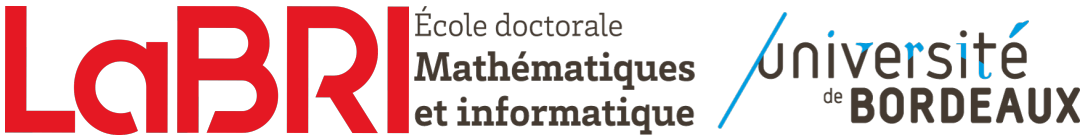
Observability and resources managements in cloud-native environnements

Nicolas Marie-Magdelaine



THÈSE PRÉSENTÉE

POUR OBTENIR LE GRADE DE

DOCTEUR
DE L'UNIVERSITÉ DE BORDEAUX

ECOLE DOCTORALE MATHEMATIQUE et

INFORMATIQUE

Laboratoire Bordelais de Recherche en Informatique

Par **Nicolas MARIE-MAGDELAINE**

Observability and resource management in cloud-native environments

Sous la direction de : **Toufik AHMED**

Soutenue le 25 Novembre 2021

Membres du jury :

M. Toufik Ahmed	Professeur des universités	Bordeaux INP	Directeur de thèse
M. Mohamed Mosbah	Professeur des universités	Université de Bordeaux	Examinateur
Mme Vèque Véronique	Professeur des universités	Université Paris-Saclay	Examinateur
M. Chaput Emmanuel	Professeur des universités	Toulouse INP	Rapporteur
M. Abbas Bradai	Maître de conférences	Université de Poitiers	Rapporteur

Acknowledgements

Ces remerciements ne peuvent pas commencer sans mentionner mon directeur de thèse, le Professeur Toufik Ahmed. Il a su dès notre rencontre, lors d'une visioconférence en 2015, me faire confiance et me confier un stage de recherche ambitieux pour l'obtention de mon master de recherche en informatique.

De notre collaboration est née une thèse. Depuis lors, il n'a cessé de m'encourager, de me soutenir, de m'accompagner, de me relever dans mes moments de découragement. Ses conseils et son expertise ont été de précieux outils pour me permettre de vivre et apprendre au cours de ces années. Aujourd'hui, je suis fier de pouvoir le remercier pour les opportunités qu'il m'a offertes et pour le soin qu'il a mis dans son enseignement.

Tout au long de cette thèse, j'ai également découvert une grande équipe. De nombreuses personnes autour du laboratoire LaBRI mais également à l'école d'ingénieur ENSEIRB-MATMECA Bordeaux INP. Ce sont des enseignants-chercheurs, des collègues doctorants, des aînés et des élèves qui m'ont offert conseil, hospitalité, joie, collaboration professionnelle et moments de convivialité.

Je ne peux pas oublier Christelle, Mohamed Lamine, Alyona, les très nombreux membres des associations étudiantes en informatique dont celle des doctorants du LaBRI, AfoDiB, que j'ai eu le privilège de vice-présider. Le Campus de l'Université de Bordeaux m'a permis de rencontrer beaucoup d'amis à travers les activités associatives et les tiers lieux tels qu'Etu'Recup.

C'est dans cette ville étudiante qu'est Bordeaux que j'ai pu rencontrer la personne qui est aujourd'hui ma partenaire de vie. Merci à toi, Anne-Claire Bonneau, de m'avoir soutenu dans les moments difficiles. Sans ta présence dans ma vie, je peux le dire, je n'aurais pas pu mener à terme ce projet.

Je remercie mes parents Charles-Edouard et Marie-Yvette Marie-Magdelaine pour m'avoir permis d'arriver aussi loin dans mes études par leur soutien sans faille et leur implication dans mes réussites, et mon oncle Ernest pour m'avoir immergé dans un milieu qui a suscité ma curiosité pour l'informatique et les réseaux dans mon adolescence.

Je me dois de citer les Marie-Magdelaine, famille et alliés, qui ont depuis mon plus jeune âge contribué à faire de moi ce que je suis maintenant.

Je remercie Florence Bonneau, Olivier Bonneau, Lucile Dijkstra et Anäelle Rousseau qui, dans ces temps difficiles de confinements et d'impossibilité de

voyager, m'ont offert une place dans leur famille et m'ont accueilli comme un des leurs.

Je n'oublie pas Sarah Delas, Tiphaine Compain, Audrey, Alexandre, Linh Chi, Jérome, Mila, Camille I.R., Florence B., Marie D., Cécile, Maïke, Isabelle, Cédric, Ben, Tomo, Thomas, Morgane, Anna, Ana et tous ceux qui sont dans mon cœur. Je vous aime, merci d'être présents dans ma vie.

J'accorde une importance spécifique à citer ici mes "mentors", ceux qui m'ont montré la voie : Guy Pignolet, Pierre-Ugo Tournoux, Denis Payet, Chris Bridges, vous avez cru en moi et avez partagé avec moi vos enseignements, sans lesquels je ne serais là.

Je remercie la Collectivité Territoriale de Martinique pour m'avoir soutenu financièrement dans les moments difficiles. Merci pour ces dispositifs qui viennent en aide aux étudiants. J'espère pouvoir, un jour, contribuer à la formation et aux succès de la jeunesse du pays.

Et bien entendu, la société Lectra pour la possibilité de réaliser ces travaux.

Contents

List of Figures

List of Tables

List of Abbreviations

AMQP	Advanced Message Queuing Protocol
API	Applications Programming Interface
ARIMA	Autoregressive integrated moving average
ARPANET	Advanced Research Projects Agency Network
AWS S3	Amazon Web Services Simple Storage Service (S3)
BLE	Bluetooth Low Energy
CaaS	Container-as-a-Service
CD	Continuous Delivery
CD	Continuous Deployment
CH	ClusterHead
CI	Continuous Integration
CIFRE	Conventions industrielles de formation par la recherche
CLI	Command Line Inter-face
CNCF	Cloud Native Computing Foundation
COTS	Commercial off-the-shelf
CPU	Central Processing Unit
DDD	Domain-Driven Design
EC2	Elastic Compute Cloud
ESB	Enterprise Service Bus
FaaS	Function-as-a-Service
FCAPS	Fault, Configuration,Accounting, Performance and Security
GPU	Graphics Processing Units
GRC	GNU Radio Companion
HTTP	Hypertext Transfer Protocol
I/O	Input / Output
IaaS	Infrastructure as a Service
IEEE	Institute of Electrical and Electronics Engineers
IOPS	input/output operations per second
IoT	Internet of Things
IT	Information technology
KPI	Key Performance Indicator
KVM	Kernel-based Virtual Machine
LAN	Local Area Network
LSTM	Long Short-Term Memory
M2M	Machine to Machine
MAPE	Monitor, Analyze, Plan, and Execute
MEC	Mobile Edge Computing
MTTD	meantime to detection
NFV	Network Function Virtualization

NFV MANO	Network Function Virtualization Management and Orchestration
NIC	Network Interfacecards
NIST	Nationa Institute of Standards and Technology
OPS	Operational engineers
OS	Operating System
PaaS	Platform as a Service
PNFs	Physical Network Functions
PoC	proof of concept
QoS	Quality of Service
RAM	Random Access Memory
RD	Research and Development
RED	Rate, Errors and Duratio
RNN	Recurrent NeuralNetwork
SaaS	Software as a Service
SARIMAX	Seasonal Auto-Regressive Integrated Moving Average with eXogenous factors
SCM	Source Code Management
SDN	Software Defined Networking
SDR	Software Defined Radio
SFC	Service Function Chain
SLA	Service-level agreements
SLO	Service LevelObjectives
SOA	Service-oriented applications
SQL	Structured Query Language
SRE	Site Reliability Engineering
TCP	Transmission Control Protocol
TSDB	Time-Series Database
UAV	Unmanned Aerial Vehicle
USE	Utilization, Saturation, Errors
UUID	Universally Unique Identifie
VLs	Virtual Links
VM	Virtual Machine
VNF	Virtual Network Functions
VNFFGs	VNF Forwarding Graphs

Abstract

Observability and resource management in cloud-native environments

Cloud Computing and Cloud-Native technologies have become the backbones of the modern Internet. Users and organizations are now relying on cloud applications for their everyday needs. However, outages and Quality of Service degradations can have disastrous impacts on our society. Moreover, web applications have become complex distributed systems, difficult to understand and operate, thus more prone to failures if not managed accordingly. As a result, it is paramount to understand, observe, prevent, detect and correct any issues that may lead to failures.

In this thesis, we propose a framework for achieving Observability in cloud-native environments. Observability envisions a deeper understanding of the complex distributed system that web applications have become, representing an improvement from traditional monitoring strategies. The proposed Observability framework is demonstrated via proof-of-concept and deployment in production environments. Furthermore, following the principle of autonomic computing, we also propose an architecture for Observability-driven auto-scaling in Cloud-Native environments. This architecture enables us to correlate auto-scaling to the application workload. We also push a step further by leveraging Machine Learning and enable proactive auto-scaling. We focus on using automation and Observability to increase the Quality of Service metrics during scaling events. Additionally, we explore and demonstrate the benefits and possibility of porting the cloud-native architecture and principles to the Internet of Things. We propose to leverage technologies such as Software-Defined Networking and Software-Defined Radio to provide flexible and re-configurable generic Internet of Things devices.

Keywords: Cloud Monitoring, Cloud Management, Cloud Platform, Cloud-Native, Observability, Virtualization

Résumé

Observabilité et gestion des ressources dans les environnements cloud-natifs

Les technologies Cloud et cloud-natif sont devenues les piliers de l'Internet moderne. Les utilisateurs et les organisations s'appuient désormais sur des applications cloud pour leurs besoins quotidiens. Cependant, les pannes et les dégradations de la qualité de service peuvent avoir des impacts désastreux sur notre société. De plus, les applications Web sont devenues des systèmes distribués complexes, difficiles à comprendre et à exploiter, donc plus sujettes aux pannes si elles ne sont pas gérées en conséquence. Par conséquent, il est primordial de comprendre, d'observer, de prévenir, de détecter et de corriger tout problème pouvant entraîner des défaillances.

Dans cette thèse, nous proposons un cadre pour atteindre l'observabilité dans les environnements cloud natifs. L'observabilité envisage une compréhension plus approfondie du système distribué complexe que sont devenus les applications Web. Le cadre d'observabilité proposé est démontré via une preuve de concept et un déploiement dans des environnements de production. De plus, suivant le principe de l'informatique autonome, nous proposons également une architecture pour la mise à l'échelle automatique basée sur l'observabilité dans les environnements cloud-natifs. Cette architecture nous permet de corréler l'auto-scaling à la charge de travail de l'application. Nous allons également plus loin en tirant parti de l'apprentissage automatique et en permettant une mise à l'échelle automatique proactive. Nous nous concentrons sur l'utilisation de l'automatisation et de l'observabilité pour augmenter les métriques de qualité de service lors des événements de mise à l'échelle. De plus, nous explorons et démontrons les avantages et la possibilité de porter l'architecture et les principes natifs du cloud vers l'Internet des objets. Nous proposons d'exploiter des technologies telles que la mise en réseau définie par logiciel et la radio définie par logiciel pour fournir des appareils Internet des objets génériques flexibles et reconfigurables.

Mots-clés : Surveillance Cloud, Gestion Cloud , Plateforme Cloud, Cloud-Natif, Observabilité, Virtualisation

Introduction

Web applications are now part of our daily lives. Our modern world relies on web applications for banking, telecommunications, transformation, healthcare, and even farming. Users and organizations expect fast, responsive, reliable, and available web applications. Cloud Computing and many new innovative technologies were created driven by this need for better, faster and reliable web applications.

Cloud computing adoption continues to rise across all organizations. From small businesses to Fortune 500 and government all walk the path of digital transformation. While the adoption of new Software-as-a-Service (SaaS) solutions may present some challenges, moving legacy business applications to the cloud can leave even large tech companies struggling.

While strategies based on transporting legacy business applications to the cloud as-is, also called "lift-and-shift", may sound attractive, they are a trap. Organizations adopting those enticing promises of easy cloud adoption may end up with more complex, more costly, and less reliable IT systems. Moving to the cloud and getting the full value out of the cloud computing paradigm requires a complete overhaul of how IT systems are created, run, and operated.

Cloud-Native Applications represent the state-of-the-art paradigm used by the large majority of organizations across the globe. Based on the cumulative innovation from Cloud Computing, microservices, automation, and decades of best practices, they represent what organizations want to strive for.

Designing and operating public-cloud-hosted cloud-native applications is a challenge full of research opportunities. Being able to follow and take part in the process of digital transformation at a large company from the beginning to completion is an exceptional opportunity.

In this thesis, we have followed and go beyond the path of a digital transformation from legacy IT to cloud-native. It was an idea environment to observe, evaluate and experiment from the early stages of the migration to cloud computing to the state-of-the-art of Public Cloud Hosted Cloud-Native Applications.

Context

I originally joined Lectra in September 2016 while they were looking for developers and engineers with experience creating and operating applications in cloud environments.

I was recruited long before any dedicated team to "cloud services production" or "operational research" was created. I joined the development of a pilot cloud-native application designed to represent a technology demonstrator for Lectra's cloud migration.

Once this demonstrator finished, I was offered the opportunity to start, in September 2017, a CIFRE Thesis around the operations of cloud services.

In this thesis, the objectives were defined as follows:

1 - Study and evaluate the operational requirements for cloud monitoring tools and solutions

2 - Deploy, as proof-of-concept and production tool, strategies and tools to observe, analyze, alert, and troubleshoot the inside of Lectra's Cloud applications.

3 - Quantify the resources allocation needs to achieve the desired quality of service for the specific needs of Lectra's microservices.

In this context, the objective of monitoring and surveillance of infrastructures and services deployed in the cloud will be one of the first to be achieved. A state-of-the-art of existing solutions will be carried out in order to determine the limits of the existing in the current and future operational frameworks.

Monitoring allows to analyze, optimize and discover what is happening in the cloud infrastructure. It also allows the precise quantification of the resource allocation needs in order to achieve the requested performance objectives and to meet the level of service quality, which can be defined contractually. This task should be the subject of in-depth research given the exotic and innovative nature of the services developed by Lectra's R&D department.

Some applications may require finer control over network latency, reliability, and expected performance. The contribution of SDN and NFV technologies should be studied in terms of network traffic control (traffic steering), traffic engineering, inter-Cloud routing management, inter-cloud data transport, and sharing of resources between different hosted tenants.

Furthermore, it will be necessary to study the different types of resource use, thus leading to the need to set up new orchestration models in order to allow the scaling of services in a reactive and/or proactive manner. The choice of one method over another will be studied qualitatively and quantitatively. Likewise, the level of service quality and therefore derivative SLAs should be able to be specified on the fly based on the current state of the systems in order to provide consumers with performance levels in line with

their needs. Dynamically resizing cloud resources and scaling virtual machines in the cloud (scaling up / down) has an undeniable impact on service performance. For this, algorithms for automatic scaling of a cloud service will also be heavily studied in order to develop the most efficient ones and to test them in real situations.

All the information collected must offer the possibility of querying at any time the context of the services, their interconnection as well as their ability to scale. Advanced knowledge of the state of services, data, and not just the infrastructure provides the ability, advanced control of deployed services and the underlying cloud infrastructure.

Problem Statement

Through this thesis, we follow the path of migration from developing desktop applications to Industry 4.0 cloud-native applications with the angle of operation and infrastructure design. This led us to major research questions including but not limited to :

- What are the requirements to operate cloud-native applications in a production environment ? How can we get information about cloud-native applications and their environment ?

Microservice architecture provides many benefits and raises new challenges such as management, monitoring, and quality of service provisioning of those microservices-based applications in the public cloud environment. In addition, in a cloud-native environment, complexity increases, which also significantly increases the difficulty to observe and visibly understand the health, performance, and behaviors of Cloud-Native Applications. Our contribution in chapter 2, presents the requirements towards Observability in Cloud-native Application lays the foundation for our Architecture Proposal for Observability presented in chapter 3.

- What is the best way to achieve efficient auto-scaling for cloud-native applications ?

Among the essential features of cloud-native applications, elasticity is significant. Cloud providers offer the capacity to provision compute, network, and storage capacity automatically. This almost unlimited and infinite provisioning capacity available at any time could make it possible to achieve exact sizing to fit volatile workloads. Indeed, uncorrelated resources provisioning to workloads may result, at best, in overspending of unused resources and, at worst, degraded quality of service and bad user experience. Our contribution in the chapter 4 presents Observability-driven auto-scaling for cloud-native applications aiming to leverage Observability to provide accurate input for auto-scaling and demonstrate the benefit of proactive auto-scaling mechanism to enhance end-to-end latency and success rate during transient scaling situations.

- How to interconnect cloud-native applications and industrial Internet of Things Objects and Machines to benefit from the scalability, flexibility, and modularity?

Proprietary IoT devices and proprietary operating systems (OS) for IoT are dominating the market today. Compliant with Industry 4.0, our IoT and connected machines are used for many use-cases and need to achieve high levels of efficiency and profitability. Lectra Industry 4.0 connected machines need to bring agility and versatility to process and operate in various environments. We aim to define an architecture framework for generic cloud-native enabled solutions for IoT to enable flexible and agile end-to-end provisioning. Our contribution, in chapter 5, brings virtualization in IoT networks based on introducing SDR, SDN, and VNF paradigm-shifting. We describe a relevant set of use-cases and deployment scenarios with a proof of concept showing how the proposed architecture framework is prototyped.

Methodology

Being part of the R&D that designed and operated the new cloud services in a large company while conducting research proved to be a challenge full of opportunities. A proof-of-concept can rarely be conducted and even adopted in a large production environment.

Our methodology was based on the participation in all the tasks of developing and operating the cloud services at Lectra R&D as a core member of the R&D team. Furthermore, being the research referent, I was pushing to add state-of-the-art publications and research to the decision process around strategies, techniques, and tools.

Multiple evaluation and comparison of strategies were run on sandboxed Lectra environment to replicate the production environment before further testing or adoption.

Finally, for some advanced proof-of-concept or destructive tests, a testbed was designed to deploy state-of-the-art infrastructures and cloud-native applications.

We were already aware of the MAPE-K principles [1] [2], and the ideas of autonomous computing and advanced automation were always guiding our work during this thesis.

Thesis Contributions

The work realized during this thesis led to a series of publications :

1. N. Marie-Magdelaine et T. Ahmed, « Proactive Autoscaling for Cloud-Native Applications using Machine Learning », in GLOBECOM 2020 - 2020 IEEE Global Communications Conference, Taipei, Taiwan, déc. 2020, p. 1-7. doi: 10.1109/GLOBECOM42002.2020.9322147. [3]

2. T. Ahmed, A. Alleg, et N. Marie-Magdelaine, « An Architecture Framework for Virtualization of IoT Network », in 2019 IEEE Conference on Network Softwarization (NetSoft), Paris, France, juin 2019, p. 183-187. doi: 10.1109/NETSOFT.2019.8806650. [4]

3. N. Marie-Magdelaine, T. Ahmed, et G. Astruc-Amato, « Demonstration of an Observability Framework for Cloud Native Microservices », in 2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM), avr. 2019, p. 722-724. [5]

Road map of the Thesis

This thesis is organized as follows :

Chapter 1 provides an overview of the cloud ecosystem, an historical understanding of the situation and useful definitions.

Chapter 2 present a detailed display of the motivation behind advanced monitoring, the pitfall of previous techniques and tools which led to the breed of a new strategy called Observability.

Chapter 3 present our architecture proposal for Observability in Cloud-Native Applications and the methodology and the components that enable us to define it.

Chapter 4 present our cloud-native framework enabling proactive auto-scaling of cloud-native applications to achieve a better quality of service.

Chapter 5 is our proposition of transposition of the cloud-native paradigm and other state-of-art technologies to IoT.

We conclude with a summary of our work, and we provide some insights for future work in Conclusion and perspectives.

Chapter 1

Concepts and Definitions

1.1 Background on cloud computing

The first computers were huge, bulky, fragile, costly, and challenging to operate. They were kept in machine rooms, with all the precautions around them and accessed limited to a handful of engineers to use them, work orders sent to computer engineers that ran them and returned the result. Time-sharing methods were used to enable multiple users' access to a computer. Very soon, terminals enabled users to interact with mainframes remotely. This communication between the client terminal and the "server" mainframe was, at first, only in Local Area Network (LAN).

However, soon, communication became nationwide with the introduction of ARPANET, a governmental project to enable access to remote computer resources. ARPANET led to the creation of the INTERNET and the World Wide Web, popularising computer usage and remote resources access.

From mail to web banking through news websites, the world wide web became trendy for organizations and people. The increased adoption of the personal computer in both organizations and homes increased the usage of the web. During the late 1990s, according to US Department of Commerce [6], it was estimated that traffic on the public Internet grew by 100 percent per year, while the mean annual growth in the number of Internet users was thought to be between 20% and 50% [7].

However, while some universities provided personal web page hosting to students and employees, web hosting was costly. A server, a computer dedicated to answering web requests, had to be powered on and connected infallibly to the Internet. Some companies soon started to offer web hosting services from advertisement-funded "free" hosting like Lycos, Geocities, AOL Hometown to private servers hosted in then called "colocation center". Soon called data centers, those places started to offer features improving connectivity, reliability, and security way above the previous "machine-room" and other "home-hosted servers". Very soon, hosting companies and web providers started to look for a way to save space and consolidate server usage.

In 1999, at the DEMO conference [8], a start-up named VMware introduced its products: Workstation, GSX, and ESX, a line of hypervisors enabling server virtualization. While tools and techniques enabling shared computer use were already out, they required specific hardware and/or operating system. Virtualization works with a hypervisor, an emulator that provides an abstraction layer between the host computer and a guest virtual machine. VMware hypervisor enabled the virtualization of the scarce and valuable servers hosted in the data center. Many competitors followed by developing their hypervisor solutions, open-source solutions such as Xen and KVM, and proprietary ones such as Microsoft Hyper-V and Oracle VM Server.

During those years, virtualization enabled to drive cost down, simplify management, consolidate servers farm, and improve reliability. By 2006, processors manufacturers (AMD [9] and Intel [10]) created new processors instructions for hardware-assisted virtualization. The same year Amazon Web Services introduced AWS Simple Storage Service (S3) and Elastic Compute Cloud (EC2), the first Storage-as-a-Service and Infrastructure-as-a-Service Public Cloud offers. Google, Microsoft, and many competitors, called Cloud Providers, followed in the following years with their Service-oriented architecture providing Infrastructures, Platforms, and Software as services available freely and sometimes free of charge to anyone with an internet connection.

In 2011, Mell and Grance in [11] from the National Institute of Standards and Technology (NIST) published the most cited and agreed upon definition of Cloud Computing. They defined the cloud essential characteristics as :

- On-demand self-service
- Broad network access
- Resource pooling
- Rapid elasticity
- Measured service

On-demand service means that cloud resources must be provisioned automatically and at will by customers without any intervention from the cloud provider. Those resources must also benefit from broad network access, making them accessible from any device and any connection through the internet.

Cloud providers must pool and regroup resources that are made available to customers in a multi-tenant model. While this created a layer of simplification from the customers, it also hid the exact location, only providing vague information such as "zones" or "data-center country."

Cloud providers must enable their customers to scale up and scale out their resources at will rapidly.

Additionally, as cloud consumers use resources in a self-service fashion, they must be billed according to the resources usage. Usage's measurement automation does this billing.

Mell and Grance also defined the service models, how services are consumed, as :

- Infrastructure as a Service (IaaS)
- Platform as a Service (PaaS)
- Software as a Service (SaaS)

Cloud providers offer multiples resources and services according to different services models. The primary factor differentiating them is the level of abstraction over the infrastructure.

The service model with the lower abstraction level is called Infrastructure-as-a-Service (IaaS). Resources offered through IaaS are bare compute, network, and storage resources. IaaS cloud customers are provided with essential resources on which they manage their operating systems and applications. However, the underlying layers, notably the data-centers networks (e.g., firewall, router), storage, servers, and virtualization are not accessible by the cloud consumers.

Cloud providers can provide the capability to consumers to deploy onto a ready-made platform. This Platform-as-a-Service (PaaS) model abstracts the virtual machine and essential resources (network, servers, storage, and operating systems) while directly supporting programming languages, libraries, services, and tools. Consumers retain some controls through environment parameters defining the platform configuration allowed by cloud providers.

Software-as-a-Service (SaaS) is the service model where cloud providers provide customers with software already installed and maintained on their cloud infrastructure. Those software and applications are accessible via web-browsers, thin clients, desktop interfaces, or even APIs. The consumer does not manage any element of the technology stack in the data-center; they only use it.

Figure 1.1 illustrates the different cloud computing services models and highlights the components ownership.

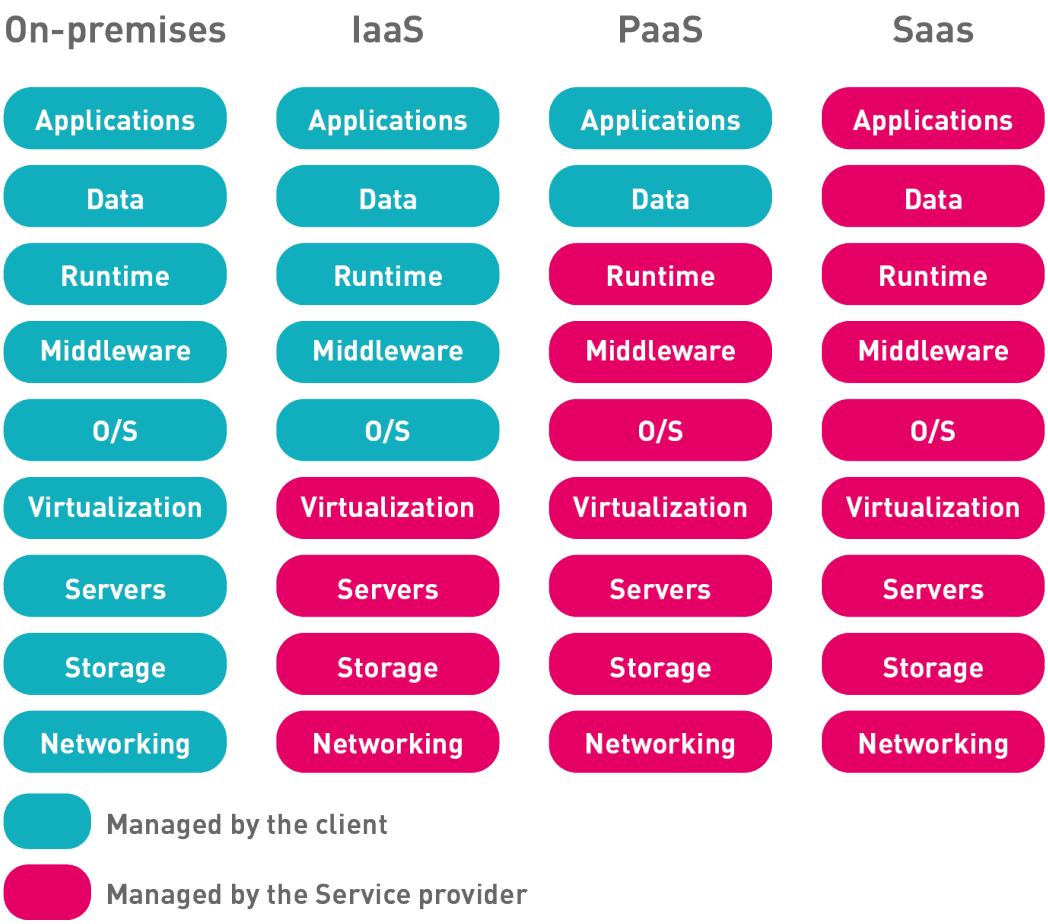


FIGURE 1.1: Cloud Computing Service Models

The deployment model defined how the cloud is accessed and by who were also defined by Mell and Grance as such :

- Private cloud
- Community cloud
- Public cloud
- Hybrid cloud

Cloud Computing deployment models define who are the cloud consumers.

In a private cloud, the cloud infrastructure is built for a single organization's specific and exclusive use (e.g., company, government). A private cloud may be built, maintained, and even hosted by third parties but remain under a single organization's sole control and use. A private cloud is different from an on-premises data center as they follow the cloud's essential characteristics and services models.

When the cloud infrastructure is provisioned for the use of a community with shared concerns, it forms a group of consumers who share their own cloud infrastructure resources. It may be owned, managed, and hosted by

the community or third parties. However, it remains dedicated to the usage of the community.

The public cloud is the most common type of cloud. It may be owned, managed, and hosted by private companies, academic institutions, or the government. However, everyone can access it and use it through the internet. The cloud provider generally hosts it.

Hybrid clouds are clouds composed of two or more types of cloud deployment models. It can be a temporary or permanent link between a private cloud and public or community cloud. It is a deployment model used for scenarios of cloud bursting or cloud migration.

We can add a new type of deployment model that is gaining traction :

- Multi-Cloud

The multi-Cloud approach uses multiple clouds as infrastructure for the same organization. The benefits are also multiple additional features available across different cloud providers, reduced vendor lock-in, higher total availability, mitigation against disasters and outages.

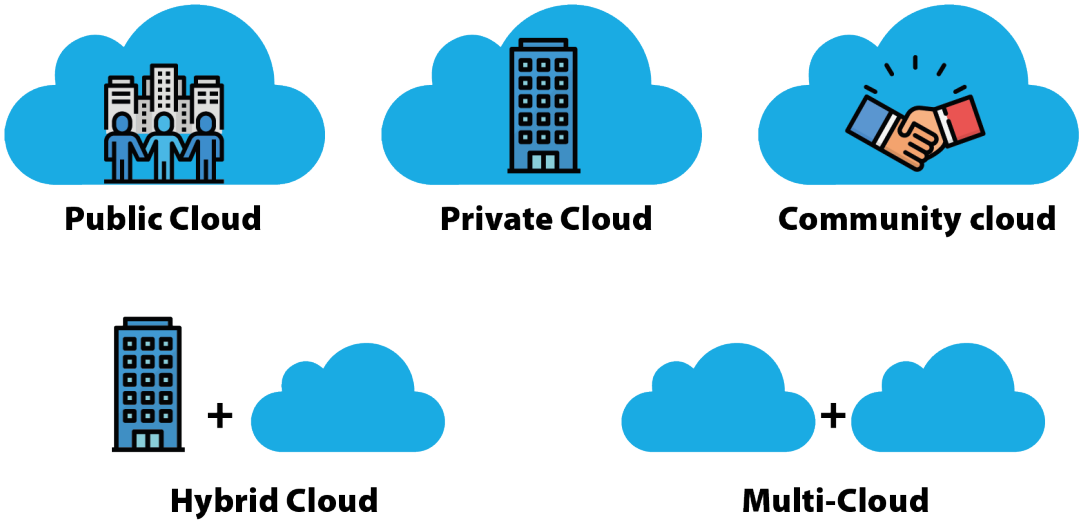


FIGURE 1.2: Cloud Computing Deployment Models

The adoption growth was phenomenal. As of 2021, 94% of enterprises use at least one cloud service. The global public cloud service market is projected to reach $266 billion in 2020.

