University of Padova, Italy

Doctoral Thesis

Service Design in the Cloud

Author:
Kiyana Bahadori

Supervisor:
Prof. Tullio Vardanega

Coordinator:
Prof. Giuseppe Sartori

A thesis submitted in fulfillment of the requirements
for the degree of Doctor of Philosophy

in the

Brain, Mind, Computer Science (BMSC)
Department of Mathematic
Series: XXXI

May 29, 2019

Abstract

In computer science, the notion of service is a broad term that incorporates any IT service delivered and accessible globally from the Internet. With the arrival of social networks and the pervasive entry of IT into business infrastructures, the population of (web-based) Internet service has grown large enough to turn the need, acquisition, delivery, maintenance and upgrade of computing resources into real strategic challenges. Along with the technology evolution and the growth experienced on the available information, the resource demand has been increasing in both science and engineering marketplaces. To satisfying the growth in resource demand, there have been attempting to enhance the resource-efficiency of delivered service to users, i.e., the effective utility from the available resources, before investing in additional compute resources, in an attempt to maximize their business competitiveness and return on investment of their infrastructures. Accordingly, one of the significant features that make cloud computing an attractive service-oriented architecture for competitive business is: elasticity, i.e., the ability to scale up/down the resources on-demand together with pay-per-use pricing model which eliminates the need for massive upfronts investment in local infrastructure, a model that was not economically feasible in the era of traditional enterprise infrastructures.Therefore, the work in this thesis had the focus on improving performance and resource utilization of service in Cloud environment.

Acknowledgements

It has been a period of intense learning for me, not only in the scientific arena but also on a personal level. Writing this dissertation has had a significant impact on me. During this almost 4 years, I have met many people who have helped me with technical discussions and a good relationship with friendship. that is why, I would like to reflect on the people who have supported and helped me so much throughout this period.

First and foremost, praises and thanks to the God, the Almighty, for His showers of blessings throughout my research work. I would like to thanks my research supervisor, Prof. Tullio Vardanega for providing funding and support.

Words can not express how grateful I am to my parents for their love, prayers, and sacrifices for educating and preparing me for my future.

I was very lucky to interact with brilliant people at the Barcelona Supercomputing Center(BSC). I would like to express my sincere gratitude to David Carrera Perez, and Josep Lluís Berral Garcia who provided me an opportunity to join their team, and for their excellent collaboration. Working on a topic, which you are passionate about, is one of the most important aspects for doctoral students. My year in Barcelona was improved by the many wonderful friends I met here. Thanks to all members of BSC (Alberto Gutierrez, Felipe, Nicola Cadenelli, Shuja-ur-Rehman Baig, David Buchaca, Aaron Call) for everything I learned from them.

My heart felts regard goes to my uncle (Fab) and his supportive wife (Anna), who provide unending inspiration and support for me. I would like to thank my best friends Maryam and Ruben for their sympathetic ear and moral support.

Finally, my thanks go to all people who have supported me to complete the research work directly or indirectly.

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

Along with the technology evolution and the growth experienced on the available information, the resource demand has been increasing in both science and engineering marketplaces. To satisfying the growth in resource demand, there have been attempting to enhance the resource-efficiency, i.e., the effective utility from the available resources, before investing in additional compute resources, in an attempt to maximize their business competitiveness and return on investment of their infrastructures.

In the past few years, the "pay-as-you-go" Cloud Computing model with its vision of offering "everything as a service" has become a dominating paradigm in the externalisation of IT resources and efficient alternative to owning and managing private data centres due to its scalability paradigm for enterprises starting, since the start of 2000s. The design of Cloud service delivery takes advantage of virtualization techniques as a key technology to improve resource-efficiency. Virtualization provides a systematic abstraction of hardware, system resources, and operating systems, to convert dedicated physical resources into virtual shared resources and simplify the complexity of management and deployment of applications. Virtualization can enable efficient collocation of workloads due to its isolation capabilities, and besides that, provides a confined environment that has specific software requirements including OS version dependencies and libraries from the underlying physical infrastructure where an application can run, dynamically adapt to the workload while being economically profitable.

Along with virtualization trend that has been massively contributing to provide resource efficiency, the emergence of parallel and distributed systems, a great effort has been put into making applications benefit efficiently from multiple computing resources.

Recently, the concept of the Internet of Things (IoT) and its required service architecture and computing paradigm has drawn significant interest from both academia and industry enhancing resource efficiency to which avoid unnecessary north(Cloud)-south(smart device/ things) bound communication of data that can be processed on the edge or intermediate nodes. IoT due to its potential applications oriented towards Smart Cities and health-care services urge experts on designing new architectures for infrastructure, platforms and services while leveraging near-data computing resource (IoT scenarios usually require low latency between sensors/actuators and computing power).

The work in this thesis had the focus on improving performance and resource utilization of services in Cloud environment. But, another major challenge that typically discourages users to use a cloud-like environment is the security concerns. As the resource capacity of near-edge IoT devices is naturally insufficient to cope with the influx of sensed data and its processing requirements, integration of the IoT system with cloud services allows opportune offloading of excess work. In several domains, the data must be protected before being offloaded to the cloud. However, for the sake of privacy, the data must be protected before being offloaded to the cloud. In this work, we also describe a security challenge concerning resource limitations of IoT devices and presents light-weight security schema for IoT ecosystem.

