected resource (steps 6-13). In step 7, it gets the remaining capacity needed to reach the minimum threshold. The reservations

Algorithm 6: Resource adjustment for intensive tasks.

Data: listLI, listHI, category
Result: Adjust required resources of intensive tasks

```
Void Adjust_res(listLI, listHI, category)
    begin
1       listFree ← getFreeResources();
2       res ← getMAX(listFree, category);
3       if res ≥ 50% × requiredResources then
4           entry ← reserve(res, category);
5           updateReservedList(category, entry);
6       else
7           remain ← getRemain(res);
8           listReservations←checkReservations(remain, listLI, listHI);
9           part ← (remain / size(listReservations));
10          newList ← Reduce(part, listReservations);
11          updateReservedList(newList);
12          entry ← reserve(res, remain);
13          updateReservedList(entry);
```

on the selected resource that can handle a resource adjustment are those that remain above the minimum threshold even if their computing capacity is reduced (step 8). In steps 9 and 10, it reduces the remaining capacity needed evenly from the reservation list. After adjustment, the list of reserved resources is updated (steps 11-13).

5.6 Conclusion

Current systems tend to approach resource and data management solutions independently when managing emerging applications and without focusing on the entire application workflows. With the increase in the quality and amount of data, it became challenging to meet the application's objectives when dealing with multiple data sources and limited resources of heterogeneous capabilities.

This chapter proposed a system that combines a data quality adaptation strategy and a workflow scheduling approach to support time-critical workflows with an accuracy constraint. It manages latency-accuracy trade-offs by controlling the quality of incoming data and distributing the workflows' tasks across the continuum based on their functionality and data input.

In Chapter 6, we validate the system's viability by implementing and deploying a data-driven application use case on *Grid'5000* testbed.

Chapter 6

Data-driven Management: the Case of Deep Learning Applications

Contents

6.1 Introduction

Deep Learning techniques have gained massive momentum in the industry over recent years, with a growing market estimated at 44.3 Billion USD by 2027 [196]. Deep Learning applications present a growing potential to extract knowledge from the analysis of streaming data with applications in numerous domains, including computer vision [197], speech recognition [198], precision medicine [199], and COVID-19 research [200].

The ecosystem supporting the emerging Deep Learning applications has become highly heterogeneous and geographically distributed. In addition, many Deep Learning applications require critical decision-making to be delivered in a timely manner with an appropriate quality even when dealing with multiple data sources [40]. Meeting the Quality of Service (QoS) requirements of Deep Learning applications on resources

with heterogeneous capacities highlights the need for data-driven management solutions and the implementation of trade-offs between the time and the quality of analysis. This chapter aims to validate the impact of data and workflow management strategies, presented in Chapter 5, on emerging intelligent data analytics systems. In particular, this chapter focuses on a time-sensitive object detection use case in an architecture relying on the use of constrained resources located at the Edge, the powerful Cloud, and along the data path.

The chapter is organized as follows: Section 6.2 presents an overview of data analytics systems with a focus on intelligent systems. The object detection use case is described in Section 6.3. Section 6.4 details the system model in the context of Deep Learning. Section 6.5 presents the evaluation of the system. Finally, Section 6.6 concludes this chapter.

6.2 Data Analytics Systems: An Overview

Data analytics is a science that provides complete data management, including the collection, cleansing, organization, storage, governance, and analysis. Given the growth in generated data, data analytics is widely used in different disciplines, such as healthcare [201], telecommunication [202], and smart cities [203]. Data analysis is a subcomponent of data analytics. It involves *processing* and *analyzing* the incoming data to derive from the information they hold meaningful insights that help create effective decision-making [204]. In short, data processing is about changing the original form of the data to a more usable and desired form using approaches such as data extraction and data filtering. Analyzing the processed data allows discovering the hidden knowledge within the incoming data.

Due to the current need to analyze data in a timely manner, data analysis is transitioning from being offline (a.k.a. batch processing) to online (a.k.a. stream processing) [205]. Analyzing a high volume of data in real-time requires the use of Machine Learning [206, 207]. This section describes data analysis workflows as well as Machine Learning with a focus on Deep Learning.

6.2.1 Data analysis workflow

The data analysis process consists of three main stages. Each stage provides a set of functionalities helping in the discovery of knowledge.

Pre-processing stage: it is an often neglected but essential step in the analysis process. It is responsible for preparing incoming data for analysis. The preparation tasks are specific to the incoming data, the goal of the data analytics application and the analysis algorithms used. This stage might include data cleansing functionalities and feature engineering [208]. The data cleansing transforms data to the appropriate shape and quality for analysis. It includes data filtering, merging, and others. Feature engineering allows the characterization of incoming data and the selection of the attributes to be used in the analysis. The extracted characteristics, such as data resolution, format, and size, contribute later to a better selection of the analysis pipeline.

Analysis stage: it is the stage of extracting the hidden knowledge from prepared data using appropriate algorithms. The selected algorithms depend on the analysis purpose and the type of the data (stream data/ historical data). The analysis tasks can classify objects, detect specific patterns, examine the dependency between attributes, predict future behaviors, etc. The algorithms used can include, among others, support vector machine (SVM) [209] and neural networks [210]. Section 6.2.2 presents briefly Machine Learning, in particular, Deep Learning.

Post-processing stage: it is responsible for processing the pieces of knowledge extracted from the analysis stage. Post-processing techniques are categorized into four classes [211]: knowledge filtering, interpretation and explanation, evaluation, and knowledge integration. ① *Knowledge filtering* corresponds to reduce the extracted knowledge and only keep those that are pertinent to the application's goal; ② *interpretation and explanation* refer to make the results understandable for end-users and machines. If a machine receives the extracted knowledge, the post-processing stage needs to formalize the results in a way to be integrated by other machines. However, if the knowledge is to be received directly by end-users, techniques of documentation and visualization are used; ③ *evaluation* corresponds to evaluating the performance of the analysis using particular criteria such as the error rate, the accuracy, and the latency; ④ *knowledge integration* refers to combining the knowledge extracted from several analysis algorithms to obtain new and more accurate knowledge.

6.2.2 Intelligent data analysis

The intelligent data analysis represents the analysis workflows that use Artificial Intelligence (AI) to extract knowledge and predict future behavior. Artificial Intelligence is any technique that aims to create intelligent computers able to address problems

usually treated by humans [212]. Machine Learning is a subset of Artificial Intelligence techniques that enables computers to learn automatically from previous experiences and perform analysis tasks without or with little human intervention using learning algorithms [213]. Several Machine Learning algorithms exist, such as K-Nearest Neighbors (K-NN), Support Vector Machines (SVM), Neural Networks (NN), and others [214]. These algorithms require data to be structured. Structured data corresponds to quantitative data that can be stored in tables of relational databases such as names, addresses, and dates. Among these techniques, Neural Networks are widely used. They represent a collection of connected units (*neurons*) organized in three types of layers: input layers, hidden layers, and output layers. The neuron is a mathematical function that takes an input and produces an output. The number of hidden layers can vary from one layer (shallow NNs) to multiple layers (deep NNs) based on the analysis needs. The neural networks learn continuously from incoming data to provide better analysis results with future data. The learning becomes deeper when the data are more complex. Complex data correspond to a large volume of unstructured data without a uniform format holding hidden patterns and interrelated features such as images, videos, and audios. Discovering patterns and understanding the complex relationships between variables is accomplished through multi-layered Neural Networks.

Deep Learning is a subset of Neural Networks that only uses multi-layered Neural Networks to build more efficient decision rules [215]. Due to the large number of hidden layers, Deep Learning can derive high-level functions from unstructured input information such as text classification [216], object detection [217], and speech recognition [218]. Many Deep Learning algorithms exist; each has a different number, type, and shape of NN's layers. An overview of several Deep Learning architectures is presented by Emmert-Streib *et al.* in [219], Lui *et al.* in [220] and van Veen in [210]. Among existing Deep Learning models, this chapter focuses on the "*You Only Look Once YOLO*" model as it is a part of the developed and implemented application use case. YOLO model is briefly presented in Section 6.2.3.

Deep Learning models exhibit higher analysis quality than traditional Machine Learning (a.k.a. shallow model) in most application scenarios such as music genre classification [221], landslide susceptibility assessment [222] and Predicting Cognitive Performance of Stroke Patients [223]. However, due to the high complexity of deep models, both their training and inference times are longer than those of shallow models. Thus, deploying intelligent data analysis into real-time data analytics systems requires the adoption of trade-off solutions between the latency and the accuracy of the analysis.

6.2.3 YOLO: You Only Look Once

Object detection consists of locating and understanding the objects located in digital images or videos. In the literature, two types of object detectors exist, one-stage detectors and two-stages detectors. The latter consists of accomplishing the detection task in two steps. First, it identifies the regions of interest within the image by scanning every location in it. A region of interest corresponds to boxes that might contain objects. Second, it adjusts the coordinates of identified boxes (so it better fits the detected objects) and classifies their detected objects. Due to the need to detect and classify objects in each box, two-stage detectors are considered slow and highly expensive in terms of computation. One of the popular two-stages detectors is Faster R-CNN, short for *Faster Region-based Convolutional Neural Networks* [224]. Furthermore, the one-stage detector consists of locating objects in images directly without scanning every location. Due to the one-step detection, one-stage detectors are considered time-efficient and can be used for real-time applications. *You Only Look Once YOLO* model is currently the best one-stage detector in Deep Learning as it outperforms existing detection models in terms of quality-speed [225]. Figure 6.1 shows the Venn diagram depicting the different areas of Artificial Intelligence and the YOLO model.

YOLO is a real-time object detection model based on Deep Learning. It detects and recognizes various objects and provides their class probabilities using a deep Neural Network called Convolutional Neural Network (CNN). Since its creation in 2015, it has been used in several disciplines such as healthcare [226, 227] and self-driving cars [228]. Four official versions of this model currently exist in the literature, and each version brought impressive improvements to the object detection field. Several variants of these

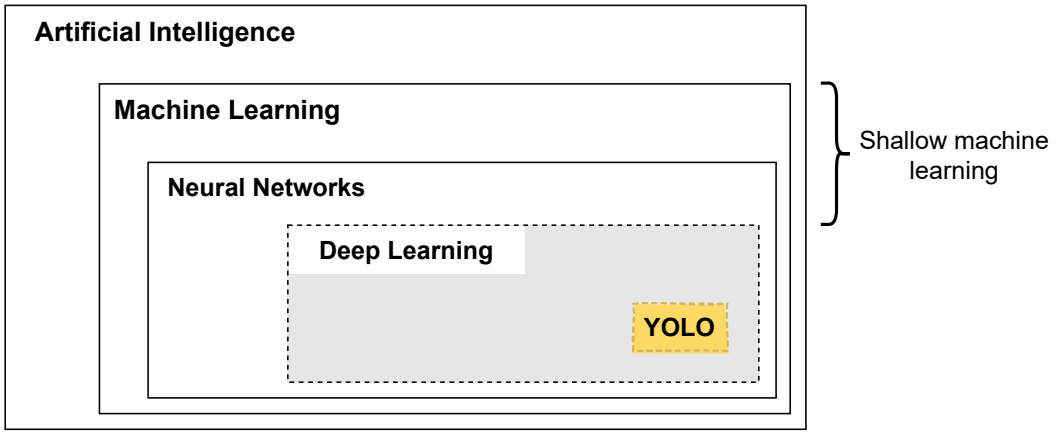


Figure 6.1 – A Venn diagram representing areas of Artificial Intelligence and the "You Only Look Once YOLO" model (inspired by Janiesch *et al.* [213])

versions are developed to meet different users' needs, such as YOLO-tiny-MSC [229], Poly-YOLO [230], YOLO-Fine [231], and others. The application use case adopted in this chapter uses the YOLO version 4 (YOLOv4) for object detection.