The Covid-19 crisis increased the cloud market by a whopping 31.4% in 2020. Cloud Computing is now a vital part of the modern world. Thus, cloud computing-related challenges, Quality of Service, reliability, availability, elasticity drives innovation, research, and capital. Soon Cloud Computing actors would gather and federate to create an initiative around Cloud Native Technologies.

1.2 Cloud Native

During the 2010s, railroad companies, travel agencies, and even apparel retailers adopted cloud computing. In addition, many organizations and companies from start-up to Fortune 500 realized that they needed to become software companies even if that was not their core business. Therefore, cloud computing led to innovations and solutions appearing everywhere, and most organizations choose to open-source those internal software projects. Open Source Projects thrived on GitHub, a SaaS provider of tools for collaborative software development. However, all those Open Source projects born from the cloud computing boom, cloud natives projects, were disorganized and faced difficulties for maintenance and leadership. At some point, tech journalists were writing about a "container orchestration war" between open source projects.

The Linux Foundation [12] was already building sustainable ecosystems around open-source projects to accelerate technology development and commercial adoption since the 2000s. In 2015, when Google pushed the open-source container orchestrator Kubernetes 1.0 release, the Linux Foundation created the Cloud Native Computing Foundation (CNCF) to federate, align, and build a sustainable ecosystem for cloud Native software.

The CNCF Cloud Native Definition [13] is as follows:

"Cloud-native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure and declarative APIs exemplify this approach.

These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.

The Cloud Native Computing Foundation seeks to drive the adoption of this paradigm by fostering and sustaining an ecosystem of open-source, vendor-neutral projects. We democratize state-of-the-art patterns to make these innovations accessible for everyone."

As of the 2020 CNCF Annual Report [14], 94% of respondents were using containers in production, with 83% using Kubernetes to orchestrate them. As of 2021, the CNCF gathers 656 members, combining a total market cap of 20.6 Trillion dollars.

In the rest of this section, we will describe the Cloud Native environment in two subsections: Cloud Native Infrastructure and Cloud-Native Application.

* Cloud-Native Infrastructure: Infrastructures are both the hardware and the software responsible for supporting applications.

* Cloud-Native Application: Cloud Native applications are specifically designed for and taking full advantage of the cloud-native environment that is to say cloud-native applications and infrastructures together.

1.2.1 Cloud Native Infrastructure

As introduced previously, Cloud Native Infrastructures represent the foundation on which applications runs. It includes every layer of the stack from the physical world up to any applications supporting business workload. From data center to orchestrator, almost all the components have evolved to adapt to the new Cloud Native paradigm. This section describes the power of the modern cloud and how it has evolved to become state-of-the-art Cloud-Native Infrastructures.

Datacenter

The Internet and cloud computing growth have created an ever-increasing need for servers, thus driving the growth for data centers. They ultimately represent the building block of every commodity available online. Far from the simple rack computer hosted in the In the 2000s, modern data centers face challenges such as reliability, security, flexibility, and ecology. Those are tough challenges when the need for more storage, compute, and network resources growth exponentially each year [15].

As more and more organizations switch to public clouds, major cloud providers are now hosting their previous IT workloads. The scale of data centers has then become a vital point of interest. Between user-generated content and enterprise workload, efficiency is the keyword. Data centers use an estimated 200 terawatt-hours (TWh) each year [16]. This is approximately 1% of global electricity demand. Some worst models predict that electricity used for information and communication technologies could exceed 20% of global electricity demand by 2030. This could happen, especially if trends like cryptocurrencies mining continue to grow [17]. This trend has pushed major data centers market players to achieve ever-increasing efficiency and reduce environmental footprints. Facebook has launched the Open Compute Project [18] an open-source initiative aiming toward power efficiency, server density, cost reduction, and environmental impact reduction. This initiative has enabled them to improve their energy efficiency on new data centers by 38% and the building cost by 24% [19]. This effort from major cloud players to increased efficiency and consolidate their data center enabled organizations to migrate their IT systems to the cloud while reducing their power consumption carbon footprint by up to 90% [20] [21].

Modern Cloud Native data centers are now called "Hyperscale data centers" because they can provision new resources and scale to ever-growing compute, networking, and storage capacities by both increasing their size and their density. Everything is optimized for cost efficiency, computing density, reliability, and sometimes even sustainability from the building to the hosted hardware. Microsoft Azure, Amazon AWS, and other large public

cloud providers are now able to design and manufacture specific components to fit those needs.

Servers

Servers in Cloud Native data centers have also evolved. Far from the single physical machine hosted in a private room, modern data center host high-density racks with up to 64 cores per blade (3rd Gen AMD Epyc CPU) and up to 24 blades per 3U rack. Some cloud providers maybe go as far as to order custom CPUs [22] and design custom hardware [23] to get the edge on their competitors.

In almost all cases, public cloud customers never interact with the physical servers, not even remotely. This is a change from the times of hosted bare-metal servers. Instead, cloud providers rent "slices" of servers called "instances" or Virtual Machines (VM). They generally take their margin on the cost per virtual CPU sold. This required the sweet spot between density, performance, and price. Some cloud instances may also feature special hardware such as Graphics Processing Units (GPU) or specific network interface cards (NIC).

Cloud computing defines this model of deployment as Infrastructure-as-a-Service (IaaS).

Virtualisation

Virtualization is a core technology behind cloud computing and cloud-native infrastructure. Virtualization enables cloud provider resource pooling. Compute, network and storage can then be sliced and served in custom quantity and quality to customers. Virtualization transforms bare-metal resources into virtual machines and provides an isolation and abstraction layer between the physical and logical servers. Virtualized servers are called Virtual Machines (VM).

A Hypervisor is a special software ran on the physical server, managing the host and acting as an abstraction layer for VM running on top of it. We distinguish two different types of hypervisors depending on the type of abstraction layer. On the one hand, Type 1 hypervisor provides a direct slice of hardware without running through the OS. One the other, Type 2 hypervisor, which slices resources through the host's operating system. Notable hypervisors are Microsoft Hyper-V [24], Linux KVM[25], Vmware ESXi [26], and Xen [27]. Public cloud providers use them. For example, Digital Ocean Public Cloud [28] has open-sourced some of the software running their cloud; we can note the usage of libvirt and KVM in their public repository [29].

Public cloud providers can almost instantly assign tenants' virtual machines to their pool of physical resources. Customers can choose the location, size, operating system, and configuration of their VMs and connect to them a few seconds later. Virtual Machines are logical objects; they can be copied,

migrated to another host, resized, backed up. In case of failure of their physical host, they are re-instantiated sometimes live on another. However, applications running on them must have been designed to accommodate those features in order for them to work correctly.

Software Defined Networking

Virtualization does not stop at servers. Data centers used to rely on a significant amount of advanced engineering dedicated to networking. However, legacy network architecture relied on many network appliances that required configuration, maintenance, and upgrades. The advent of Cloud Computing and the new software paradigm pushed the need for a significant evolution in networking. Software-Defined Network (SDN) emerged promoting network solutions that are directly programmable, agile, centrally manageable, open standards-based, and vendor-neutral. Virtualization of network functionality that was physical appliance is called Network function virtualization. This enables the replacement of physical network appliances by generic computing hardware. Together, SDN and NFV enable the creation of Virtual Network Functions (VNF) such as virtual routers, virtual switches, virtual 4G eNodeB [30], and even 5G-NR gNodeB. Google was among the first to deploy a production level SDN network with their B4 project in 2011 [31], but nowadays, all the major network vendors offer SDN solutions [32][33][34]. Together, Cloud Computing and Software-Defined networks have revolutionized the IT industry.

Containerization

Running isolated programs on top of a host operating system was not a new idea. OS-level virtualization Solutions such as BSD Jails or chroot existed. However, those were crude and far from easy use by developers. Until a day in 2013, Docker was introduced. It quickly became the industry standard for containers. Containers are lightweight packages containing a file system image and associated configuration. They contain binaries and required libraries but none of the operating system or kernel. It enables packaging, shipping, deployment of applications with their dependencies without the entire underlying operating system. Containers differentiate themselves from virtual machines that run on top of a hypervisor (type 1 or 2) with interaction with emulated or real hardware resources. Containers interact with their host kernel. Figure 1.3 illustrates how containerization is another step of resource sharing. Containers provide some isolation and the possibility to throttle resource usage, allowing multiple containers to run on a single host. Containers, being lighter, can be dynamically instantiated, created, duplicated, deleted much faster than virtual machines.

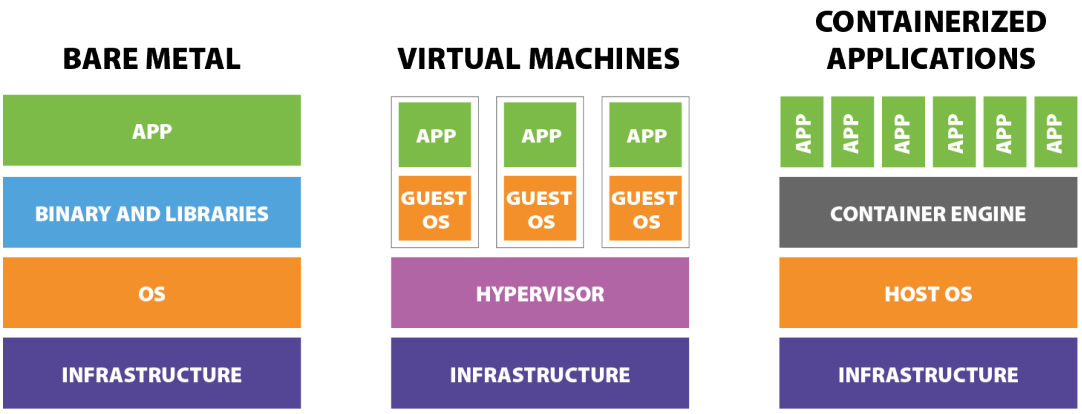


FIGURE 1.3: Bare-metal, Virtual Machines, and Containerized Applications

Serverless Computing

Serverless computing is a cloud computing service model derived from Software-as-a-Service (SaaS) from which the cloud provider manages the entire application support stack. Developers concentrate on writing code that is transparently run by the cloud provider platform. On paper, serverless computing solves all the system administration issues linked to cloud-native applications and offers developers an experience that just works. However, serverless computing is sprinkled with limits, fallacies, and pitfalls [35].

When Google created Google Cloud Platform, their first product was Google App Engine. The idea was to provide customers with a platform managed by Google instead of requiring customers' management. Platform-as-a-Service (PaaS) raised the abstraction layer higher; however, the justified fear of limitations, vendor lock-in, portability concern meant that PaaS did not become the major deployment model in cloud-native solutions. Unlike any other IaaS solutions, based on VMs or Containers, PaaS Cloud Providers enable their customers to write valuable business code without wasting time with operation and administration. This type of approach is called "No-OPS" and is focused on reducing the time to market. Despite several trials, PaaS has not reached the level of adoption of Containers.

Function-as-a-Service (FaaS) is a model of cloud computing deployment which relies on independent atomic functions that run on-demand. Run on-demand means that they do not run continuously but only when solicited. They start up, process the eventual payload with their trigger event, and often terminates within milliseconds. FaaS offers both a very high abstraction design and low cost. Customers do not manage the underlying infrastructure, nor anything except their code. FaaS provides state-of-the-art and built-in security, scaling, monitoring, logging, and debugging capability. A major downside resides in the vendor lock-in risk, Functions environment are often plagued with compatibility and interoperability issues.

Cloud providers have integrated containers in their offers with Container-as-a-Service (CaaS). CaaS enables customers to deploy their containers to the

cloud on a managed abstraction layer. The advantages rely upon the portability and compatibility of standardized containers avoiding vendor lock-in. Container-as-a-Service is seducing offers for organizations that have already containerized their applications. It offers them the ability to hand over container orchestration and focus on creating and maintaining their containers. It solves some limits and issues of PaaS and FaaS but lowers the abstraction layer.

While serverless computing seems to solve many of the issues encountered by most organizations, its many drawbacks have pushed the adoption of orchestrated containers as the most adopted solution for cloud-native infrastructures.

Automation

Among the characteristics of Cloud offers, there is self-service capability through Applications Programming Interface (API). It enables customers to automate their interactions with cloud providers. Cloud adoption and software architecture trends have multiplied the number of unique resources that require management. Operation engineers are faced with the challenges of caring for thousand of abstracted virtualized resources. If left unchecked, security vulnerability or even outages can occur. Manual operations take time and add risk. Human errors, like manual misconfiguration, increased downtime, and decreased reliability. Infrastructure can be defined with code and executed with perfect repeatability, thus reducing cost, increasing speed, and reducing risk. This approach called Infrastructure-as-Code is part of a movement called DevOps, combining software development (Dev) and IT operations (Ops).

Dynamic Management

When an organization has automated their Cloud Native infrastructure, they are only an "if block" away to implement dynamic management. Indeed, cloud computing allows flexible and on-demand resources allocation and de-allocation. Furthermore, by correlating resource usage with automation, organizations can closely match their needs with the provisioned resources, thus reducing over-provisioning. This process of matching cloud resources to usage is called autoscaling.

The use-cases of Dynamic management are limitless, as long as APIs are available, code can be written to modify infrastructure dynamically.

Orchestration

Orchestration is the final form of automation. Orchestrators take care of every aspect of management from deployment to end-of-life. While dynamic management tools execute tasks and follow a process. Orchestration runs during the entire lifetime of infrastructures. Orchestration makes sure

that both infrastructure and applications run as desired, taking decisions on automating deployment, configuration, scaling, diagnostic, management.

1.2.2 Cloud Native Applications

In the same fashion as infrastructure, web application design has profoundly evolved. Every aspect of web applications life cycle has advanced, how they are designed, coded, tested, packaged, released, and operated. Our modern world runs on applications that we expect to be reliable, fast, permanent, and ubiquitous. Organizations need to offer seamless, reactive, and always-on services to ever-demanding users. Google, Amazon, Facebook, Salesforce, Apple, and many others cannot afford outages or downtimes. Organizations' perquisites have pushed innovation and research to achieve the current state of the art. This section will describe every aspect of Cloud-Native Applications, from how they are built to their strengths and weaknesses.

Microservice Architecture

Alongside the evolution of infrastructure, web applications have evolved during the last two decades.

At the beginning of the 2000s, web applications were designed as "monolith," also called 3-tier. One web server serving as a frontend to the users, one application server running the business logic, and a database storing the data. This design, while simple, had many flaws. Development was very slow in a huge codebase containing the entire application. It was not easy to understand for new developers and challenging to maintain even for old ones. The process to modify, test and deploy were slow, heavy, sometimes painful as risks increased with the number of lines of codes and the age of the project. The technological choice in languages and framework had to be kept for the project's entire lifetime. Scaling or modify the underlying infrastructures of those monolith web applications was sometimes impossible. In order to solve those issues, enable some modularity and interoperability, organizations started to build service-oriented applications (SOA) and Enterprise Service Bus (ESB). However, it was clear that those designs needed more refinement. SOA and ESB quickly evolved, and among web applications, a major architecture design has emerged in the quest to take advantage of the new cloud technologies. Microservice Architecture is nowadays the primary design adopted to create new web applications and reform old ones. Microservice has been adopted by Google, Amazon, Netflix, and most of the world.

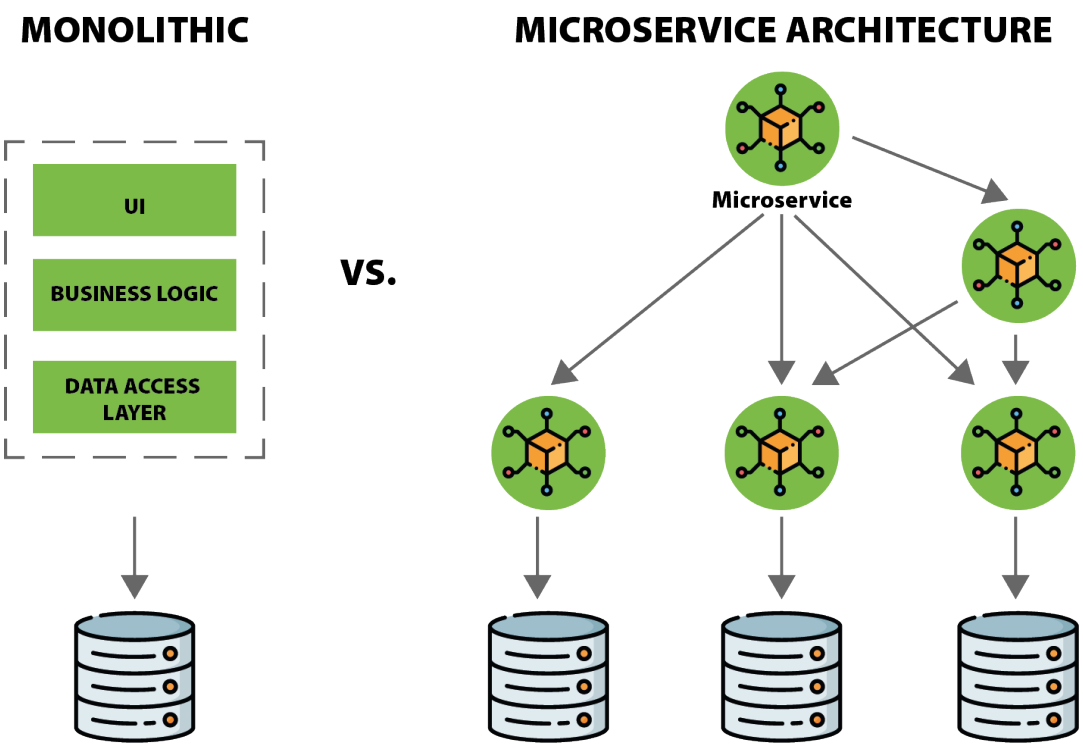


FIGURE 1.4: Monolith vs Microservices Architecture

Microservice Architecture aims to guide towards small, autonomous, decoupled components working together. Microservice architecture provides many benefits but also some new challenges.

Multi-tier and Service-oriented architecture were already ways to divide large monolith applications into smaller components. However, they were not successful in solving the issues of the monolith while bringing the new issues of complex distributed systems. The microservice approach brings more than just decoupling; it comes with a set of best practices. From those best practices [36], [37], [38], [39], the microservice architecture enables a set of features: Independent deployability, Domain Driven Design, Data ownership and information hiding, Small sized, and Flexibility and modularity. They can be detailed as follows:

- Independent deployability

One of them is independent deployability; being an independent, self-contained microservice offers freedom of deployment. At any given time, microservices can be upgraded, re-deployed, at will, at any moment without impact on other microservices or users. In addition, independent deployability enables developers to push their changes more often, increasing the number of releases and thus decreasing the risk associated with new releases. Independent deployability also accelerates development time leading to a shorter time to market.

- Domain Driven Design

Microservice architecture also benefits from Domain-Driven Design (DDD) by defining clear boundaries between microservices. DDD limits the scope of a microservice, and the team efforts focus on delivering a specific business feature. As a result, it becomes easier and faster to deliver helpful business features.

- Data ownership and information hiding

Microservices architecture imposes data ownership by not sharing databases but API from which other microservices interact to get data. Data ownership is required for independent and loose coupled services that do not require data integrity checks between multiple services.

- Small sized

Being small is a significant benefit; microservice are sized around atomic business features or functionality. The more features a service contains, the more lines of codes need to be written. The codebase grows with every additional feature, eventually becoming too large for small teams to understand completely. As a result, it becomes more difficult to fix bugs, add features, and overall maintain an extensive service.

Small codebases also make smaller executables and lighter containers which can be started quickly, shorting the overall time to start the entire application.

Small service can be rewritten or refactored quickly, allowing for faster "fail-fast" approach.

- Flexibility and modularity

Web applications built upon microservices can be extended, reduced, and modified by creating, refactoring, and even retiring microservices. This enables a more flexible and forgiving approach to software engineering. If a feature needs to be modified, added, or removed, a monolith of code does not need to be rewritten, tested, and released.

Each microservices being independent and only interacting through API and events. They can be written in any language, use any database or third-party services. This ability of microservice architecture to be built with different languages and technologies is called technological heterogeneity. Technological heterogeneity enables more flexibility than previous architectures. Languages and technologies can be tested with limited impact and consequences. Some features may better fit some technologies enabling microservices to use the best choice for their specific use-case.

- Robustness

Moreover, in case of failure, either from bugs in the code or external reasons, microservice architecture can limit the "blast radius" of the failure. Only the feature powered by the affected microservice is most likely to suffer. Moreover, microservices do not share internal data and state, thus making the whole application more resilient.

- Scalability

Even in the case of performance issues due to under-sizing, microservice architecture can scale by running more replicas of the saturated features. While monolith needs to be scaled entirely and relies on huge dedicated hardware to fit their needs, microservice can run on much smaller, less powerful server and achieve better scaling.

Microservice architecture has brought a list of advantages to organizations that have switched to the new paradigm. We can cite:

- Flexibility and modularity
- Independent deployability
- Faster and more frequent releases
- Robustness
- Scalability

However, alongside those advantages, some new challenges as appeared. Difficulties arose from transforming the single monolith executable into a myriad of microservices, each one with its own needs. This intrinsically defined microservices-based application as complex distributed systems. Most of the complexity that was inherently in the codebase has moved to operations.

1.2.3 Operations

The operations regroup all the management, deployment, integration, maintenance, monitoring of applications, networks, and servers. In cloud times, building networks and deploy servers is not about moving hardware but writing code. Google pioneered, in 2003, this new paradigm of operations and called it Site Reliability Engineering (SRE) [40]. During the same time, a set of practices combining software development (Dev) and IT operations (Ops) called DevOps emerged. Both prescribe the same philosophies:

TABLE 1.1: Table SRE DevOps

DevOps	SRE
Reduce organization silos	Share ownership with developers by using the same tools and techniques across the stack
Accept failure as normal	Have a formula for balancing accidents and failures against new releases
Implement gradual change	Encourage moving quickly by reducing costs of failure
Leverage tooling and automation	Encourages "automating this year's job away" and minimizing manual systems work to focus on efforts that bring long-term value to the system
Measure everything	Believes that operations is a software problem, and defines prescriptive ways for measuring availability, uptime, outages, toil, etc.

In order to implement the principles and concepts promoted by Agile, DevOps and SRE organizations adopted a series of tools.

- Source Code Management (SCM)
  Individuals or teams of developers need to manage their source code. They need to backup, version, centralized it, and collaborate around it. Source Code Management tools such as Git and its implementation GitHub, GitLab, or Bitbucket offers those features.
- Agile Project Management
  Agile methodologies such as Scrum and Kanban are frameworks used for developing, delivering, and guiding teams to develop software. While those techniques can be used with paper and blackboards, software implementations are now the norm in the software industry.
- Continuous Integration (CI)
  Continuous integration relates to the best practice of integrating all the developers' work as frequently as possible. Indeed the longer developers work independently on the shared code more difficult merging all the modifications becomes. Continuous integration promotes the automation of merging developers' work and work in conjunction with automated testing. The application or microservice is tested and built automatically with each developers' commis to the source code. Other requirements such as compliance with regulations and licensing may be tested in this step.

- Continuous Delivery (CD)
  Beyond continuous integration, continuous delivery offers the possibility for humans to review the result of continuous integration and decide if the latest produced release can be deployed to production.
- Continuous Deployment (CD)
  Another step further, continuous deployment simply automates the entire pipeline of processes from the developers' commits to the deployment on the production environment.
- Monitoring
  Once applications are deployed, operational engineers (OPS) must check their health state and make sure they are up and running as expected. Monitoring systems are set up in order to retrieve health states for servers and applications. When a malfunction occurs, they must raise the alarm and alert OPS. OPS are responsible for the diagnosis and repair of defective servers and applications. DevOps and SRE best practices encourage the use of automated monitoring and alerting systems measuring availability, uptime, and outages.

Together those tools implementing those features are known as a toolchain. In previous paradigms, the processes carried out by the DevOps toolchains were mainly manual. However, the microservice architecture brought additional work to the operations. More components are coded, released, tested, deployed, and monitored. New strategies and tools are needed to adapt to operate those new cloud-native applications. Those operational requirements are even more significant when organizations use cloud-native applications in production settings. They then have to equip themselves to ensure their Cloud-Native Applications' quality of Service (QoS) in the stringent production environment. Indeed ensuring the correct behavior and health of Cloud-Native Applications is complex due to their distributed nature.

1.3 Conclusion

Technologies have evolved. We have seen new programming languages, new architecture paradigms, and new team management strategies. From the Agile movement to the DevOps methodologies, software engineering has seen numerous evolutions. As a result, cloud-Native Applications built around microservice, containerization, automation, and hosted in the public cloud are now the norm.

During the last decade, every aspect of building and operating web applications has changed. At the same time, users and organizations are now expecting more than ever performance and reliability from now-ubiquitous web applications. Together those factors contribute to new challenges for organizations, the software industry, and academic research. As stated previously, in this thesis, we will focus on the challenges around management, monitoring, and quality of service of those microservices-based applications in the public cloud environment.

Chapter 2

Towards Observability in Cloud Native Application

Among the challenges of operating Cloud-Native Applications in the production environment, maintaining them and knowing when they experience failures or reduced quality of service is of utmost importance. Both Cloud providers and cloud users need to monitor resources usage, performance, and availability. Moreover, in case of failures, partial or total outages, they rely on monitoring data to diagnose, troubleshoot, and ultimately repair. Without that information, the operation of cloud-native applications can be compared to flying a plane without visibility nor instruments.

The Cloud Native paradigm shift has brought an overall increase of complexity, which also greatly increased the difficulty to observe and visibly understand the health, performance, and behaviors of Cloud Native Applications.

In this chapter, we will present a detailed display of the motivation behind advanced monitoring, the pitfall of previous techniques and tools which led to the breed of a new strategy called Observability.

2.1 Monitoring : an essential prerequisite for production environment

Our modern world relies upon and expects web applications, independently of their architecture, to run 24/7 at optimal performance levels. Yet, the target of 100% availability remains an illusory mirage [41]. Every year, major cloud providers and services experience major failures from performance degradation to complete outages. Nowadays, minutes after failures, social networks are filling with posts complaining about unexpected downtimes [42]. Worst some failures may propagate between cloud services and bring entire portions of the Internet down with them [43].

Moreover, some outages have been caused by cyber-attacks such as DDoS attacks [44] or security breaches. Sony Playstation, Electronics Arts, Valve Steam and many others companies have been victims of hackers from not using DDoS detection and mitigation services.

In addition to the loss of services to customers, outages cause financial and reputation harm to organizations [41]. A list of 2020 outages reveals that no company is safe from outages [45]. Microsoft, Google, Amazon, IBM, Slack, Github, and Zoom provide services essential to professional activity worldwide.

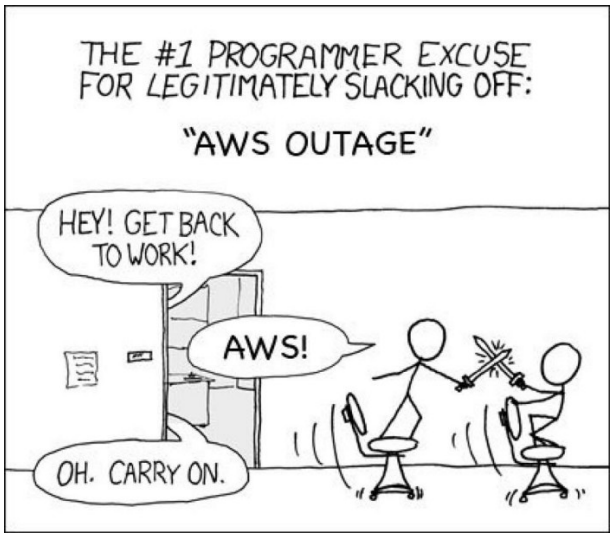


FIGURE 2.1: Webcomic sarcasm about Amazon Web Servcices outages [xkcd.com]

Millions of dollars can be lost; users can switch to competitors: those are events that any organization wants to avoid, and when they happen, reduce and limit their impact. Cloud providers strive to achieve better availability and often commit themselves to Service-level agreements (SLA). For example, Microsoft Azure virtual machine service provides a 10% credit to users affected with a monthly uptime of less than 99.95% (21.92 minutes downtime monthly). Whenever those SLAs are violated, cloud consumers can ask to be compensated. Monitoring uptime is more than a technical necessity; it is also a commercial, financial and legal one.

Additionally, Cloud-Native Applications hosted on Public Cloud are billed on a "pay-as-you-go" basis, meaning that an increase in resources usages is directly correlated to an increase in the billed amount. As a result, monitoring new resources used and billed becomes a challenge to avoid over-budget spending.

Monitoring is crucial to check the health of applications. Modern monitoring does more than alerting when failures happen. They can prevent some issues that, if unnoticed, would have cause outages. Bugs and issues may have been hidden in the development environment and only reveal themselves once connected to the production environment.

Even security becomes more and more challenging in the context of public-cloud, internet-connected resources are scanned by robots waiting at the first unpatched vulnerability to penetrate systems.

In the previous paradigm, OPS would simply take care of "The Server", a carefully maintained system running a single application. Any issue was easily diagnosed; if something was wrong, it was "The server". Fleets of hundreds of dynamically instantiated containers running on auto-scaled virtual machines in the Cloud represent a very much more complex challenge. Monitoring is required for almost every aspect of Cloud-Native Applications' lifecycle.

2.2 Related Work on Cloud Monitoring

Majors surveys in literature, Aceto *et al.*, 2013 [46], Ward and Barker, 2014 [47], Fatema *et al.*, 2014 [48] and Syed *et al.*, Syed *et al.*, [49], compiling quality research, provide a wide view and in-depth analysis of cloud monitoring. They describe and cite state-of-art requirements, methods, strategies, motivations and tools to monitor cloud infrastructures.