1.1 Motivation Challenges

In this thesis, my research objective was to get the most value out of the amalgamation of (1) Cloud service delivery means, the important steps forward being made in (2) the software architecture concept, with containers, microservices, in (3) the development process revolving around the DevOps agility, and (4) the deployment options in IoT ecosystems allowed by the increasing heterogeneity of hardware architectures to embrace parallel computing including accelerators such as Graphics Processing Units (GPUs) vs CPU trade off, that directly influences the scaling of computation performance and cost.

To this end, the combination of our research work presented in this dissertation explores design and implantation aspect of elastic scalable service(C1), proposed effective solutions to improve elastic service delivery across container orchestration platform(C2), and proposed distributed machine learning technique in IoT ecosysyem which enhance resource efficiency (C3), finally describe a security challenge concerning resource limitations of IoT devices and presents light-weight security schema for IoT ecosystem (C4) in the following evolving Cloud environment.

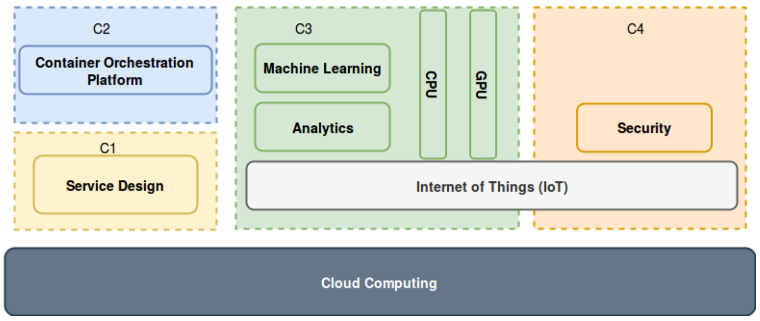


FIGURE 1.1: Summary of contributions.

1.2 Contributions

Provided the presented context, our contribution in this PhD work explored the design and implementation principles that warrant rapid elastic scalability to software services deployed on the Cloud and Fog computing paradigms. Our research work in this project has consisted of the following steps:

The focus of this thesis is to improve service scalability, by increasing resource efficiency on virtualized environments within design and implementation technology. To archive that we:

- We drew a trajectory that, starting from a better understanding of the principal service design features, related them to the microservice architectural style and its implications on elastic scalability, most notably dynamic orchestration, and concludes reviewing how well state-of-the-art technology fares for their implementation. This contribution is corroborated by the following paper:
  - c1. Kiyana Bahadori, Tullio Vardanega, "Designing and Implementing Elastically Scalable Services - A State-of-the-art Technology Review". In proceedings of the 8th International Conference on Cloud Computing and Services Science, CLOSER 2018, Funchal, Madeira, Portugal, March 19-21, 2018.

- In the level of Platform-as-a-Service(PaaS) and on the abstraction level of container, we have presented and discussed our view of how the dynamic orchestration capabilities required for containerized microserviced-based applications responds to the DevOps demand for increased agility in the whole development cycle. We have argued that Machine Learning is a necessary ingredient to enable the realization of sound and effective automation rules. To prove our point we presented the results from an experimental implementation of an Elastic Container Service (ECS) hosted on AWS, augmented with our machine learning implements, to increase the agility in deployment of containerized application. We evaluated that prototype against a few experimental scenarios, which showed the lower latency achieved by our method for application deployment. This contribution resulted in the following paper:
  - c2. Kiyana Bahadori, Tullio Vardanega, "DevOps Meets Dynamic Orchestration", In proceeding of the First international workshop on software engineering aspects of continuous development and new paradigms of software production and deployment, DevOps 2018, Toulouse, France,March 5-6, 2018. (In print).
- We have extended our interest to a specific innovative distributed application for a traffic modeling and prediction service, designed scenario, drawn from a Smart City context. We present a distributed machine learning approach for road traffic modeling and prediction, designed for a city-wide scenario based on the Fog computing paradigm. We display an architecture for large scale Floating Car Data (FCD), data modeling on the Edge, leveraging parallelism on deep learning algorithms in a city-wide scenario towards prediction. To validate this approach, we used real traffic logs from one week of Floating Car Data (FCD) in the city of Barcelona, provided by one of the largest road-assistance companies in Spain, comprising thousands of vehicles from their fleet only in the city of Barcelona.
  - c3. Kiyana Bahadori, Shuja-ur-Rehman Baig, Alberto Gutierrez-Torre, Waheed Iqbal, Josep Lluís Berral, David Carrera, Tullio Vardanega, "Towards Modeling Traffic Data on Edge Infrastructures using Distributed Deep Learning". The International Journal of eScience, Future Generation Computer Systems (FGCS)(submitted)
- The work in this thesis had the focus on improving performance and resource utilization of services in Cloud environment. But, another major challenge that typically discourages users to use a cloud-like environment is the security concerns. As the resource capacity of near-edge IoT devices is naturally insufficient to cope with the influx of sensed data and its processing requirements, integration of the IoT system with cloud services allows opportune offloading of excess work. In several domains, the data must be protected before being offloaded to the cloud. However, for the sake of privacy, the data must be protected before being offloaded to the cloud. In this work, we also describe a security challenge concerning resource limitations of IoT devices and presents light-weight security schema for IoT ecosystem.