6.3 Object Detection Use Case

This section presents an intelligent data analysis application based on Deep Learning. It defines the set of microservices in the application workflow and applies the proposed categorization approach presented in Section 5.5.1 to classify each task.

6.3.1 Definition

The object detection use case is a time-sensitive application that identifies and locates objects in images or videos (sequence of images). Figure 6.2 shows the workflow of the object detection use case. As depicted in the figure, the workflow consists of 4 microservices:

1. `Resize` microservice is responsible for receiving incoming frames and modifying their resolution to suit the input data size of the following task. This service belongs to the pre-processing stage.

2. `YOLOv4-416` microservice is responsible for detecting objects in frames with resolution 416p using the YOLO model. It is a part of the analysis stage.

3. `YOLOv4-512` microservice is another microservice responsible for detecting objects in incoming frames using the YOLO model. Differently from `YOLOv4-416` microservice, this microservice analyses frames with resolution 512p. It is a part of the analysis stage.

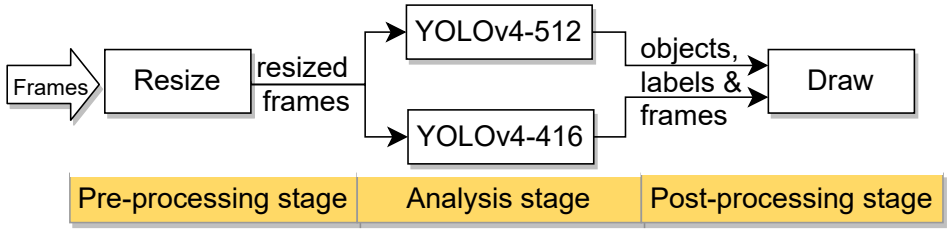


Figure 6.2 – Workflow of an object detection Deep Learning application showing the stages, dataflow and tasks.

4. `Draw` microservice is the last microservice in the workflow. It receives the frames and the analysis result from the analysis stage. It is responsible for marking the detected and identified objects on the frames and save them locally. This task belongs to the post-processing stage.

6.3.2 Tasks categorization

In Section 5.5.1, three categories of tasks were identified based on the resource requirements: Non-Intensive (NI), Low-Intensive (LI), and High-Intensive (HI) tasks. Based on our observations in deploying and running Deep Learning applications, tasks within a Deep Learning workflow can be classified as follows. The categorization of each task in the object detection use case is shown in Table 6.1.

- *In pre-processing and post-processing stages*: the tasks that do not manage their own database are considered NI. Database microservices are considered HI due to their storage demand. In the application use case, `Resize` microservice is considered as NI and `Draw` microservice as HI.
- *In the analysis stage:* the shallow Machine Learning models are LI. However, concerning Deep Learning models, their characteristics have an impact on their performance, such as their parameters and the number of Neural Network layers. In this work, we only consider the data resolution to categorize Deep Learning models. For each type of Deep Learning models in the analysis stage (YOLO, Faster RCNN, etc.), the lowest resolution task is considered LI. For example, in the object detection use case, YOLOv4-416 is considered LI, and YOLOv4-512 is considered HI.

Table 6.1 – The CPU, memory and storage requirements of the object detection use case. Legend: o: non-intensive; +: low-intensive; ++: high-intensive.

Resources / **Task**	CPU	Memory	Storage
`Resize`	o	o	o
`YOLOv4-416`	+	+	o
`YOLOv4-512`	++	++	o
`Draw`	o	o	++

In practice, users need to specify in the description of the submitted Deep Learning workflow the type of each task (pre-processing, shallow, etc.). The system will then automatically assign each type to a category, as mentioned above.

6.4 System Modeling: The Case of Deep Learning

The general formulation of the proposed system and the latency/accuracy performance models were presented in Section 5.3 page 73. This section presents extra details on the system modeling in the context of Deep Learning applications.

Workflow Modeling. The application workflow (W) is composed of 3 stages: Pre-processing stage (Pr), Analysis stage (A), and Post-processing stage (Po). Each stage is composed of a set of microservices providing the required functionalities. Let $F_{Pr} = \{m_i^{Pr}, 1 < i \leq x^{Pr}\}$, $F_A = \{m_i^A, 1 < i \leq x^A\}$ and $F_{Po} = \{m_i^{Po}, 1 < i \leq x^{Po}\}$ correspond to the set of microservices selected to provide the pre-processing, analysis, and post-processing functionalities, respectively. The workflow assigned to a user u can be presented as $W_u = \{m_i\} = \{F_{Pr} \cup F_A \cup F_{Po}\}$. The analysis stage has Z Deep Learning models. These models can be of different types (Faster RCNN, YOLO, etc.) or the same type but support different input quality (YOLO416, YOLO512, etc.). Let $D = \{d1, d2, ..., dZ\}$ represent the set of models in the system and $Q = \{q1, q2, ..., qL\}$ the set of data qualities supported.

End-to-End Latency Model. In the context of Deep Learning, the end-to-end latency of a job J from a data source u corresponds to the time needed to complete all the stages of the data analytics workflow. Based on Chapter 5 page 76, the latency model is formulated as:

$$T_J^u = \mu \cdot T_{reduce} + T_{Pr} + T_A + T_{Po},$$

where T_{reduce} refers to the time required to reduce the job's quality via the management microservice *Reduce* (see Figure 5.1 page 74). μ is a binary value that indicates whether a data adaptation was needed for this data source or not. T_{Pr}, T_A, and T_{Po} correspond to the time spent in completing the microservices F_{Pr}, F_A and F_{Po} of the pre-processing, analysis, and post-processing stages, respectively. The latency of each stage is formulated as the sum of its microservices' response time. Thus, the response

time of the processing stages are formulated as follows:

$$T_{Pr} = \sum_{i=1}^{x^{Pr}} RT(m_i^{Pr}) \quad \text{where } m_i^{Pr} \in F_{Pr}$$
$$T_A = \sum_{i=1}^{x^A} RT(m_i^A) \quad \text{where } m_i^A \in F_A$$
$$T_{Po} = \sum_{i=1}^{x^{Po}} RT(m_i^{Po}) \quad \text{where } m_i^{Po} \in F_{Po}$$

The response time of a microservice is defined as $RT(m_i) = \alpha \cdot T_D + Trans(m_j, m_i) + T_E$. The transfer time of the job between the data producer and data consumer $Trans(m_j, m_i)$ depends on the data size S (in Mbits) to be sent over the network (details in formula 5.3 page 77). In the object detection use case, the data size is given by $S = \gamma \cdot (w \times h)$ with (w, h) correspond to the width and height of the frame, and γ is the number of bits required to represent the information carried by one pixel. S might correspond to the original data size or the reduced size if the data quality adaptation was applied. Additionally, the execution time T_E depends on the throughput capacity of the chosen microservices (details page 77). In the analysis stage A of the Deep Learning workflow, the throughput depends on the chosen Deep Learning models and the quality of incoming data. As experimentally proved in [15] and [232], the models provide a slower analysis speed with high data quality than with low-quality data. In addition, [233] showed that with the same data, some Deep Learning models perform faster than others.

Analysis Accuracy Model. The analysis accuracy of the object detection use case corresponds to the detection accuracy of Deep Learning models. It is measured using the metric *F1_score*, a weighted average of the *Precision* and *Recall* evaluation metrics. It is presented as follows:

$$F1\_score = 2 \times \frac{Precision \times Recall}{Precision + Recall},$$

where *Precision* corresponds to the ratio of correctly predicted positive observations among the total predicted positive observations and *Recall* is the ratio of correctly predicted positive observations among all observations. These metrics are formulated

as follows:

$$Precision = \frac{TP}{(TP + FP)} \qquad Recall = \frac{TP}{(TP + FN)}$$

where True Positive (TP) refers to the total number of correctly detected objects; False Positive (FP) refers to the total number of incorrectly detected objects; finally, False Negative (FN) refers to the total number of undetected objects. The *Intersection over Union (IoU)* metric [234] is used to identify the TP, FP, and FN. IoU is a number from 0 to 1 that specifies the amount of overlap between the predicted and ground truth (i.e., actual) objects. If $IoU \geq 0.5$ and the label is accurate, the detection is TP. However, if $IoU \geq 0.5$ and the label is false, the detection is FP. In case the $IoU < 0.5$ and the label is accurate, the detection is FP.

It has been experimentally observed in [15, 232, 233] that the quality of incoming data and the chosen Deep Learning model impact the accuracy of the results. As showed in [15, 232], the relationship between accuracy and data quality is formulated as a concave exponential function of three coefficients $\mathcal{E}(q, d) = \alpha 1_d - \alpha 2_d \times e^{-q/\alpha 3_d}$. It reflects that a higher data quality q produces a better analysis accuracy, and the analysis accuracy gain decreases at a high quality. In this work, $\{\alpha 1, \alpha 2, \alpha 3\}$ are constant coefficients of a Deep Learning model d.

6.5 Evaluation of Latency Optimization

The evaluation aims to show the impact of the proposed system on the end-to-end processing latency of the object detection use case (Figure 6.2) when dealing concurrently with incoming data. The use case is representative of the general Deep Learning problem as it deals with tasks of different categories and requires leveraging limited and heterogeneous resources to achieve real-time performance. This section presents the evaluation methodology and the results of a set of experiments. In addition, we discuss some insights and takeaways from the evaluation.

6.5.1 Methodology overview

6.5.1.1 Testbed

The evaluation of the system is performed on the large-scale platform *Grid'5000* [173]. It represents a distributed testbed designed to support experimental-driven research in

parallel and distributed systems. The experimental setup of the Edge-to-Cloud continuum consists of a total of 14 nodes in the *nova* cluster. Each node is originally equipped with two processors of 8 cores each, 64 GB memory, and 598 GB storage (HDD). Among the reserved nodes, there are 11 Edge nodes, 2 Fog nodes, and 1 Cloud node. The original node capacity is not entirely allocatable/available for Edge, Fog, and Cloud nodes. The allocatable capacity of each type of node is described in Table 6.2. The network connection between nodes of the same and different types has a delay and bandwidth as given in Table 6.3. The emulation of the continuum on *Grid'5000* is achieved via the framework E2Clab [235]. It provides, among others, a network emulation feature that allows the specification of Edge-to-Cloud communication constraints (delay, loss, and bandwidth). In addition, it permits the virtualization of the cluster's physical network to separate Edge, Fog, and Cloud nodes into independent virtual networks/subsets.

The system has 19 data sources on the Edge. They generate 25 frames per second in the default resolution of $512p$. The frames are from the COCO2017 validation dataset [236] of size 1GB. It is an object detection dataset with 91 defined classes.

Table 6.2 – Resource capacity of Edge, Fog and Cloud nodes.

Layer	Cores	RAM (GiB)	Storage HDD
Edge	1	2	2GB
Fog	4	32	20GB
Cloud	8	64	500GB

Table 6.3 – Delay and bandwidth of network connections between nodes.

Layer	Average Delays(ms)	Uplink Bandwidth
Edge-Edge	1	1Gbps
Fog-Fog	1	1Gbps
Cloud-Cloud	1	10Gbps
Edge-Fog	4	30Mbps
Fog-Cloud	5	10Gbps

6.5.1.2 Platform

The platform is built on top of Kubernetes 1.15. We implemented and deployed the object detection use case presented in Figure 6.3. The two object detector models in the analysis stage `YOLOv4-512` and `YOLOv4-416` are pre-trained on the COCO dataset and provide an accuracy AP_{50} equals to 64.9% and 62.7%, respectively.