This section will detail state-of-art motivations, requirements, methods, strategies, and tools to monitor cloud infrastructures. Then we will analyze how they perform compared to existing cloud monitoring requirements.

2.2.1 Motivation for Cloud Monitoring

Motivation, also called purposes for cloud monitoring, depends on the perspective (providers and/or users), and the deployment model of concerned Cloud.

According to Syed *et al.*, [49], motivations can be reduced to 3 main categories: Billing, Performance monitoring and Efficient use of Resources.

Ward and Barker, 2014 [47] added additional categories: Performance uncertainty, SLA enforcement, Defeating abstraction, Load balancing Latency, Service Faults, Location.

Fatema *et al.*, 2014 [48] defined the motivation as Accounting and billing, SLA management, Service/resource, provisioning, Capacity planning, Configuration management, Security and privacy assurance, Fault management.

Aceto *et al.*, 2013 [46] also defined similar categories: Capacity and resource planning, Capacity and resource management, Datacenter management, SLA management, Billing, Troubleshooting, Performance management, Security management.

TABLE 2.1: State of the art on Cloud Monitoring ...

	Motivation for cloud monitoring	Cloud monitoring requirements
Aceto *et al.*, 2013 [46]	Capacity and resource planning Capacity and resource management Data center management SLA management Billing Troubleshooting Performance management Security management	Scalability Elasticity Adaptability Timeliness Autonomicity Comprehensiveness, Extensibilty, Intrusiveness Resilience, Reliability, Availability Accuracy
Fatema *et al.*, 2014 [48]	Accounting and billing SLA management Service / resource provisioning Capacity planning Configuration management Security and privacy assurance Fault management	Scalability Portability Non-intrusiveness Robustness Multi-tenancy Interoperability Customizability Extensibility Shared resource monitoring Usability Affordability Archivability
Ward and Barker, 2014 [47]	Performance uncertainty SLA enforcement Defeating abstraction Load balancing latency Service faults Location	Scalable Cloud aware Fault tolerance Autonomic Multiple granularities Comprehensiveness Time sensivity
Syed *et al.*, 2017 [49]	Billing Efficient use of resources Performance monitoring	Agent-less Robustness Operational insight

Those four main authors roughly defined the same motivations for cloud monitoring. We can synthesize them as follows:

Accounting and Billing

Cloud resources being offered as pay-as-you-go, accurate consumption measurement must be done, requiring information such as the number of CPU computing hours, bandwidth used in and out of providers' networks, amount of storage used, et cetera [50].

Depending on the deployment model, the data used to measure usage and bill are not the same. While Infrastructure-as-a-Service (Iaas) resources are billed according to usual metrics, complex Platform-as-a-Service (Paas) and SaaS resources may require advanced and/or complex metrics which are difficult to foresee (example scenarios azure pricing calculator). Aceto *et al.*, 2013 [46] emphasized complex cloud resources that necessitate advanced monitoring. Surveys Aceto *et al.*, 2013 [46] and Fatema *et al.*, 2014 [48] both describe the need to monitor usage both by cloud providers and cloud consumers in order to get more transparency and verify billing.

Capacity planning

Both surveys Aceto *et al.*, 2013 [46] and Fatema *et al.*, 2014 [48] describe capacity planning as a motivation for cloud monitoring. Before cloud computing, capacity planning, especially for web applications, was one of the most challenging tasks for engineers and developers. (ref scholar : capacity planning web). They had to quantify the resources that had to be purchased before product launch. They used testing and stress-test to determine an estimated amount of workload that would need to be supported. Failure to do so resulted in degraded SLA, outages, and downtimes. Since cloud computing, cloud consumers can use as many resources as they can afford as long as their architecture is able to scale up [51]. Monitoring remains an important issue for both Cloud consumers and providers. It enables consumers to adapt their resources provisioning to workload. As for providers, they need monitoring tools to make sure they have enough resources to provide virtually infinite scale [52].

Performance Management

Cloud service providers are responsible for providing standardized resources by design. Each instance of the same product is supposed to behave similarly with the same level of performance. However, for many reasons such as aging, network issues, or others, some instances may not perform as expected and, even worse, be defective. Syed *et al.*, [49] point out how performance monitoring is important for both cloud providers and cloud consumers in order to verify that SLAs are not violated in those cases. Ward and al tested multiples instances of virtual machines and observed an up to 29% difference in performance between instances sold as being the same. Other works in literature [53] [54] conducted the same experiment with similar results. Ward and Barker, 2014 [47] define Cloud monitoring performance needs as essential:

• "To quantify the performance of a newly instantiated VM deployment to produce an initial benchmark that can determine if the deployment offers acceptable performance."

• In order to examine performance jitter to determine if a deployment is dropping below an acceptable performance baseline.

• To detect stolen CPU, resource over-sharing, and other undesirable phenomena.

• To improve instance type selection to ensure that the user achieves the best performance and value."

Moreover, performance for the cloud consumer is not only about compute performance; it can be about latency. All nodes and instances that are provisioned inside one cloud service provider's data center may not exhibit the same latency values. Ward and Barker, 2014 [47] noted this as an additional need for monitoring.

SLA Management

Aceto *et al.*, 2013 [46], Ward and Barker, 2014 [47], Fatema *et al.*, 2014 [48] and Syed *et al.*, [49] all discuss the subject of SLA management. They all agree on the necessity of monitoring QoS to verify SLA compliance. Aceto *et al.*, 2013 [46] see SLA enforcement in the context of cloud resources as a natural driving force to achieve more realistic SLAs and better pricing models e.g., [55] [56]. Ward and Barker, 2014 [47] describe how monitoring SLAs from a cloud consumer point of view is very important to detect when a cloud service provider's SLAs are unable to protect its users. It enables consumers to migrate to more robust architecture to protect themselves, demand compensation for breached SLAs, or migrate to other cloud providers. SLA management may also be a contractual requirement in some regulations (e-health, government data,...). In those cases, monitoring and enforcing SLA becomes more than motivation; it is a requirement [46].

Troubleshooting

The Cloud represents a complex infrastructure and a major challenge to troubleshoot [46], [48]. Indeed, looking for the root cause inside an immense amount of resources, each made from several layers of abstraction, is difficult [46], [47], [48]. Both cloud providers and cloud consumers need to monitor to troubleshoot. Depending on the service model, the root cause may lie under the responsibility of either providers or consumers. Ward and Barker additionally describe how monitoring of public cloud infrastructure, especially when the consumers are entirely dependant, helped avoid some disruptions and enabled a faster return to operational states.

Security

Fatema *et al.*, 2014 [48] discuss how security was a major issue in the adoption of cloud computing. Monitoring to detect breaches, intrusions, or attacks is essential in a public-faced multi-tenant environment. Monitoring for security reasons is a major motivation that must not be bypassed as the consequences may be dire [46].

2.2.2 Requirements for Cloud Monitoring

Cloud monitoring systems will not operate in the same environment and with the same constraints as legacy monitoring systems in the previous paradigm. In order to achieve the previously defined purposes, cloud monitoring systems need to conform to a new set of requirements. Aceto *et al.*, 2013 [46], Ward and Barker, 2014 [47] and Fatema *et al.*, 2014 [48] all describe requirements for cloud monitoring systems as follows:

Scalability

Cloud infrastructure can scale up from dozens to tens of thousand instances in a very short amount of time. While being a major advantage for the application's capacity to handle the high volatility of workloads, this capacity is a major challenge for cloud monitoring systems. Cloud monitoring systems must be able to cope with those scale events dynamically by exploiting the elasticity and scalability of cloud infrastructure. Aceto *et al.*, 2013 [46], Ward and Barker, 2014 [47], Fatema *et al.*, 2014 [48] and Syed *et al.*, [49] all emphasize this requirement as being essential. A scalable monitoring system must be able to collect, transfer and analyze monitoring data without impairing the normal operations of the Cloud. In addition to coping with the increase in the number of resources, Cloud monitoring systems must cope with the dynamic and changing nature of the monitored resources. Aceto *et al.*, 2013 [46] emphasize elasticity which, while being similar to the scalability requirements, is nuanced. While scalability is about adaptation to an increase in quantity, elasticity represents an adaptation to an increase of different resources (high cardinality).

Adaptability, extensibility and comprehensiveness

Cloud monitoring systems must cope and adapt with the rapid growth of cloud computing, being able to fit into new architectures composed of new types of services. Both Aceto *et al.*, 2013 [46] and Fatema *et al.*, 2014 [48] point out how important those requirements are for cloud monitoring systems. In order to do so, they must be able to adapt their way of processing monitoring data to specific different use-cases. They also must be able to integrate additional modules, also known as "plugins", to monitor in new ways or new types of resources. Aceto *et al.*, 2013 [46] point out how comprehensiveness and extensibility are related. They both depend on the capacity of the cloud monitoring system to support different types of resources using built-in knowledge or with plugins.

Portability and Interoperability

Public cloud providers innovate in a very competitive market. This results in a very heterogeneous cloud environment composed of different APIs, platforms, services. Monitoring multiple actors and their thousand of different resources is a challenge that monitoring systems must address. Cloud-monitoring system must be portable from one Cloud to another but also be able to operate on multi-cloud at once ([48]).

Cloud awareness

In addition to a large variety of resources and performance tiers, cloud computing also add a large variety of costs as stated by ward an all. Those aspects need to be taken into account by the cloud monitoring system, identifying cost and performance issues linked to cloud application requirements such as QoS or latency ([47]).

Timeliness

As pointed out by Aceto *et al.*, 2013 [46], "a monitoring system is timely if detected events are available on time for their intended use [57].” Indeed, depending on the amount of data the monitoring needs to process as well as the saturation level, an increase in latency between the collected data and the raised alerts may exist. Ward and Barker, 2014 [47] point out this requirement by describing the monitoring latency as "the time between a phenomenon occurring and that phenomenon being detected, arises due to a number of causes." In order to be effective, a cloud monitoring system must provide a way to reduce or avoid monitoring latency, especially in the event when the architecture is scaled up.

Accuracy

As highlighted by Aceto *et al.*, 2013 [46], accuracy is of paramount importance in order to correctly identify problems and their causes. In cloud systems, it is all the more important as inaccurate monitoring can have a significant impact on providers, causing money loss because of SLAs violations. In his review, Aceto *et al.*, 2013 [46] cite two issues linked to the accuracy of cloud monitoring systems that are discussed in the literature. The first is related to the workload used to perform the measurements, while the second is related to the virtualization techniques used in the Cloud that can impact measurement accuracy.

Autonomicity

As explained by Aceto *et al.*, 2013 [46], autonomicity is a requirement of paramount importance in cloud systems that are, by nature, highly volatile. In order to avoid service interruptions, cloud monitoring systems need to self-manage and react without manual intervention to unpredictable changes,

faults, and performance degradation. However, this requirement is not easily achieved, and the issues linked to its implementation are addressed by several studies [57] [58] [59] [60] [61] [62]. In their review, Ward and Barker, 2014 [47] cite autonomic as a requirement and insists on the need for greater degrees of autonomic behavior in cloud systems that auto-scale and rapidly change.

Non-intrusiveness

As described by Aceto *et al.*, 2013 [46], a Cloud monitoring tool is intrusive if its adoption requires significant modification to the Cloud. Non-intrusiveness thus becomes both a requirement and a challenge in cloud environments composed of a large number of resources. Fatema *et al.*, 2014 [48] states that a monitoring tool should consume as little resource capacity as possible on the monitored systems so as not to hamper the overall performance of the monitored systems [63].

Usability

According to Fatema *et al.*, 2014 [48], Cloud monitoring tools should, in order to be useable, facilitate deployment, maintenance, and human interaction.

Multi-tenancy

Public cloud offers multiple tenants the ability to share physical resources and instances. Many works in literature have discussed the necessity of guaranteeing service level agreements and virtual machine monitoring [64] [65] [66]. Fatema *et al.*, 2014 [48] explain how cloud monitoring tools must have tenant isolation in case of multi-tenancy.

Robustness, Resilience, Reliability and availability

Cloud monitoring systems must exhibit characteristics that provide them with robustness, resilience, reliability, and high availability. Indeed, cloud computing resources may expect transient failures [47]. Cloud monitoring systems must function properly even though their underlying resources might expect failures [47]. Resources can change, be moved, instantiated and deleted while the cloud monitoring system is active [48]. Moreover, they must be aware of those events and report them. For activities such as Billing and SLA management, cloud monitoring systems cannot afford failures that may compromise those activities [46].

Affordability

Stated by Fatema *et al.*, 2014 [48], a cloud monitoring system must be affordable. An open-source cloud monitoring system is an advantage compared to expensive proprietary solutions. This represents one of the main aspects behind the popularity of Cloud adaptation.

Archivability

As pointed out by Fatema *et al.*, 2014 [48], "The availability of historical data can be useful for analyzing and identifying the root cause of a problem in the long term".

2.2.3 Open research issues and challenges

Cloud monitoring research has been a prolific area both in the scientific and the industrial sector in recent years. The major surveys in our literature review have exhaustively compared all the scientific and commercial monitoring tools available and checked their compliance with the requirements they have defined. Their works give us an accurate overall picture of state-of-the-art cloud monitoring.

Aceto *et al.*, 2013 [46] underline "how some features, namely Intrusiveness, Resilience, Reliability, Availability, and Accuracy are not explicitly considered or advertised by most commercial or open-source solutions for Cloud monitoring" and "properties highly valued for Cloud services are currently not central for most of the analyzed Cloud monitoring platforms themselves". This means that most of the solutions analyzed by [46] did not fit the requirements for cloud monitoring.

They explicitly stated that "monitoring systems must be refined and adapted to different situations in environments of large scale and highly dynamic like Clouds." While many issues of cloud monitoring have received attention from the research community with important results, considerable efforts are required to achieve maturity and seamless integration in complex architecture. They emphasized a list of future research directions in cloud monitoring.

- Effectiveness
- Efficiency
- New monitoring techniques and tools
- Cross-layer monitoring
- Cross-domain monitoring: Federated Clouds, Hybrid Clouds, multi-tenancy services
- Monitoring of novel network architectures based on Cloud
- Workload generators for Cloud scenarios
- Energy and cost efficient monitoring
- Standard and common testbeds and practices

According to Fatema *et al.*, 2014 [48], "Monitoring in Clouds is an area that is yet to be fully realized". They observed "that the realization of some desirable capabilities such as scalability, robustness, and interoperability, is

still a challenge.". In addition, they also point out that "none of the tools surveyed have capabilities for verifiable metering and service KPI monitoring.". They state that "General purpose monitoring tools [Legacy/previous generation tool] are commonly designed with a client-server architecture where the client resides on the monitored object and communicates information to the server."

"These tools were designed for monitoring fixed-resource environments where there was no dynamic scaling of resources.".Fatema *et al.*, 2014 [48] argue that "in designing future monitoring tools especially for Clouds, these challenges must be addressed."

Ward and Barker, 2014 [47] point out that older tools designed for and from previous paradigms are a poor fit for cloud monitoring. They add that there is a trend toward newly design tools for cloud monitoring that better fit the new requirements. Ward and Barker also add that the area of monitoring tools being a solution in itself might be over. The new answer might be to envision monitoring as an engineering practice with new strategies, knowledge, and jobs descriptions.

In their survey, the most recent in the literature, Syed *et al.*, [49] stated that monitoring is still facing several challenges, notably, Cloud-native monitoring issues. Syed *et al.*, [49] insisted on the difference in nature between cloud monitoring solutions made from and for legacy architectures and their inadequacy when used with modern dynamic cloud-native architectures. Cloud-native monitoring issues are listed as the need for "agent-less, robustness, and operational insight features for optimal monitoring."

As seen in our state-of-the-art, existing monitoring solutions do not fit the requirements for cloud monitoring. With the advent of Cloud Native Architecture, the gap between the existing tools and strategies and the production monitoring requirements became even greater.

2.3 From Monitoring to Observability

As said previously, Cloud-Native Applications are built upon dynamically orchestrated microservice powered by multiple public cloud resources. By their inherent nature, this represents a fundamental change compared to the previous architecture. While monitoring in the Cloud Infrastructure-as-a-Service (IaaS) model, i.e., virtual machines, has been largely discussed in the literature, it was obvious by our state-of-the-art that the monitoring solutions, strategies, and architectures were not up to the task for novel cloud architectures such as Cloud-Native Applications.

Indeed Cloud Computing represented an already steep increase in the difficulty for monitoring, the advent of exponentially more complex Cloud-Native Applications relying on containerization, orchestration, Paas, Saas, and many new practices increasing scalability and dynamic life-cycles proved to be an even greater challenge.

Applications running across multiple clouds, with thousand of microservices with millions of users, are beyond the scope of monitoring systems. The potential of issues and questions from new failures is beyond traditional paradigms.

Around 2013, facing the exponential increase in complexity of systems, some Site Reliability Engineers (Twitter: @gphat @kylembrandt) started thinking about a new paradigm inspired by control theory, Observability.

Stack Overflow, the largest platform for system administrators and IT professionals, made a post about "The Importace of Observability" [67] in 2013. Twitter defined the concept of Observability in two posts between 2013 [68], 2016 [69], and a publication in IEEE Symposium on Large Data Analysis and Visualization 2013 [70]. Uber joined with a post and talk about the implementation of Observability in 2016 [71]).

In its original context, the term was defined as : "In control theory, Observability is a measure of how well internal states of a system can be inferred from knowledge of its external outputs."

In computer science, we define Observability as : "Observability is the ability to understand the inner working of systems and applications by observing the external outputs."

In our publication [5], we stated that: "While monitoring mainly reside in connecting up/down checks to data extracted from servers and websites [46] [72], Observability hopes to bring understanding to the internals of applications and infrastructures."

Thus Observability does not replace monitoring; it can be seen as a necessary extension of monitoring for operating Cloud-Native Applications. Observability represents an extension of monitoring; it aims to both fulfill the same purposes and additional ones that are required from operating Cloud-Native Applications.

We previously listed in our state-of-the-art a list of motivations and requirements for cloud monitoring that we summarise in table 2.2.

TABLE 2.2: Cloud Monitoring motivations and requirements

Motivations	Requirements
Accounting and Billing	Scalability
Capacity planning	Adaptability
Performance Management	Extensibility
SLA Management	Comprehensiveness
Troubleshooting	Portability
Security	Interoperability
	Cloud awareness
	Timeliness
	Accuracy
	Autonomicity
	Non-intrusiveness
	Usability
	Multi-tenancy
	Robustness
	Resilience
	Reliability
	Availability
	Affordability
	Archivability

Observability inherits all these properties but has new motivations as follows:

- Decisional observability

As automation is gaining momentum thanks to the DevOps movement, broader situation awareness is necessary. Decision algorithms need to access at the same time to finer granularity and broader information in order to infer the best course of action for each decision. Orchestration tools such as Kubernetes [73] require each microservice to expose information in a machine-readable format to make an automatic decision [74]. Some proprietary IT management solutions such as [75] offer automatic event mitigation; however, setting up those systems is expensive and labor-intensive as they require defining each failure case and attaching the necessary actions. Decisional Observability enables new applications such as automatic multi-dimensional auto-scaling. With new information about the microservices state, health, and load, decisions can be made even before issues arise. An example of these types of decisions is automatic preemptive auto-scaling where new microservice instances can be started before errors happen by correlating the gateway's increased load with forecasted traffic.

- Dependency and Topology discovery

Cloud-native applications can become very complex distributed systems with hundreds of microservices dynamically linked. Understanding how those microservices interact is a challenge. This is further complicated by the short-lived nature of some allocated resources. Business processes that use many different microservices inside cloud-native applications are very common and notoriously difficult to troubleshoot. Cloud-native monitoring tools must provide insights on the dependencies between business services and microservices, as well as a complete logical topology of the interactions between application microservices.

- Managed resource monitoring

Software as a Service (SaaS) and Backend as a Service (BaaS) solutions are commonly used in support of many business and application needs from authentication (e.g., Auth0[76]) to mailing (e.g., Sendgrid [77]) and from storage (e.g., Amazon Web Service S3[78]) to publish-subscribe buses (e.g. Google PubSub [79]). Dependency on these external services is vital to keep the applications up and running, and any failure of one of those services can result in Internet-wide impact. For instance, the February 28th, 2017 Amazon S3 outage [43] affected a large part of amazon cloud services, any cloud applications that had dependencies on them, and even IoT devices that relied on those cloud services [80] were impacted.

Multi-cloud, redundant, and highly available design is crucial to avoid the failure of the entire application when such events happen. Monitoring is essential in support of the implementation of recovery mechanisms [81].

- System-wide transactional visibility

Troubleshooting and performance management in distributed systems require the ability to follow in/out requests of every microservice along the path. The ability to monitor the execution flow along with topological data is paramount for any data-centric operational need, from troubleshooting to billing.

Chapter 3

Architecture Proposal for Observability in Cloud Native Applications

We carefully analyzed the state-of-the-art at the beginning of this thesis. Architectures, strategies, and tools available at the time did not fit the strict requirements that were enacted for usage in both our research and production environments. In this chapter, we describe the methodology and the components that enabled us to define our architecture proposal for Observability in Cloud-Native Applications.

3.1 Methodology

From our point of view, cloud computing and cloud-native applications started an exponential increase in complexity that caused human operational engineers to be submerged in data, issues, and challenges. Both the DevOps movement and scientists agree that humans do not scale well. Even the most skilled engineer cannot understand, visualize and take decisions on cloud-native applications as large as the ones built by Twitter, Uber, or Google.

However, there is a field of computer science, autonomic computing, that proposes a solution. Introduced by Paul Horn [82], VP of research at IBM, autonomic computing is defined as "Computing systems that can manage themselves given high-level objectives from administrators."

Cloud-Native Applications and Infrastructures being already highly automated and orchestrated in other aspects, could we not do the same with Observability for Cloud-Native Environments?

We decided to follow the principle of MAPE-K [1] [2], the structure behind autonomic managers and adapt it in order to monitor and gain observability on Cloud-Native Applications.

The result of this reasoning is the approach described in Figure 3.1. Decomposing the requirements in multiple functional entities enables to escape the tenacious hype factor in IT and construct a framework dedicated to offers the maximum Observability in cloud-native architecture.

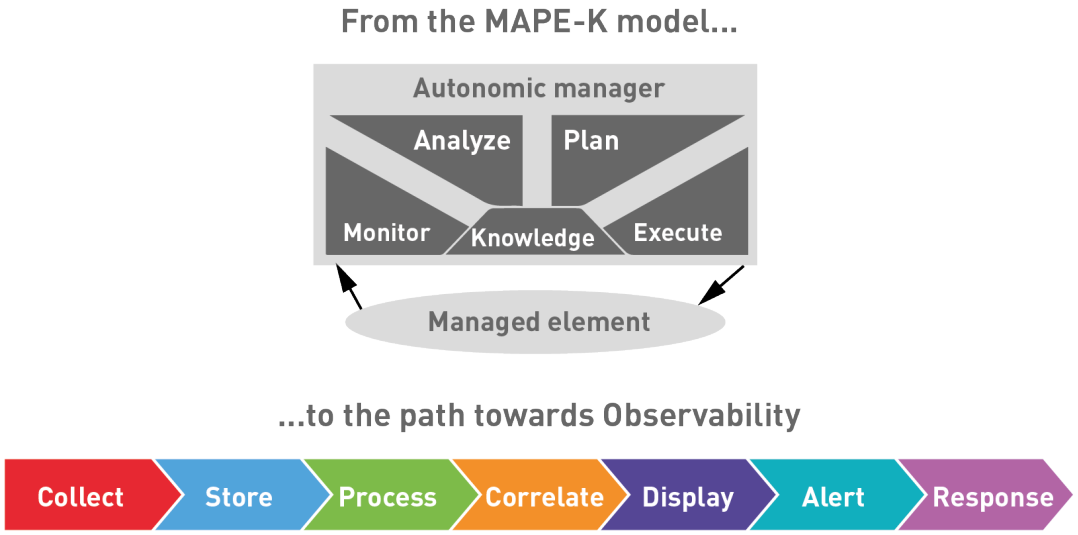


FIGURE 3.1: Path towards observability.

Our path to Observability is described as a chain of processes starting from collecting monitoring raw data through response automation with processes such as:

- Collect
- Store
- Process
- Correlate
- Display
- Alert
- Response

This path is gradually implementable and continuously improvable, enabling iterative progress towards Observability. In our approach, each specific environment implements an observability path using the tools and solutions that best fit their constraints. In enterprise and production environments, this approach enables to use commercial off-the-shelf or open-source software that fulfills requirements for some steps of the process, thus greatly facilitating implementation and adoption.

Collect

The first step of Observability is data collection. Our framework requires raw data for cloud-native applications, infrastructures, middleware, and third parties. These data represent the raw material whose quality depends on all other observability path processes.

Depending on the source, those data may be pulled or pushed. Components that are too short-lived to be scraped send their data to an intermediate gateway from which the data is collected. Most cloud-native architecture

components benefit from built-in auto-discovery features. This enables the collection of monitoring data without explicitly specifying the list of monitoring endpoints. The final role of this data also has an influence on the collection frequency and the delay since live events. For example, data that will be used for alerting purposes is very delay-sensitive, as each second increases the meantime to detection (MTTD) KPI. On the opposite, data collected for regulatory reasons just needs to be reliably collected and available whenever needed.

Monitoring data can mainly come in different forms, such as logs, metrics, and traces. Logs are journals of discrete events in a text form; they generally contain additional numerical values. The main purpose of those logs is to record past events for later diagnostic, drill-downs, and root cause analysis. Modern logs are structured, e.g., Formatted as JSON, so they can be ingested easily by centralized logging systems taking care of collecting, indexing, and storage. However, the number of logs collected only increase in size with time and service usage growth.

Lighter and more agile monitoring data can be needed, especially for alerting and graphing purposes. Metrics, mostly in the form of time series, offer light simple key-value data linking a timestamp, labels, and numerical values. They represent real-time values or counters and thus can be plotted to dynamically observe evolutions in the system. Moreover, being numerical values, they are easier to aggregate, average, summed, or modify in any mathematical way. This is especially useful when trying to get insights based on multiple metrics sources. But excluding the few labels that may be attached to them, metrics are decorrelated from the application context.

When it comes to having contextual meaning, traces are one of the most useful forms of data. They represent transactions or end-to-end user requests, tracing the cascade of events from the initial request going through every service and application until the final answer is sent to the user. It is the aggregation of all the logs and metrics created by a single request during its processing. Even if collecting all traces enables developers and operational engineers, the volume of data is directly linked to the activity, and thus storage cost and scalability issues may arise.

Independently from their types, collected data must come with extra information such as synchronized time and labels. Those metadata will prove themselves very useful when processing tasks will be performed. Additionally, a first level of filtering and sampling can be implemented during monitoring data collection, reducing both bandwidth, storage, and processing power requirements in the following steps.

Store

After collection, monitoring data must be stored in a way that makes it accessible for all the steps mentioned in our observability process. Those

databases can be centralized or distributed but must be indexed and specialized in monitoring data types. Indeed monitoring data present some particularities that may cause performances issues with general purpose relational and NoSQL databases. In addition, in the cloud-native applications ecosystem, because of scaling, those databases must be elastic and/or scalable in order to absorb great changes in ingress traffic.

The collected information must be entirely searchable with centralized databases or distributed using indexation applied on at least timestamps and at best the most important labels.

Scaling in cloud-native environments may create situations were duplication exists in labels. Those databases must also be able to de-duplicate incoming data.

Process

With convenient and fast access to the monitoring data via API, algorithms can clean or extract more information from the original monitoring data. Depending on the processing tasks, they can be implemented to be executed as soon as the data is collected. Usage of data streaming pipelines offers real-time processing capabilities. Less time-sensitive batch jobs are periodically executed or only when needed at querying time. Alerting purpose necessitates real-time data or as close as possible. This justifies processing tasks and rules applied on the fly as soon as the raw data is created and stored. On the opposite, analysis needs such as quarterly or yearly analysis can be processed later. This temporal requirement towards processing raw monitoring data motivates the placement of algorithms relatively to data sources.

By using advanced algorithms, the processing of raw monitoring data enables insights such as trending, scoring, anomalies detection or forecast.

Correlate

Along with the functional process block, the correlate block is one of the most important in the observability paradigm.

Cloud-native applications are complex multilevel distributed systems. Issues and failures can happen at any point and propagate themselves. Many techniques can be used to correlate events. Detailed and consistent labeling on monitoring data enables to correlate cloud-native applications, their replicas, and the full stack of cloud resources linked. Synchronized timing across the cloud-native infrastructure enables to detect events that exhibit the same timestamps. Anomalies detection and pattern matching can pinpoint issues that reproduce themselves across a variety of microservices. Correlation ID can transform logs into transactions to follow interaction from the initial request to the egress point.

Correlation events, logs, traces, and metrics and presenting them directly to operation engineers can greatly improve the system's reliability.

Integrating tools or writing extensions that automate this functionality is a great leap towards Observability.

Display

Displaying of the live and historical monitoring data on a dashboard enables teams to get visual feedback on systems health. This is used in conjunction with data visualization techniques to realize manual diagnosis or root cause analysis by plotting values and displaying logs lines.

Alert

More than just collecting information useful for finding the root cause of failures, it is often necessary to alert the operational team of relevant issues without them having to constantly stare at monitoring dashboards. Alerts need to be more sophisticated than just forwarding raw metrics and meaningful rules must be used to choose relevant monitoring data to be sent to the operational team. Alerts can also be made on processed data that can reveal anomalies, patterns and correlations that could not have been detected with regular numerical values.

Response

As automation is gaining momentum thanks to the DevOps movement, broader situation awareness is necessary. Decision algorithms need to access at the same time to finer granularity and broader information in order to infer the best course of action for each decision. Orchestration tools such as Kubernetes require from each microservice to expose information in a machine-readable format so an automatic decision can be made. Decisional Observability enables new applications such as automatic multi-dimensional auto-scaling. With new information about the microservices state, health, and load, decisions can be made even before issues arise. An example of these types of decisions is automatic preemptive auto-scaling, where new microservice instances can be started before CPU/RAM saturation or erratic behaviors happens by correlating the gateway's or middleware increased load with forecasted traffic.