- c4. Seyed Farhad Aghili, Kiyana Bahadori, Hamid Mala, Tullio Vardanega, " A Lightweight Attribute-Based Encryption Scheme for IoT-Cloud Integrated Systems Using Dynamic Attributes". International Conference on Computer Communications and Networks (submitted)

Next, we further detail the five contributions of this thesis.

1.3 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 introduces some basic concepts about service and virtualization and their associated technologies; Chapter 3 presents design and implementing aspect of elastically, scalable service and review the state of the art technology; Chapter 4 presents the results from an experimental implementation of an Elastic Container Service (ECS) hosted on AWS; Chapter 5 presents and evaluate a distributed machine learning approach for road traffic modelling and prediction, designed for a city-wide scenario based on the Fog computing paradigm; Chapter 6 describe a security challenge concerning resource limitations of IoT devices and presents light-weight security schema for IoT ecosystem; Finally, Chapter 7 presents the conclusions and the future work of this thesis.

Chapter 2

BACKGROUND

The fundamental concepts used during the elaboration of the other chapters are briefly introduced in this chapter and further described and discussed in the other chapters, along with this thesis. We first describe the notion of service and its related elastic scalability challenge to meet resource efficiency. Then, we detail the architecture and, other requirements, which plays a crucial role to meet this requirement.

2.1 The Notion of Service in the Cloud Computing

In computer science, the notion of service is a broad term that incorporates any IT service delivered and accessible globally from the Internet. With the arrival of social networks and the pervasive entry of IT into business infrastructures, the population of (web-based) Internet service has grown large enough to turn the need, acquisition, delivery, maintenance and upgrade of computing resources into real strategic challenges. Interestingly, the prospect of fast and affordable on-demand service delivery over the Internet proceeds from the very notion of Cloud computing [39, 6, 95, 56, 22]. The old central concept of Cloud in terms of **Computation as a utility** together with the vision of offering "everything as a service" has become a dominating paradigm in the externalization of IT resources for enterprises starting, since the start of the 2000s. Cloud computing has emerged as a successful and ubiquitous paradigm for service-oriented computing, where computing infrastructure and solutions are delivered as a service. In fact, Cloud by changing the relationship between business and infrastructure via offering higher-level of value to the businesses, set this trend to accelerate, resulting in the impressive growth of the infrastructure as a Service (IaaS) [139]. In 2017-2018, the use of the top Cloud service providers (Amazon, Microsoft Azure) had increased respectively from 60% to 63% and 19% to 25% [139].

Over the last years, the development of information technologies has increased the demands of resources. To satisfying this growth, there have been attempting to enhance the resource-efficiency of delivered services to the users, i.e., the effective utility from the available resources, before investing in additional compute resources, in an attempt to maximize their business competitiveness and return on investment of their infrastructures.

Accordingly, one of the significant features that make cloud computing an attractive service-oriented architecture for competitive business is: **elasticity** [63], i.e., the ability to scale up/down the resources on-demand together with **pay-per-use** pricing model which eliminates the need for massive upfronts investment in local infrastructure, a model that was not economically feasible in the era of traditional enterprise infrastructures. The recent advances in Cloud computing and the enterprise interest in exploiting its potential with its distinguishing traits provide a broad spectrum of capabilities that service providers could offer to meet compute service needs, and ultimately to deliver cost-effective end-user application usage experiences [142, 6, 22, 74].

At the big-picture level, the most critical characteristic in the service delivery model of Cloud is the inherent asymmetry in the economies of scale [56, 6]. It provides an organization with access to the near limitless availability of powerful computing resources in a consumption-based model that previously would be beyond their reach financially and operationally. From the service provider's perspective, cloud computing is an ideal way to leverage unused physical resources by "lending" them to consumers that are willing to pay for their use. However, the ability to afford those benefits to the user is contingent on attaining rapid elasticity in service design and implementation.

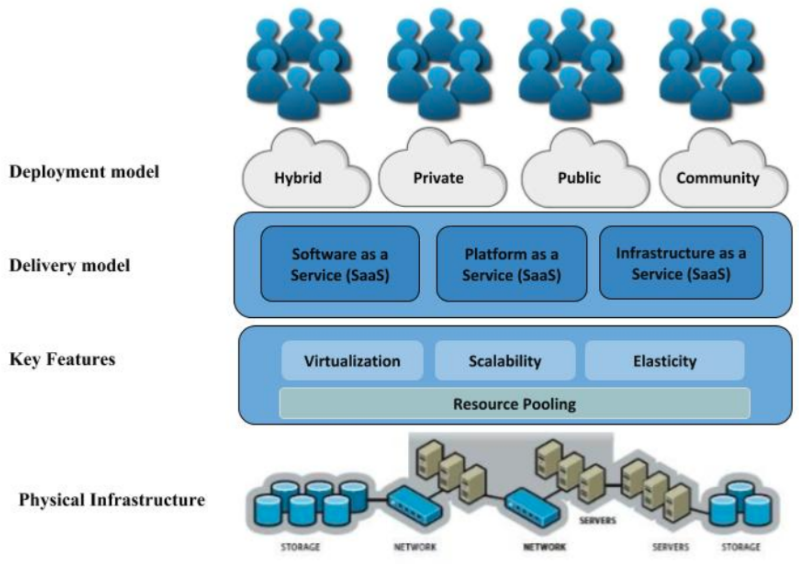


FIGURE 2.1: Visual model of Cloud Computing

Elasticity is defined as the ability to conserve resources at a fine grain with as fast as possible lead time allows matching resource to workload much more closely which result in cost savings for service providers [95, 63]. According to [58]

> "Elasticity is the degree to which a system is able to adapt to workload changes by provisioning and de-provisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible."