Based on the data-driven scheduling approach presented in this work, the mapping of the application use case on the available resources is as follows: `Resize` microservice on the Edge, `YOLOv4-416` microservice on the Fog, and `YOLOv4-512` and `Draw` microservices on the Cloud. The platform is illustrated in Figure 6.3. Since there are few tasks in the deployed object detection use case, each task can reserve the entire computing capacity of the node on which it is deployed. Regarding the two services placed on the Cloud, they share the resources by half. The `Resize` microservice has 11 instances distributed on all the Edge nodes, `YOLOv4-416`, `YOLOv4-512`, and `Draw` microservices have one instance each.

The system only supports two data qualities, 512*p* and 416*p*. Therefore, the data adaptation strategy consists of selecting whether the quality of data entering the system remains at 512*p* or should be reduced to 416*p*. Thus, due to the data quality adaptation strategy and the data-driven service discovery adopted, the deployed microservices constitute two possible processing workflows for system's users:

- *Cloud-only* configuration: it uses the `Resize` service on the Edge for pre-processing, the `YOLOv4-512` model, and `Draw` service located on the Cloud for the analysis and post-processing stages, respectively. Data sources assigned to this pipeline are those generating data of quality 512*p*.

- *Fog-only* configuration: it uses the `Resize` service on the Edge for pre-processing, the `YOLOv4-416` model on the Fog for the analysis, and the `Draw` service located on the Cloud for the post-processing stage. Data sources assigned to this pipeline are those with adapted data of quality 416*p*.

The system evaluation consists of four experiments in total, each running for 20 minutes. Experiments 1 and 2 use all system data sources generating data in the default frame rate and resolution. Experiments 3 and 4 use only one data source. The purpose of these experiments is to measure the average makespan and accuracy of the

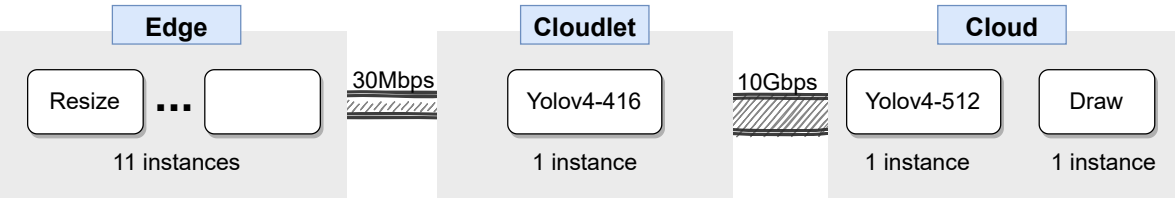


Figure 6.3 – The distribution of the Deep Learning object detection use case across the Edge-to-Cloud continuum.

Deep Learning object detection use case during runtime with and without the latency-accuracy trade-off solution. In experiments 1 and 3, no data adaptation strategy is used. As a result, the quality of generated data stays equal to 512×512. Due to that, all data sources are assigned to the pipeline with a Cloud-only analysis configuration. However, in the second experiment, the data adaptation strategy was used once. Among all data sources, the quality of the generated data by only 1 data source is reduced to 416×416. So, unlike the rest, this data source is assigned to the pipeline with the Fog-only analysis configuration. In experiment 4, the data adaptation strategy is applied to the single data source used. Results of these experiments are presented in Section 6.5.2.

6.5.2 Evaluation results

Figure 6.4 shows the average system makespan variation with 19 data sources with and without using the data adaptation. In experiment 1, without any data adaptation, the system takes up to around 5.7 hours to analyze the data generated by 19 data sources during a 20 minutes test. However, in experiment 2, when applying the data adaptation strategy on one data source, the average system makespan was reduced to around 2.6 hours. The gain in the average system makespan between experiments 1 and 2 is shown in Figure 6.5. Results show that using the data adaptation strategy when dealing with high load helped accelerate the system analysis up to 54.4% (≈3.1 hours).

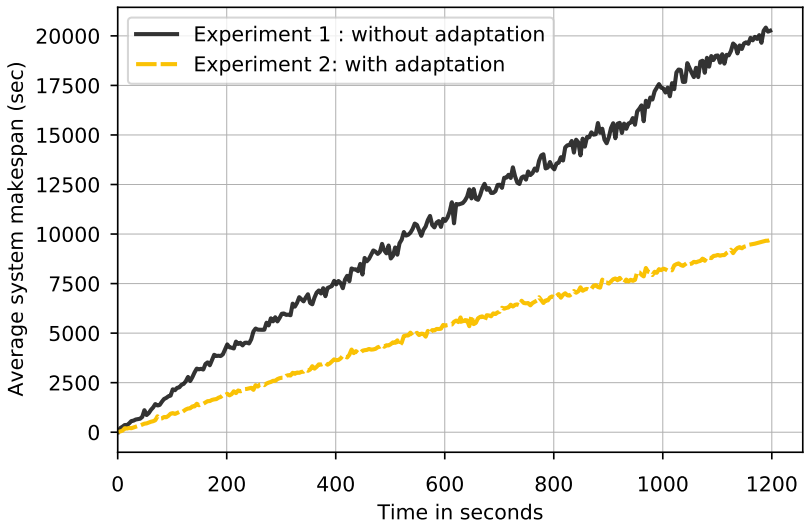


Figure 6.4 – With 19 data sources, the system makespan with data adaptation in experiment 2 is lower than in experiment 1, where no trade-off solution is used.

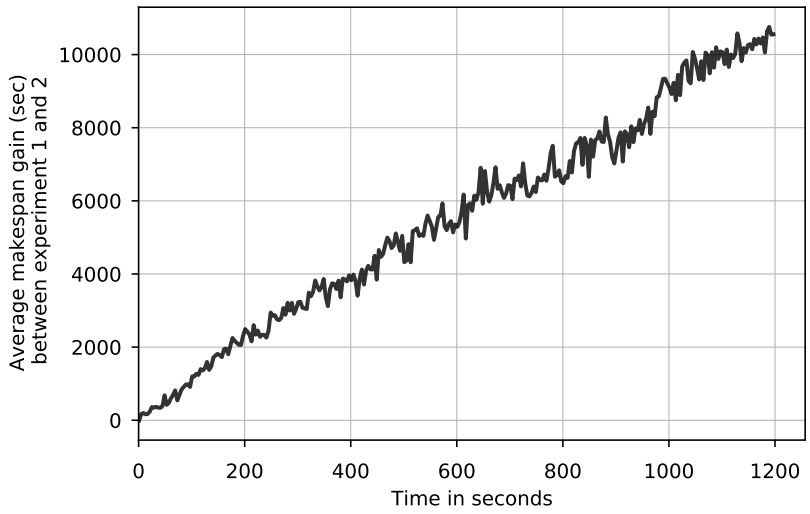


Figure 6.5 – Up to 54.4% gain in average system makespan between experiment 2 (with data adaptation) and experiment 1 (without data adaptation).

Figure 6.6 shows the variation of average system *F1-score* with and without the data adaptation strategy. Results show that system accuracy decreased from 71.19% to 63.82% after reducing the quality of one data source from 512×512 to 416×416. The system accuracy remains higher than the default accuracy threshold fixed to 50%. The obtained system accuracy depends mainly on the models used. In this work, the accuracy results are for the pre-trained YOLOv4 models.

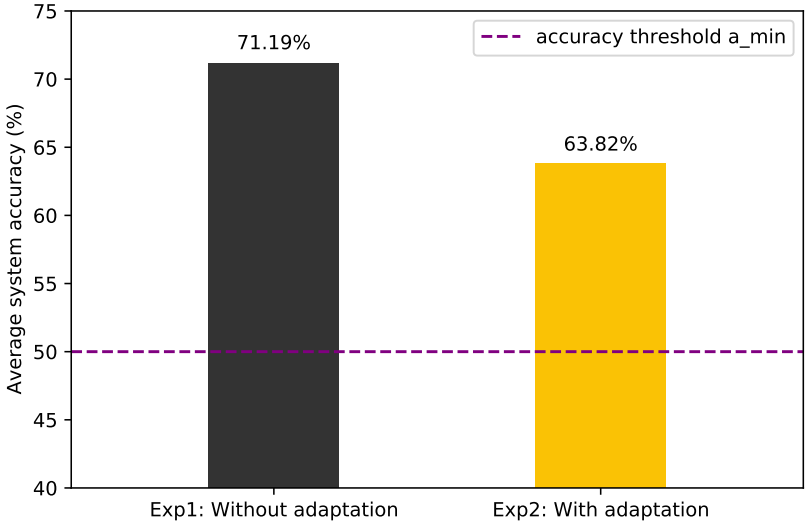


Figure 6.6 – With 19 data sources, average system accuracy decreased in experiment 2 (with data adaptation) compared to experiment 1 (without data adaptation). Despite this, it remains higher than a fixed threshold equals to 50%.

Figure 6.7 shows the average system makespan of a single data source with and without using the data quality adaptation strategy. In experiment 3, without any data adaptation, the system takes up to around 1 hour to analyze all the data generated by the single data source. However, when applying the data adaptation strategy in experiment 4, the average system makespan is around 1.3 hours.

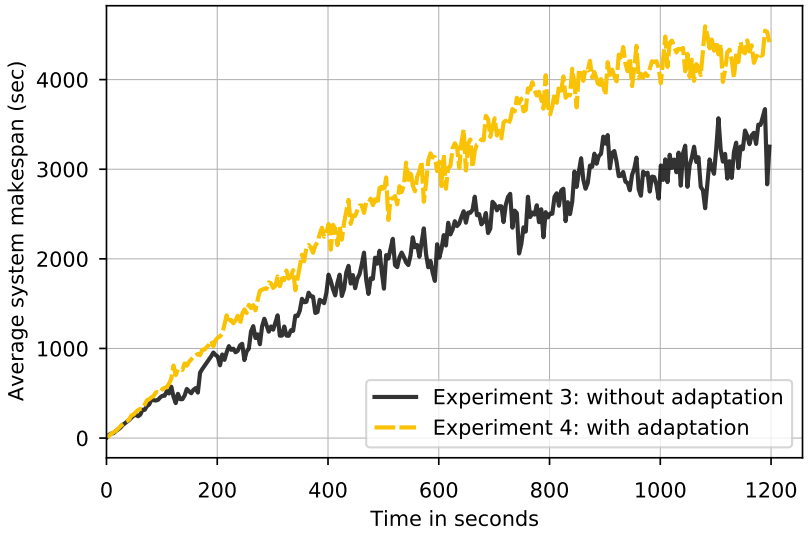


Figure 6.7 – With a single data source, the average system makespan with data adaptation in experiment 4 is higher than in experiment 3, where no trade-off solution is used.

6.5.3 Discussion

During the system evaluation, we observed that when the Fog-Cloud network is constrained due to high load, the Edge-only configuration for analysis pipelines can be an interesting approach, as it is shown in experiment 2. However, if no high load in the system as in experiments 3 and 4, the Cloud of 10Gbps bandwidth performs better in terms of makespan than the Edge-only pipeline configuration. This indicates that, for Deep Learning applications, applying the data adaptation strategy to maximize the utility function is only possible when the Fog-Cloud network performance is constrained. Otherwise, the Cloud-only pipeline configuration is the better choice in terms of average system makespan.