3.2 Cloud Native Observability Data Sources and Types

Multiple data sources including logs, metrics, and traces obtained at different monitoring layers must be harnessed to achieve Observability in cloud-native microservices. We discuss in this section the different monitoring layers, monitoring abstractions, and data types.

- Monitoring abstraction layers

In the literature, the Cloud Computing environment has been stratified into seven separate layers: facility, network, hardware, operating system,

middleware, application, and finally, the user. The microservices architecture mandates that a new layer, "Microservices", be added.

- Facility: this layer forms the base of the data center. It covers the physical infrastructure necessary to host and run all the equipment. It constitutes the first data source.

- Network: A large number of communication links and switching and routing devices are used to ensure a fast and reliable connectivity between servers within the data center and with the outside world, i.e., the tenant and end-users through the Internet.

- Hardware: this refers to CPU, memory and, storage resources. In the cloud.

- Operating System (OS): hosted, for example, by physical servers within virtual machines in order to run applications.

- Middleware: provide an environment to develop and operate applications. Examples include the Docker Engine, the Java JVM, or any runtime environment abstracting the operating system for the application.

- Microservices: this layer is added to represents a set of functions executed on demand and triggered by events. As discussed earlier, the challenge is to monitor microservices through their lifetime, i.e., from the time a microservice registers a new instance to its termination. The stateless nature of some microservices further complicates the problem.

- Application: a collection of different microservices running on top of the previous layers in the cloud system. Correlating different monitoring information from different microservices to maintain a consistent view on how the application performs when it is running is very challenging.

- User: By using his web browser, the end-user can connect any front-end mobile app or a machine connected remotely, such as in an M2M (Machine to Machine) or IoT (Internet of Things) scenario.

Depending on the service layer, the deployment model, and business roles, the players in the cloud computing environment will be either producers or consumers of monitoring data. From the perspective of a cloud tenant developing microservices-based applications, many of the layers are not accessible, and the complexity due to the distributed nature of microservices makes the retrieval of monitoring data available at these layers even more difficult. Monitoring can be provided at different abstraction levels, from low-level physical hardware to high-level user applications. This nuance depends on the service model but may also be created in order to obtain more meaningful data. For example, Microsoft Cosmos DB uses a custom metric called "RU/s" [83]. Based on a formula using memory, CPU, and IOPS, this complex metric is generated by user requests. This metric is considered high level compared to those that compose it. The monitoring depth depends on the monitoring hooks and tools made available to retrieve monitoring data on the system's health and performance.

- Monitoring data types

Logs

A log is a record of immutable events. It exists either as a log list stored in a file commonly called a log file or as discrete events sent for recording to a log management system. Logs can exist in many forms, from human-readable logs to binaries. Logs whose only purpose is to record events for human operators are generally plaintext formatted. Even if such a format is useful for in-place debugging, they become tricky to manipulate when parsing the logs. Binary logs are not meant for humans and generally serve as journals in file systems, databases, or any transactional system. For such systems, these types of logs provide synchronization, recovery points, and replication. There are also structured logs which are logs that can be read, understood, and interpreted by humans and applications alike. This is the case for JSON files commonly used by developers.

Logs are very useful sources of information; they record the details of what happened inside each request. They can be easily queried using simple "grep" or "jq" in the case of JSON files. Most of the use cases like debugging, auditing, root cause analysis, billing validation, users KPIs or any exploratory troubleshooting can be done using log files.

It is worth noting that logs are storage expensive. One way to circumvent the issue is to use log management systems [84] or [85].The main purpose of such systems is to centralize and manage logs. They can aggregate and index a large number of logs but are still costly. Logs are essential for any production-grade system and constitute an important data source for monitoring and Observability.

Metrics

Metrics are sets of numbers with timestamps and labels attached to them that give information about a component or process. They can be counters or gauges that give insights by themselves and can also be recorded and graphed over time to analyze trends. Metrics are easier to export, retrieve, and store compared to logs. They can be compressed, indexed, and queried faster and with more flexibility. Aggregating, averaging, or summing them over time can resolve retention issues.

Metrics are popular in the cloud infrastructure because they are very scalable; they don't increase overhead when the system's usage increases. They are essential for Observability since they constitute a light, efficient and precise way to obtain information from cloud microservices architectures at scale. As the overhead only increases with the number of different metrics recorded and not with the number of data points, it is important to focus on a limited set of relevant metrics for most cases.

Metrics can also be manipulated mathematically and correlated with other metrics for complex analysis. They can also be used to trigger alerts in near real-time.

In cloud-native microservices-based applications, metrics are highly required for any operational team to achieve Observability.

Traces

Data traces are end-to-end representations of a series of causally-related distributed events across a service operation from the initial request up to the service delivery or failure. A single trace enables to follow the flow of the request and the changes that happened during the path across the distributed system. In complex distributed systems like microservices in public cloud environments, traces are the panacea for troubleshooting, debugging, performance analysis, and any similar use cases. Even difficult issues like asynchronous executions become visible using data traces.

- Monitoring strategy and methods

Gathering a large volume of monitoring data is only a means to achieve Observability. While access to logs, metrics, and traces is essential, these are only raw bits representing values and events. Only once they are parsed, cleaned, structured, processed, and correlated, those data become information able to satisfy the requirements of Observability. In this section, we will discuss the strategies and methods used to select, recover and correlate data for achieving Observability.

- Blackbox monitoring

Blackbox monitoring refers to a monitoring strategy that treats systems as black boxes and examines them from the outside. Before agile and DevOps practices, Operation engineers (Ops) were in charge of keeping servers up and running for hosting applications. They did not have any way to change the delivered binary. The most convenient way was to monitor the underlying operating system and server general information like CPU usage, RAM saturation, and disk I/O. This type of approach is particularly useful in scenarios where third-party apps are being deployed. Even in a microservices environment, being able to obtain some low-level information remains useful and enables operators to achieve better system integration by being more aware of other components.

- Whitebox monitoring

Whitebox is a more recent approach to tackle monitoring by retrieving information from inside the applications and systems. Instead of measuring host, network, or operating system levels, the focus is on the application and how it is running and performing. This type of monitoring needs to be implemented in a DevOps fashion. Developers and operation engineers (Ops) work together to implement whitebox monitoring. While developers instrument data exporters into their applications, Ops configure monitoring tools to gather those data. The main difference between blackbox monitoring and whitebox monitoring is the difference between symptom detection and cause determination. Blackbox monitoring reveals symptoms while whitebox eases identification of underlying causes.

- Computational based tests

Computational-based tests use CPU time and power of the monitored resources to determine the liveness as well as any information that can be deduced by executing either the running service or specific algorithms. This type of active testing enables to determine performance, stability, or base value to evaluate the max capacity of the service in its running environment. In environments without a continuous flow of requests, computational tests can help detect errors or outages that can be avoided before impacting real clients.

- Network based tests

Network tests refer to all tests that can be done at the networking layer. These types of tests can be either passive (e.g., counting packets, measuring throughput and bandwidth) or active by injecting packets that can measure round-trip times or jitter along the network interfaces between services or endpoints.

According to Google SRE books [86], at least four golden signals have to be monitored. (1) Latency, which represents the elapsed time on how long each request needs to be processed and if the request was successful or not. (2) Traffic monitoring measuring the throughput in terms of request per second, number of concurrent sessions, basic network I/O, and the load of a component. The monitoring data can be retrieved, shipped, and stored as time series metrics where a set of labels and values are attached for each timestamp. (3) Errors representing the rate of failures that can be used to diagnose a specific component of the system. It also allows for discriminating bad requests sent by users from hidden bugs, regressions, or attacks. (4) Saturation, which is usually the performance degradation of a component before reaching 100 % utilization. This consists in observing the performance of a system or component before the tipping point, where higher load means lower performance and increased response time (latency).

- RED (Rate, Errors and Duration)

One of the monitoring processes derived from the four golden metrics is the RED method for Rate, Errors, and Duration. By obtaining the number of requests per second served by the services, including the number of failed requests and how many times each request lasted, most of the obvious issues can be pinpointed. However, this method is clearly linked to the availability of high-level white-box metrics.

- USE (Utilization, Saturation, Errors)

Another method from the literature is USE for Utilization, Saturation, Errors. It extols the value of checking the aforementioned control values for every resource in the system. It can be used in contexts where inside metrics are not available, as the resource lemma indicates, but not limited to CPUs, memory, network interfaces, storage devices, etc. In this case, the USE metrics need to be interpreted with high utilization level that can be correlated to

issues such as increased latency or even signs of bottlenecks when achieving 100%. Saturation ideally needs to be kept at 0, and any increase represents extra time spent for jobs. Any errors reported are worth investigating because even if they are not caused by the system, they still represent misuse by clients.

- Correlation

As mentioned previously, correlation is a great way to navigate monitoring data. However, this needs to be plan ahead, while many applications and systems already assign a Universally Unique Identifier (UUID) to requests, some need to be developed with these criteria in mind. By filtering logs, metrics, and traces with this unique correlation ID, engineers can determine all the services, machines, and tier services that this request has been through. Any gaps also help to identifies "area of darkness" in the system, components that are insufficiently connected to the observability framework.

3.3 Observability Architecture Framework

Being equipped with the knowledge of our requirements, our methodology, and our data sources, we were able to design a framework that would enable us to achieve Observability in cloud-native applications. Our methodology enables us to define clear roles for our observability framework. Instead of relying on one single monitoring, our observability approach relies on the same principles of the cloud-native paradigm, it is a distributed system of microservice together contributing to achieving Observability. That approach enables scalability, robustness and resiliency as each part of the system is a separate failure domain.

Our framework begins with collecting data in all forms possible, logs, traces, and metrics, from our cloud-native environment. Extracting data from our Cloud Native Infrastructure components, Cloud Native Applications, Cloud providers APIs, and other dependencies such as Software-as-a-Service Third-Party Service Provider APIs. Those data can be recovered via APIs, data exporter (agents) when possible, instrumentation of apps, or even captured via running service mesh. Those data then enter our observability framework, which stores it in cloud storage. From there, the collected data can be processed, filtered, requested, correlated in any way or fashion. Those data then become knowledge that can be used to display KPI (Service Level Objectives SLO), triggered alerts, and even automated actions.

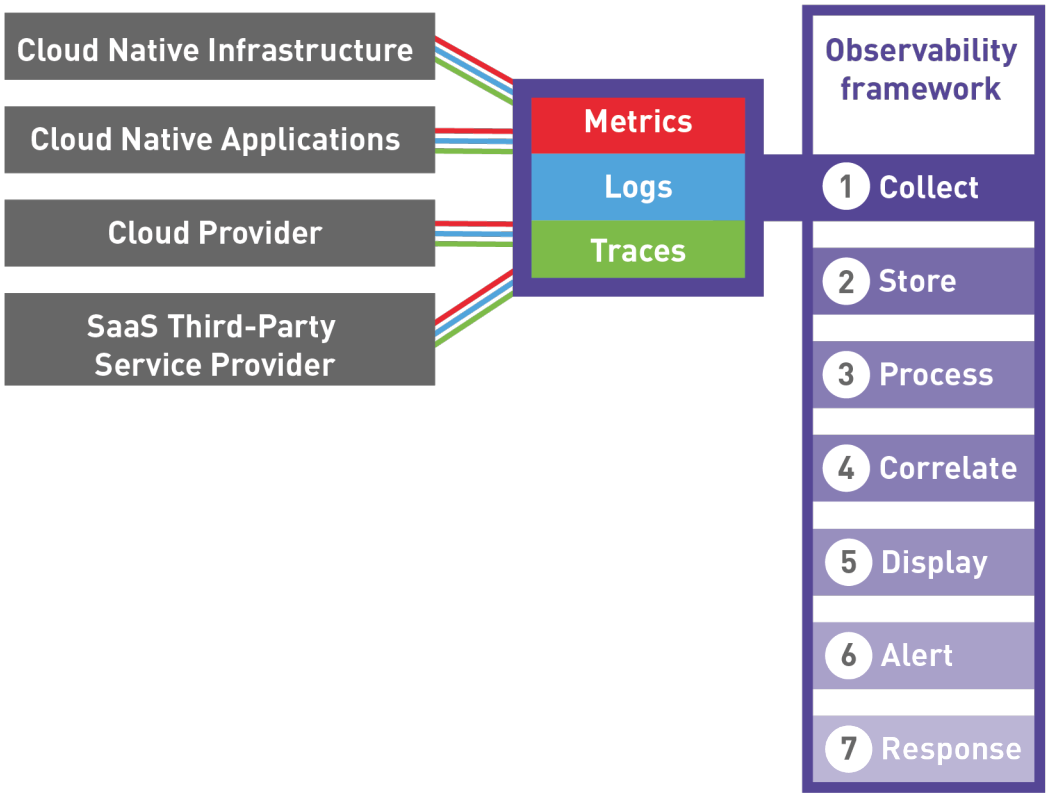


FIGURE 3.2: Architecture framework for observability in cloud-native architecture

Our architecture framework is built following the cloud-native paradigm; each component must run in a cloud-native environment and thus benefit from it. Thus in our framework, all the components can be instantiated, configured, and maintained alongside Cloud-native applications with the same tools and best practices. The components of our framework can be instantiated on the same cloud infrastructure or on dedicated instances for additional resiliency and high availability purposes.

3.4 Implementation of our Architecture for Observability in Cloud Native Applications

We implemented a proof of concept (PoC) observability framework in an industrial production environment within LECTRA [87] EPA:LSS [88], which is an industry-leading technology company specialized in software and soft fabric cutting equipment for fashion, technical textiles, furniture and automotive. Being an important player in the 4.0 industry, this production environment offers many SaaS cloud services.

As a constraint, all the created microservices rely on the Microsoft Azure cloud infrastructure. While we could have opted for a fully open-source stack for this implementation, the existing contract dictated the choice of some commercial components. During the previous years, new services were

designed and developed progressively. Many new technologies and tools became successively available both on the Azure platform and in the open-source ecosystem.

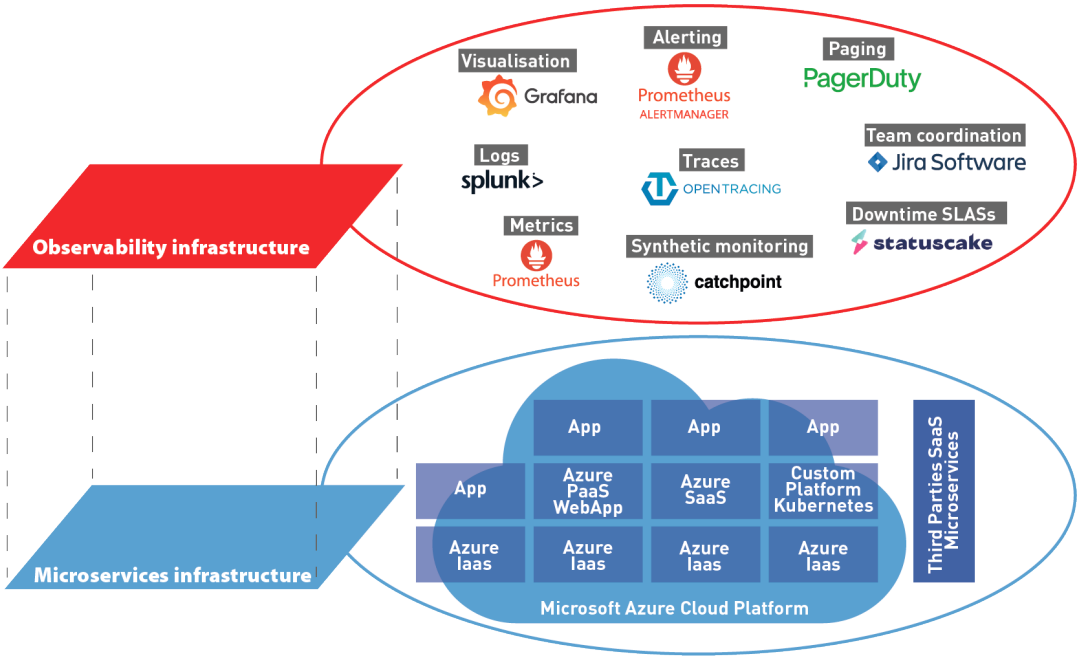


FIGURE 3.3: Illustration of different generations of microservices deployed in our environment, along with the observability framework set up in our PoC.

Some of our earliest microservices run on an IaaS service model (VMs) and we progressively added managed PaaS (WebApps), then our own PaaS infrastructure of containers orchestrated on top of IaaS resources. All our generations of microservices are developed and deployed in continuous integration or continuous deployment.

When considering third-party services, there are many interdependencies. Thus, we needed to develop new tools and methods to achieve a high level of Observability and guarantee production-grade SLAs.

Figure 3.3 shows both microservice infrastructure and Observability framework. The observability framework provides a number of functionalities and build on highly customizable tools such as:

- Logs recovery, storage and management [85]
- Metrics recovery, storage and management [89]
- Metrics visualization and correlation [90]
- Alerting [91]
- Paging [92]
- Team Coordination [93]

- Synthetic monitoring [94] [95]
- Service level agreement evaluation [95]

As displayed in Figure 3.3, many off-the-self tools and services compose our observability layer. Each one of them is in charge of either collecting, processing, or displaying the information that enables Observability on our microservices-based applications.

Directly connected to our microservices are information-gathering components. Splunk and Prometheus are used to collect and manage respectively logs and metrics. Both Slunk and Prometheus come with exporters and APIs that allow for the collection of data from infrastructure components and applications. Meanwhile, Status Cake and Catchpoint realize content liveness checking and synthetic monitoring. Those four previously cited tools, Slunk, Prometheus, Status Cake, and Catchpoint, enable the collection of both "inside" whitebox and outside blackbox information.

FIGURE 3.4: Details of a Dashboard of our production Observability stack at Lectra

Other tools in our observability layer are given an information processing role. Alertmanager uses the metrics centralized by Prometheus to launch automatic actions but also enables custom triggering, such as alerts forwarded to PagerDuty. Grafana enables operational engineers to visualize the data in a human-readable form. After some event, our operational engineers write post-mortem that becomes part of our knowledge base and greatly helps whenever the issue happens again.

Even if the tools included as part of the observability framework are off-the-shelf software, they have been customized to fit our specific needs. Furthermore, we implemented our service logic, business rules, and domain knowledge directly inside each component. As an example, the liveness of a business service is implemented by considering all the internal and external dependencies. In addition, we created rules that proactively detect failures based on patterns related to historical outages. During the alpha stages of

our development, our failure detection system was mainly based on the complaints of developers and alpha users. Soon we implemented a log management system to display our custom logs and errors on dashboards. However, this approach only allows the information to be retrieved several minutes after the occurrence of events. In order to overcome this limitation, a lighter and more reactive metric-based system was implemented. This system is based on Prometheus and required adding metric exporters inside microservices. This system has proven to be much faster than log-based alerting.

Our log and metric collection system is hosted on the same cloud infrastructure that hosts our applications. Furthermore, we needed third parties solutions to get accurate and independent measurements. This is where third parties synthetic monitoring plays a role (StatusCake, Catchpoint). With these additions, we were able to identify any malfunctioning microservices and be alerted in less than 30 seconds.

3.4.1 Results

To illustrate the results of our observability approach, we provide some results in Figure 3.5 of the total downtime and in Figure 3.6 of the mean downtime. The values shown in the figures are aggregated from our production environment and are represented in months since the microservice was connected to our observability suite. These two key performance indicators show how fast issues and outages are detected and fixed. Using our observability framework to gather information on the microservice-based applications and using the information and orchestration rules, we were able to greatly improve availability. It is worth noting that before deploying our framework, humans were in charge of watching basic dashboards displaying limited hardware statistics and investigate whenever they recognize patterns corresponding to an issue.

By using those advanced dashboards and alerts, operational engineers can pinpoint issues faster and even detect trends that may lead to future failures. Figure 4.4 shows an example of one of those advanced dashboards dedicated to getting a precise idea of how a microservice is behaving. The top four numbers indicate, from left to right, the number of automatic restarts, scale-up events, queued messages, or dead letter messages that happened during the period. Below the first row, we find graphs displaying inputs, outputs (HTTP and AMQP), and interaction with third-party SaaS services (Search and Database Service). This dashboard greatly helps operational engineers and developers to pinpoint issues whenever alerts are raised.

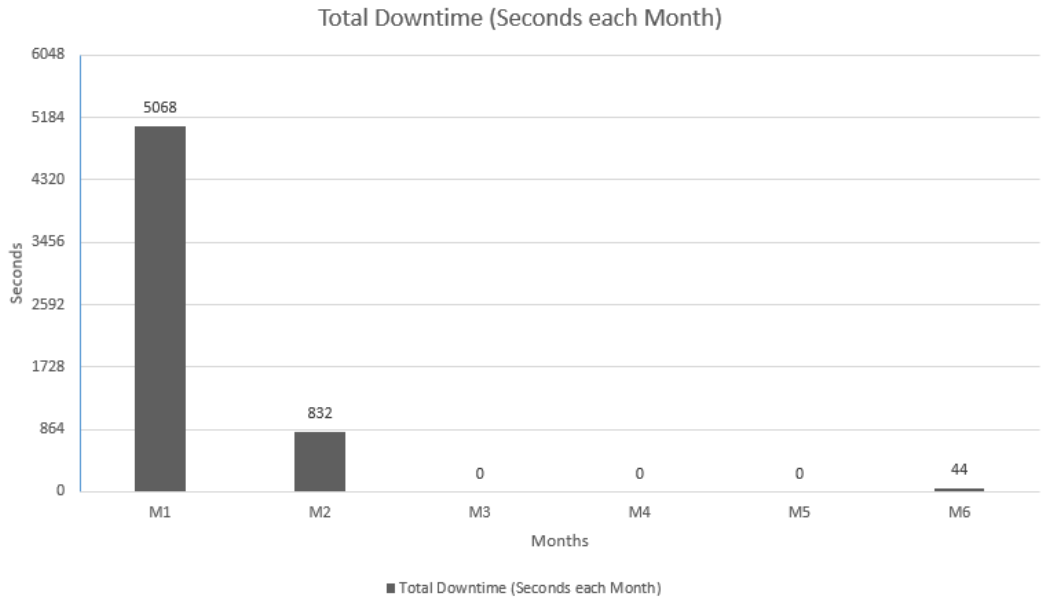


FIGURE 3.5: Total downtime (Seconds per Month) for equipped microservices

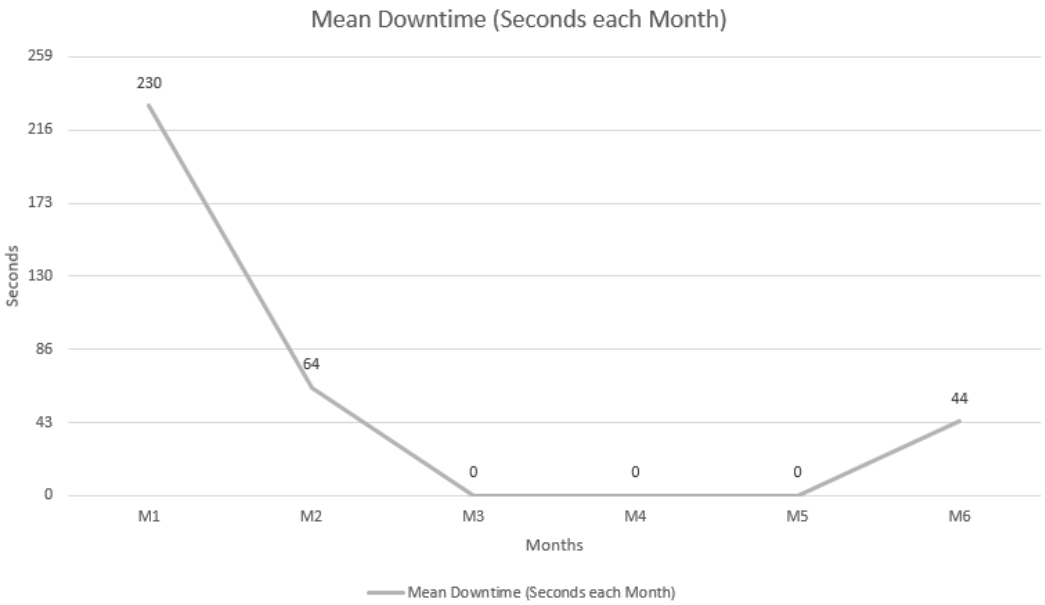


FIGURE 3.6: Mean downtime (Seconds per Month) for equipped microservices

3.5 Conclusion

Achieving the design of an Observability Framework and its implementation within Lectra systems is a major accomplishment and a a milestone in this thesis. While monitoring is a well-established subject and keyword in research, Observability has not yet gained traction with less than ten publications on the subject on IEEE Explore at the time of writing.

One of those publications being "Demonstration of an Observability Framework for Cloud-Native Microservices", an interactive live demonstration of the implementation of our Observability Framework within Lectra production environment. Participants were able to discover the difference between the traditional monitoring approach and Observability by asking custom requests that have not been pre-programmed and get immediate insights.

Developers and Engineers at Lectra also quickly adopted the new tools and this implementation was quickly extended and is now the foundation of global production operation at Lectra. Many changes that were needed to interconnect the different components of our framework have resulted as a contribution (merge accepted) to open source tools.

In this thesis, Observability is a first step toward a better quality of Service in Cloud Native Environments. It represents the "sensory" part of the autonomous system paradigm. Observability represents the base on which we developed our research work during this thesis.

FIGURE 3.7: Dashboard of our production Observability stack at Lectra

Chapter 4

Observability driven Auto-scaling for Cloud Native Applications

4.1 Introduction

Cloud-native technologies empower organizations to build and run scalable applications in modern, dynamic environments. Containers, orchestrators, service meshes, microservices, immutable infrastructure, and declarative APIs enable loosely coupled systems that are resilient, manageable, and observable. In addition, devOps and Agile practices essential to the cloud-native paradigm enable software developers and operations engineers to collaborate to deliver higher and faster software quality.

Nowadays, cloud-native has become the de-facto standard in the industry. According to the Cloud-native Computing Foundation (CNCF) in their 2018 survey, "Production usage of CNCF projects has grown more than 200% on average since December 2017".

The microservices architectural approach consists in developing the application as a collection of small services. Each one is independent and implementing atomic functionalities. As a result, businesses services requests are answered by multiple microservices interacting with each other. As they are independent components, microservices can be deployed, upgraded, scaled up and down, and restarted independently, thus enabling fast and frequent releases on live applications with little or no impact on end-users.

Containerized microservices offer more efficiency and speed than virtual machine ones. Containers can be instantiated with the speed and ease of any operating system process. Multiple containers can run on the same operating system and benefit from isolation from each other.

This ability to mutualize resources and reduce overhead makes containers the ideal form factor for microservices. Containers also offer flexibility with management API that enables complete automation of their lifecycle. Features such as orchestrators performing automatic rolling updates, basic autoscaling, and many more can use this API.

However, all the advantages that can be obtained by migrating to the cloud-native paradigm come with new challenges. Migration to the cloud-native paradigm is a challenge by itself. New projects starting from scratch can be developed following the cloud-native approach, but existing (legacy) applications cannot become cloud-native applications just by implementing a "lift and ship" project. Complete refactoring is often necessary. Developers and infrastructure engineers must learn, adapt, and code using new languages, design patterns, and technologies. Once developed, cloud-native applications reveal new challenges, as their new design cannot be operated in the same fashion. Cloud-native applications are, in fact, large and complex distributed systems.

Since 2015, researchers, engineers, and technicians have worked on many of those issues. Orchestration, observability-oriented monitoring, and the design of large dynamic clusters hosting cloud-native applications are well-known problems. However, complete and satisfactory automation of cloud-native applications life cycle has not yet been achieved with satisfaction.

In the cloud-native paradigm, instances of microservices are immutable. Failures may result from bugs, corruptions, issues coming from the underlying infrastructure, or third-party dependencies. Most of those microservices issues can be solved by re-instantiating the incriminated components (kill and restart). These practices called "Cattle over Pets" put an emphasis on automated provisioning and deployment and greatly improve mean-time-to-repair by using re-instantiation as failure mitigation.

Additionally, among their new challenges, cloud-native applications also face high volatility. As cloud resources are available as a commodity, cloud-native applications are designed to be scaled up and down. This scaling, also called auto-scaling when automatized, is a key element of production-ready cloud-native applications.

Two important surveys in the literature [96] [97] have reviewed the work on auto-scaling, and they provided a detailed taxonomy related to web applications in the cloud, mainly for auto-scaling based on VMs. However, the emergence of containers and cloud-native applications has brought new characteristics that make the existing works in the literature not suitable, and solutions are still left to be explored [97].

In this chapter, we present our cloud-native framework enabling proactive autoscaling of cloud-native applications to achieve a better quality of service. Our framework uses proactive autoscaling algorithms based on Long Short-Term Memory (LSTM) to try to improve the end-to-end latency for cloud-native applications. We developed a proof of concept to demonstrate this framework. We also discuss the implementation of Observability-driven autoscaling inside Lectra's production environment.

4.2 Related Work on Auto-scaling

Cloud computing has revolutionized how workloads are designed and hosted. Cloud providers offer their customers the capacity to unilaterally provision compute, network, and storage capacity automatically. Moreover, those resources can be provisioned and released at will, enabling applications to be scaled up and down on demand. The almost unlimited and infinite provisioning capacity available at any time could make it possible to achieve exact sizing at any time for any application. This aspiration to provision resources with little to no over-provisioning and/or under-provisioning is motivating research in the area of dynamic cloud provisioning.

In one of the earliest papers on the subject of dynamic provisioning, Urgaonkar et al. [98] explored multi-tier applications and argued that "dynamic provisioning of multi-tier Internet applications raises new challenges not addressed by prior work on provisioning single-tier applications." While cloud computing was not available at the time, they had a glimpse of the challenges that were coming and identified the opportunities offered by proactive provisioning.

Later in the first workshop on Automated control for data centers and clouds (ACDC '09), [99] presented issues that make feedback control in a cloud computing infrastructure different from other computer systems. In [100], Ming Mao and Marty Humphrey presented an architecture enabling auto-scaling in a public cloud environment, which rests on the "MAPE" principle: Monitor, Analyze, Plan, and Execute. Highlighting the importance of monitoring in the auto-scaling behavior, they used advanced monitoring metrics that give better indications of a cloud application's quality of service.