Speed and *precision* are the qualifying traits of elasticity [63]. As the number of requests for particular service increases (respectively, decreases), the speed of resource provisioning (respectively, releasing) should vary in accord with the speed of demand variation.

The demand of elasticity has come up with the need for autonomic systems capable of managing themselves to replace tedious and complex operations with human agents. Thanks to autonomic techniques, elasticity just does that. In a 2001 manifesto [73], IBM was the first to coin the conceptual architecture of autonomic computing in the form of an autonomic loop with different properties of autonomic system, each corresponding to a particular purpose. This model as it shown in Figure 2.2 details the different components that allow an autonomic manager to achieve dynamic scaling (elasticity), namely Monitor, Analyze, Plan, Execute and Knowledge (MAPE-K loop).

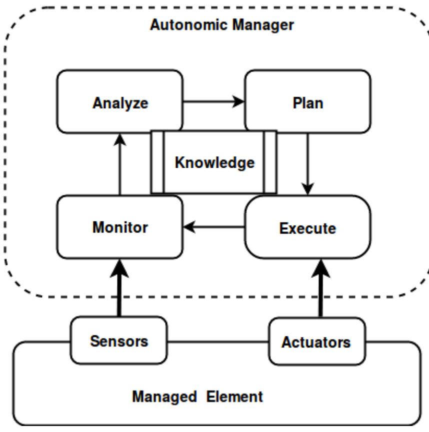


FIGURE 2.2: Autonomic system'control loop

2.1.1 Virtualization

The design of Cloud service delivery takes advantage of virtualization techniques to provide a confined environment from the underlying physical infrastructure where an application can run, dynamically adapt to the workload while being economically profitable. Virtualization is the key technology that utilizes resources by creating a virtual logical partition from a shared pool of resources to meet the principle of elastic scalability for service [56, 6]. To this end, hypervised virtualization has been a significant enabler for cloud computing. Its primary bounty includes isolation (which it assures) and self-sufficiency, that is, no external dependency and perfect loose coupling (which it allows for), both being much-desired qualities for service-oriented applications. In addition to, or perhaps as a reflection of, being too resource-costly, however, classic hypervised virtualization technology is fundamentally unable to respond to the rapidity requirements put forward by elastic horizontal scaling [103, 43].

Accordingly, an important step relevant to the trend in elastic service delivery is the shift of virtualization from the hardware level (virtual machines) to the operating system level (containers) as is depicted in Figure 2.3. In 2013, the Linux container technologies with critical features such as kernel namespaces, and Cgroups, created an ecosystem consisting of isolated layered container image format that revolutionizes the way software is being designed and deployed [103].

A container is composed of two parts: a pile of images and one container. Images are a collection of different read-only file systems stacked on top of each other using the Advanced multi-layered Unification File-system (AuFS), which implements union mounting that allows separate file systems or directories to be overlaid into a single coherent file system. A container is the writable image that sits on the top layer of the image stack [103, 75, 43].
The container uses the kernel of the host operating system to run multiple root file systems. The name "container" designates each such root file system [69] and holds the code and libraries that constitute the contained application while sharing the host operating system with all other entities that run on it. This sharing is a blessing but also a weakness; the latter because it requires the application to run on the host OS, renouncing interoperability. Containers use `namespaces` to cater for isolation among processes and `cgroups` to limit and control resources usage in an individual process. As it shown in Figure 2.3 the basic idea of containers is to package the entire application and its required dependencies into one isolated, lightweight, executable unit to run in virtualized environment [103].

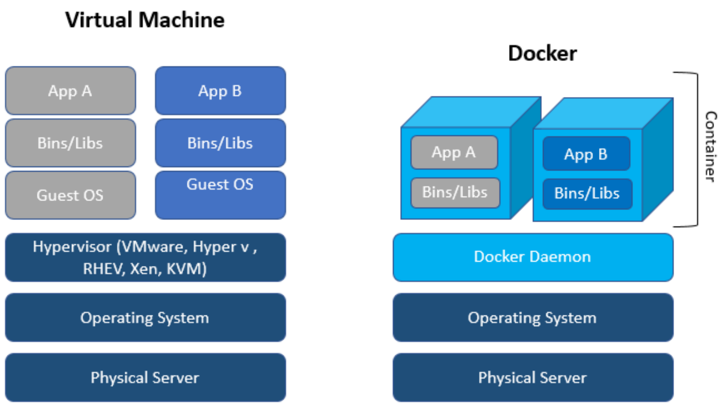


FIGURE 2.3: Virtualization

The resulting artifacts are lightweight, immutable image of software that reside on an already provisioned operating system, which therefore need only a few seconds to bootstrap [50]. Interestingly, allocating the required amount of resources needed for every individual container precisely is a great transitive facilitator to achieving more efficient usage of the underlying infrastructure. However, this practice also adds one level of abstraction (the container layer) to the problem of managing the deployment infrastructure of the application. Moreover, containerization achieves portability by isolating the application and its dependencies within self-contained units that are agnostic to programming languages and deployment platform [75]. Docker [69] is a de-facto standard and the most popular implementation of the container concept.