6.6 Conclusion

Deep Learning applications are among the emerging applications facing challenges in meeting their real-time requirements when dealing with multiple data sources and limited resources of heterogeneous capabilities. This chapter illustrated the viability of the proposed data quality and workflow management strategies presented in Chapter 5 by implementing and deploying an object detection use case on *Grid'5000*. Evaluation results showed a gain in average system makespan reaching up to 54.4% compared to a Cloud-only pipeline configuration in a multi-user scenario with an analysis accuracy remaining higher than a fixed threshold.

Chapter 7

Conclusion and Perspectives

Contents

7.1 Summary

The development of devices and network technologies is driving the evolution of current applications towards processing generated data to extract information capable of strengthening businesses, academia, and the quality of human life. Among emerging applications, Deep Learning applications present expectations of near-real-time processing to maintain the effectiveness of their results. The need to guarantee the latency requirement has changed the application and infrastructure designs traditionally adopted to deliver services close to data sources while ensuring their computing needs. Running Deep Learning applications on current infrastructures has several challenges. This thesis focused on three challenges related to the applications, data producers, and infrastructures:

1. **Decoupled and heterogeneous services.** Emerging microservice-based applications are designed as a set of independent services managed by different entities and not designed to work together automatically. In addition, several

implementations exist for the same functionality where each provides a particular Quality of Service, accepts specific data input, and often does not have explicit identifiers. Therefore, the ability to discover available services by their identifiers is not guaranteed, and the selection of one implementation over the other can impact the real-time performance of applications.

2. **Fluctuating incoming load.** The number of data producers is dynamic where new data sources can join the system and others leave. These data sources can have different data characteristics (such as resolution and type). Dealing with multiple users generating data concurrently increases the latency of deployed applications. Thus, with the fluctuating load during runtime, there is no guarantee of the real-time requirement.

3. **Heterogeneous and distributed infrastructures.** Existing infrastructure designs consist of geographically distributed resources with different computing and bandwidth capacities. Additionally, Deep Learning applications consist of processing tasks with different computing and network requirements. Therefore, deploying the processing tasks of Deep Learning applications on heterogeneous resources impact their processing latency.

This thesis addressed the challenges mentioned above to create a system for Deep Learning applications in distributed infrastructures through several contributions. Section 7.2 describes the contributions made in this thesis. Then, in Section 7.3, we discuss the perspectives that our research opens.

7.2 Contributions

7.2.1 Designing scalable data-driven service discovery system

Traditional service discovery approaches look up the location of specific services using their identifiers. Adopting traditional discovery approaches in current microservice-based applications is inefficient as it prevents discovering the microservices without identifiers and selecting the appropriate implementations that fit users' needs. In addition, existing approaches to reduce the impact of fluctuating load on the applications consist of scaling services based on metrics related to infrastructure or application without considering traffic heterogeneity. In Chapter 3 and Chapter 4, we studied the

challenges of discovering microservices based on incoming data rather than services identifiers while guaranteeing a Quality of Service, even with fluctuating load, using data-related metrics.

Data-driven service discovery framework

Chapter 3 presented a data-driven service discovery approach that allows data producers (external users or microservices) to discover the functionalities provided without the need for redevelopment. The proposed approach is built on a data-centric microservices description and a Peer-to-Peer data-driven architecture to ensure the system's resiliency and cover wide geographical areas. We described in detail the steps of the discovery using an illustrative example.

Data-driven microservice adaptation scheme

Chapter 4 presented an adaptation scheme that scales deployed microservices according to the load. Unlike existing scaling approaches, this approach manages microservices by data type. The increase of load with specific data characteristics triggers a scale out of microservices targeted by this load. However, the microservices designed to work with other data types are not affected. Similarly, a drop in the load triggers a decrease in the capacity of targeted microservices to prevent the misuse of the system's resources. Since the system's computing capacity is limited, it might resort to load shedding in order to prevent performance degradation.

Integration into the Istio Service Mesh project

Leveraging a Service Mesh abstracts the complexity of microservices interactions. However, existing Service Mesh projects do not support the data-driven discovery approach. In Chapter 4, we integrated the proposed discovery framework into the *Istio* Service Mesh project [91]. We conducted a set of experiments to evaluate the proposed framework on *Grid'5000* [173]. Results showed that the platform could adapt and maintain an acceptable system latency and percentage of accepted requests during runtime. In addition, the system is effectively using the system resources by adapting the reserved resources according to the load.

7.2.2 Leveraging a latency-accuracy trade-off approach

Only adopting scaling and load shedding algorithms to manage the latency of Deep Learning applications is insufficient when dealing with limited resources of heteroge-

neous capabilities and a high volume of incoming data. Hence, there is a need to adopt data and workflow management approaches. Existing work tends to target these two aspects independently, rarely manages the entire Deep Learning workflow, and lacks general formulations of proposed approaches. Chapter 5 targeted these limitations by presenting a system design that enables trade-offs between the time and quality of data processing. The system was then validated in Chapter 6 in the context of Deep Learning using an object-detection use case.

Latency-accuracy trade-off approach based on tasks categorization

Chapter 5 presented a general formulation of a latency-accuracy trade-off method that combines a data quality adaptation and workflow scheduling approaches. The quality adaptation approach selects the data quality configuration for existing data sources. The selected qualities provide the fastest workflow processing latency without an accuracy violation. The workflow scheduling approach relies on the use of constrained resources located at the Edge, the Fog, and the powerful Cloud nodes. The scheduling priorities are specified using a task categorization strategy based on the functionality provided and the accepted data quality.

Experimental evaluation in the context of Deep Learning

We conducted an experimental evaluation on the testbed *Grid'5000* to validate the proposed trade-off method on a Deep Learning application use case. The evaluation methodology and results are presented in Chapter 6. The evaluation results showed a gain in average system makespan reaching up to 54.4% when using the proposed trade-off method compared to a Cloud-only workflow configuration in a multi-user scenario. During the evaluation, the application accuracy remains higher than a fixed threshold.

7.3 Perspectives

Our work opens several perspectives. In this section, we discuss the most promising ones, which, if addressed, will extend the capability of data-driven management in current systems.

Dynamic Latency-accuracy trade-off approach

Chapter 5 presented a latency-accuracy trade-off approach based on a combination of a data quality adaptation and a workflow scheduling strategies. The proposed approach

is static, as the number of reserved resources (Edge, Fog, and Cloud nodes) and the complexity of incoming data are fixed during runtime. In addition, the adaptation strategy presented in this work and in the majority of existing online configuration approaches [100, 101] are triggered periodically. During runtime, in each time period, the system checks whether a reconfiguration is needed. However, in real-life deployments, the infrastructures are dynamic where new nodes can join the network, and others leave. Additionally, the content of incoming data is continuously changing and the more complex it is, the more resources are needed. For example, in the object detection use case defined in Chapter 6, images can have different sizes and numbers of objects. Therefore, analyzing images with one detected object of large size is less resource-intensive than processing an image with 100+ small objects. Thus, in practice, even though periodic approaches help optimize the system performance, they cannot prevent the performance degradation that might happen within the specified period. Also, using short periods is not efficient due to the high computing cost of the profiling process [180]. Therefore, designing approaches to cope with the dynamicity of current environment is required.

Resource assignment tool for data-driven workflows

Chapter 5 and Chapter 6 presented and demonstrated a task categorization strategy. In this strategy, the amount of computing resources assigned to each task category are predefined based on our observations during the evaluation of the Deep Learning application. An interesting research direction is the design of a resource assignment tool for Deep Learning workflows. It considers incoming data, type of tasks, and resource heterogeneity to generate a resource assignment plan. Furthermore, this tool can leverage artificial intelligence techniques to select an optimal resource assignment plan providing the best analysis performance. This tool can be integrated into our system to automatically distribute workflows on heterogeneous resources with minimum resource waste.

Control plane for Edge-to-Cloud Continuum

The Edge-to-Cloud continuum is a promising design for emerging applications. While significant research and development exist at specific places along this continuum, few existing approaches include the entire computing continuum as a collective whole [18]. Thus, the Edge-to-Cloud continuum did not establish concrete maturity yet due to the lack of management tools. The extreme heterogeneity in the resources and ap-

plications requires the design of a control plane able to balance the requirements of deployed applications with what is possible in practice using trade-off strategies. This control plane must be external to the applications' deployments. This thesis focuses on the latency-accuracy trade-off to manage the latency constraint. However, for most emerging applications, other trade-off solutions are also required to deal with other performance constraints such as accuracy-energy consumption trade-off, energy-latency trade-off, and/or cost-latency trade-off, among others.

Leverage hybrid nanoservice/microservice applications

In data-driven management, extracting early insights from data before being transferred across the continuum is a key for performance optimizations in emerging applications. However, deploying processing services near data producers is hampered by the constrained resource capacity at the Edge of the network and the stack of resources (CPU, memory, storage, and others) required by microservices. The concept of *nanoservice* is a recently emerging paradigm to build applications in Cloud-based systems with constrained resource capacity. While the microservice is defined as a single "functionality" within the application, the nanoservice is emerging as a lightweight microservice dedicated to performing a "single-purpose granular operation" [237]. In the literature, the boundary of an operation is either logical [238] or physical [239] and requires fewer resources stack than normal microservices. Hence, designing approaches that leverage nanoservices- and microservices-based workflows can increase the processing capabilities at the edge of the network.

Bibliography

[1] Kevin Fauvel, Daniel Balouek-Thomert, Diego Melgar, Pedro Silva, Anthony Simonet, Gabriel Antoniu, Alexandru Costan, Véronique Masson, Manish Parashar, Ivan Rodero, et al. A distributed multi-sensor machine learning approach to earthquake early warning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 403–411, 2020.

[2] Marzieh Fathi, Mostafa Haghi Kashani, Seyed Mahdi Jameii, and Ebrahim Mahdipour. Big data analytics in weather forecasting: a systematic review. *Archives of Computational Methods in Engineering*, pages 1–29, 2021.

[3] Andre Luckow, Kartik Rattan, and Shantenu Jha. Exploring task placement for edge-to-cloud applications using emulation. In *2021 IEEE 5th International Conference on Fog and Edge Computing (ICFEC)*, pages 79–83. IEEE, 2021.

[4] Andrés Villa-Henriksen, Gareth T.C. Edwards, Liisa A. Pesonen, Ole Green, and Claus Aage Grøn Sørensen. Internet of things in arable farming: Implementation, applications, challenges and potential. *Biosystems Engineering*, 191:60–84, 2020.

[5] DOMO. Data never sleeps 9.0. `https://www.domo.com/learn/infographic/data-never-sleeps-9`, 2021. Last checked October 2021.

[6] Gonçalo Carvalho, Bruno Cabral, Vasco Pereira, and Jorge Bernardino. Edge computing: current trends, research challenges and future directions. *Computing*, 103(5):993–1023, 2021.

[7] Pooyan Habibi, Mohammad Farhoudi, Sepehr Kazemian, Siavash Khorsandi, and Alberto Leon-Garcia. Fog computing: a comprehensive architectural survey. *IEEE Access*, 8:69105–69133, 2020.

[8] Carla Mouradian, Diala Naboulsi, Sami Yangui, Roch H. Glitho, Monique J. Morrow, and Paul A. Polakos. A comprehensive survey on fog computing: State-of-the-art and research challenges. *IEEE Communications Surveys Tutorials*, 20(1):416–464, 2018.

[9] accenture.com. "cloud native: Application development at the speed of business". `https://www.accenture.com/_acnmedia/PDF-90/Accenture-Cloud-Native-POV-Final.pdf`. Last checked September 2021.

[10] Martin Fowler James Lewis. Microservices, a definition of this new architectural term. `https://www.martinfowler.com/articles/microservices.html`. Last checked August 2021.