Recent works in literature and surveys [96] [97] give an overview of the auto-scaling techniques and architectures that have been investigated. However, none of these works have tackled proactive auto-scaling in cloud-native applications and the new challenges that come with them. Containers and micro-services-based applications are more volatile, more difficult to monitor, and present resource allocation issues mainly in terms of isolation. Compared to VM-based micro-service applications, cloud-native applications are based on containers sharing a common underlying infrastructure, dynamically managed and built on service meshes, services discovery, messaging, etc., making this resulting distributed system complex to observe and orchestrate.

We tackled the problem of scaling of volatile cloud-native applications (container-based applications), and we propose a framework for auto-scaling in cloud-native applications. Our framework aims to achieve a specific QoS goal by forecasting system states metrics (e.g., forecasting metrics such as a request or load metric) using learning-based forecast models with LSTM. Forecasted metrics are used to dynamically adjust the resource pool horizontally (number of replicas) and vertically (resources pool). Furthermore, we demonstrate this framework in a real cloud-native environment.

4.3 Proposed Architecture

In this section, we provide a set of requirements, algorithms, and techniques that, once combined, aim to provide proactive auto-scaling on cloud-native applications. Our proactive microservice works using a learning-based model to dynamically forecast the system states and provide insights enabling proactive auto-scaling.

4.3.1 Architecture for Observability driven Auto-Scaling

Our architecture is fully cloud-native powered. It is built on top of cloud IaaS (infrastructure-as-a-service) resources. This cloud can be a public, private, or hybrid cloud as long as it fulfills the requirements to qualify as a cloud provider. This means, among other requirements, that it has the ability to provide on-demand a virtually unlimited amount of resources that can be used for autoscaling. It must provide complete APIs for provisioning, management, and monitoring. This implies that the resources pool, made of virtual machines, can be created, resized in size and number both on-demand and programmatically via API.

An orchestrator manages the resources pool and, therefore, can use the cloud API to scale up and down the resources pool. Among those resources, some are reserved for the orchestrator needs, providing API for interactions, scheduling, control-loops, and storage of configurations and cloud-native applications manifests. Those functions run on selected virtual machines hosts from the resources pool. Those are called "masters" which only run control plane workloads and are separated from the "workers."

The “masters” and “workers” hosts communicate seamlessly over an overlay network (see Figure 4.1). They are also equipped with an essential container runtime, as all the workload, from control and data plane, are containerized. The "worker" hosts provide resources where the orchestrator deploys cloud-native applications according to the user-provided manifests.

Cloud-native applications are containerized orchestrated microservices with automated lifecycles, which are developed according to best practices [101]. They are generally stateless, immutable and they can be natively load-balanced. This robustness allows the orchestrator to treat them as "cattle" instead of "snowflakes", which means that they can be created, updated, destroyed transparently, and as many times as required in production environments [102].

4.3.2 Using resource monitoring and observability

One of the major challenges related to the auto-scaling of cloud-native applications is the lack of precise and relevant monitoring information that can provide the actual system state. Indeed, in a complex containerized distributed system such as cloud-native applications, it is difficult to determine the health state of the entire system. In this context of increasingly complex

architectures, it is necessary to get more visibility on infrastructures and applications.

Both the industrial and scientific worlds are tackling those new issues. One of the approaches that are gaining momentum is called Observability. [103],[104],[105],[5] . Observability does not replace traditional monitoring but extends it. While monitoring mainly aims to collect and display raw data, observability aims to offer an intuitive presentation of applications' internal states and behaviors through the collection, processing, correlation, and display of various information.

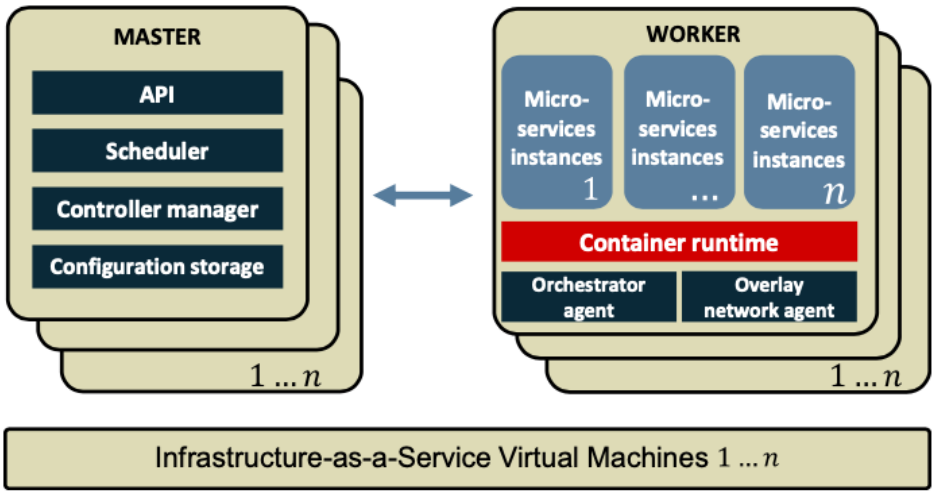


FIGURE 4.1: Cloud-native Orchestrated Platform

In order to operate, understand, and optimize cloud-native applications and infrastructures, we adopted an observability approach based on the "three pillars of observability": logs, metrics, and traces. They represent the raw data needed to get an inside view of the applications and microservices' health states and behaviors. Logs, metrics, and traces are recorded for all microservices instances, whether they are support or business microservices. Logs are centralized and recorded with timestamp and tags enabling thereafter correlation with other data.

Metrics also benefit from the same process being recorded and exported as time-series in a time-series database (TSDB). Traces are collected, enabling correlation between all the microservices. Those correlations can pinpoint bottlenecks in clients' and internal requests. All the components necessary to collect, store, process, correlate, display, alert, and respond to those data are cloud-native. They are orchestrated within the same resources pool alongside the data plane microservices. Once processed, the collected observability data is made available through API enabling automation.

4.3.3 Auto-scaling framework

Auto-scaling consists of dynamically adjusting the resources allocated to elastic applications according to goals. Those resources can be any component of the cloud-native architecture, virtual machines, or containers. The resources pool can be auto-scaled by adding more virtual machines, thus increasing the number of "worker" hosts where the orchestrator can instantiate more containers.

Cloud-native applications are horizontally auto-scaled by instantiating more replicas of their containerized components. Thus, auto-scaling in the cloud-native paradigm is closely linked to a resource allocation problem [97]. Auto-scaling via dynamic resizing of the existing containers is called vertical auto-scaling.

Our work focuses on the horizontal auto-scaling of cloud-native applications as it follows the best practices for cloud-native design. Auto-scaling is a continuous self-managed process where a control-loop monitor, analyze, plan and execute as shown in Figure 4.2.

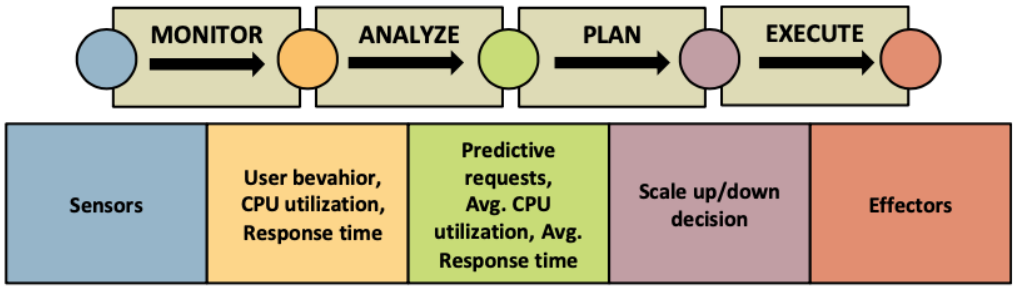


FIGURE 4.2: MAPE Process in our auto-scaling framework

This control loop is run by the orchestrator. At regular set intervals, it uses observability data such as latency or incoming requests to sense the application's state, analyzes the queried values according to an algorithm, checks rules, and then sends a decision to scale to the desired replica count.

This process has some weaknesses (See Algorithm 1). The first weakness is that CPU usage is a metric that is very distant from the QoS experienced by the user. Another one is that QoS degradations are experienced during transient periods. Indeed, the reactive time, composed of the decision and scaling time, represents a period during which the system is incorrectly sized.

In order to solve the issue related to a lack of causality between the auto-scaling sensory input and the QoS experienced by the user, we choose to use observability data. This approach enables us to find accurate information that can be linked to the auto-scaler. For example, we can use the number of requests received by said microservice instead of using CPU usage for a containerized microservice that shares a worker host CPU. This approach is called reactive auto-scaling on custom metrics (as described by Algorithm 2).

Algorithm 1: Basic CPU Auto-scaling Algorithm

Input: The set of applications deployed in our system, A_n
Output: The number of pods of each application
foreach *each application* A_t *deployed in our system* **do**
    $U_{cpu} \leftarrow$ the average CPU utilization ratio of A_t
    $N_{pod} \leftarrow$ the number of pods of A_t
    if $U_{cpu} \geq 0.5$ **then**
        $N_{pod} = N_{pod} + 1$
    else
        if $Ucpu \leq 0.4$ **then**
            if $N_{pod} \geq 1$ **then**
                $N_{pod} = N_{pod} - 1$
    PODSCALER (A_t, N_{pod})

Algorithm 2: Custom Metric Auto-scaling Algorithm

Input: The set of applications deployed in our system, A_n
Output: The number of pods of each application
foreach *application* A_t *deployed in our system* **do**
    $M_c \leftarrow$ the average utilization ratio for metric value of A_t
    $M_d \leftarrow$ the avg. utilization ratio for desired metric of A_t
    $R_c \leftarrow$ the current replica count of pods of A_t
    $N_{pod} \leftarrow$ the number of pods of A_t
    $R_d \leftarrow \frac{R_c * M_c}{M_d}$
    if $R_d \neq R_c$ **then**
        if $N_{pod_{MIN}} \leq R_d \leq N_{pod_{MAX}}$ **then**
            $N_{pod} = R_d$
    PODSCALER (A_t, N_{pod})

The other issue of QoS degradation during transient periods is tackled by proactive autoscaling, which can be achieved by forecasting the system states and autoscaling according to this forecast (see Algorithm 3). This forecast can be obtained by many different methods. In our architecture, we choose to focus on Long Short-Term Memory Recurrent Neural Network. Other methods exist, such as Box-Jenkins methods (ARIMA, SARIMAX, ...), which offer capabilities to forecast time series but are based on linear regression to capture temporal structures. As demonstrated by Siami-Namini *et al.*, [106], they are largely outperformed by Machine Learning based LSTM networks in forecasting time series. This is especially true for highly volatile time series such as the ones we encounter in cloud-native environments.

Long Short-Term Memory (LSTM) [107] is a kind of Recurrent Neural Network (RNN) with the capability of remembering the values from earlier stages for future use. Unlike regression-based algorithms, LSTM RNN offers a strong capacity in handling small discrete patterns in time series. Deep learning can use the large time-series data produced by cloud-native platforms to solve some of their remaining challenges.

Our architecture benefits from the democratization of deep-learning solutions. A deep learning forecast microservice can be included in our platform control place. This predictor microservice linked to our observability component can publish forecasts for some selected values on which the orchestrator will act. This forecast can greatly help solve transient QoS issues experienced while scaling cloud-native applications under burst loads by providing early information on events requiring rapid auto-scaling.

Algorithm 3: Proactive Metric Auto-scaling Algorithm

Input: The set of applications deployed in our system, A_n
Output: The number of pods of each application
foreach *application* A_t *deployed in our system* **do**
    $PM_c \leftarrow$ the predicted average utilization ratio for metric value of $A_t + \Delta$ with $\Delta = forecast\ time$
    $M_c \leftarrow$ the average utilization ratio for metric value of A_t
    $M_d \leftarrow$ the avg. utilization ratio for desired metric of A_t
    $R_c \leftarrow$ the current replica count of pods of A_t
    $N_{pod} \leftarrow$ the number of pods of A_t
    $R_d \leftarrow \frac{R_c * PM_c}{M_d}$
    if $R_d < R_c$ **then**
        if $N_{pod_{MIN}} \leq R_d \leq N_{pod_{MAX}}$ **then**
            $N_{pod} = R_d$
    PODSCALER (A_t, N_{pod})

4.4 Implementation and Evaluation

4.4.1 Enterprise Production Implementation

One of the goals of this thesis was to produce usable results of our research and implement them in Lectra's production environment. While it was not possible to implement the proactive approach with production constraints, we were able to fully implement the Observability-driven reactive autoscaling.

We provide an illustrated explanation of how our observability framework is leveraged for autoscaling a microservice as presented in Figure 4.3. This custom auto-scaling is based not on microservices resources usage (CPU) but also on observing the application message-bus queuing system. This set-up can also be applied for microservices that cannot exploit system metrics to determine functional load status.

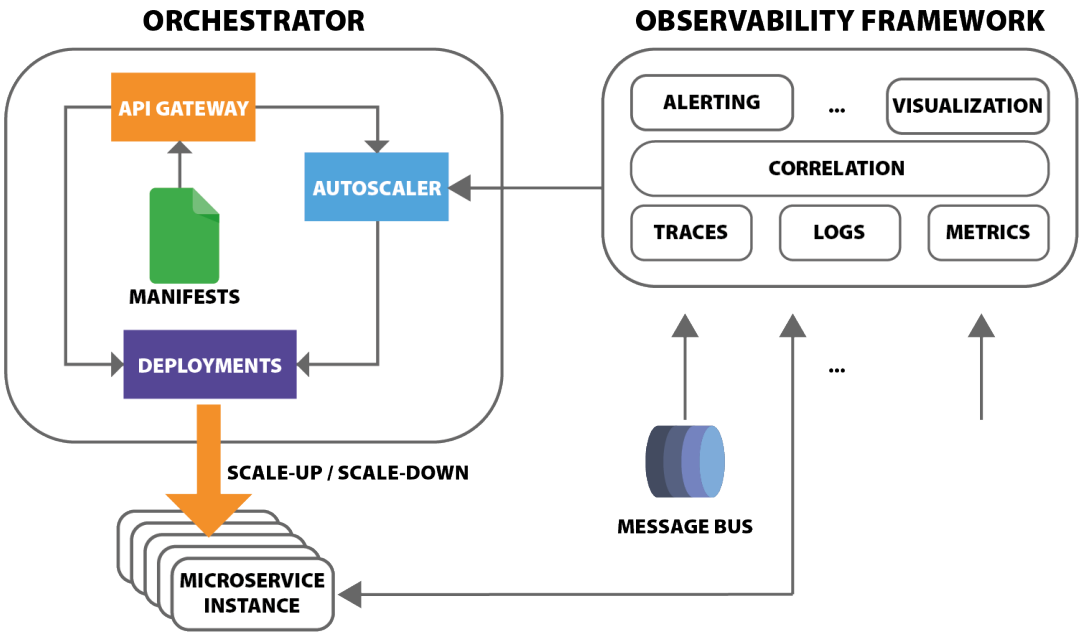


FIGURE 4.3: Elements of the observability driven auto-scaling orchestrator.

Figure 4.3 illustrates some interaction between core elements that enable observability driven auto-scaling. Microservices are deployed by the orchestrator using a manifest file containing all the necessary parameters to deploy and run an instance. This manifest also specifies the autoscaler behavior and which data is used in the decision algorithm. The instances then continuously communicate with the observability framework for reporting the necessary information. Our approach focuses on custom parameters that are not directly linked to instances. Those parameters are used to drive the orchestrator autoscaler (scale-up / scale-down). This is especially useful in hybrid infrastructures where all microservices and components are not hosted in the same orchestrated environment.

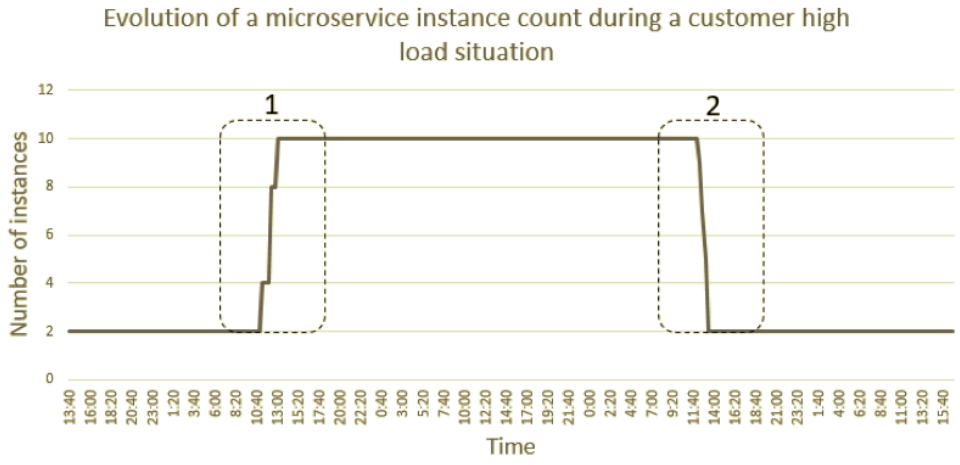


FIGURE 4.4: Microservices auto-scaling based on message-bus metrics (Screenshot from our grafana dashboard)

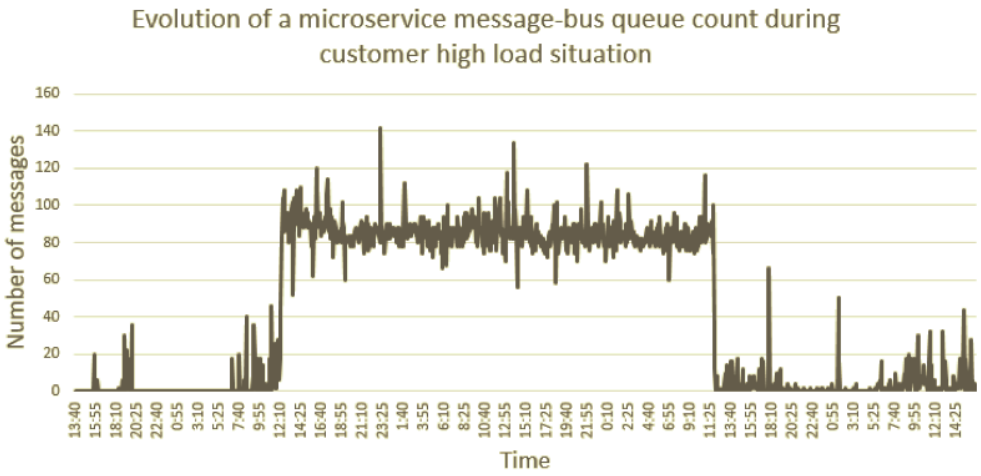


FIGURE 4.5: Increase in incoming message during the scaling event in Figure 4.4

In Figure 4.4, the auto-scaling rule for microservices is linked to the amount of work sent over the message-bus queue as reported in Figure 4.5. There are two distinctive events of (1) scale-up and (2) scale-down. The baseline for any microservice is set to two instances, which is our base requirement for any highly available production environment. At (1) of Figure 4.4, the message-bus queue of the microservice fills up; the orchestrator then increases the number of instances gradually to 10, which is the maximum number of instances allowed for this microservice. Then at (2), when all workloads are processed, the orchestrator reduces back the number of instances to 2.

4.4.2 Proof-of-Concept Implementation

Production constraints and changes in strategies have pushed us to develop and implement our Observability-driven proactive autoscaling Proof on Concept (PoC) outside Lectra's production environment. We have chosen to implement our Proof on Concept (PoC) on public cloud virtual machines hosted by Digital Ocean cloud provider. Nowadays, most of cloud providers offer solutions for managed Kubernetes master nodes (Amazon EKS, Microsoft AKS, and Google GKE). This enabled us to benefit from a real and up-to-date cloud-native platform for our experiments. Our platform is composed of a pool of [3 to 20] virtual machines, with the following characteristics: 4 vCPU, 8GB of RAM. Those machines are equipped with either Intel Xeon Skylake (2.7 GHz, 3.7 GHz turbo) or Intel Xeon Broadwell (2.6 GHz).

Additionally, Digital Ocean enabled us to use cluster autoscaling. In opposition to other experiments conducted within laboratory on-premises server resources, our PoC can horizontally scale its platform by adding more worker nodes programmatically. Our worker nodes accommodate all the cloud-native applications required for our experiment. Those applications can be divided into two groups: business applications and support applications. Both are hosted non-discriminatively on our worker nodes.

For our experiment, we use a cloud-native microservices demo application called Online Boutique [108], which is composed of a 10-tier microservices application as described in Figure 4.6. This application is written in 5 different languages (C#, Go, Node.JS, Python, and Java), communicating with each other using gRPC. This web-based e-commerce app - where users can browse items, add them to the cart, and purchase them - was created by Google teams in 2015 to serve as a design example and an experimental tool and application.

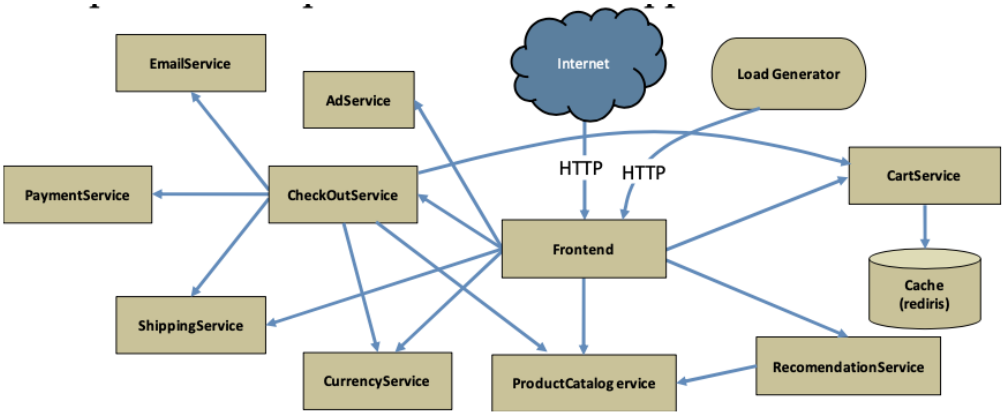


FIGURE 4.6: Online Boutique cloud application architecture

Our solution for proactive autoscaling is based on observability data. Thus, we deployed an observability stack composed of Kube-state-metrics [109], Linkerd [110], and Prometheus [89]. This stack fulfills the requirements

established in our architecture design. Kube-stats-metrics generate and expose metrics about Kubernetes API objects. Those metrics provide insights on Kubernetes object status and health. A complete Prometheus stack, a time series collection and storage component, is also implemented along with a visualization tool and Linkerd, service mesh for Kubernetes that implements transparent measuring proxies and provides advanced networking information for each microservice.

Kube-stats-metrics, Linkerd, Prometheus, and the predictor microservices are all installed inside the Kubernetes cluster alongside the online boutique microservices.

In the Kubernetes vocabulary, each containerized instance of a microservice is deployed in a "Pod". By using a services mesh such as Linkerd in our cluster, we also equipped each pod with a sidecar component acting as a proxy. All those proxies report their metrics to the service mesh controller, which in turn exposes itself to our Prometheus stack.

The time-series database provides information required by our predictor to forecast values that will be used by the horizontal pod autoscaler to operate. Our predictor is a microservice running within the same infrastructure as all the others, whose purpose is to process metrics from known microservices with a model developed to forecast values that can be used for auto-scaling.

This microservice uses a model in conjunction with selected metrics and/or events and published forecasted metrics which are used by standard Horizontal Pod Auto-scaler Kubernetes objects. This approach does not require custom control loops (controllers) to be integrated within Kubernetes internals. Moreover, models can be validated by data scientists before being used on real production clusters. They can also refine models following the same DevOps continuous process as developers and automate deployment in production.

In our implementation, a Keras model uses LSTM based RNN to predict values (number of requests along with end-to-end application latency) which are used in the proactive metric auto-scaling algorithm. As our experiments rely on generated and simulated traffic, teaching our model with our simulated data would necessarily mean a perfect forecast. We decided to use a real dataset extracted from a real application to demonstrate that we could implement the forecast of clients' requests using a state-of-the-art model tuned by data scientists [111] [112]. Figure 4.7 demonstrates the effectiveness of LSTM forecasting we put in place.

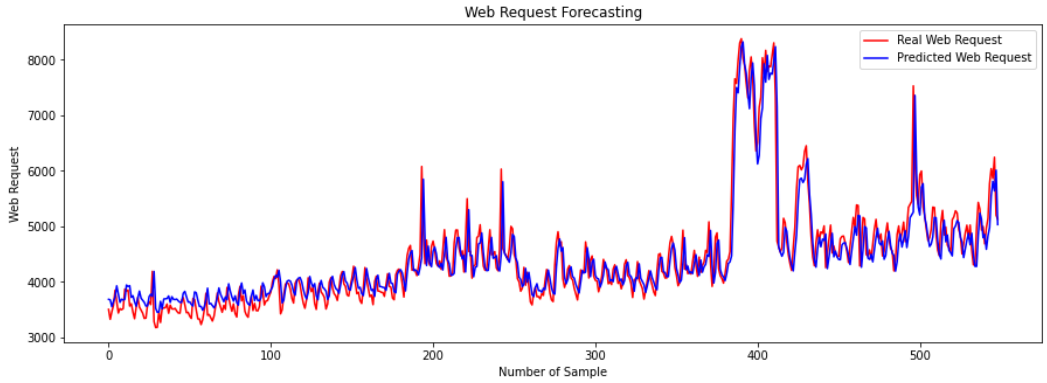


FIGURE 4.7: Web request time series forecasting using LSTM

4.4.3 Test description and results

As stated previously, our experiment runs on Digital Ocean public cloud platform. We also have an additional server containing a load generation microservice using Locust.io [113], which simulates connections from real clients navigating and using every component of our online boutique. These tests simulate a sudden influx of 500 users in different configurations of our platform's autoscaling system.

- **Experiment A**: it aims to evaluate the behavior of our platform without any autoscaling system. Results are presented in figure 4.9.
- **Experiment B**: it uses a well-known resources usage reactive autoscaling system. In our implementation, we monitor the CPU usage of all the microservices and duplicate them when a CPU saturation situation is detected to maintain an average of 50% usage across all replicas (see Algorithm 1). Results are presented in figure 4.10.
- **Experiment C**: it aims to show the behavior of our application with an autoscaling system based on the number of actual client requests. It scales our microservices to maintain a ratio of requests per replica optimized to get the best latency (see Algorithm 2). Results are presented in figure 4.11.
- **Experiment D**: it leverages machine learning LSTM forecasts using TensorFlow and Keras based on clients' requests values as in Experiment C. Those forecasted values are then injected into the observability platform system and then are used by the Kubernetes control plane to autoscale our online boutique application. This architecture is described in Figure 4.8. Our predictor was built to predict any microservice metrics using any monitoring data (see Algorithm 3).Results are presented in figure 4.12.

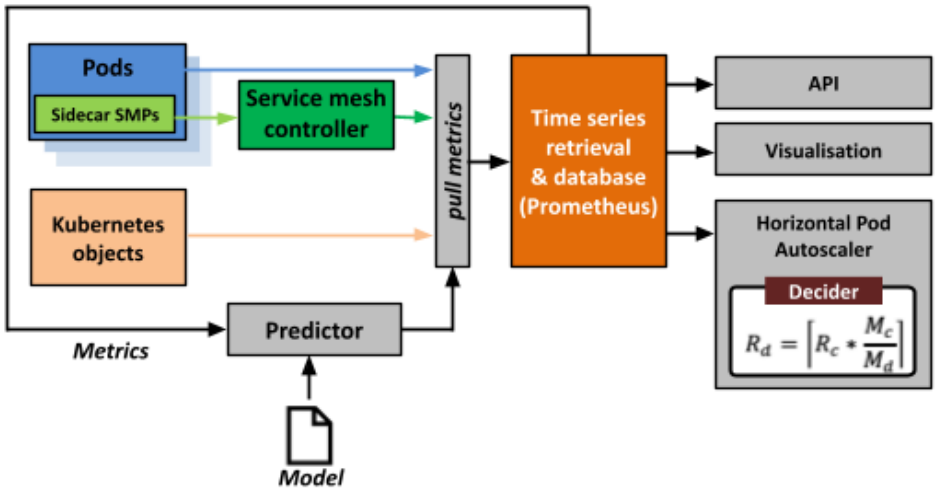


FIGURE 4.8: Components of the proposed proactive auto-scaler

Our experiments demonstrate the benefit of our proactive auto-scaling solution in enhancing end-to-end latency and the success rate (see results in Figure 4.12). As shown in the results of Experiment D, we obtain the best tradeoff between the end-to-end latency and the period during which the system is in the transient situation (auto-scaling period). During this transient period, the system is under resources shortage and needs to provision new worker nodes. Once new work nodes are ready, the orchestrator can instantiate new replicas to improve QoS.