2.1.2 DevOps Methodology

The evolution gives the evidence of the progressive maturation in the interpretation of the Cloud Computing offering in the conceptualisation of software development and releases process across the SaaS, PaaS and IaaS layers of the Cloud reference architecture. Along with this maturation, the software element of the Cloud stack starts to take an increasingly prominent role in the realization of service-based innovation. Netflix [100] is an excellent example of a successful transition to an entirely new (software enabled) system architecture which required migrating existing data centre-based applications to the public Cloud (AWS) leveraging its horizontal scalability and agility needed to succeed.

Over the last few years, as innovation accelerates and customer needs rapidly evolve, a novel technology stack is sought that helps inject responses to the demand of accelerated agility and speed into the software development lifecycle. In this respect, the adoption of Cloud-aware agile development methodologies [40], now known as DevOps, is a fundamental shift in how leading-edge companies are starting to manage software and IT performance, where speed, agility, security and stability are business imperatives [76, 13, 52, 45, 52].

The DevOps concept rests on the adoption of automation solutions to create and manage tasks ("items" in the Agile TODO list) as dynamic assets in terms of *Infrastructure as Code* (IaC) [136, 98, 36]. Its high pressure on automation, extended from application development to the maintenance and operation infrastructure, fosters more in-depth attention to the performance of infrastructure management [45]. Therefore, the IaC paradigm is a natural fit for Cloud infrastructure where a resource can be provisioned and configured through APIs thus helping balance *time*, *resource*, and *quality* which are the critical assets in elastic service delivery in the Cloud environment [136].

Realizing software releases in terms of as-a-service offerings, which is natural for web and Cloud service development, naturally positions their providers as the early adopters of DevOps principles after the agility and speed that they promise [13]. In this context, DevOps embraces the operational aspects of defining a service as a product item that needs to be deployed, scaled, maintained, monitored and supported through its life cycle. Not surprisingly, therefore, players such as Amazon [25] provide a DevOps-focused way of creating and maintaining their infrastructure.

2.1.3 Cloud-native Application

As businesses grow, it becomes strategically necessary for providers to assure the ability to more users, in more locations, with a broader range of devices, while maintaining responsiveness, managing costs, and not falling over. In this context, the initial idea and concept of Cloud native computing with its service delivery model changed the relationship between businesses and infrastructure. Later, the pursuit of business growth changes the relationship; it moves forward to change the relationship between applications and infrastructure in terms of Cloud-native application.

Cloud-native application introduced a new way to design, implement resilience and scalable ecosystem around constraint inherent to Cloud environment [80, 79, 129, 123]. The Cloud Native Computing Foundation defines the core concepts for Cloud-native as a containerized unit of application that dynamically orchestrated while its architecture (microservice oriented) leverage horizontal scalability and utilize the advantages of the Cloud computing model [78, 21, 49].

The Cloud-native application development focused on how an application is built, delivered, and operated regardless of location in order to leverage the advantages of the Cloud computing [129, 79]. Taking advantage of Cloud services means using agile and scalable components like containers to deliver discrete and reusable features that integrate into well-described ways, even across technology boundaries like multicoloured, which allows delivery teams to iterate using repeatable automation and orchestration rapidly.

At a high level, Figure2.4 depicted the reference architecture for Cloud-native enterprise system using container-based technologies. On the subject of a reference

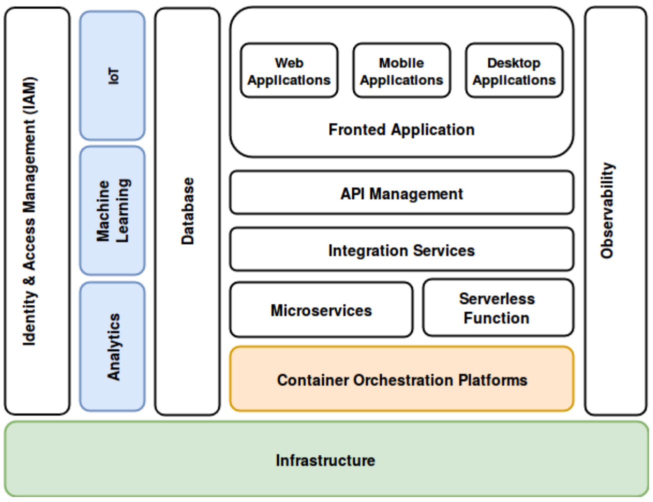


FIGURE 2.4: A reference architecture for Cloud-native enterprise system

architecture for Cloud-native architecture, we have highlighted our contributions of this thesis concerning the resource efficiency of service design in the Cloud environment.