[11] The International Data Corporation (IDC). "idc futurescape: Worldwide it industry 2019 predictions". `https://www.idc.com/research/viewtoc.jsp?containerId=US44403818`. Last checked September 2021.

[12] Eduard Gibert Renart, Daniel Balouek-Thomert, and Manish Parashar. An edge-based framework for enabling data-driven pipelines for iot systems. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 885–894, 2019.

[13] Anurag Choudhry and Anshu Premchand. Microservices and devops for optimal benefits from iot in manufacturing. In *Proceedings of International Conference on Recent Trends in Machine Learning, IoT, Smart Cities and Applications*, pages 375–384. Springer, 2021.

[14] Muhammad Ali, Ashiq Anjum, M Usman Yaseen, A Reza Zamani, Daniel Balouek-Thomert, Omer Rana, and Manish Parashar. Edge enhanced deep learning system for large-scale video stream analytics. In *2018 IEEE 2nd International Conference on Fog and Edge Computing (ICFEC)*, pages 1–10. IEEE, 2018.

[15] C. Wang, S. Zhang, Y. Chen, Z. Qian, J. Wu, and M. Xiao. Joint configuration adaptation and bandwidth allocation for edge-based real-time video analytics. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, pages 257–266, 2020.

[16] Eduard Gibert Renart, Javier Diaz-Montes, and Manish Parashar. Data-driven stream processing at the edge. In *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*, pages 31–40, 2017.

[17] Yuuichi Teranishi, Takashi Kimata, Hiroaki Yamanaka, Eiji Kawai, and Hiroaki Harai. Dynamic data flow processing in edge computing environments. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 935–944. IEEE, 2017.

[18] Pete Beckman, Jack Dongarra, Nicola Ferrier, Geoffrey Fox, Terry Moore, Dan Reed, and Micah Beck. Harnessing the computing continuum for programming our world. *Fog Computing: Theory and Practice*, pages 215–230, 2020.

[19] Ahmed Oussous, Fatima-Zahra Benjelloun, Ayoub Ait Lahcen, and Samir Belfkih. Big data technologies: A survey. *Journal of King Saud University - Computer and Information Sciences*, 30(4):431–448, 2018.

[20] Ravi Pratap Singh, Mohd Javaid, Ravinder Kataria, Mohit Tyagi, Abid Haleem, and Rajiv Suman. Significant applications of virtual reality for covid-19 pandemic. *Diabetes & Metabolic Syndrome: Clinical Research & Reviews*, 14(4):661–664, 2020.

[21] Fabio Arena, Giovanni Pau, and Mario Collotta. A survey on driverless vehicles: From their diffusion to security. *J. Internet Serv. Inf. Secur*, 8:1–19, 2018.

[22] Sidi Ahmed Mahmoudi, Mohammed Amin Belarbi, Saïd Mahmoudi, Ghalem Belalem, and Pierre Manneback. Multimedia processing using deep learning technologies, high-performance computing cloud resources, and big data volumes. *Concurrency and Computation: Practice and Experience*, 32(17):e5699, 2020.

[23] Ch Cheevers, M Bugajski, A Luthra, S McCarthy, P Moroney, and K Wirick. Virtual and augmented reality–how do they affect the current service delivery and home and network architectures? arris enterprises llc, 2016.

[24] Yuezhi Zhou, Di Zhang, and Naixue Xiong. Post-cloud computing paradigms: a survey and comparison. *Tsinghua Science and Technology*, 22(6):714–732, 2017.

[25] Alef, the 5g edge api company. `https://alefedge.com/about-alefedge/`. Last checked August 2021.

[26] Raj Jain and Subharthi Paul. Network virtualization and software defined networking for cloud computing: a survey. *IEEE Communications Magazine*, 51(11):24–31, 2013.

[27] Jalal H Kiswani, Sergiu M Dascalu, and Frederick C Harris Jr. Cloud computing and its applications: A comprehensive survey. *International Journal of Computer Applications IJCA*, 28, 2021.

[28] Mamta Agiwal, Abhishek Roy, and Navrati Saxena. Next generation 5g wireless networks: A comprehensive survey. *IEEE Communications Surveys & Tutorials*, 18(3):1617–1655, 2016.

[29] Iman Azimi, Arman Anzanpour, Amir M. Rahmani, Pasi Liljeberg, and Tapio Salakoski. Medical warning system based on internet of things using fog computing. In *2016 International Workshop on Big Data and Information Security (IWBIS)*, pages 19–24, 2016.

[30] Shreshth Tuli, Nipam Basumatary, Sukhpal Singh Gill, Mohsen Kahani, Rajesh Chand Arya, Gurpreet Singh Wander, and Rajkumar Buyya. Healthfog: An ensemble deep learning based smart healthcare system for automatic diagnosis of heart diseases in integrated iot and fog computing environments. *Future Generation Computer Systems*, 104:187–200, Mar 2020.

[31] Andrea Pazienza, Roberto Anglani, Giulio Mallardi, Corrado Fasciano, Pietro Noviello, Corrado Tatulli, and Felice Vitulano. Adaptive critical care intervention in the internet of medical things. In *2020 IEEE Conference on Evolving and Adaptive Intelligent Systems (EAIS)*, pages 1–8. IEEE, 2020.

[32] A. Beloglazov, R. Buyya, Young Choon Lee, and Albert Y. Zomaya. A taxonomy and survey of energy-efficient data centers and cloud computing systems. *ArXiv*, abs/1007.0066, 2011.

[33] Hamed Tabrizchi and Marjan Kuchaki Rafsanjani. A survey on security challenges in cloud computing: issues, threats, and solutions. *The journal of supercomputing*, 76(12):9493–9532, 2020.

[34] Hedda R Schmidtke. Location-aware systems or location-based services: a survey with applications to covid-19 contact tracking. *Journal of Reliable Intelligent Environments*, 6(4):191–214, 2020.

[35] Yasmine Harbi, Zibouda Aliouat, Allaoua Refoufi, and Saad Harous. Recent security trends in internet of things: A comprehensive survey. *IEEE Access*, 2021.

[36] Daniel Balouek-Thomert, Eduard Gibert Renart, Ali Reza Zamani, Anthony Simonet, and Manish Parashar. Towards a computing continuum: Enabling edge-to-cloud integration for data-driven workflows. *The International Journal of High Performance Computing Applications*, 33(6):1159–1174, 2019.

[37] Jaber Almutairi and Mohammad Aldossary. A novel approach for iot tasks offloading in edge-cloud environments. *Journal of Cloud Computing*, 10(1):1–19, 2021.

[38] Seyed Hossein Mortazavi, Mohammad Salehe, Carolina Simoes Gomes, Caleb Phillips, and Eyal de Lara. Cloudpath: A multi-tier cloud computing framework. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[39] Zhengyuan Pang, Lifeng Sun, Zhi Wang, E. Tian, and Shiqiang Yang. A survey of cloudlet based mobile computing. *2015 International Conference on Cloud Computing and Big Data (CCBD)*, pages 268–275, 2015.

[40] Daniel Balouek-Thomert, Ivan Rodero, and Manish Parashar. Harnessing the computing continuum for urgent science. *SIGMETRICS Perform. Eval. Rev.*, 48(2):41–46, November 2020.

[41] Dragi Kimovski, Dijana C Bogatinoska, Narges Mehran, Aleksandar Karadimce, Natasa Paunkoska, Radu Prodan, and Ninoslav Marina. Cloud—edge offloading model for vehicular traffic analysis. In *2020 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*, pages 746–753. IEEE, 2020.

[42] Muhammad Ijaz, Gang Li, Huiquan Wang, Ahmed M El-Sherbeeny, Yussif Moro Awelisah, Ling Lin, Anis Koubaa, and Alam Noor. Intelligent fog-enabled smart healthcare system for wearable physiological parameter detection. *Electronics*, 9(12):2015, 2020.

[43] Scott Carey. What is cloud-native? the modern way to develop software. `https://www.infoworld.com/article/3281046/what-is-cloud-native-the-modern-way-to-develop-software.html`. Last checked August 2021.

[44] Frank Moley. Microservices foundations. `https://www.linkedin.com/learning/microservices-foundations/bounded-context`. Last checked August 2021.

[45] Mark Richards. Software architecture patterns: Layered architecture. `https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html`. Last checked August 2021.

[46] Gunjan Samtani and Dimple Sadhwani. Web services and application frameworks (. net and j2ee). In *Web Services Business Strategies and Architectures*, pages 273–289. Springer, 2002.

[47] Jonas Fritzsch, Justus Bogner, Alfred Zimmermann, and Stefan Wagner. From monolith to microservices: A classification of refactoring approaches. In *International Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, pages 128–141. Springer, 2018.

[48] Olaf Zimmermann. An architectural decision modeling framework for service-oriented architecture design. 2009.

[49] Naghmeh Niknejad, Waidah Ismail, Imran Ghani, Behzad Nazari, Mahadi Bahari, et al. Understanding service-oriented architecture (soa): A systematic literature review and directions for further investigation. *Information Systems*, 91:101491, 2020.

[50] Mike P Papazoglou and Willem-Jan Van Den Heuvel. Service oriented architectures: approaches, technologies and research issues. *The VLDB journal*, 16(3):389–415, 2007.

[51] Uwe Zdun, Carsten Hentrich, and Wil MP Van Der Aalst. A survey of patterns for service-oriented architectures. *International journal of Internet protocol technology*, 1(3):132–143, 2006.

[52] Rama Akkiraju, Joel Farrell, John Miller, Meenakshi Nagarajan, Marc-Thomas Schmidt, Amit Sheth, and Kunal Verma. "web service semantics—wsdl-s". `https://www.w3.org/Submission/WSDL-S/`. Accessed July, 2021.

[53] Marc Hadley. "web application description language". `https://www.w3.org/Submission/wadl/`. Last checked July 2021.

[54] Jacek Kopeckỳ, Tomas Vitvar, Dieter Fensel, and Karthik Gomadam. hrests & microwsmo. *CMS WG Working Draft*, 2009.

[55] Omer Aziz, Muhammad Shoaib Farooq, Adnan Abid, Rubab Saher, and Naeem Aslam. Research trends in enterprise service bus (esb) applications: A systematic mapping study. *IEEE Access*, 8:31180–31197, 2020.

[56] Fabrizio Montesi and Janine Weber. Circuit breakers, discovery, and api gateways in microservices, 2016.

[57] Kseniia Kyslova. Service-oriented architecture: When to use and how to benefit from it? `https://proxify.io/de/articles/benefits-of-service-oriented-architecture`. Last checked August 2021.

[58] Sam Newman. Building microservices: Designing fine-grained systems. *Oreilly & Associates Inc*, 2015.

[59] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering*, pages 195–216, 2017.

[60] Olaf Zimmermann. Microservices tenets. *Computer Science-Research and Development*, 32(3):301–310, 2017.

[61] Adrian Cockcroft. Migrating to microservices. `https://www.infoq.com/presentations/migration-cloud-native/`. Last checked August 2021.

[62] Davide Taibi and Kari Systä. From monolithic systems to microservices: A decomposition framework based on process mining. 2019.

[63] Anup Kalia, Jin Xiao, Rahul Krishna, Saurabh Sinha, Maja Vukovic, and Debasish Banerjee. Mono2micro: A practical and effective tool for decomposing

monolithic java applications to microservices. *arXiv preprint arXiv:2107.09698*, 2021.

[64] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. *IEEE Cloud Computing*, 4(5):22–32, 2017.

[65] Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software*, 146:215–232, 2018.

[66] Antonio Messina, Riccardo Rizzo, Pietro Storniolo, Mario Tripiciano, and Alfonso Urso. The database-is-the-service pattern for microservice architectures. In *International Conference on Information Technology in Bio-and Medical Informatics*, pages 223–233. Springer, 2016.