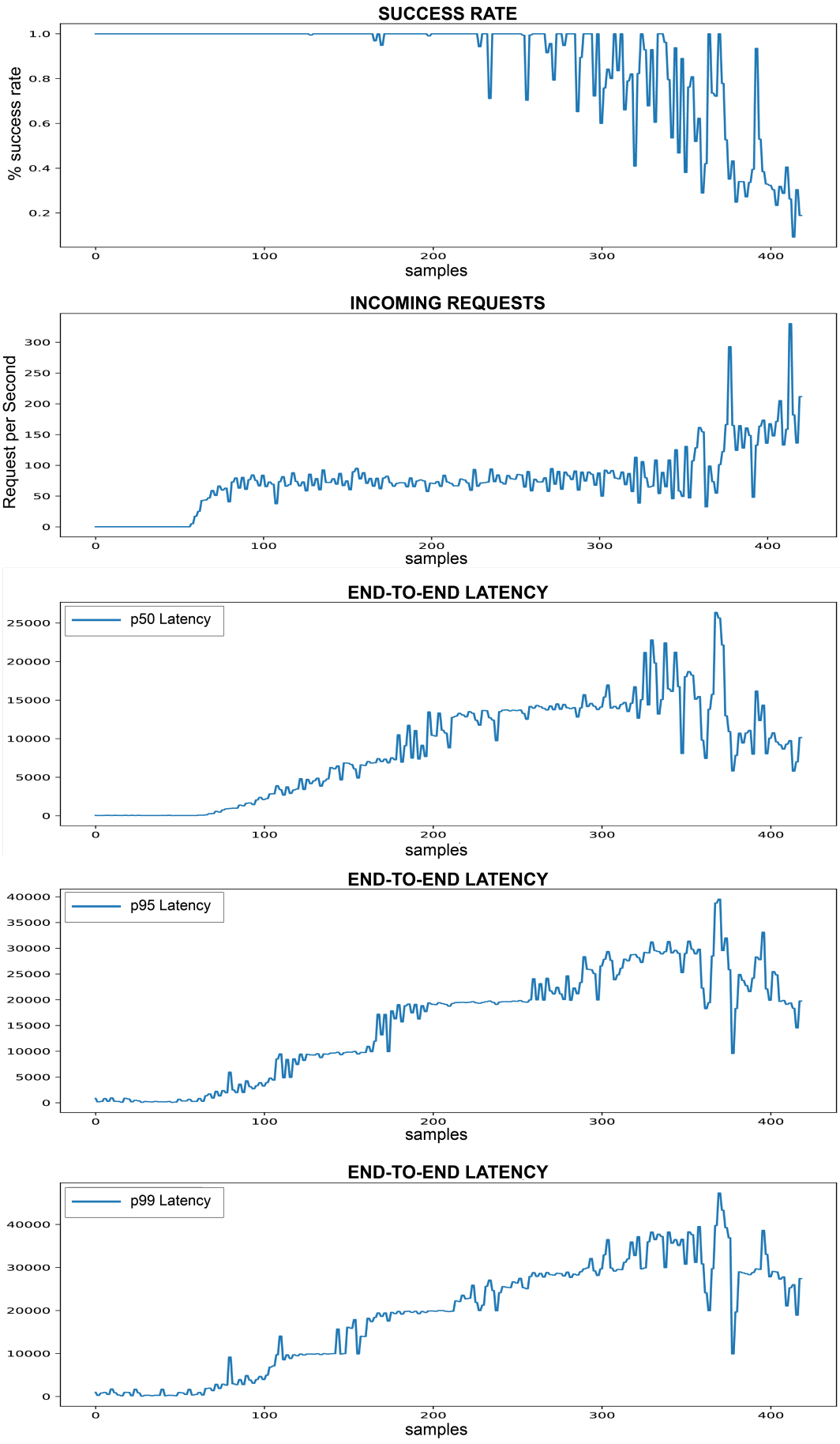


FIGURE 4.9: Results from our Proof of Concept : Experience A

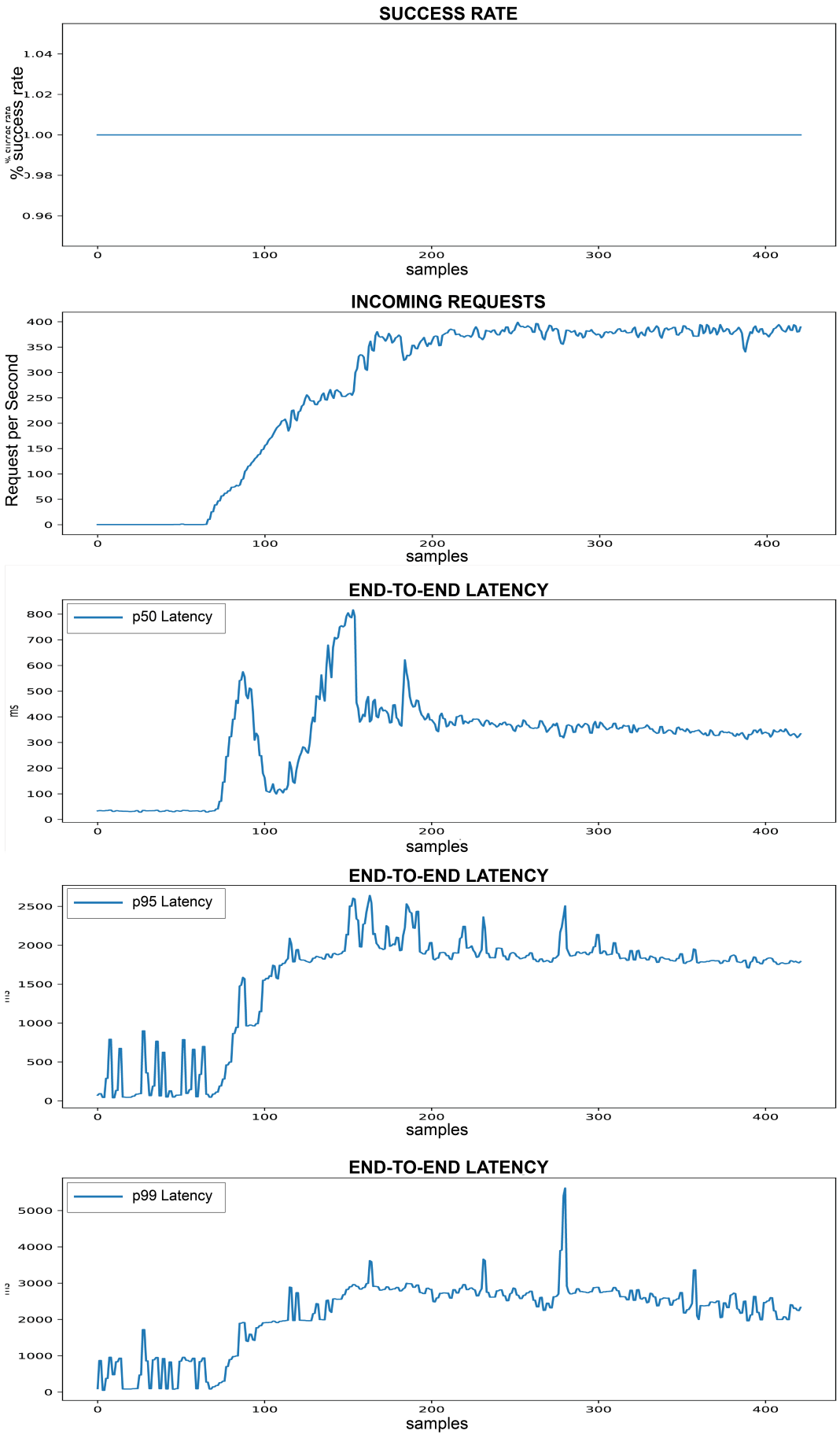


FIGURE 4.10: Results from our Proof of Concept : Experience B

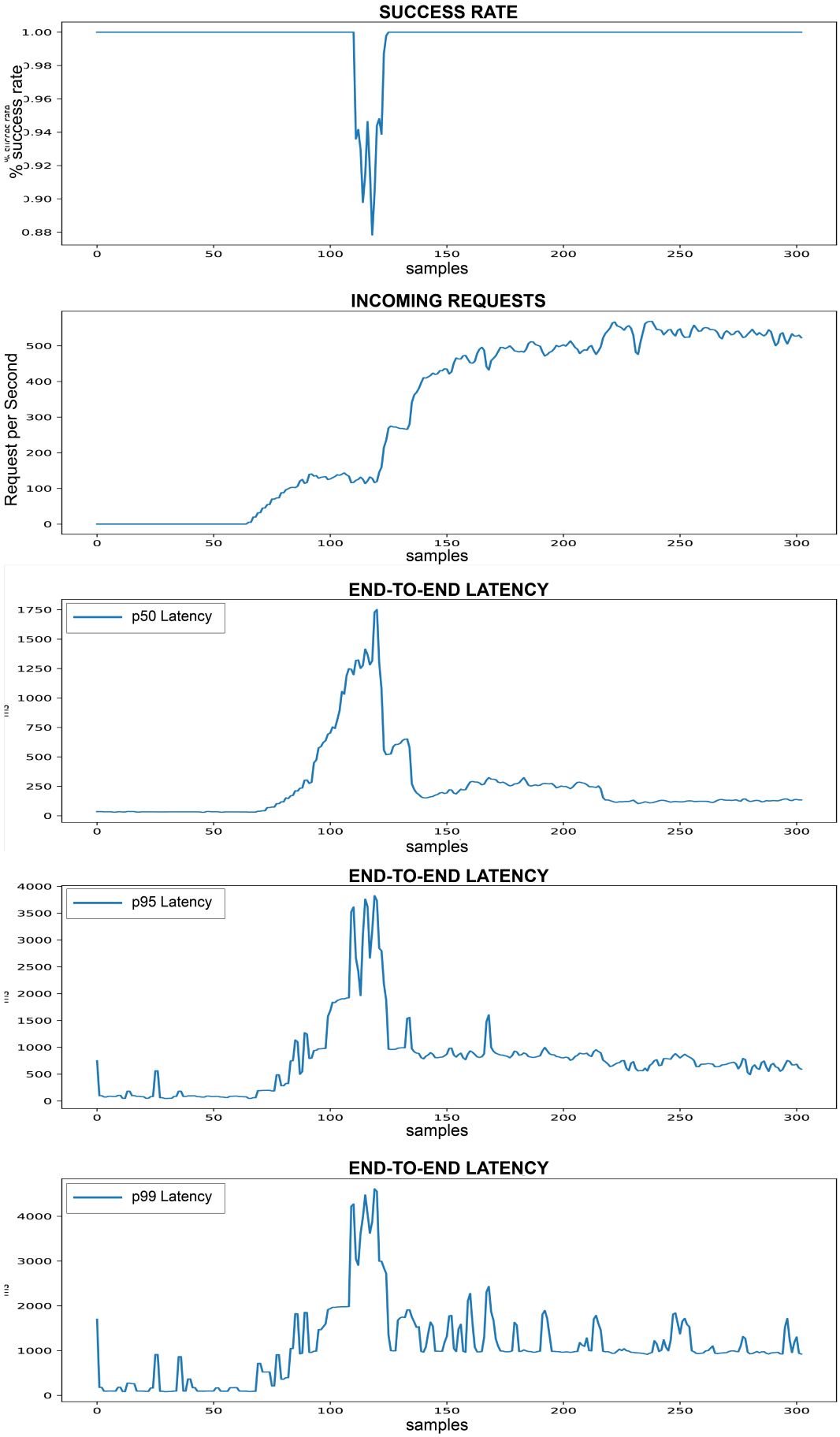


FIGURE 4.11: Results from our Proof of Concept : Experience C

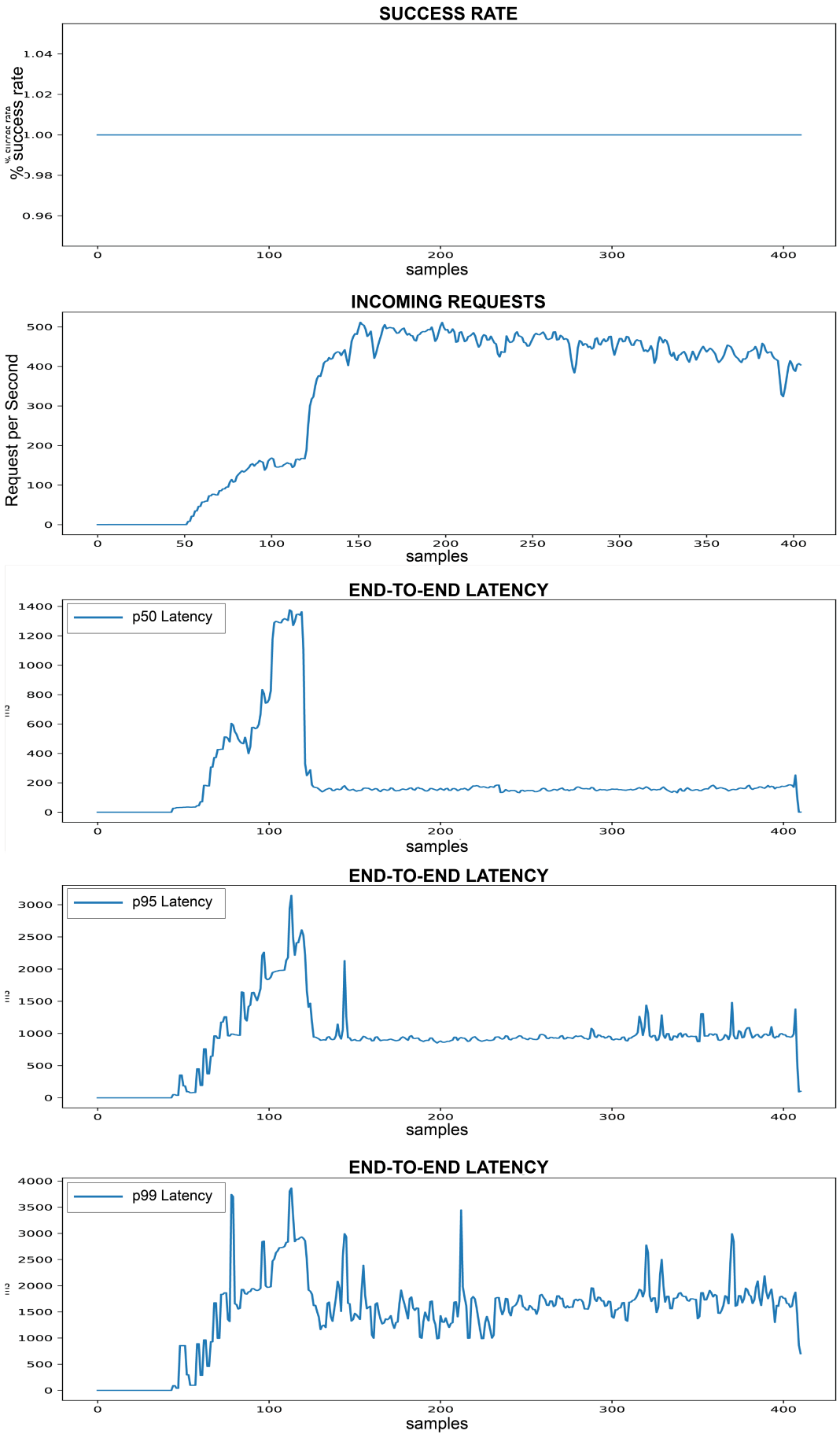


FIGURE 4.12: Results from our Proof of Concept : Experience D

Figure 4.12 (Incoming requests) displays the number of HTTP responses served by the frontend of "Online Boutique" to the simulated users' requests. All the responses are displayed with the correct ones and the totality of the HTTP return code. Figure 4.12 (success rate) displays the end-to-end availability during all the experiment. This availability is measured as the percentage of succeeded requests on the total of issued users requests. A request has succeeded if the status code returned by the frontend is 2xx (such as 200 OK). Errors or timeouts in one or multiple microservices running in the application affect the availability. Figure 4.12 (end-to-end latency p99) shows the 99th percentile of the application end-to-end latency for simulated users' requests.

As we can see, the number of responses served increases with autoscaling functionalities (Experiments B, C, D). Experiment A exhibits the behaviors associated with resources saturation as shown by the dramatic increase in the end-to-end latency and the increase in the number of erratic answers that affects the service availability. This is expected for a single non-autoscaled application under extreme load.

Furthermore, we observe many failures in the microservices of the "Online Boutique" which causes the application to be inoperative at the end of experiment A. Experiments B and C exhibit both a better behavior using CPU autoscaling and custom metrics observability based autoscaling, respectively. However, the application is kept under stress by inefficiently scaling CPU, thus showing worse latency performance. On the other hand, by reacting heavily under current load (i.e., client requests), experiment C produces both erroneous answers and low end-to-end availability. As displayed by experiment D, those types of surge traffic are the situations where proactively auto-scaled cloud-native applications shine.

Experiment D is an improvement over the issues of previous auto-scaling systems by avoiding the problems experienced while scaling under heavy load and thus showing a more reliable behavior. As we can see in Figure 4.12, the number of requests per second in Experiment D is higher, and the availability remains at 100% over the lifetime of the experiment, while surge traffic in Experiment C leads to errors and timeouts. Furthermore, the latency is lower in Experiment D. Thus, we show in this experiment a good compromise in all the metrics. Machine learning and deep learning are now technologies mature enough to provide production tools to implement proactive auto-scaling. This will enable better performance, lower latency, and fewer errors than other types of cloud-native auto-scaling.

The results graphs from experience A, B, C, D in Figure 4.12 are kept separated for scaling reasons. Indeed some result produce number that would be off chart or unreadable.

4.5 Conclusion

As cloud-native adoption is becoming the standard for new IT developments, organizations need tools and techniques able to achieve better QoS, reliability, and lower costs for their cloud-native applications. The democratization of machine learning and deep learning solutions is now offering a panel of new possibilities. The proactive auto-scaling framework presented aims to propose a new path for auto-scaling in cloud-native environments hoping to circumvent issues such as provisioning exact resources and managing Flash Crowds traffic that lead to related QoS degradations. Innovation happens at the borders of disciplines. By seizing this opportunity and applying machine learning to cloud-native orchestration, we might tackle some challenges of autonomous computing.

Cloud-native applications and infrastructures already offer natively by design some features such as basic self-healing and control-loop. Bringing machine learning and deep learning and applying them to observability and monitoring data of cloud-native applications is still in its infancy. Our framework is a step further to demonstrate this relationship.

Our success with implementing our Observability driven reactive autoscaling inside a production environment reinforces our conviction that advanced autoscaling can greatly contribute to improving quality of Service in Cloud Native Applications. We envision implementing our proactive solution in an industrial production environment.

Chapter 5

An Architecture Framework for Virtualization of IoT Applications: Cloud-native and Observability in IoT

IoT can benefit from Cloud Computing, Cloud Native Applications and Observability. The concept and architecture detailed in the previous chapter can be applied to multiple use cases from core network observability, through cloud, edge, and even IoT. This chapter is our proposition of transposition of the cloud native paradigm and other state-of-art technologies to IoT.

5.1 Introduction

The Internet of Things industry is constantly looking for new improvement features that support cutting-edge service innovations with deployment velocity and business agility. As major key drivers in the evolution of IoT, Software Defined Networking / Radio (SDN, SDR) and Network Function Virtualization (NFV) are being positioned as central technology enablers towards decoupling IoT hardware from service deployment, leveraging an increasing in the number and type of services supported over a single deployed IoT platform. The support of virtualization concept within the IoT devices is envisioned to achieve critical cost reduction in the service offering, and at the same time, being able to easily bring a new set of innovative service into the market. This allows IoT device vendor to open their platform to a great extent, and at the same time, it enables IoT service provider to avoid vendor lock-in with proprietary hardware and software technologies. The IoT service provider will gain greater control over the IoT devices by simplifying network management with centralized management and control of IoT devices from multiple vendors, and creating opportunities for collaboration and interoperability.

Adopting SDR, SDN and NFV offers endless expectations that covers

many distinct operational areas, such as fine-grained radio control and monitoring, advanced signal processing, e.g. collaborative signal processing, distributed enforcement of QoS, network resources management, in-network processing and storage of data including fusion and detection of complex events, etc.

The path toward the commoditization of IoT devices into a generic white-box devices able to support most of IoT functions extended with out-of-tree functions will result also in scalability benefits since the underlying resources can be mobilized when needed according to the aggregated demands, reducing the amount of specialized IoT resources to be provisioned and making feasible other modes of feature evolution. This concept sheds up light on new ways of designing IoT devices in order to improve IoT service offering with the support of virtualization and decentralization, to allow multiple isolated environments to be executed in a single device. As a result, cooperation among heterogeneous devices from multiple vendors, within an IoT network, is facilitated and at the same time, it provides resource efficiency and cost-benefit.

We propose an IoT NFV architecture framework that leverages the above-mentioned advantages and providesflexible and reconfigurable solutions to create and deploy new customized on-demand virtualized IoT services. Our proposed architecture framework abstracts the underlying physical IoT network resources into a set of logical resources (radio, sensing capabilities, computational, memory and storage) used for service orchestration in terms of hosting, chaining and managing of IoT functions. Thus, this chapter includes the following main contributions:

- Proposing an architecture framework to bring virtualization in IoT network based on introducing SDR, SDN and VNF paradigm shifting.
- Exploiting the virtualization paradigm introduced for IoT, to describe a set of use-cases and deployment scenarios.
- Presenting a proof of concept showing how the proposed architecture framework can be prototyped using Raspberry Pi acting as IoT devices and Docker containers as IoT functions.

The rest of this chapter is organized as follows. Section II reviews some background knowledge and state of the art. Section III presents the proposed architecture that brings virtualization in IoT network and introduces the concept of Generic IoT device. Section IV presents some use-cases that can be derived by focusing on the advantages of the solutions in addressing well-known IoT challenges. Section V presents a proof of concept implementation. Finally, main conclusions are drawn in Section VI.

5.2 Background and State of the Art

IoT networks are complex and difficult to design, manage and operate in order to offer IoT services. These networks are composed of multiple,

potentially heterogeneous IoT devices, acting as sensors and actuators, interconnected with each other and with IoT gateways. These latter offer connectivity to the rest of the Internet up to Cloud, where the service is exposed (stored, analyzed, presented and shared) to end-users. The IoT architecture involves a multitude of software and hardware components and calls for a wide range of protocols to allow device-to-device and device-to-cloud communications. Consequently, one of the major requirements for the successful and wide deployment of such environment to offer IoT services 2019 IEEE Conference on Network Softwarization (NetSoft)183 concerns the efficient design, management and operation of IoT networks while addressing complex problems related to massively distributed applications, scalability support, and upgrading. Control architecture and virtualization using SDN and NFV are actually changing the way networks are designed. The efforts that led to these concepts are clearly related to the long research background on supporting programmable packet processing, active networks, in-band and out-band control, separation of control and data, network operating system, network virtualization, etc. Although network virtualization has played an important role in computer networks for many years, the virtualization concept for IoT is still in its infancy.

Many works suggested that the control plane for IoT is the appropriate place to introduce flexibility and programmability in order to achieve a multiservice environment with more reliability and guaranteed performance. Accordingly, the introduction of SDN in IoT has been proposed in many research papers [114] [115] [116] [117] [118] [119] and various lessons have been learned for future deployments related to (1) seamless integration of wireless sensor networks and mobile networks, (2) ability to modify the network behavior according to user needs, (3) unified view on accessing, configuring and operating IoT cloud systems, (4) global optimizations with centralized methods, and (5) solving scalability issues within large IoT deployments, among many others.

A very little attention is given to IoT virtualization and the introduction of NFV concept for providing IoT service as a network slice offering an IoT Service deployment over virtualized IoT architecture. The survey [120] presented a set of works tackling SDN and virtualization solutions for IoT, but most of the items examined were concentrated on SDN applications. The study [121] suggested a new multi-layered IoT architecture involving SDN and NFV and proposed some use cases and founding principles for building an IoT infrastructure. In [122], we started an investigation on how NFV concept can be extended to IoT to allow decoupling IoT functionalities from specific dedicated devices and we proposed an energy efficient solution for the placement of IoT Service Function. However, the studied works lack a clear vision on the way to design a virtualization architecture framework for IoT network. In our case, we suggest the virtualization of full stack IoT functions using generic IoT device, from radio resources using SDR to network functions using SDN and NFV concepts. We also believe that the adoption of SDR, SDN and NFV over IoT networks will be a key enabler towards

more flexible and agile end-to-end service provisioning that allows overcoming several existing limitations in terms of operational flexibility, evolvability and large-scale deployment.

5.3 Paradigm shift toward generic IoT Device

Proprietary IoT devices and proprietary operating systems (OS) for IoT are dominating the market today, putting critical pressure on IoT service providers for always adopting vendor lock-in solutions which lack portability and interoperability. A paradigm shift has started to take place to adopt commercial off-the-shelf (COTS) devices with a shift to standards-based hardware such as the case of solutions from the Department of Defense (DoD) [123] used for military and tactical communication. These COTS devices benefit from competitive pricing, interoperability and best practices while most of them are using open source OS [124], protocols and tools.

This paradigm shift opens a new perspective for leading IoT movement toward supporting Software Defined Radio / Networking (SDN, SDR) and Network Function Virtualization (NFV). The objective is to remove the dependency from dedicated and specialized physical IoT devices as well as abstracting physical resources into virtual resources that can be allocated when needed. With this approach, IoT service providers can concentrate on developing and optimizing software functions that can be dynamically orchestrated to respond more efficiently to changing market demands.

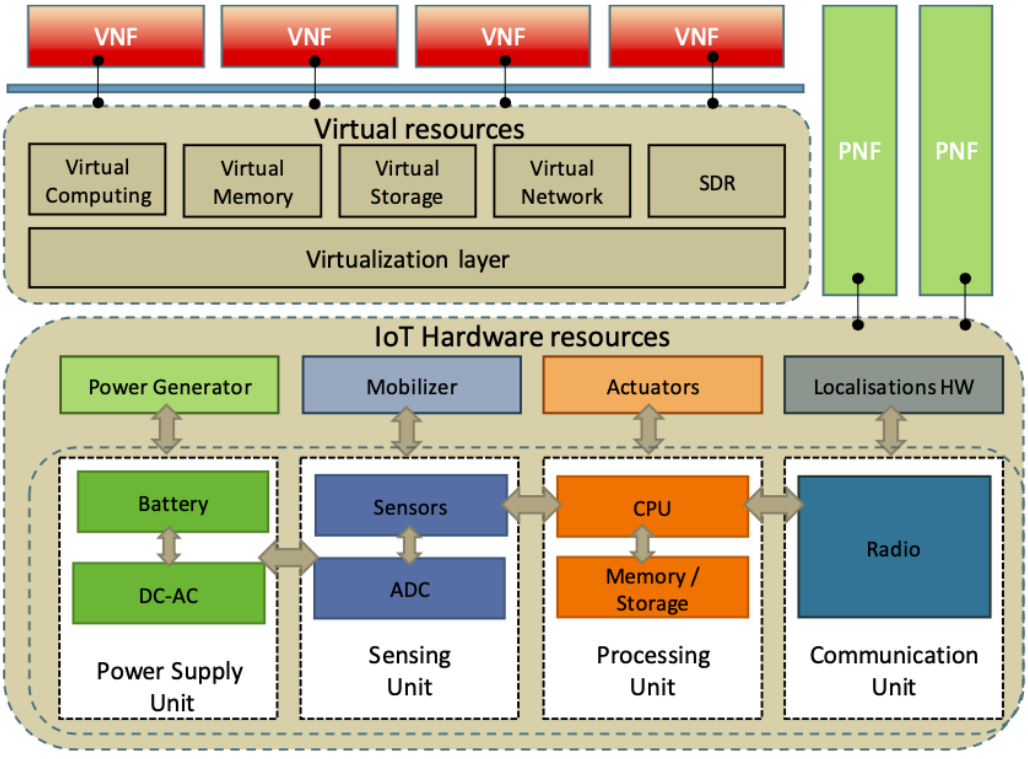


FIGURE 5.1: Paradigm shift for IoT devices

Figure 5.1 shows our proposal mapping of an IoT device to ETSI guideline for NFV reference architecture framework [125]. The decoupling of hardware and software exposes a new set of entities such as the virtualization layer, virtualized resources, the Virtualized Network Functions (VNFs). This mapping is augmented with SDR capabilities to allow access, control and management of the radio spectrum. VNFs can be chained with other VNFs and/or Physical Network Functions (PNFs) to realize a Network Service (NS). The PNFs are part of physical resources that remain as hardware components such as sensing capability that cannot be virtualized. The Network Services are created based on associated VNF Forwarding Graphs (VNFFGs), Virtual Links (VLs), PNFs, VNFs. In addition, the management and orchestration functions are not illustrated in this figure. Those functions are part of the IoT service provider infrastructure network.

The virtualization techniques used inside an IoT device are similar to those used for servers and cloud computing. The virtualization allows sharing of the physical resources so that multiple network functions (workloads in general) are executed while they are co-located with each other in a fully isolated environment. This represents a progressive manner to design, deploy and manage IoT Service. The deployed IoT devices (generic IoT device, IoT gateway) form what we called, an IoT network segment that is usually connected to service provider cloud infrastructure (cloud segment) to deliver a full end-to-end IoT Service. The cloud runs a set of process for data aggregating, correlation, analysis, classification, visualization, etc. A fog computing or mobile edge computing (MEC) can be deployed in the vicinity of the IoT network or within the backhaul to reduce the heavy burden on the cloud and to improve the service performance.

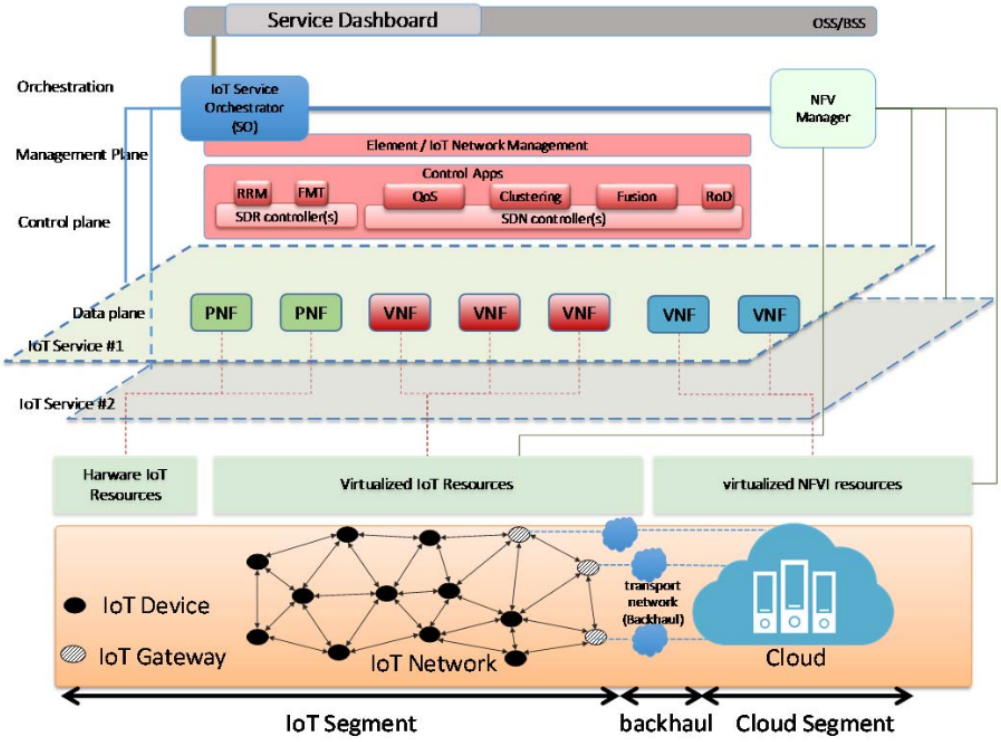


FIGURE 5.2: Virtualization architecture framework for IoT network

Figure 5.2 illustrates the architecture framework for virtualization of IoT network segment when connected to the cloud to deliver an end-to-end IoT service. This architecture is also aligned with MANO (Network Function Virtualization Management and Orchestration) framework. We focus on control and management planes only for IoT segment. In this architecture, we suppose that each IoT device follows the paradigm shift proposed in Figure 5.1. An IoT Service is represented by a sequence of PNFs and VNFs instances (IoT VNF and cloud-based VNF), chained together to compose an IoT Service (i.e. Network Service) that requires a particular amount of resources in terms of sensing capabilities, radio for communication, computational, memory and storage. In this context, the placement, management, chaining, and orchestration operations of these VNFs should be carefully considered to meet the required performances along with energy constraints to support diverse IoT services over the same shared infrastructure.