Chapter 3

ELASTICALLY SCALABLE SERVICE

Cloud technology, with innovative ideas to host and on-demand delivery of service, creates an undeniable benefit of fast and affordable compute acceleration over the Internet. To this end, elastically scalable design and implementation for service are essential for today's modern service providers to the fulfillment of Service Level Agreements (SLAs) by dynamically re-allocating workloads and resources. In this chapter, in the quest for a service-based architecture that can guide service design, we argue for the adoption of the microservice architecture style. Subsequently, we look into state-of-the-art technology to assist the implementation of elastic scalability, concentrating on dynamic orchestration, which is one particular facet of that general quality.

3.1 Design and Implementation Priniciple

The first widespread use of the notion of service, concerning software systems, arose as part of the Service Oriented Architecture (SOA) initiative. In the context of SOA, the term *service* is defined as an independent logical unit, which provides functionality for a specific business process and can be composed with other services to form an application [48, 89]. Another connotation of service emerged in Cloud Computing as part of the X-as-a-service model, coined to evoke the manner of contract-based, pay-as-you-go utility delivering [6, 56]. The union of the two perspectives sets a most ambitious goal: building a software application by agile aggregation of service units in such a way that the exposed capabilities are realized with the least resource consumption and then delivered and consumed as metered utilities with assured quality of service.

Cloud computing has imposed itself as an attractive novel operational model for hosting and delivering IT services over the Internet [6, 56, 22]. Its central tenet of as-a-service provisioning, to the user (from the application perspective) and to the service provider (from the implementation perspective), meets the technology and economic requirements of today's acquisition pattern of IT infrastructure [6, 56].
In fact, the promised benefits of cost-effectiveness favoring pay-as-you-go pricing model, scalability [1, 138] and reliability, appeal major IT companies to meet their business objectives in cloud environment [5, 22]. However, from service provider point of view, the cost is outweighed by economic benefits of *elasticity* and transference of risk especially in an occurrence of over and under-provisioning of resource [56]. In that context, the goal of service providers is to design and implement service portfolios capable of satisfying given service level objectives (SLO), in the face of fluctuating user demand, while keeping operational costs at bay.

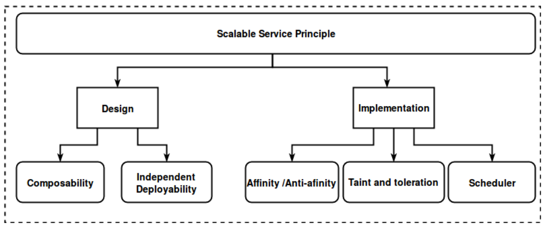


FIGURE 3.1: Scalable service principle

To this end, service design and implementation must both seek and provide for elastic scalability [63]. Figure 3.1 has shown the design and implementation principles required to meet this demand.

3.1.1 Scalable Service Design Principle

Fundamentally, service design is the activity of planning, and organizing a business's resource, implementing change to improve a service's quality and making it meet the user's and customer's needs for that service. It can be used to improve an existing service or to create a new service from scratch. The general principles of service design are to focus the designer's attention on generic requirements of all services. For example, the powering of an online shopping site rests on a multiplicity of services, ranging displaying, carting, payment processing, monitoring and shipping, which may either be procured from third-parties or provided by specialized development units inside the business organization.
Enterprise IT architectures have to leverage and improve the existing suite of applications to address changing the business requirements rapidly [76, 67]. Other than for silo-ed monoliths [113], the functionality provided by a single service is normally insufficient to respond alone to all user requests. In other words, a service normally needs to collaborate with other services to pursue a specific business goal, and the fabric of such collaboration may change with the goal. This collaborative trait requires services to warrant *composability* [120], i.e., the ability to participate, unchanged, in different aggregates as business demand requires [126, 110].

3.1.2 Composability

Service composability means the ability of new services from combining existing services. Following the mentioned example of the online shopping site, composability improves the ability of services to be composed into larger blocks, thereby enabling rapid (re)use of the functionality [120, 93]. Services need to be composable so that smaller services can be combined into more extensive services that provide a specific value. The principle of service-oriented architecture knows as service composability, as shown in Figure 3.2 can be broken up into five basic principles. In the following, we explain briefly all the five principles needed for service composability.