[67] Cristian Gadea, Mircea Trifan, Dan Ionescu, and Bogdan Ionescu. A reference architecture for real-time microservice api consumption. In *Proceedings of the 3rd Workshop on CrossCloud Infrastructures & Platforms*, CrossCloud '16, New York, NY, USA, 2016. Association for Computing Machinery.

[68] Antonio Messina, Riccardo Rizzo, Pietro Storniolo, and Alfonso Urso. A simplified database pattern for the microservice architecture. In *The Eighth International Conference on Advances in Databases, Knowledge, and Data Applications (DBKDA)*, pages 35–40, 2016.

[69] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Architectural patterns for microservices: A systematic mapping study. In *CLOSER*, pages 221–232, 2018.

[70] Rodrigo Laigner, Yongluan Zhou, Marcos Antonio Vaz Salles, Yijian Liu, and Marcos Kalinowski. Data management in microservices: State of the practice, challenges, and research directions, 2021.

[71] Saulo S. de Toledo, Antonio Martini, and Dag I.K. Sjøberg. Identifying architectural technical debt, principal, and interest in microservices: A multiple-case study. *Journal of Systems and Software*, 177:110968, 2021.

[72] Claus Pahl, Antonio Brogi, Jacopo Soldani, and Pooyan Jamshidi. Cloud container technologies: a state-of-the-art review. *IEEE Transactions on Cloud Computing*, 7(3):677–692, 2017.

[73] Tasneem Salah, M. Jamal Zemerly, Chan Yeob Yeun, Mahmoud Al-Qutayri, and Yousof Al-Hammadi. Performance comparison between container-based and vm-based services. In *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*, pages 185–190, 2017.

[74] Nane Kratzke. About microservices, containers and their underestimated impact on network performance. *arXiv preprint arXiv:1710.04049*, 2017.

[75] Marcelo Amaral, Jorda Polo, David Carrera, Iqbal Mohomed, Merve Unuvar, and Malgorzata Steinder. Performance evaluation of microservices architectures using containers. In *2015 IEEE 14th International Symposium on Network Computing and Applications*, pages 27–34. IEEE, 2015.

[76] Lianping Chen. Microservices: architecting for continuous delivery and devops. In *2018 IEEE International conference on software architecture (ICSA)*, pages 39–397. IEEE, 2018.

[77] Tony Mauro. Adopting microservices at netflix: Lessons for architectural design. `https://www.nginx.com/blog/Microservices-at-netflix-architectural-best-practices/`. Last checked August 2021.

[78] staci Kramer. The biggest thing amazon got right: The platform. `https://gigaom.com/2011/10/12/419-the-biggest-thing-amazon-got-right-the-platform/`. Last checked August 2021.

[79] Karan Parikh Steven Ihde. From a monolith to microservices + rest: the evolution of linkedin's service architecture. `https://www.infoq.com/presentations/linkedin-Microservices-urn/`. Last checked August 2021.

[80] Maha Driss, Daniah Hasan, Wadii Boulila, and Jawad Ahmad. Microservices in iot security: Current solutions, research challenges, and future directions. *arXiv preprint arXiv:2105.07722*, 2021.

[81] Juan-Manuel Fernandez, Ivan Vidal, and Francisco Valera. Enabling the orchestration of iot slices through edge and cloud microservice platforms. *Sensors*, 19(13), 2019.

[82] Kleanthis Thramboulidis, Danai C. Vachtsevanou, and Alexandros Solanos. Cyber-physical microservices: An iot-based framework for manufacturing systems. *CoRR*, abs/1801.10340, 2018.

[83] Björn Butzin, Frank Golatowski, and Dirk Timmermann. Microservices approach for the internet of things. In *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–6. IEEE, 2016.

[84] Sergio Trilles, Alberto González-Pérez, and Joaquín Huerta. An iot platform based on microservices and serverless paradigms for smart farming purposes. *Sensors*, 20(8):2418, 2020.

[85] Docker Inc. "docker: Empowering app development for developers". `https://www.docker.com/`. Last checked July 2021.

[86] Cloud Native Computing Foundation. "kubernetes: Production-grade container orchestration". `https://kubernetes.io/`. Last checked August 2021.

[87] Twitter. "finagle library". `https://twitter.github.io/finagle/`. Last checked July 2021.

[88] Netflix. "hystrix library". `https://github.com/Netflix/Hystrix`. Last checked July 2021.

[89] NASSCOM Insights. "service mesh - manage service-to-service communication within your microservice application efficiently". `https://community.nasscom.in/communities/it-services/service-mesh-manage-service-service-communication-within-your-microservice`. Accessed July, 2021.

[90] Linkerd service mesh. `https://linkerd.io/`. Last checked July 2021.

[91] Istio service mesh. `https://istio.io/`. Last checked July 2021.

[92] "consul: Service discovery". `https://www.consul.io/docs/discovery/services`. Last checked July 2021.

[93] App mesh: Application-level networking for all your services. `https://aws.amazon.com/app-mesh/`. Last checked July 2021.

[94] Sachin Manpathak. Kubernetes service mesh: A comparison of istio, linkerd and consul. `https://platform9.com/blog/kubernetes-service-mesh-a-comparison-of-istio-linkerd-and-consul/`, October 2019. Last checked July 2021.

[95] Envoy proxy. `https://www.envoyproxy.io/`. Last checked July 2021.

[96] Vincent Bushong, Amr S Abdelfattah, Abdullah A Maruf, Dipta Das, Austin Lehman, Eric Jaroszewski, Michael Coffey, Tomas Cerny, Karel Frajtak, Pavel Tisnovsky, et al. On microservice analysis and architecture evolution: A systematic mapping study. *Applied Sciences*, 11(17):7856, 2021.

[97] Kevin Fauvel, Daniel Balouek-Thomert, Diego Melgar, Pedro Silva, Anthony Simonet, Gabriel Antoniu, Alexandru Costan, Véronique Masson, Manish Parashar, Ivan Rodero, et al. A distributed multi-sensor machine learning approach to earthquake early warning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 403–411, 2020.

[98] Cloud Native Computing Foundation (CNCF). "cncf cloud native interactive landscape". `https://landscape.cncf.io/card-mode?category=scheduling-orchestration&grouping=category`. Last checked September 2021.

[99] M. Ali, A. Anjum, O. Rana, A. R. Zamani, D. Balouek-Thomert, and M. Parashar. Res: Real-time video stream analytics using edge enhanced clouds. *IEEE Transactions on Cloud Computing*, pages 1–1, 2020.

[100] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J. Freedman. Live video analytics at scale with approximation and delay-tolerance. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 377–392, Boston, MA, March 2017. USENIX Association.

[101] Junchen Jiang, Ganesh Ananthanarayanan, Peter Bodik, Siddhartha Sen, and Ion Stoica. Chameleon: scalable adaptation of video analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 253–266, 2018.

[102] Q. Liu, S. Huang, J. Opadere, and T. Han. An edge network orchestrator for mobile augmented reality. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 756–764, 2018.

[103] Redowan Mahmud, Adel N. Toosi, Kotagiri Ramamohanarao, and Rajkumar Buyya. Context-aware placement of industry 4.0 applications in fog computing

environments. *IEEE Transactions on Industrial Informatics*, 16(11):7004–7013, 2020.

[104] Eduard Renart, Daniel Balouek-Thomert, Xuan Hu, Jie Gong, and Manish Parashar. Online decision-making using edge resources for content-driven stream processing. In *2017 IEEE 13th International Conference on e-Science (e-Science)*, pages 384–392. IEEE, 2017.

[105] Melvin E Conway. How do committees invent. *Datamation*, 14(4):28–31, 1968.

[106] "owl web ontology language". `https://www.w3.org/TR/2004/REC-owl-guide-20040210/#StructureOfOntologies`. Last checked July 2021.

[107] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. Unraveling the web services web: an introduction to soap, wsdl, and uddi. *IEEE Internet computing*, 6(2):86–93, 2002.

[108] Aabhas V. Paliwal, Basit Shafiq, Jaideep Vaidya, Hui Xiong, and Nabil Adam. Semantics-based automated service discovery. *IEEE Transactions on Services Computing*, 5(2):260–275, 2012.

[109] Jian Wu, Liang Chen, Zibin Zheng, Michael Lyu, and Z. Wu. Clustering web services to facilitate service discovery. *Knowledge and Information Systems*, 38, 01 2014.

[110] "what is openapi?". `https://swagger.io/docs/specification/about/`. Last checked July 2021.

[111] "restful api modeling language". `https://raml.org/`. Last checked July 2021.

[112] "api blueprint". `https://apiblueprint.org/`. Last checked July 2021.

[113] "linkerd: Service profiles". `https://linkerd.io/2.10/tasks/books/#service-profiles`. Last checked July 2021.

[114] Kaarthik Sivashanmugam, Kunal Verma, Amit Sheth, and John Miller. Adding semantics to web services standards. *Proceedings of the International Conference on Web Services*, 05 2003.

[115] Gomes Porfirio, Cavalcante Everton, Rodrigues Taniro, Batista Thais, Delicato Flavia C., and Pires Paulo F. A federated discovery service for the internet of things. In *Proceedings of the 2nd Workshop on Middleware for Context-Aware Applications in the IoT*, M4IoT 2015, page 25–30, New York, NY, USA, 2015. Association for Computing Machinery.

[116] Rohallah Benaboud, Ramdane Maamri, and Zaidi Sahnoun. Agents and owl-s based semantic web service discovery with user preference support. *International journal of Web and Semantic Technology*, 4(2):57–75, Apr 2013.

[117] Nawaz Falak, Qadir Kamran, and Ahmad H. Farooq. Semreg-pro: A semantic based registry for proactive web service discovery using publish-subscribe model. In *2008 Fourth International Conference on Semantics, Knowledge and Grid*, pages 301–308, 2008.

[118] Meriem Aziez, Saber Benharzallah, and Hammadi Bennoui. Service discovery for the internet of things: Comparison study of the approaches. In *2017 4th International Conference on Control, Decision and Information Technologies (CoDIT)*, pages 0599–0604, 2017.

[119] Asma Adala, Nabil Tabbane, and Sami Tabbane. A framework for automatic web service discovery based on semantics and nlp techniques. *Adv. MultiMedia*, 2011, January 2011.

[120] Davis John and Rajasree M S. Restdoc: Describe, discover and compose restful semantic web services using annotated documentations. *International journal of Web & Semantic Technology*, 4:37–49, 01 2013.

[121] Amit P Sheth, Karthik Gomadam, and Jon Lathem. Sa-rest: Semantically interoperable and easier-to-use services and mashups. *IEEE Internet Computing*, 11(6):91–94, 2007.

[122] Meriem Achir, Abdelkrim Abdelli, and Lynda Mokdad. A taxonomy of service discovery approaches in iot. In *2020 8th International Conference on Wireless Networks and Mobile Communications (WINCOM)*, pages 1–6, 2020.

[123] Anind Dey. Understanding and using context. *Personal and Ubiquitous Computing*, 5:4–7, 02 2001.

[124] Santosh Pattar, Dwaraka S. Kulkarni, Darshil Vala, Rajkumar Buyya, Venugopal K. R., S.S. Iyengar, and L.M. Patnaik. Progressive search algorithm for service discovery in an iot ecosystem. In *2019 International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1041–1048, 2019.

[125] Dina Hussein, Son Han, Gyu Myoung Lee, Noel Crespi, and Emmanuel Bertin. Towards a dynamic discovery of smart services in the social internet of things. *Computers & Electrical Engineering, Elsevier*, 58, 01 2017.