In particular, as illustrated in Figure 5.2, the following entities are considered as building blocks for deploying IoT services.

- IoT Physical Network Functions, essentially for sensing and radio communication and IoT Virtual Network Functions that represent the software part of an IoT device that runs as an isolated code on top of a virtualization layer.
- SDR-based applications and SDR controller for the realization of radio signal processing applications and radio-based control. In particular,

using SDR for radio spectrum will allow an IoT device to be augmented with a set of wide range of software-based signal processing blocks connected to each other through flowgraphs that describe the data flow between those software blocks. The reconfigurability of the flowgraphs is a key feature provided by SDR. In this case, an IoT device will no more utilize a single specific radio with specific signal processing and protocol, but instead, it will use a general-purpose radio that supports a wide-range of signal-processing applications that can be controlled through a centralized logic.

- SDN-based application and SDN controller for the realization of IoT-level control such as IoT routing, multipath forwarding, load balancing, clustering, in-network data fusion, etc.
- Element Management functions / IoT Network Management, which provide a package of management functions, e.g. Fault, Configuration, Accounting, Performance and Security (FCAPS) management, for IoT.

5.4 Use cases and scenarios

We present in the following some use cases that can be efficiently developed over the framework. The objective is to highlight some of the advantages of the solutions while addressing challenges related to deployment, configuration and operation, seamless integration with Internet, global optimizations, and scalability among others. Major advantages come with the use of generic IoT device augmented with full-stack virtualization capabilities that provide enhanced flexibility and re-configurability, software-based feature updates, simplified deployment procedures, and easy control of network topologies. These use cases are supported by a set of message sequence charts that are not illustrated for simplicity reasons.

5.4.1 Clustering

In the context of IoT and more generally device-to-device communication, the organization of the devices into clusters can be greatly enhanced with SDR, SDN, and NFV. There is a huge amount of work in the literature that tackled the problem of sensors network clustering to ensure low energy consumption as described by different surveys on this topic [126] [127]. The clustering objectives for IoT are multiple: enhancing connectivity and communicating between nodes, providing load-balancing and fault-tolerance, and maintaining hierarchical topology to support scalability.

However, as the cluster-head is selected based on some criteria, the selection process and the associated cluster-head functionalities should be provided in a flexible manner. Two relevant key features provided by our architecture framework related to clustering process could be (1) In-network processing, such as data aggregation and fusion, (2) Dynamic clustering, rotation and backup of cluster-head.

5.4.2 Tracking

Target tracking techniques have been widely studied in the literature of wireless sensor networks [128]. The tracking consists of state-estimation of the target using techniques ranging from single-node to collaborative methods.

The majority of these methods employs active prediction-based scheme coupled with selective activation of nodes activities. The nodes are then waked-up on-demand to follow the target path. In this context, the proposed virtualized IoT architecture framework can enhance the target tracking by extracting useful information using in-network processing, fine-grained radio control and monitoring, dynamic clustering, collaboration between nodes, coordination between communication-related and sensing-related operations, etc.

5.4.3 Tactical networking and high dynamic network

Tactical networks and high-dynamic networks are composed of extremely heterogeneous wireless and ad-hoc mobile nodes. They rely on a wide range of ground sensors, robots and UAVs aiming to provide mission-critical applications. These networks suffer from limited bandwidth and intermittent connectivity due to communication range, interference, mobility, and overhead induced by security requirements. As consequences, channel conditions and network topology vary over time. The service requirements are also extremely heterogeneous ranging from simple data dissemination to real-time communication with QoS support.

This framework can be used to efficiently support service requirement for efficient tactical networks communication such as:

1. Radio communication reprogramming and cognitive radio,
2. Network awareness,
3. Cross-layer networking.

5.5 Proof of concept and implementation

This section describes the primary implementation realized through a set of Raspberry PI 3 model B v1.2 acting as IoT devices. Each device is composed of single-board computer powered by a quad-core ARMv8 BCM2837B0 Cortex-A53 running at 1.2 GHz and equipped with a BCM43438 wireless LAN and Bluetooth Low Energy (BLE) chips. The board is also featured with 40 GPIO pins that can be controlled in software as input and output pin and used for wide range of purposes such as wiring many types of sensors and actuators. A list of 50 sensors is provided in [129].

Figure 5.3 illustrates our target application. The network is composed of devices having multiple sensing capabilities that can be activated when

needed. The devices are deployed in a single building and we simulate a temperature measurement as an IoT service.

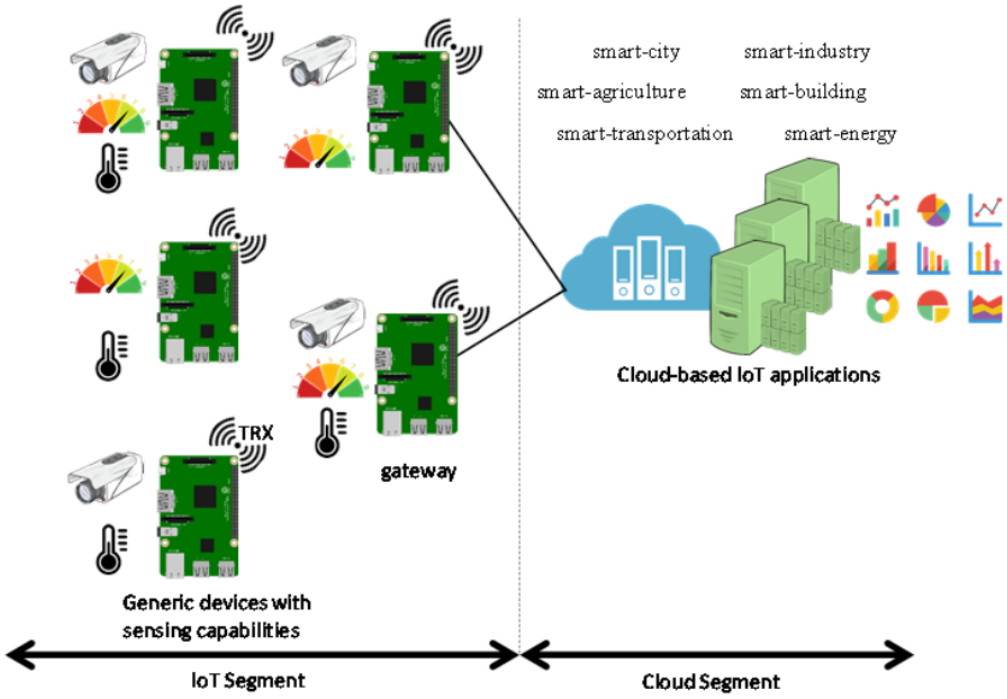


FIGURE 5.3: Network topology used as the target application

5.5.1 SDR Capabilities

Each device is augmented with a low-cost digital USB 2.0 stick with RTL2832U chipset initially designed for DVB-T reception but can support SDR capabilities. Another alternative is to use LimeSDR-Mini TODO USB stick which provides Transmission (TX) and Reception (RX) capabilities with SDR.

There exist a large set of general-purpose software supporting SDR such as SoapySDR [130], CubicSDR [131], Gqrx SDR [132] and GNU Radio [133], which are used for monitoring and analyzing radio signal reception. In particular, GNU Radio is an open source toolkit for software development that provides building blocks for signal processing. The GNU Radio Companion (GRC) tool allows designing a flow graph by connecting different blocks with input/output connections. Furthermore, tools such as rpitx [134] can allow the Raspberry Pi to transmit over a wide range of frequencies from 5 KHz up to 1500 MHz. This is achieved by connecting the GPIO 4 (Pin 7) with a band-pass filter and a wire acting as an antenna. This setup turns the Raspberry Pi as a general radio frequency transmitter. The rpitx tool can accept an I/Q signal as an input and can transmit I/Q signal back on a specific frequency.

The combination of rpitx and Gnu Radio opens new capabilities and perspectives to ensure flexible radio manipulation and reconfiguration while

helping to increase scalability, agility and enabling better use of radio resources. Our first setup allows us to have a generic IoT device that is easy to configure towards a new transmission technology and protocols, so it can adapt dynamically according to needs and scenarios.

This proposed generic IoT device offers unmatched levels of programmability with the support of a high range of frequency bands and communication standards.

5.5.2 VNF Capabilities

The VNF capabilities that allow multiple isolated environments to be executed on a single Raspberry Pi device are provided by the virtualization layer based on containers technology using Docker [135]. This represents a fast way of creating, installing and running independent manageable Linux containers. Furthermore, to define, run and manage multi-container applications, Docker Compose is used to provide simple service composition facilities.

We create an online repository from Docker Hub which is, in our case, the place where the IoT service provider is supposed to host the image of the different VNFs available to be on-boarded on the device. We show in the next section a scenario of service chain composition for clustering management for our temperature measurement as an IoT service.

5.5.3 Service Function Chain Composition and Clustering Management

We suppose that based on the battery depletion level a specific Cluster Head (CH) is elected. This CH is in charge of collecting the temperature level from all the Cluster Members (CM). Two VNFs are defined as docker images: Temperature_CM_VNF and Temperature_CH_VNF.

The first VNF for the CM reads periodically the actual temperature from the PNF sensor. The second VNF for CH, performs a data fusion by providing the mean value (average) of the received temperature. The orchestration capability is done by a simple script based on CLI (Command Line Interface). However, it is possible to use advanced tools such as Kubernetes (k8s) which is an open-source software for automating deployment, scaling, and management of containerized applications.

The logic sequence for the orchestration in our case is

1. A CLI for Docker client contacts the Docker daemon running locally in each device,
2. The Docker daemon pulls the appropriate image from the Docker Hub either Temperature_CM_VNF (for cluster member) or Temperature_CH_VNF (for cluster head),

3. The Docker daemon creates a new container from that image and runs the executable that produces the output, and
4. The Docker daemon streams that output to the Docker client, and then forwards the output to the next VNF in the SFC.

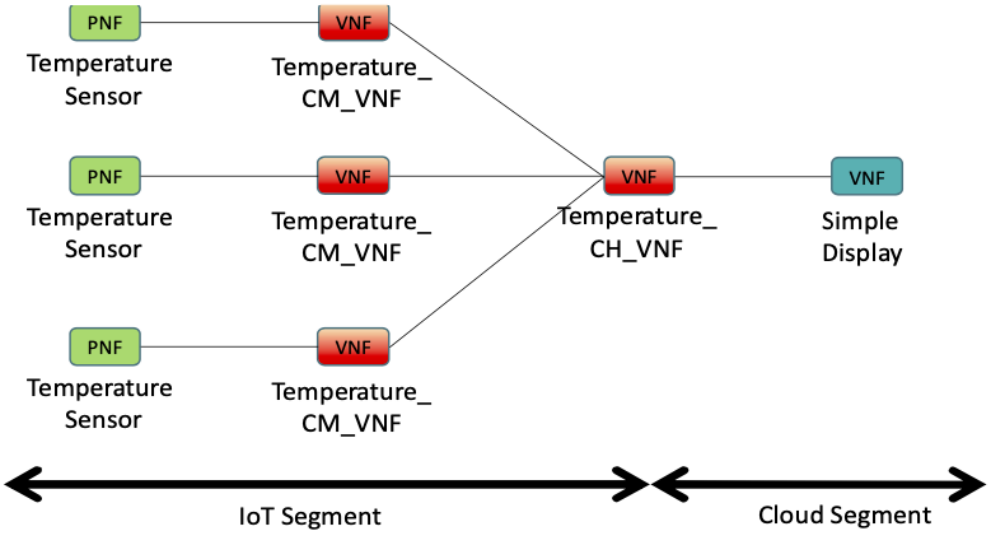


FIGURE 5.4: SFC composition for temperature measurement as an IoT service.

Figure 5.4 shows the SFC composition for temperature measurement as an IoT service. This SFC is arranged as a PNF, two IoT VNFs and one cloud VNF used for displaying the data.

It is worth noting that an IoT device is able to manage multiple containers at the same time and one device can participate in multiple IoT services in a parallel manner.

When the CH is updated according to the energy criteria, an updated SFC composition is triggered by the service orchestrator in order to install new image or to migrate existing container from Docker images (Temperature_CM_VNF and Temperature_CH_VNF) on the appropriate member (either CH or CM).

In our case, the temperature service is stateless and there is no need to synchronize the different contexts of VNFs. Context and state synchronization can be solved using Docker Swarm so that the new CH joins the "Cluster head swarm" before the old CH leaves it.

Our implementation demonstrates the ability of our solution to create and deploy new customized on-demand virtualized IoT services by managing an IoT network dynamically through the virtualization of full stack IoT functions, from radio communication processing functions to high-level processing capabilities.

This result in a more flexible and re-configurable network that suits a variety of IoT business and market applications.

5.6 Conclusion

We proposed an IoT NFV architecture framework that leverages existing key enablers technologies, namely SDR, SDN and NFV, in providing flexible and re-configurable solutions for creating and deploying new customized on-demand virtualized IoT services.

We supported the introduction of generic IoT device capable of accepting new transmission technology by (re)programming the radio functions and offering multiple isolated environments to execute multiple softwarized functions in a virtualized environment. We identified a set of relevant use cases and we realized a proof of concept implementation to demonstrate the feasibility of the proposed architecture framework. Wide acceptance of this concept will open a new era of opportunities for IoT.

Conclusion and perspectives

The Internet has become more than just a means of communication: Web applications are now part of our daily lives. Individuals and organizations of all types and sizes rely upon web services for a large variety of use-cases. The introduction of Cloud Computing lowered the entry barrier for web applications. It became possible to host web applications with computing resources available in a pay-as-you-go fashion. As many new and old organizations benefited from this opportunity, we witnessed a soar of new technologies, use-cases, and new challenges. Indeed, while cloud computing and cloud-native approaches such as cloud-native applications have revolutionized how we build and operate software, they also created new challenges.

Modern cloud-native applications have solved many difficulties presented by previous architectures, which make them faster to develop. However, this result came at a great cost for simplicity. Cloud-native applications have increased moving parts, third-party dependencies, release frequency, and need for expertise. At the same time, users have normalized the idea that their applications must perform 24/7 in a fast, reliable, and convenient way. Both users and organizations rely on cloud-native applications on a day-to-day basis. In this context, the need to understand those applications, know their health states and operate them with such high demands for Quality of Service is of paramount importance.

From very earlier on, Researchers and companies alike foresaw those challenges. As early as 2001, IBM VP Jeffrey O. Kephart and D.M Chess announced that the increase in complexity would have to be tackled. They introduced a concept called "Autonomous Computing" where high-level interfaces could abstract the difficulty of managing and operating complex systems. This concept was based on a self-managed control loop, which allowed monitoring data, analyzing them, and planning and executing preprogrammed knowledge. Automation and Orchestration are desirable features in the context of cloud-native applications. This is why, in this thesis, we have pursued the goal of implementing more autonomous computing features into cloud-native applications.

Our first contribution on Observability allowed us to gain visibility and understand what happened inside and around cloud-native applications and infrastructure. At the time of writing, the concept of Observability and its differentiation from traditional monitoring is gaining traction and desirable in the IT industry. We contributed to defined requirements, characteristics to achieve Observability in a modern cloud computing environment, such as

Cloud-native applications hosted in the public Cloud. We defined strategies and methods that can be brought together and build an architecture framework that can achieve Observability on a cloud-native environment. This framework was then deployed into a commercial production environment to improve the engineers' ability to diagnose outages and failures while improving the quality of service. This observability framework eased the work and assisted both developers and operation engineers in their missions. This work was published and demonstrated at IEEE IM 2019. [5]

Our second contribution focused on auto-scaling, as our state-of-the-art and field exploration showed a need for more research in this area. While many works were conducted on the subject, most of them focused on virtual machines, basic metrics, and previous architecture. Our research was based on using our observability data to drive our auto-scaling, following our path based on autonomous computing. We first described an architecture able to use our observability framework as an input for a production orchestrator and auto-scaler. We choose real usage metrics from the communication bus between microservices as the input for the auto-scaler. This enabled us to scale efficiently outside the saturation situation that can arise with CPU usage-driven auto-scaling. This solution was also ported to a production environment and now helps Lectra to achieve satisfactory QoS in the equipped microservices. This part was also demonstrated at IM 2019.

We decided to push further Observability-driven auto-scaling with Proactive Auto-Scaling for Cloud-Native Applications. We leveraged state-of-the-art machine learning and service mesh inside a dedicated testbed free from Lectra's constraints. This enabled us to achieve a proof-of-concept where machine learning forecasted time-series could be used to detect and predict a sudden increase in traffic, namely flash-crowd and proactively auto-scale to avoid saturation situation during the scale-up period. This work has been the subject of a publication at Globecom 2020. [3]

Our third contribution focused on applying the innovations and technologies available from cloud-native applications to the Internet of Things. IoT objects are now more powerful than ever and able to host containers. Containerization alongside virtualization, Software-defined Radio (SDR), and Software-defined Networking (SDN) pave the way for general purposes IoT object architecture connected to the Cloud and able to communicate with sensors. Furthermore, SDR and programmable radio are essential enablers for IoT devices. Those technologies could create cloud IoT devices accessible like any other cloud resource. This work was published at NETSOFT 2019. [4]

We believe that our work contributed to improving the way cloud-native applications can be operated. Autonomous computing and advanced orchestration are of paramount importance for operation engineers and developers alike in their mission to deliver better, faster, more reliable cloud-native applications. Now a very hot topic in the industry followed by all the major players, Observability is a field that requires more research to clearly define

and characterize the subject. In 2021, the CNCF announced a new project around Observability-driven auto-scaling and envisioned bringing it to the masses [136].

While not focused on security, our work also encompasses the possibility for disaster recovery, resilience to cyber-attack. Indeed, observability systems enable easier protection and monitoring against threats. Automation and Orchestration by employing a "cattle vs. pets" approach enable systems to be recovered and restored in less time in case of intrusion, defacing, and other disasters such as data-center outages. While not the most cost-efficient, auto-scaling increases the system's resistance to DDoS attacks and reduces fall-outs on legitimate users.

We believe that in a production environment, machine learning-enabled auto-scaling could power use-cases such as delay reduced auto-scaling from zero and thus reduce cost, energy consumption and carbon footprint by not keeping some microservices in always-on configuration. As more IoT objects and sensors need to be connected to the Internet and with the advent of edge and fog technologies, SDR and NFV could enable frequency-agile re-configurable gateways for multiple use-cases.

Résumé en Français

Internet est devenu plus qu'un simple moyen de communication : les applications Web font désormais partie de notre quotidien. Les individus et les organisations de tous types et de toutes tailles s'appuient sur des services Web pour une grande variété de cas d'utilisation. L'introduction du Cloud Computing a abaissé la barrière d'entrée pour les applications Web. Il est devenu possible d'héberger des applications Web avec des ressources informatiques désormais disponibles en mode de paiement à l'utilisation. L'adoption du cloud computing par les organisations et les entreprises de toutes tailles a été fulgurante. Cela a stimulé l'innovation et nous avons assisté à une montée en flèche de nouvelles technologies, de cas d'utilisation et de nouveaux défis. Les applications Web sont passées d'une conception monolithique à un système distribué reposant sur la virtualisation, la conteneurisation, l'orchestration et les microservices.

Les applications cloud natives modernes ont résolu de nombreuses difficultés présentées par les architectures précédentes, ce qui les rend plus rapides à développer. Cependant, alors que ces innovations permettent aux développeurs d'être plus créatifs et productifs, cela a également conduit à une augmentation spectaculaire de la complexité. Ce résultat a eu un coût élevé pour la simplicité d'opération. Les applications natives du cloud ont augmenté le nombre de composants, les dépendances avec des tiers, la fréquence de publication et le besoin d'expertise.

En effet, alors que le cloud computing et les approches cloud natives telles que les applications cloud natives ont révolutionné la façon dont nous concevons et exploitons des logiciels web, ils ont également créé de nouveaux défis. Dans le même temps, les utilisateurs, qui s'appuient sur des applications cloud, ont accepté l'idée que leurs applications doivent fonctionner 24h/24 et 7j/7 de manière rapide, fiable et pratique. Les ingénieurs opérationnels se sont retrouvés avec des outils, des stratégies et des techniques créés pour exploiter les architectures logicielles précédentes alors que la nature et la complexité de leurs missions étaient révolutionnées. Dans ce contexte, la nécessité de comprendre ces applications cloud natives modernes, de connaître leur état de santé et de les exploiter avec des exigences aussi élevées en matière de qualité de service est d'une importance primordiale.

De nombreux travaux dans la littérature ont évalué les exigences pour l'exploitation et la surveillance des applications cloud [46] [47] [48] [49], ils

ont unanimement déclaré que les outils, les techniques et les stratégies créés pour les architectures héritées étaient en deçà de les exigences nécessaires.

Très tôt, les chercheurs comme les entreprises ont anticipé ces défis. Dès 2001, le vice-président d'IBM Jeffrey O. Kephart et DM Chess ont annoncé qu'il fallait s'attaquer à l'augmentation de la complexité. Ils ont introduit un concept appelé "Autonomous Computing" où des interfaces de haut niveau pourraient faire abstraction de la difficulté de gérer et d'exploiter des systèmes complexes. Ce concept était basé sur une boucle de contrôle autogérée, qui permettait de surveiller les données, de les analyser, de planifier et d'exécuter des connaissances préprogrammées. L'automatisation et l'orchestration sont des fonctionnalités souhaitables dans le contexte des applications cloud natives. C'est pourquoi, dans cette thèse, nous avons poursuivi l'objectif d'implémenter des fonctionnalités de calcul plus autonomes dans des applications natives du cloud.

Cela nous a conduit à des questions de recherche majeures, notamment, mais sans s'y limiter :

- Quelles sont les exigences pour exploiter des applications cloud natives dans un environnement de production ?

- Comment obtenir des informations sur les applications cloud natives et leur environnement ?

- Quel est le meilleur moyen d'obtenir une mise à l'échelle automatique efficace pour les applications natives du cloud ?

Proposition d'architecture pour l'observabilité dans les applications cloud natives

Notre première contribution sur l'observabilité nous a permis de gagner en visibilité et de comprendre ce qui s'est passé à l'intérieur et autour des applications et infrastructures cloud natives. Au moment de la rédaction de cet article, le concept d'observabilité et sa différenciation par rapport à la surveillance traditionnelle sont de plus en plus connus et approuvés dans l'industrie informatique. Nous avons contribué à définir les exigences, les caractéristiques pour atteindre l'observabilité dans un environnement de cloud computing moderne, telles que les applications natives du cloud hébergées dans le cloud public.

Nous avons défini des stratégies 5 et des méthodes qui peuvent être réunies et construire un cadre d'architecture qui peut atteindre l'observabilité sur un environnement cloud natif. Ce cadre a ensuite été déployé dans un environnement de production commerciale pour améliorer la capacité des ingénieurs à diagnostiquer les pannes et les pannes tout en améliorant la qualité de service. Ce cadre d'observabilité a facilité le travail et a aidé à la fois les développeurs et les ingénieurs d'exploitation dans leurs missions. Ce travail a été publié et démontré à IEEE IM 2019. [5]

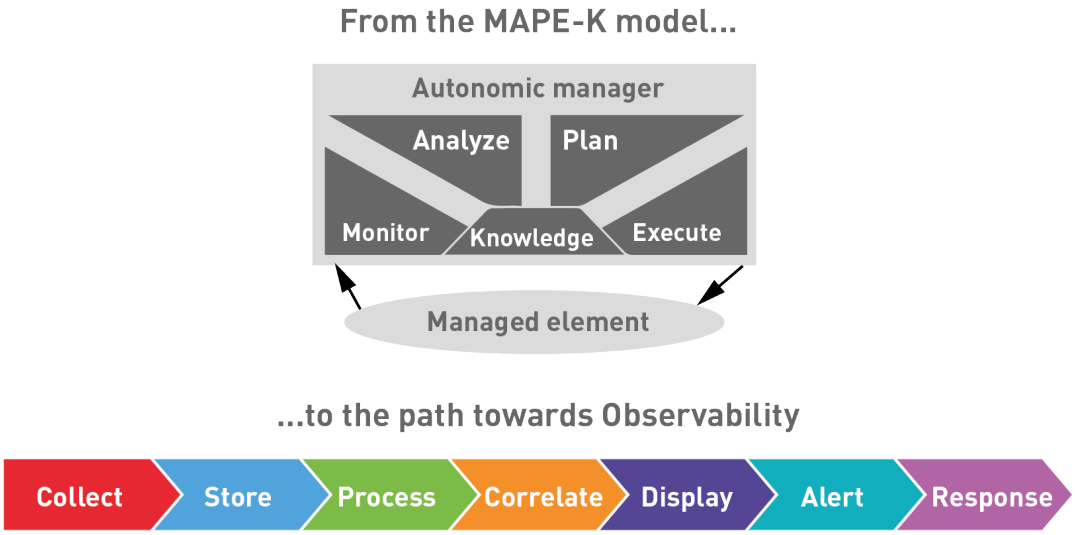


FIGURE 5: Path towards observability.

Auto-Scaling (Mise à l'échelle) automatique axée sur l'observabilité pour les applications cloud natives

Notre deuxième contribution s'est concentrée sur l'auto-scaling, car notre état de l'art et notre exploration sur le terrain ont montré un besoin de plus de recherche dans ce domaine. Si de nombreux travaux ont été menés sur le sujet, la plupart d'entre eux se sont concentrés sur les machines virtuelles, les métriques de base et l'architecture précédente. Notre recherche était basée sur l'utilisation de nos données d'observabilité pour piloter notre auto-scaling, en suivant notre voie basée sur l'informatique autonome. Nous avons d'abord décrit une architecture capable d'utiliser notre cadre d'observabilité comme entrée pour un orchestrateur de production et un auto-scaler. Nous choisissons les métriques d'utilisation réelles du bus de communication entre les microservices comme entrée pour l'auto-scaler. Cela nous a permis d'évoluer efficacement en dehors de la situation de saturation qui peut survenir avec la mise à l'échelle automatique basée sur l'utilisation du processeur. Cette solution a également été portée sur un environnement de production et permet désormais à Lectra d'atteindre une QoS satisfaisante dans les microservices équipés. Cette partie a également été démontrée à l'IM 2019.

Nous avons décidé de pousser plus loin la mise à l'échelle automatique basée sur l'observabilité avec la mise à l'échelle automatique proactive pour les applications natives du cloud. Nous avons utilisé un apprentissage automatique (Machine Learning) et un maillage de services (Service Mesh) de pointe au sein d'un banc d'essai dédié, libéré des contraintes de Lectra. Cela nous a permis de réaliser une preuve de concept où les séries temporelles prévues par l'apprentissage automatique pourraient être utilisées pour détecter et prédire une augmentation soudaine du trafic, à savoir une foule flash et auto-scaling proactive pour éviter une situation de saturation pendant la période d'auto-scaling. . Ce travail a fait l'objet d'une publication à

Globecom 2020. [3]

Un cadre d'architecture pour la virtualisation des applications IoT : natif du cloud et observabilité dans l'IoT

Notre troisième contribution s'est concentrée sur l'application des innovations et des technologies disponibles à partir des applications natives du cloud à l'Internet des objets. Les objets IoT sont désormais plus puissants que jamais et capables d'héberger des conteneurs. La conteneurisation aux côtés de la virtualisation, la radio définie par logiciel (SDR) et la mise en réseau définie par logiciel (SDN) ouvrent la voie à une architecture d'objets IoT à usage général connectée au cloud et capable de communiquer avec des capteurs. De plus, le SDR et la radio programmable sont des catalyseurs essentiels pour les appareils IoT. Ces technologies pourraient créer des appareils IoT cloud accessibles comme toute autre ressource cloud. Ce travail a été publié à NETSOFT 2019. [4]

Conclusion

Nous pensons que notre travail a contribué à améliorer la façon dont les applications cloud natives peuvent être exploitées. L'informatique autonome et l'orchestration avancée sont d'une importance primordiale pour les ingénieurs d'exploitation et les développeurs dans leur mission de fournir des applications cloud natives meilleures, plus rapides et plus fiables. Devenu un sujet très branché dans l'industrie suivi par tous les grands acteurs, l'observabilité est un domaine qui nécessite plus de recherche pour définir et caractériser clairement le sujet. En 2021, le CNCF a annoncé un nouveau projet autour de la mise à l'échelle automatique basée sur l'observabilité et a envisagé de le porter aux masses [136].

Bien qu'ils ne soient pas axés sur la sécurité, notre travail englobe également la possibilité de reprise après sinistre, la résilience aux cyberattaques. En effet, les systèmes d'observabilité permettent une protection et une surveillance plus faciles contre les menaces. L'automatisation et l'orchestration en utilisant une approche « cattle vs pets » permettent aux systèmes d'être récupérés et restaurés en moins de temps en cas d'intrusion, de dégradation et d'autres catastrophes telles que les pannes de centre de données. Bien qu'elle ne soit pas la plus rentable, l'auto-scaling augmente la résistance du système aux attaques DDoS et réduit les retombées sur les utilisateurs légitimes.

Nous pensons que dans un environnement de production, la mise à l'échelle automatique activée par l'apprentissage automatique (Machine Learning) pourrait alimenter des cas d'utilisation tels que retarder la mise à l'échelle

automatique réduite à partir de zéro et ainsi réduire les coûts, la consommation d'énergie et l'empreinte carbone en ne gardant pas certains microservices en configuration allumé en permanence. Alors que davantage d'objets et de capteurs IoT doivent être connectés à Internet et avec l'avènement des technologies de pointe et de brouillard (fog), SDR et NFV pourraient permettre des passerelles reconfigurables agiles en fréquence pour de multiples cas d'utilisation.