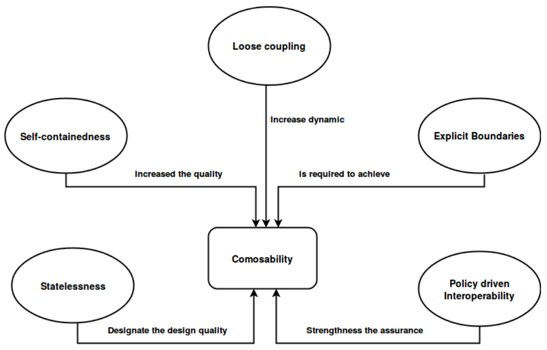


FIGURE 3.2: Service composability principle

- **Explicit boundaries**
  - A service possesses an interface specification that describes its functionality and exposes for the use of other services. That interface, separate from the corresponding implementation, forms the boundary of that service and contains all that one needs to know to interact with it. The boundary for service is defined through a contract [27]. A contract contains a schema and associated service policies (for a style of interaction, persistence, guarantee, etc.), and is published using different mechanisms, initially via WSDL [26], and later through REST API [90] and GRPC [137]. The schema defines the structure of the data message, agnostic to programming languages (hence intrinsically interoperable), needed to request a specific service functionality.
- **Policy-driven interoperability**
  - Interoperability is a concern with the ability of software systems, using different platforms and languages to communicate with each other. W3C's notion of web service was the first practical solution to assure network-based interoperability [33, 31]. Service policies provide a configurable set of semantic stipulations concerning service expectations or requirements expressed in a machine-readable language. For example, an online shopping service may require a security policy enforcing a specific service level (requiring an ID) and the users who do not comply are not allowed to continue. The security policy can be used with other related services such as shipping. Augmenting service interfaces with policy stipulations strengthens the assurance of service guarantees across technology boundaries. So, a rich policy-driven interoperable interface enables the movement of a service from one platform to another without changing its functionalities.
- **Loose coupling**
  - Flexibility in the system is achieved by making services loosely coupled and autonomous, which enables them to be composed without affecting the other parts of the system [48]. Loose coupling refers to the degree to which the functionalities of services depends upon each other. A well-defined service boundary goes hand in hand with the self-contained-ness of the functionality set exposed in its service interface, and therefore with

the degree of loose coupling and autonomy of its implementation. Loose coupling is when service does what your service interface declares (without incurring chains of dependencies) and does not place arbitrary constraints on your context of use (hence being fit for reuse in various compositions) [49]. Loose coupling is gained by limiting each service's knowledge about the implementation of other services through an explicit contract. Leveraging loose coupling also promotes reusability: organizations can adapt their applications to new market requirements over time.

- **Self-containedness**

  – A service interface is self-contained if its implementation can be independently managed (for instance, replicated or migrated) and versioned (so long as in a backwards-compatible manner) without affecting the rest of the system [48]. However, this is an ideal case which should be followed, and in some cases, it might not be possible to follow. For example, a service might provide new API versions or a different, backwards-incompatible interface or transport mechanism (e.g. a new API interface might not return all data that was returned by a previous version of the API due to privacy-related requirements that came in force after the publication of the first version of the API, or new security requirements mandate that TLSv1 transport should be outphased). In this case, the services that depend on this microservice must be adapted.

- **Statelessness**

  – The separation between service execution and state information enables replication, which is one of the essential dimensions of scalability [1]. Statelessness [48] is the name that designates this design quality. An important design principle for service composability is statelessness. A stateless service does not retain any information regarding the service itself or any previous request to it and lets the database layer handle the maintenance of a client's state. This principle allows for separation between a service and its state information, which favors replication and reusability for a service.

3.1.3 Independent Deployability

Another essential trait of a service that helps meet the requirements of elastic scalability is being *independently deployable* (desirably, by fully automated machinery [55]). It provides an ability for a developer to deploy, update, test and investigate directly on a particular service without affecting the rest of the system. Therefore each service can be scaled independently of other services. Moreover, the isolation of each service addresses the challenge of the technology stack. This, in fact, is entirely different from using monolithic application where components must be deployed together.

3.1.4 Scalable Service Architecture

The existence of previously mentioned key traits within a service as composability and independent deployability emphasized required migration in the context of the application architecture from the traditional monolithic towards a service-oriented architecture. The monolithic architecture defined an application as a single large unit which offers several services [55]. If any part of the application fails the whole set of services goes down, which represents a single point of failure. Any change made to a small module in a monolith application requires the entire application to be rebuilt and re-deployed, which causes a long downtime. Monolithic application architecture also suffers from the dependency hell problem [96] and imposed technology lock-in for developers.

The real essence of service-oriented architecture as a service-based model consists of breaking down a traditional monolithic application into units of service to reduce the complexity of the system. The evolution of software architecture in the last decades has shifted from Monolithic architecture towards service-oriented architecture and, more recently, microservice architecture. Service-oriented architecture can be seen as a step in this evolution process, which brought up the idea of service as a fundamental building block of an application. In a microservice architecture, a system is composed of independently deployable services, which interact with each other via light-weight communications mechanisms using standard data formats and protocols across well-defined interfaces [42, 128, 99]. Microservice is an architectural style that originated from SOA [89, 99].

Microservice architecture is defined as[99, 101, 55] as an approach to developing a single application composed of a set of independently deployable services, which interact with each other via light-weight communications mechanisms. The term microservice is defined as a small, cohesive, independent function which implements a single business capability [42]. However, there is no predefined size limit for a microservice. Each microservice is expected to maintain focus on combining related functionalities into a single business capability. Each microservice has the capability of being independently deployed by fully automated deployment machinery [55]. The major difference between microservices and SOAs comes from the higher *granularity* and *high independence* of services which brings the beneficial aspects in maintainability and extend-ability [99]. On the solid footsteps of successful adopters such as Amazon [72], LinkedIn [68], Netflix [92], the endpoint of that transformative journey should arguably be the microservice architecture.
One driving criterion that helps cut the service boundary right is to focus on the scalability concern [132, 2, 41]. Scalability is the ability to deploy cost-effective strategies for extending one's capacity [138]. Scalability can be obtained in two ways: vertical or horizontal scaling as it has shown in Figure 3.3. Vertical scaling (aka scale-up) is the ability to increase, in quantity or capacity, the resources availed to a single instance of the service of interest. The extent of this strategy is evidently upper-bounded by the capacity of the hosting server [133, 132]. In fact, this approach requires investing heavily in more powerful computers to accommodate the demand, which requires enormous capital expending. Therefore; the vertical type of scaling is limited by the size of a single server(physical/virtual) in which the application can only get resources as big as the size of the server [133, 15]. Horizontal scaling (aka scale-out) is the ability to aggregate multiple units, transparently, into a single logical entity to adapt to different workload profiles [15]. In this approach, application architecture needs to be designed in such a way that allows efficiently distributing and

scaling for their components. The system attains horizontal scaling through replication mechanism to handle the workload. Replication is one central dimension to this strategy.