[126] In-Young Ko, Han-Gyu Ko, Angel Jimenez Molina, and Jung-Hyun Kwon. Soiot: Toward a user-centric iot-based service framework. *ACM Trans. Internet Technol.*, 16(2), April 2016.

[127] Hong Qing Yu and Stephan Reiff-Marganiec. Automated context-aware service selection for collaborative systems. In *International Conference on Advanced Information Systems Engineering*, pages 261–274. Springer, 2009.

[128] Talal Ashraf Butt, Iain Phillips, Lin Guan, and George Oikonomou. Adaptive and context-aware service discovery for the internet of things. In *Internet of things, smart spaces, and next generation networking*, pages 36–47. Springer, 2013.

[129] "dns service discovery". `http://www.dns-sd.org/`. Last checked July 2021.

[130] Milosh Stolikj, P. Cuijpers, J. Lukkien, and N. Buchina. Context based service discovery in unmanaged networks using mdns/dns-sd. *2016 IEEE International Conference on Consumer Electronics (ICCE)*, pages 163–165, 2016.

[131] Firas Albalas, Wail Mardini, and Majd Al-Soud. Aft: Adaptive fibonacci-based tuning protocol for service and resource discovery in the internet of things. In *2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 177–182, 2017.

[132] Seokhwa Kim, Keuntae Lee, and J. Jeong. Dns naming services for service discovery and remote control for internet-of-things devices. *2017 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 1156–1161, 2017.

[133] Manoj Parameswaran, Anjana Susarla, and Andrew B Whinston. P2p networking: an information sharing alternative. *Computer*, 34(7):31–38, 2001.

[134] Cédric Tedeschi. *Peer-to-Peer Prefix Tree for Large Scale Service Discovery*. Theses, Ecole normale supérieure de lyon - ENS LYON, October 2008.

[135] Eddy Caron, Frédéric Desprez, and Cédric Tedeschi. Enhancing computational grids with peer-to-peer technology for large scale service discovery. *Journal of Grid Computing*, 5(3):337–360, 2007.

[136] Eddy Caron, Florent Chuffart, Haiwu He, and Cédric Tedeschi. Implementation and evaluation of a p2p service discovery system: Application in a dynamic large scale computing infrastructure. In *2011 IEEE 11th International Conference on Computer and Information Technology*, pages 41–46. IEEE, 2011.

[137] Juan Li, Nazia Zaman, and Honghui Li. A decentralized locality-preserving context-aware service discovery framework for internet of things. In *2015 IEEE International Conference on Services Computing*, pages 317–323, 2015.

[138] Simone Cirani, Luca Davoli, Gianluigi Ferrari, Remy Leone, Paolo Medagliani, Marco Picone, and Luca Veltri. A scalable and self-configuring architecture for service discovery in the internet of things. *IEEE Internet of Things Journal*, 1:508–521, 10 2014.

[139] Jiang Rui and Sun Danpeng. An agricultural service oriented information discovery technology for internet of things. In *2016 International Conference on Smart Grid and Electrical Automation (ICSGEA)*, pages 268–271, 2016.

[140] Elli Rapti, Anthony Karageorgos, Catherine Houstis, and Elias Houstis. Decentralized service discovery and selection in internet of things applications based on artificial potential fields. *Service Oriented Computing and Applications*, 11(1):75–86, 2017.

[141] Yuwei Wang. Towards service discovery and autonomic version management in self-healing microservices architecture. In *Proceedings of the 13th European Conference on Software Architecture - Volume 2*, ECSA '19, page 63–66, New York, NY, USA, 2019. Association for Computing Machinery.

[142] Petar Krivic, Pavle Skocir, and Mario Kusek. Agent-based approach for energy-efficient iot services discovery and management. In *KES international symposium on agent and multi-agent systems: technologies and applications*, pages 57–66. Springer, 2018.

[143] Rana Helal and Amr ElMougy. An energy-efficient service discovery protocol for the iot based on a multi-tier wsn architecture. In *2015 IEEE 40th Local Computer Networks Conference Workshops (LCN Workshops)*, pages 862–869, 2015.

[144] Sameh Ben Fredj, Mathieu Boussard, Daniel Kofman, and Ludovic Noirie. Efficient semantic-based IoT service discovery mechanism for dynamic environments. In *IEEE 25th Annual International Symposium on Personal, Indoor, and Mobile Radio Communication (PIMRC), 2014*, pages 2088 – 2092, washington, United States, September 2014.

[145] Chris Richardson. Pattern: Client-side service discovery. `https://microservices.io/patterns/client-side-discovery.html`. Last checked July 2021.

[146] Chris Richardson. Pattern: Server-side service discovery. `https://microservices.io/patterns/server-side-discovery.html`. Last checked July 2021.

[147] Netflix. Netflix open source software center. `https://netflix.github.io/`. Last checked July 2021.

[148] Amazon AWS. Elastic load balancing. `https://aws.amazon.com/elasticloadbalancing/`. Last checked July 2021.

[149] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in cloud computing: What it is, and what it is not. In *10th International Conference on Autonomic Computing (ICAC 13)*, pages 23–27, San Jose, CA, June 2013. USENIX Association.

[150] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), March 2014.

[151] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of grid computing*, 12(4):559–592, 2014.

[152] Kubernetes: Horizontal pod autoscaler. `https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/`. Last checked August 2021.

[153] Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, and Philippe Merle. Autonomic vertical elasticity of docker containers with elasticdocker. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pages 472–479, 2017.

[154] Anthony Kwan, Jonathon Wong, Hans-Arno Jacobsen, and Vinod Muthusamy. Hyscale: Hybrid and network scaling of dockerized microservices in cloud data centres. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 80–90, 2019.

[155] Sourav Dutta, Sankalp Gera, Akshat Verma, and Balaji Viswanathan. Smartscale: Automatic application scaling in enterprise clouds. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 221–228, 2012.

[156] Philipp Hoenisch, Ingo Weber, Stefan Schulte, Liming Zhu, and Alan Fekete. Four-fold auto-scaling on a contemporary deployment platform using docker containers. In *International Conference on Service-Oriented Computing*, pages 316–323. Springer, 2015.

[157] Jingqi Yang, Chuanchang Liu, Yanlei Shang, Bo Cheng, Zexiang Mao, Chunhong Liu, Lisha Niu, and Junliang Chen. A cost-aware auto-scaling approach using the workload prediction in service clouds. *Information Systems Frontiers*, 16(1):7–18, 2014.

[158] Shveta Verma and Anju Bala. Auto-scaling techniques for iot-based cloud applications: a review. *Cluster Computing*, pages 1–35, 2021.

[159] Amazon ec2 auto scaling. `https://docs.aws.amazon.com/autoscaling/ec2/userguide/what-is-amazon-ec2-auto-scaling.html`. Last checked August 2021.

[160] Google cloud compute engine: Autoscaling groups of instances. `https://cloud.google.com/compute/docs/autoscaler`. Last checked August 2021.

[161] Tian Ye, Xue Guangtao, Qian Shiyou, and Li Minglu. An auto-scaling framework for containerized elastic applications. In *2017 3rd international conference on big data computing and communications (BIGCOM)*, pages 422–430. IEEE, 2017.

[162] Muhammad Abdullah, Waheed Iqbal, Arif Mahmood, Faisal Bukhari, and Abdelkarim Erradi. Predictive autoscaling of microservices hosted in fog microdata center. *IEEE Systems Journal*, 15(1):1275–1286, 2020.

[163] João Nunes, Thiago Bianchi, Anderson Iwasaki, and Elisa Nakagawa. State of the art on microservices autoscaling: An overview. In *Anais do XLVIII Seminário Integrado de Software e Hardware*, pages 30–38, Porto Alegre, RS, Brasil, 2021. SBC.

[164] Manuel Gotin, Felix Lösch, Robert Heinrich, and Ralf Reussner. Investigating performance metrics for scaling microservices in cloudiot-environments. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 157–167, 2018.

[165] Abeer Abdel Khaleq and Ilkyeun Ra. Intelligent autoscaling of microservices in the cloud for real-time applications. *IEEE Access*, 9:35464–35476, 2021.

[166] Salman Taherizadeh, Vlado Stankovski, and Jin-Hee Cho. Dynamic multi-level auto-scaling rules for containerized applications. *The Computer Journal*, 62(2):174–197, 2019.

[167] Hashicorp consul. `https://www.hashicorp.com/products/consul/multi-platform-service-mesh`. Last checked July 2021.

[168] what is dns? how dns works. `https://www.cloudflare.com/learning/dns/what-is-dns/`. Last checked August 2021.

[169] Registrator. `https://github.com/gliderlabs/registrator`. Last checked July 2021.

[170] Joyent. `https://github.com/joyent/containerpilot`. Last checked July 2021.

[171] etcd key-value store. `https://etcd.io/`. Last checked July 2021.

[172] Oracle. Oracle data sheet: Mysql cluster. `https://www.mysql.com/products/cluster/mysql-cluster-datasheet.pdf`. Last checked August 2021.

[173] Desprez and al. Adding virtualization capabilities to the grid'5000 testbed. In Ivan I. Ivanov, Marten van Sinderen, Frank Leymann, and Tony Shan, editors, *Cloud Computing and Services Science*, pages 3–20, Cham, 2013. Springer International Publishing.

[174] Grafana. `https://grafana.com/`. Last checked August 2021.

[175] Prometheus. `https://prometheus.io/`. Last checked August 2021.

[176] kube-state-metrics. `https://github.com/kubernetes/kube-state-metrics`. Last checked August 2021.

[177] Zheyi Chen, Junqin Hu, Xing Chen, Jia Hu, Xianghan Zheng, and Geyong Min. Computation offloading and task scheduling for DNN-based applications in cloud-edge computing. *IEEE Access*, 2020.

[178] M. Satyanarayanan, P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, W. Hu, and B. Amos. Edge analytics in the internet of things. *IEEE Pervasive Computing*, 14(2):24–31, 2015.

[179] Ben Taylor, Vicent Sanz Marco, Willy Wolff, Yehia Elkhatib, and Zheng Wang. Adaptive deep learning model selection on embedded systems. *SIGPLAN Not.*, 53(6):31–43, June 2018.

[180] Woo-Joong Kim and Chan-Hyun Youn. Lightweight online profiling-based configuration adaptation for video analytics system in edge computing. *IEEE Access*, 8:116881–116899, 2020.

[181] Xukan Ran, Haolianz Chen, Xiaodan Zhu, Zhenming Liu, and Jiasi Chen. Deepdecision: A mobile deep learning framework for edge video analytics. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 1421–1429. IEEE, 2018.

[182] Chien-Chun Hung, Ganesh Ananthanarayanan, Peter Bodik, Leana Golubchik, Minlan Yu, Paramvir Bahl, and Matthai Philipose. Videoedge: Processing camera streams using hierarchical clusters. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 115–131. IEEE, 2018.

[183] Qiang Liu and Tao Han. Dare: Dynamic adaptive mobile augmented reality with edge computing. In *2018 IEEE 26th International Conference on Network Protocols (ICNP)*, pages 1–11. IEEE, 2018.

[184] Ali Reza Zamani, Moustafa AbdelBaky, Daniel Balouek-Thomert, Juan J Villalobos, Ivan Rodero, and Manish Parashar. Submarine: A subscription-based data

streaming framework for integrating large facilities and advanced cyberinfrastructure. *Concurrency and Computation: Practice and Experience*, 32(16):e5256, 2020.