Bibliography

[1] D. Sinreich. "An architectural blueprint for autonomic computing". en. In: *undefined* (2006). URL: https://www.semanticscholar.org/paper/An-architectural-blueprint-for-autonomic-computing-Sinreich/47c37d43f43e2be57f6f2bc668979f784911e953.

[2] Eric Rutten, Nicolas Marchand, and Daniel Simon. "Feedback Control as MAPE-K Loop in Autonomic Computing". In: *Software Engineering for Self-Adaptive Systems III. Assurances*. Ed. by Rogério de Lemos, David Garlan, Carlo Ghezzi, and Holger Giese. Cham: Springer International Publishing, 2017, pp. 349–373. ISBN: 978-3-319-74183-3.

[3] Nicolas Marie-Magdelaine and Toufik Ahmed. "Proactive Autoscaling for Cloud-Native Applications using Machine Learning". en. In: *GLOBECOM 2020 - 2020 IEEE Global Communications Conference*. Taipei, Taiwan: IEEE, Dec. 2020, pp. 1–7. ISBN: 978-1-72818-298-8. DOI: 10.1109/GLOBECOM42002.2020.9322147. URL: https://ieeexplore.ieee.org/document/9322147/.

[4] Toufik Ahmed, Abdelhamid Alleg, and Nicolas Marie-Magdelaine. "An Architecture Framework for Virtualization of IoT Network". en. In: *2019 IEEE Conference on Network Softwarization (NetSoft)*. Paris, France: IEEE, June 2019, pp. 183–187. ISBN: 978-1-5386-9376-6. DOI: 10.1109/NETSOFT.2019.8806650. URL: https://ieeexplore.ieee.org/document/8806650/.

[5] N. Marie-Magdelaine, T. Ahmed, and G. Astruc-Amato. "Demonstration of an Observability Framework for Cloud Native Microservices". In: *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. Apr. 2019, pp. 722–724.

[6] Dave Henry, Sandra Cooke, and Sabrina Montes. "THE EMERGING DIGITAL ECONOMY". en. In: (), p. 259.

[7] *Cloud Adoption Statistics in 2021*. en-US. URL: https://hostingtribunal.com/blog/cloud-adoption-statistics/.

[8] *DEMO*. URL: https://web.archive.org/web/20180215063848/http://www.demo.com/ehome/index.php?eventid=29414&.

[9] *AMD64 Virtualization Codenamed "Pacifica" Technology*. URL: https://web.archive.org/web/20120305061511/http://www.mimuw.edu.pl/~vincent/lecture6/sources/amd-pacifica-specification.pdf.

[10] *Intel® VT Intel® Virtualization Technology (VT) in Converged Application Platforms*. en. URL: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/virtualization-tech-converged-application-platforms-paper.pdf.

[11] Peter Mell and Timothy Grance. "The NIST Definition of Cloud Computing". en. In: (), p. 7.
[12] *Linux Foundation*. en-US. URL: https://www.linuxfoundation.org/.
[13] *CNCF Cloud Native Definition v1.0*. original-date: 2015-12-07T19:29:57Z. URL: https://github.com/cncf/toc/blob/c0bee0289989c3dc4b5105755859d6564d02DEFINITION.md.
[14] *CNCF Annual Report 2020*. en-US. URL: https://www.cncf.io/cncf-annual-report-2020/.
[15] *Modern Data Centers Battle Exponential Data Growth*. en. Jan. 2018. URL: https://www.datacenterknowledge.com/industry-perspectives/modern-data-centers-battle-exponential-data-growth.
[16] Nicola Jones. "How to stop data centres from gobbling up the world's electricity". en. In: *Nature* 561.7722 (Sept. 2018). Bandiera_abtest: a Cg_type: News Feature Number: 7722 Publisher: Nature Publishing Group Subject_term: Energy, Engineering, Research data, Computer science, pp. 163–166. DOI: 10.1038/d41586-018-06610-y. URL: https://www.nature.com/articles/d41586-018-06610-y.
[17] *Chip Industry Sees Danger of AI in Explosion of Electricity Use - Bloomberg*. URL: https://www.bloomberg.com/news/articles/2020-07-21/chip-industry-sees-danger-of-ai-in-explosion-of-electricity-use.
[18] *Facebook Launches Open Compute Project*. en-US. Apr. 2011. URL: https://about.fb.com/news/2011/04/facebook-launches-open-compute-project/.
[19] *Open Compute Project*. en. URL: https://www.opencompute.org/about.
[20] *Efficacité – Centres de données – Google*. fr. URL: https://www.google.com/intl/fr/about/datacenters/efficiency/.
[21] Eric Masanet, Arman Shehabi, Jiaqi Liang, Lavanya Ramakrishnan, XiaoHui Ma, Valerie Hendrix, Benjamin Walker, and Pradeep Mantha. "The Energy Efficiency Potential of Cloud-Based Software: A U.S. Case Study". In: (June 2013). DOI: 10.2172/1171159. URL: https://www.osti.gov/biblio/1171159.
[22] *Why Amazon (AMZN), Google (GOOGL), Microsoft (MSFT) Are Designing Own Chips - Bloomberg*. URL: https://www.bloomberg.com/news/articles/2021-03-17/why-amazon-amzn-google-googl-microsoft-msft-are-designing-own-chips.
[23] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. "Azure Accelerated Networking: SmartNICs in the Public Cloud". In: *Proceedings*

of the 15th USENIX Conference on Networked Systems Design and Implementation. NSDI'18. USA: USENIX Association, 2018, 51–64. ISBN: 9781931971430.

[24] *Hyper-V*. en. Page Version ID: 1038147745. URL: https://en.wikipedia.org/w/index.php?title=Hyper-V&oldid=1038147745.

[25] *Kernel-based Virtual Machine*. en. Page Version ID: 1024533628. URL: https://en.wikipedia.org/w/index.php?title=Kernel-based_Virtual_Machine&oldid=1024533628.

[26] *VMware ESXi*. en. Page Version ID: 1043508143. URL: https://en.wikipedia.org/w/index.php?title=VMware_ESXi&oldid=1043508143.

[27] *Xen*. URL: https://en.wikipedia.org/wiki/Xen#XENSERVER.

[28] *DigitalOcean*. en. URL: https://www.digitalocean.com/.

[29] *DigitalOcean GitHub*. en. URL: https://github.com/digitalocean.

[30] Roberto Riggio, Abbas Bradai, Tinku Rasheed, Julius Schulz-Zander, Slawomir Kuklinski, and Toufik Ahmed. "Virtual network functions orchestration in wireless networks". In: *2015 11th International Conference on Network and Service Management (CNSM)*. 2015, pp. 108–116. DOI: 10.1109/CNSM.2015.7367346.

[31] Subhasree Mandal. "Experience with B4: Google's Private SDN Backbone". In: Santa Clara, CA: USENIX Association, July 2015.

[32] *Cisco*. fr. URL: https://www.cisco.com/c/fr_fr/index.html.

[33] *Ubiquiti*. en. URL: https://www.ui.com/.

[34] *Aruba*. fr. URL: //www.arubanetworks.com/fr/.

[35] Jörn Kuhlenkamp, Sebastian Werner, and S. Tai. "The Ifs and Buts of Less is More: A Serverless Computing Reality Check". In: *2020 IEEE International Conference on Cloud Engineering (IC2E)* (2020), pp. 154–161.

[36] S. Newman. *Building Microservices*. O'Reilly Media, 2021. ISBN: 978-1-4920-3397-4. URL: https://books.google.fr/books?id=ZvM5EAAAQBAJ.

[37] A. Davis. *Bootstrapping Microservices with Docker, Kubernetes, and Terraform: A project-based guide*. Manning Publications, 2021. ISBN: 978-1-61729-721-2. URL: https://books.google.fr/books?id=QKQbEAAAQBAJ.

[38] R. Mitra and I. Nadareishvili. *Microservices: Up and Running*. O'Reilly Media, 2020. ISBN: 978-1-4920-7540-0. URL: https://books.google.fr/books?id=Gj0LEAAAQBAJ.

[39] S. Newman. *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O'Reilly Media, 2019. ISBN: 978-1-4920-4779-7. URL: https://books.google.fr/books?id=ota_DwAAQBAJ.

[40] *Google - Site Reliability Engineering*. URL: https://sre.google/.

[41] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. "Why Does the Cloud Stop Computing? Lessons from Hundreds of Service Outages". In: *Proceedings of the Seventh ACM Symposium on Cloud Computing*. SoCC '16. New York, NY, USA: Association for Computing Machinery, 2016, 1–16. ISBN: 9781450345255. DOI: 10.1145/2987550.2987583. URL: https://doi.org/10.1145/2987550.2987583.

[42] *Downdetector (@downdetector) / Twitter*. fr. URL: https://twitter.com/downdetector.

[43] Amazon Web Services. *Summary of the Amazon S3 Service Disruption in the Northern Virginia (US-EAST-1) Region*. URL: https://aws.amazon.com/message/41926/.

[44] Catalin Cimpanu. *Hacker who launched DDoS attacks on Sony, EA, and Steam gets 27 months in prison*. en. URL: https://www.zdnet.com/article/hacker-who-launched-ddos-attacks-on-sony-ea-and-steam-gets-27-months-in-prison/.

[45] Colin Bartlett. *5 Biggest Outages of Q2 2020*. en-US. Aug. 2020. URL: https://statusgator.com/blog/2020/08/21/5-biggest-outages-of-q2-2020/.

[46] Giuseppe Aceto, Alessio Botta, Walter de Donato, and Antonio Pescapè. "Cloud monitoring: A survey". In: *Computer Networks* 57.9 (2013), pp. 2093–2115. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2013.04.001. URL: https://www.sciencedirect.com/science/article/pii/S1389128613001084.

[47] Jonathan Stuart Ward and Adam Barker. "Observing the clouds: a survey and taxonomy of cloud monitoring". In: *Journal of Cloud Computing* 3.1 (2014), p. 24. ISSN: 2192-113X. DOI: 10.1186/s13677-014-0024-2. URL: https://doi.org/10.1186/s13677-014-0024-2.

[48] Kaniz Fatema, Vincent C. Emeakaroha, Philip D. Healy, John P. Morrison, and Theo Lynn. "A survey of Cloud monitoring tools: Taxonomy, capabilities and objectives". In: *Journal of Parallel and Distributed Computing* 74.10 (2014), pp. 2918–2933. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2014.06.007. URL: https://www.sciencedirect.com/science/article/pii/S0743731514001099.

[49] Hassan Jamil Syed, Abdullah Gani, Raja Wasim Ahmad, Muhammad Khurram Khan, and Abdelmuttlib Ibrahim Abdalla Ahmed. "Cloud monitoring: A review, taxonomy, and open research issues". In: *Journal of Network and Computer Applications* 98 (2017), pp. 11–26. ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2017.08.021. URL: https://www.sciencedirect.com/science/article/pii/S1084804517302783.

[50] *Microsoft Azure Pricing Calculator*. en. URL: https://azure.microsoft.com/en-us/pricing/calculator/.

[51] Google. *Bringing Pokémon GO to life on Google Cloud*. URL: https://cloudplatform.googleblog.com/2016/09/bringing-Pokemon-GO-to-life-on-Google-Cloud.html.

[52] Mary Jo Foley. *European users reporting they're hitting Azure capacity constraints*. en. URL: https://www.zdnet.com/article/european-users-reporting-theyre-hitting-azure-capacity-constraints/.

[53] Simon Ostermann, Alexandria Iosup, Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick Epema. "A Performance Analysis of EC2 Cloud Computing Services for Scientific Computing". In: *Cloud Computing*. Ed. by Dimiter R. Avresky, Michel Diaz, Arndt Bode, Bruno

Ciciani, and Eliezer Dekel. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 115–131. ISBN: 978-3-642-12636-9.

[54] Yaakoub El-Khamra, Hyunjoo Kim, Shantenu Jha, and Manish Parashar. "Exploring the Performance Fluctuations of HPC Workloads on Clouds". In: *2010 IEEE Second International Conference on Cloud Computing Technology and Science*. 2010, pp. 383–387. DOI: 10.1109/CloudCom.2010.84.

[55] *Preemptible Virtual Machines*. en. URL: https://cloud.google.com/preemptible-vms.

[56] *Introducing B-Series, our new burstable VM size | Blog Azure et mises à jour | Microsoft Azure*. fr. URL: https://azure.microsoft.com/fr-fr/blog/introducing-b-series-our-new-burstable-vm-size/.

[57] Chengwei Wang, Karsten Schwan, Vanish Talwar, Greg Eisenhauer, Liting Hu, and Matthew Wolf. "A Flexible Architecture Integrating Monitoring and Analytics for Managing Large-Scale Data Centers". In: *Proceedings of the 8th ACM International Conference on Autonomic Computing*. ICAC '11. New York, NY, USA: Association for Computing Machinery, 2011, 141–150. ISBN: 9781450306072. DOI: 10.1145/1998582.1998605. URL: https://doi.org/10.1145/1998582.1998605.

[58] Gregory Katsaros, Georgina Galizo, Roland Kübert, Tinghe Wang, J. Oriol Fitó, and Daniel Henriksson. "A MULTI-LEVEL ARCHITECTURE FOR COLLECTING AND MANAGING MONITORING INFORMATION IN CLOUD ENVIRONMENTS". In: *Proceedings of the 1st International Conference on Cloud Computing and Services Science - Volume 1: CLOSER*, INSTICC. SciTePress, 2011, pp. 232–239. ISBN: 978-989-8425-52-2. DOI: 10.5220/0003388602320239.

[59] Rizwan Mian, Patrick Martin, and Jose Luis Vazquez-Poletti. "Provisioning Data Analytic Workloads in a Cloud". In: *Future Gener. Comput. Syst.* 29.6 (Aug. 2013), 1452–1458. ISSN: 0167-739X. DOI: 10.1016/j.future.2012.01.008. URL: https://doi.org/10.1016/j.future.2012.01.008.

[60] Waheed Iqbal, Matthew N. Dailey, David Carrera, and Paul Janecek. "Adaptive Resource Provisioning for Read Intensive Multi-Tier Applications in the Cloud". In: *Future Gener. Comput. Syst.* 27.6 (June 2011), 871–879. ISSN: 0167-739X. DOI: 10.1016/j.future.2010.10.016. URL: https://doi.org/10.1016/j.future.2010.10.016.

[61] Vincent C Emeakaroha, Marco AS Netto, Rodrigo N Calheiros, Ivona Brandic, Rajkumar Buyya, and César AF De Rose. "Towards autonomic detection of SLA violations in Cloud infrastructures". In: *Future Generation Computer Systems* 28.7 (2012), pp. 1017–1029.

[62] Anas Ayad and Uwe Dippel. *Agent-based monitoring of virtual machines*. Vol. 1. Journal Abbreviation: Proceedings 2010 International Symposium on Information Technology - Visual Informatics, ITSim'10 Pages: 6 Publication Title: Proceedings 2010 International Symposium on Information Technology - Visual Informatics, ITSim'10. July 2010. DOI: 10.1109/ITSIM.2010.5561375.

[63] Spyridon V. Gogouvitis, Vassileios Alexandrou, Nikoletta Mavrogeorgi, Stefanos Koutsoutos, Dimosthenis Kyriazis, and Theodora Varvarigou. "A Monitoring Mechanism for Storage Clouds". In: *2012 Second International Conference on Cloud and Green Computing*. 2012, pp. 153–159. DOI: 10.1109/CGC.2012.26.

[64] Daniel Tovarnakk and Tomas Pitner. "Towards multi-tenant and interoperable monitoring of virtual machines in cloud". en. In: *2012 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. Timisoara: IEEE, Sept. 2012, pp. 436–442. ISBN: 978-1-4673-5026-6. DOI: 10.1109/SYNASC.2012.55. URL: https://ieeexplore.ieee.org/document/6481063/.

[65] Peer Hasselmeyer and Nico d'Heureuse. "Towards holistic multi-tenant monitoring for virtual data centers". In: *2010 IEEE/IFIP Network Operations and Management Symposium Workshops*. 2010, pp. 350–356. DOI: 10.1109/NOMSW.2010.5486528.

[66] Xu Cheng, Yuliang Shi, and Qingzhong Li. "A multi-tenant oriented performance monitoring, detecting and scheduling architecture based on SLA". In: *2009 Joint Conferences on Pervasive Computing (JCPC)*. 2009, pp. 599–604. DOI: 10.1109/JCPC.2009.5420114.

[67] *The Importance of Observability - Server Fault Blog*. URL: https://blog.serverfault.com/2013/11/26/the-importance-of-observability/.

[68] *Observability at Twitter*. URL: https://blog.twitter.com/engineering/en_us/a/2013/observability-at-twitter.html.

[69] *Observability at Twitter: technical overview, part I*. URL: https://blog.twitter.com/engineering/en_us/a/2016/observability-at-twitter-technical-overview-part-i.html.

[70] Winston Lee, Arun Kejariwal, and Bryce Yan. "Chiffchaff: Observability and analytics to achieve high availability". In: *2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*. Oct. 2013, pp. 119–120. DOI: 10.1109/LDAV.2013.6675168.

[71] Uber Engineering. *Observability at Uber Engineering: Past, Present, Future*. Mar. 2016. URL: https://www.youtube.com/watch?v=2JAnmzVwgP8.

[72] Juan Gutierrez-Aguado, Jose M Alcaraz Calero, and Wladimiro Diaz Villanueva. "IaaSMon: Monitoring Architecture for Public Cloud Computing Data Centers". en. In: *J Grid Computing* 14.2 (June 2016), pp. 283–297.

[73] *Production-Grade Container Orchestration*. en. URL: https://kubernetes.io/.

[74] *Configure Liveness, Readiness and Startup Probes*. en. Section: docs. URL: https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/.

[75] Recommended Links. *ITOM - Enterprise IT Operations Management - ServiceNow*. en. URL: https://www.servicenow.com/products/it-operations-management.html.

[76] *Auth0: Secure access for everyone. But not just anyone.* fr. URL: https://auth0.com/fr.

[77] *Sendgrid*. URL: https://sendgrid.com/.

[78] *AWS Amazon S3*. fr-FR. URL: https://aws.amazon.com/fr/s3/.
[79] *Cloud Pub/Sub*. URL: https://cloud.google.com/pubsub.
[80] Rand Hindi. *AWS S3 outages breaking IoT in connected homes*. en. URL: https://medium.com/snips-ai/thanks-for-breaking-our-connected-homes-amazon-c820a8849021.
[81] M M Alshammari, A A Alwan, A Nordin, and I F Al-Shaikhli. "Disaster recovery in single-cloud and multi-cloud environments: Issues and challenges". In: *2017 4th IEEE International Conference on Engineering Technologies and Applied Sciences (ICETAS)*. Nov. 2017, pp. 1–7.
[82] *Paul Horn directs IBM Research into autonomic computing development*. en. URL: https://www.computerworld.com/article/2796791/paul-horn-directs-ibm-research-into-autonomic-computing-development.html.
[83] markjbrown. *Request Units as a throughput and performance currency in Azure Cosmos DB*. en-us. URL: https://docs.microsoft.com/en-us/azure/cosmos-db/request-units.
[84] *Elasticsearch : Le moteur de recherche et d'analyse distribué officiel*. fr-fr. URL: https://www.elastic.co/fr/elasticsearch.
[85] *Splunk*. URL: https://www.splunk.com.
[86] *Google - Site Reliability Engineering*. URL: https://landing.google.com/sre/book/chapters/monitoring-distributed-systems.html.
[87] *Technology solutions for fashion, automotive and furniture | Lectra*. en. URL: https://www.lectra.com/en.
[88] *Is Now An Opportune Moment To Examine Lectra SA (EPA:LSS)?* URL: https://simplywall.st/stocks/fr/software/epa-lss/lectra-shares/news/is-now-an-opportune-moment-to-examine-lectra-sa-epalss.
[89] Prometheus. *Prometheus*. en. URL: https://prometheus.io/.
[90] *Grafana: The open observability platform*. en. URL: https://grafana.com/.
[91] Prometheus. *Alertmanager*. en. URL: https://prometheus.io/docs/alerting/latest/alertmanager/.
[92] *PagerDuty*. en. URL: https://www.pagerduty.com/.
[93] Atlassian. *Jira*. en. URL: https://www.atlassian.com/software/jira.
[94] *Catchpoint*. URL: https://www.catchpoint.com/.
[95] *StatusCake: Website Monitoring with Uptime Monitoring solution*. en-GB. URL: https://www.statuscake.com/.
[96] Parminder Singh, Pooja Gupta, Kiran Jyoti, and Anand Nayyar. "Research on Auto-Scaling of Web Applications in Cloud: Survey, Trends and Future Directions". en. In: *Scalable Computing: Practice and Experience* 20.2 (May 2019), pp. 399–432. ISSN: 1895-1767. DOI: 10.12694/scpe.v20i2.1537. URL: https://www.scpe.org/index.php/scpe/article/view/1537.
[97] Chenhao Qu, Rodrigo N. Calheiros, and Rajkumar Buyya. "Auto-Scaling Web Applications in Clouds: A Taxonomy and Survey". en. In: *ACM Computing Surveys* 51.4 (July 2018), pp. 1–33. ISSN: 03600300. DOI: 10.

1145/3148149. URL: http://dl.acm.org/citation.cfm?doid=3236632.3148149.

[98] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal. "Dynamic Provisioning of Multi-tier Internet Applications". en. In: *Second International Conference on Autonomic Computing (ICAC'05)*. Seattle, WA, USA: IEEE, 2005, pp. 217–228. DOI: 10.1109/ICAC.2005.27. URL: http://ieeexplore.ieee.org/document/1498066/.

[99] Harold C Lim, Shivnath Babu, Jeffrey S Chase, and Sujay S Parekh. "Automated control in cloud computing: challenges and opportunities". en. In: (2009), p. 6.

[100] Ming Mao and Marty Humphrey. "Auto-scaling to minimize cost and meet application deadlines in cloud workflows". In: *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ISSN: 2167-4337. Nov. 2011, pp. 1–12.

[101] *The Twelve-Factor App*. URL: https://12factor.net/.

[102] Pini Reznik, Jamie Dobson, and Michelle Gienow. *Cloud native transformation: practical patterns for innovation*. English. OCLC: 1099533905. 2019. ISBN: 978-1-4920-4890-9.

[103] Cindy Sridharan. *Distributed Systems Observability*. Undetermined. OCLC: 1103253397. Place of publication not identified: O'Reilly Media, Inc., 2018. ISBN: 978-1-4920-3342-4. URL: https://www.safaribooksonline.com/library/view/title/9781492033431/?ar?orpq&email=^u.

[104] Russ Miles. *Chaos engineering observability: bringing chaos experiments into system observability*. English. OCLC: 1102269482. 2019. URL: http://proquest.safaribooksonline.com/?fpi=9781492051046.

[105] R. Picoreti, A. Pereira do Carmo, F. Mendonça de Queiroz, A. Salles Garcia, R. Frizera Vassallo, and D. Simeonidou. "Multilevel Observability in Cloud Orchestration". In: *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech)*. Aug. 2018, pp. 776–784. DOI: 10.1109/DASC/PiCom/DataCom/CyberSciTec.2018.00134.

[106] Sima Siami-Namini, Neda Tavakoli, and Akbar Siami Namin. "A Comparison of ARIMA and LSTM in Forecasting Time Series". en. In: *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, Dec. 2018, pp. 1394–1401. ISBN: 978-1-5386-6805-4. DOI: 10.1109/ICMLA.2018.00227. URL: https://ieeexplore.ieee.org/document/8614252/.

[107] Klaus Greff, Rupesh K. Srivastava, Jan Koutník, Bas R. Steunebrink, and Jürgen Schmidhuber. "LSTM: A Search Space Odyssey". In: *IEEE Transactions on Neural Networks and Learning Systems* 28.10 (Oct. 2017). Conference Name: IEEE Transactions on Neural Networks and Learning Systems, pp. 2222–2232. ISSN: 2162-2388. DOI: 10.1109/TNNLS.2016.2582924.

[108] *Online Boutique*. original-date: 2018-08-03T18:32:18Z. URL: https://github.com/GoogleCloudPlatform/microservices-demo.

[109] *kube-state-metrics*. original-date: 2016-05-06T06:10:13Z. URL: https://github.com/kubernetes/kube-state-metrics.

[110] *Linkerd*. en. URL: https://linkerd.io.

[111] *Machine Learning Mastery*. en-US. URL: https://machinelearningmastery.com/.

[112] *Kaggle*. en. URL: https://www.kaggle.com/.

[113] *Locust*. URL: https://locust.io/.

[114] Q. Zhu, Ruicong Wang, Qi Chen, Y. Liu, and Weijun Qin. "IOT Gateway: BridgingWireless Sensor Networks into Internet of Things". In: *2010 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing* (2010), pp. 347–352.

[115] Ángel Leonardo Valdivieso Caraguay, Alberto Benito Peral, Lorena Isabel Barona López, and Luis Javier García Villalba. "SDN: Evolution and Opportunities in the Development IoT Applications". In: *International Journal of Distributed Sensor Networks* 10.5 (2014), p. 735142. DOI: 10.1155/2014/735142. eprint: https://doi.org/10.1155/2014/735142. URL: https://doi.org/10.1155/2014/735142.

[116] Stefan Nastic, Sanjin Sehic, Duc-Hung Le, Hong-Linh Truong, and Schahram Dustdar. "Provisioning Software-Defined IoT Cloud Systems". In: *Proceedings of the 2014 International Conference on Future Internet of Things and Cloud*. FICLOUD '14. USA: IEEE Computer Society, 2014, 288–295. ISBN: 9781479943579. DOI: 10.1109/FiCloud.2014.52. URL: https://doi.org/10.1109/FiCloud.2014.52.

[117] Pascal Thubert, Maria Rita Palattella, and Thomas Engel. "6TiSCH centralized scheduling: When SDN meet IoT". en. In: *2015 IEEE Conference on Standards for Communications and Networking (CSCN)*. Tokyo, Japan: IEEE, Oct. 2015, pp. 42–47. ISBN: 978-1-4799-8927-0 978-1-4799-8928-7. DOI: 10.1109/CSCN.2015.7390418. URL: http://ieeexplore.ieee.org/document/7390418/.

[118] Samaresh Bera, Sudip Misra, and Athanasios V. Vasilakos. "Software-Defined Networking for Internet of Things: A Survey". en. In: *IEEE Internet of Things Journal* 4.6 (Dec. 2017), pp. 1994–2008. ISSN: 2327-4662. DOI: 10.1109/JIOT.2017.2746186. URL: http://ieeexplore.ieee.org/document/8017556/.

[119] Alex Mavromatis, Aloizio Pereira Da Silva, Koteswararao Kondepu, Dimitrios Gkounis, Reza Nejabati, and Dimitra Simeonidou. "A Software Defined Device Provisioning Framework Facilitating Scalability in Internet of Things". In: *2018 IEEE 5G World Forum (5GWF)*. 2018, pp. 446–451. DOI: 10.1109/5GWF.2018.8516955.

[120] Nikos Bizanis and Fernando A. Kuipers. "SDN and Virtualization Solutions for the Internet of Things: A Survey". In: *IEEE Access* 4 (2016), pp. 5591–5606. DOI: 10.1109/ACCESS.2016.2607786.

[121] Nathalie Omnes, Marc Bouillon, Gael Fromentoux, and Olivier Le Grand. "A programmable and virtualized network amp; IT infrastructure for the internet of things: How can NFV amp; SDN help for

facing the upcoming challenges". In: *2015 18th International Conference on Intelligence in Next Generation Networks*. 2015, pp. 64–69. DOI: 10.1109/ICIN.2015.7073808.

[122] Riad Kouah, Abdelhamid ALLEG, Abir Laraba, and Toufik Ahmed. "Energy-Aware Placement for IoT-Service Function Chain". In: *2018 IEEE 23rd International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*. Barcelona, France: IEEE, Sept. 2018, pp. 1–7. URL: https://hal.archives-ouvertes.fr/hal-01990850.

[123] OpenSystems Media. *The shift to standards-based hardware for military communications: What role will COTS systems play? - Military Embedded Systems*. en. URL: http://militaryembedded.com/comms/communications/the-cots-systems-play.

[124] *Wearables Operating Systems and Platforms*. en. URL: https://www.abiresearch.com/market-research/product/1030999-wearables-operating-systems-and-platforms/.

[125] Ultan Mulligan. *ETSI - NFV*. en-gb. URL: https://www.etsi.org/technologies/nfv.

[126] Ameer Ahmed Abbasi and Mohamed Younis. "A Survey on Clustering Algorithms for Wireless Sensor Networks". In: *Comput. Commun.* 30.14–15 (Oct. 2007), 2826–2841. ISSN: 0140-3664. DOI: 10.1016/j.comcom.2007.05.024. URL: https://doi.org/10.1016/j.comcom.2007.05.024.

[127] Olutayo Boyinbode, Hanh Le, Audrey Mbogho, Makoto Takizawa, and Ravi Poliah. "A Survey on Clustering Algorithms for Wireless Sensor Networks". In: *2010 13th International Conference on Network-Based Information Systems*. 2010, pp. 358–364. DOI: 10.1109/NBiS.2010.59.

[128] Oualid Demigha, Walid-Khaled Hidouci, and Toufik Ahmed. "On Energy Efficiency in Collaborative Target Tracking in Wireless Sensor Network: A Review". In: *IEEE Communications Surveys Tutorials* 15.3 (2013), pp. 1210–1222. DOI: 10.1109/SURV.2012.042512.00030.

[129] *50 of the most important Raspberry Pi Sensors and Components*. en-US. Jan. 2016. URL: https://tutorials-raspberrypi.com/raspberry-pi-sensors-overview-50-important-components/.

[130] *SoapySDR*. en. URL: https://github.com/pothosware/SoapySDR.

[131] *CubicSDR*. en-CA. URL: https://cubicsdr.com/.

[132] *Gqrx SDR*. en-US. URL: https://gqrx.dk/.

[133] *GNU Radio*. URL: https://www.gnuradio.org/.

[134] *rpitx*. original-date: 2015-10-21T16:06:52Z. URL: https://github.com/F5OEO/rpitx.

[135] *Docker*. en. URL: https://www.docker.com/.

[136] *KEDA*. en-us. URL: https://keda.sh/ (visited on 09/24/2021).