[185] Prateeksha Varshney and Yogesh Simmhan. Characterizing application scheduling on edge, fog, and cloud computing resources. *Software: Practice and Experience*, 50(5):558–595, 2020.

[186] Haoyu Wang, Zetian Liu, and Haiying Shen. *Job Scheduling for Large-Scale Machine Learning Clusters*, page 108–120. Association for Computing Machinery, New York, NY, USA, 2020.

[187] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*, 2018.

[188] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, Chen Meng, and Wei Lin. Dl2: A deep learning-driven scheduler for deep learning clusters. *arXiv preprint arXiv:1909.06040*, 2019.

[189] Michael Zhang, Chandra Krintz, and Rich Wolski. Stoic: Serverless teleoperable hybrid cloud for machine learning applications on edge device. In *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE, 2020.

[190] Zheyi Chen, Junqin Hu, Xing Chen, Jia Hu, Xianghan Zheng, and Geyong Min. Computation offloading and task scheduling for DNN-based applications in cloud-edge computing. *IEEE Access*, 2020.

[191] Wenjia Zheng, Yun Song, Zihao Guo, Yongchen Cui, Suwen Gu, Ying Mao, and Long Cheng. Target-based resource allocation for deep learning applications in a multi-tenancy system. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2019.

[192] Eduard Gibert Renart, Alexandre Da Silva Veith, Daniel Balouek-Thomert, Marcos Dias De Assunção, Laurent Lefevre, and Manish Parashar. Distributed operator placement for iot data analytics across edge and cloud resources. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 459–468. IEEE, 2019.

[193] Joby P P. Qos aware resource scheduling in internet of things-cloud environment. *International Journal of Scientific and Engineering Research*, 6:294–297, 01 2015.

[194] Qi Zhang, Mohamed Faten Zhani, Raouf Boutaba, and Joseph L Hellerstein. Dynamic heterogeneity-aware resource provisioning in the cloud. *IEEE transactions on cloud computing*, 2(1):14–28, 2014.

[195] Eduard Gibert Renart, Daniel Balouek-Thomert, and Manish Parashar. Edge based data-driven pipelines (technical report), 2018.

[196] Global Industry Analysts. Global deep learning industry. `https://www.reportlinker.com/p05798338/Global-Deep-Learning-Industry.html`, 2021. Last checked October 2021.

[197] Athanasios Voulodimos, Nikolaos Doulamis, Anastasios Doulamis, and Eftychios Protopapadakis. Deep learning for computer vision: A brief review. *Computational intelligence and neuroscience*, 2018, 2018.

[198] Zixing Zhang, Jürgen Geiger, Jouni Pohjalainen, Amr El-Desoky Mousa, Wenyu Jin, and Björn Schuller. Deep learning for environmentally robust speech recognition: An overview of recent developments. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2018.

[199] Francesco Piccialli, Vittorio Di Somma, Fabio Giampaolo, Salvatore Cuomo, and Giancarlo Fortino. A survey on deep learning in medicine: Why, how and when? *Information Fusion*, 66:111 – 137, 2021.

[200] Connor Shorten, Taghi M Khoshgoftaar, and Borko Furht. Deep learning applications for covid-19. *Journal of Big Data*, 8(1):1–54, 2021.

[201] Panagiota Galetsi and Korina Katsaliaki. A review of the literature on big data analytics in healthcare. *Journal of the Operational Research Society*, 71(10):1511–1529, 2020.

[202] Hira Zahid, Tariq Mahmood, Ahsan Morshed, and Timos Sellis. Big data analytics in telecommunications: literature review and architecture recommendations. *IEEE/CAA Journal of Automatica Sinica*, 7(1):18–38, 2019.

[203] Safa Ben Atitallah, Maha Driss, Wadii Boulila, and Henda Ben Ghézala. Leveraging deep learning and iot big data analytics to support the smart cities development: Review and future directions. *Computer Science Review*, 38:100303, 2020.

[204] Thomas A Runkler. *Data analytics*. Springer, 2020.

[205] Saeed Shahrivari. Beyond batch processing: Towards real-time and streaming big data. *Computers*, 3, 03 2014.

[206] Erwin Adi, Adnan Anwar, Zubair Baig, and Sherali Zeadally. Machine learning and data analytics for the iot. *Neural Computing and Applications*, 32(20):16205–16233, 2020.

[207] Maryam M Najafabadi, Flavio Villanustre, Taghi M Khoshgoftaar, Naeem Seliya, Randall Wald, and Edin Muharemagic. Deep learning applications and challenges in big data analytics. *Journal of big data*, 2(1):1–21, 2015.

[208] Jason Brownlee. *Data preparation for machine learning: data cleaning, feature selection, and data transforms in Python*. Machine Learning Mastery, 2020.

[209] Himani Bhavsar and Mahesh H Panchal. A review on support vector machine for data classification. *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)*, 1(10):185–189, 2012.

[210] Rama Akkiraju, Joel Farrell, John Miller, Meenakshi Nagarajan, Marc-Thomas Schmidt, Amit Sheth, and Kunal Verma. The neural network zoo. `https://www.asimovinstitute.org/neural-network-zoo/`, 2016. Accessed September 2021.

[211] Ivan Bruha and Fazel Famili. Postprocessing in machine learning and data mining. *SIGKDD Explorations*, 2:110–114, 12 2000.

[212] Michael Haenlein and Andreas Kaplan. A brief history of artificial intelligence: On the past, present, and future of artificial intelligence. *California management review*, 61(4):5–14, 2019.

[213] Christian Janiesch, Patrick Zschech, and Kai Heinrich. Machine learning and deep learning. *Electronic Markets*, pages 1–11, 2021.

[214] Mohammad Saeid Mahdavinejad, Mohammadreza Rezvan, Mohammadamin Barekatain, Peyman Adibi, Payam Barnaghi, and Amit P. Sheth. Machine learning for internet of things data analysis: a survey. *Digital Communications and Networks*, 4(3):161–175, 2018.

[215] Aston Zhang, Zachary C Lipton, Mu Li, and Alexander J Smola. Dive into deep learning. *arXiv preprint arXiv:2106.11342*, 2021.

[216] Kamran Kowsari, Kiana Jafari Meimandi, Mojtaba Heidarysafa, Sanjana Mendu, Laura Barnes, and Donald Brown. Text classification algorithms: A survey. *Information*, 10(4):150, 2019.

[217] Zhong-Qiu Zhao, Peng Zheng, Shou-tao Xu, and Xindong Wu. Object detection with deep learning: A review. *IEEE transactions on neural networks and learning systems*, 30(11):3212–3232, 2019.

[218] Zixing Zhang, Jürgen Geiger, Jouni Pohjalainen, Amr El-Desoky Mousa, Wenyu Jin, and Björn Schuller. Deep learning for environmentally robust speech recognition: An overview of recent developments. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 9(5):1–28, 2018.

[219] Frank Emmert-Streib, Zhen Yang, Han Feng, Shailesh Tripathi, and Matthias Dehmer. An introductory review of deep learning for prediction models with big data. *Frontiers in Artificial Intelligence*, 3:4, 2020.

[220] Hongyu Liu and Bo Lang. Machine learning and deep learning methods for intrusion detection systems: A survey. *applied sciences*, 9(20):4396, 2019.

[221] Alexander Schindler, Thomas Lidy, and Andreas Rauber. Comparing shallow versus deep neural network architectures for automatic music genre classification. In *FMT*, pages 17–21, 2016.

[222] Dieu Tien Bui, Paraskevas Tsangaratos, Viet-Tien Nguyen, Ngo Van Liem, and Phan Trong Trinh. Comparing the prediction performance of a deep learning neural network model with conventional machine learning models in landslide susceptibility assessment. *CATENA*, 188:104426, 2020.

[223] Sucheta Chauhan, Lovekesh Vig, Michele De Filippo De Grazia, Maurizio Corbetta, Shandar Ahmad, and Marco Zorzi. A comparison of shallow and deep

learning methods for predicting cognitive performance of stroke patients from mri lesion images. *Frontiers in Neuroinformatics*, 13:53, 2019.

[224] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *arXiv preprint arXiv:1506.01497*, 2015.

[225] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4:optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934*, 2020.

[226] N Suresh Kumar, Amit Kumar Goel, and S Jayanthi. A scrupulous approach to perform classification and detection of fetal brain using darknet yolo v4. In *2021 International Conference on Advance Computing and Innovative Technologies in Engineering (ICACITE)*, pages 578–581. IEEE, 2021.

[227] Fetulhak Abdurahman, Kinde Anlay Fante, and Mohammed Aliy. Malaria parasite detection in thick blood smear microscopic images using modified yolov3 and yolov4 models. *BMC bioinformatics*, 22(1):1–17, 2021.

[228] Yingfeng Cai, Tianyu Luan, Hongbo Gao, Hai Wang, Long Chen, Yicheng Li, Miguel Angel Sotelo, and Zhixiong Li. Yolov4-5d: An effective and efficient object detector for autonomous driving. *IEEE Transactions on Instrumentation and Measurement*, 70:1–13, 2021.

[229] Dongsheng Li, Yujie Zhang, Junping Xiang, Jianfei Li, and Yu Zou. Yolo-tiny-msc: A tiny neural network for object detection. In *Journal of Physics: Conference Series*, volume 1873, page 012073. IOP Publishing, 2021.

[230] Petr Hurtik, Vojtech Molek, Jan Hula, Marek Vajgl, Pavel Vlasanek, and Tomas Nejezchleba. Poly-yolo: higher speed, more precise detection and instance segmentation for yolov3. *arXiv preprint arXiv:2005.13243*, 2020.

[231] Minh-Tan Pham, Luc Courtrai, Chloé Friguet, Sébastien Lefèvre, and Alexandre Baussard. Yolo-fine: one-stage detector of small objects under various backgrounds in remote sensing images. *Remote Sensing*, 12(15):2501, 2020.

[232] José Ignacio Fernández-Villamor, Carlos Angel Iglesias, and Mercedes Garijo. Microservices-lightweight service descriptions for rest architectural style. In *ICAART (1)*, pages 576–579, 2010.

[233] Jonathan Hui. Object detection: speed and accuracy comparison (Faster R-CNN, R-FCN, SSD, FPN, RetinaNet and YOLOv3). `http://www.lighterra.com/papers/videoencodingh264/`, 2018. Last checked August 2021.

[234] Adrian Rosebrock. Intersection over union (iou) for object detection. `https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/`, 2016. Accessed September, 2021.

[235] Daniel Rosendo, Pedro Silva, Matthieu Simonin, Alexandru Costan, and Gabriel Antoniu. E2Clab: Exploring the Computing Continuum through Repeatable, Replicable and Reproducible Edge-to-Cloud Experiments. In *Cluster 2020 - IEEE International Conference on Cluster Computing*, pages 1–11, Kobe, Japan, September 2020.

[236] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014.

[237] Frans Panduwinata, Pujianto Yugopuspito, et al. Nanoservices as generalization services in service-oriented architecture. In *2017 International Conference on Soft Computing, Intelligent System and Information Technology (ICSIIT)*, pages 131–137. IEEE, 2017.

[238] Erkki Harjula, Pekka Karhula, Johirul Islam, Teemu Leppänen, Ahsan Manzoor, Madhusanka Liyanage, Jagmohan Chauhan, Tanesh Kumar, Ijaz Ahmad, and Mika Ylianttila. Decentralized iot edge nanoservice architecture for future gadget-free computing. *IEEE Access*, 7:119856–119872, 2019.

[239] Xinwen Wang, Tiancheng Yuan, Yu-Ju Huang, and Robbert van Renesse. Disaggregated applications using nanoservices. 2021.

List of Figures

List of Tables

List of Algorithms