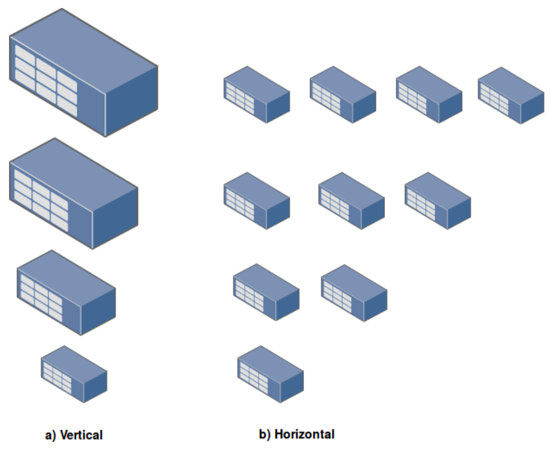


FIGURE 3.3: Horizontal vs Vertical scaling

To meet the goals stated earlier, we need to pursue *elastic* scalability (aka elasticity), that is, the combination of strategy and means that allow dynamic resource provisioning and releasing, while preserving service continuity, and that is amenable to full automation [6, 63]. *Speed* and *precision* are the qualifying traits of elasticity [63]. As the number of requests for particular service increases (respectively, decreases), the speed of resource provisioning (respectively, releasing) should vary in accord with the speed of demand variation, fully transparent to the user.

The latter quality, which one might call frugality, prevents wastefulness in the ratio of provisioning over need, by matching the footprint of resource deployment to the level of demand as exactly as the grain of service design allows.It is the balance of those two qualities that sanctions the goodness of the service boundary.

In the monolithic approach, in case of increased workload on a specific service, the scaling solution is to duplicate the entire application. This results in complexity and over-provisioning of resources for the other existing components, which is not beneficial for enterprises due to being inefficient. Similar reasoning can be applied to service-oriented architecture since services are large, not isolated and they can only scale at a large granularity. This is where microservices come up to facilitate by providing small services that could be deployed and scaled independently.

In a microservice architecture, due to stateless and independent deployability of service, each particular service can be easily replicated, and it is possible to achieve the beneficial aspects of scaling by utilizing resources more efficiently. Consequently, it is important for an application architecture to be capable of applying elastic scalability mechanisms to meet the requirement of the users and provide efficient utilization of resources. Deploy web applications in the cloud has shown [135] an independent ability of microservices to be developed, deployed, scaled and operated and monitored. Therefore, microservice architecture facilitates elastic scalability in the system and also provides efficient resource utilization due to having independent deployablity as a characteristic for its services. Numerous studies (e.g[**microiot1**, **microiot**, 11, 41, 12, 135]) show that adopting the microservice architecture helps pursue those goals. To better understanding, we illustrate the methodology which monolithic and microservice architecture uses to handle the scalability in Figure 3.4.

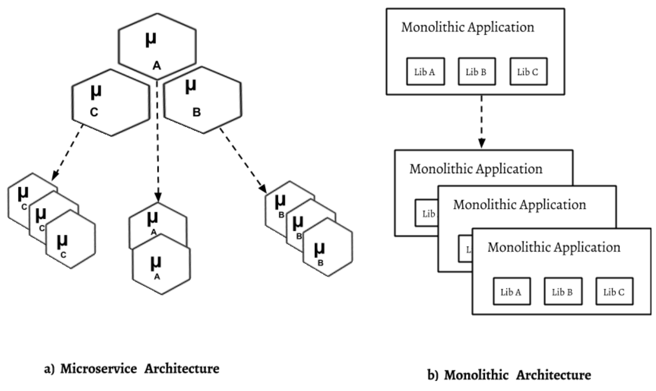


FIGURE 3.4: Microservice and Monilith scaling

3.2 Container Orchestration

In the microservices architecture, each service needs to be provided with its required resources in a fast, reliable and cost-effective way, and then needs to be run, upgraded and replaced independently. These requirements speak of containers [103]. Container-based virtualization, which originates from the Linux Container project (LXC), as we describe in Chapter2 is much better apt than the virtual machine at the scaling, while still assuring excellent isolation [103]. Because of their leaner nature, containers consume less resources (so that one single host can accommodate many containers) and are lighter and easier to house and transport, faster to deploy, boot and shut down than virtual machines [43]. It is not surprising, therefore, that containers were found to be up to 10 times faster to bootstrap than virtual machines [50].

There are two approaches to deploying an application using containers: one-to-one and one-to-many [112]. The classification criterion is the number of services housed in the container, which defines its *granularity*. The one app (service) per container approach allows each service to run on a dedicated container so that one container hosts only one service, and each service is packaged as a container image. This solution eases horizontal scaling, provides fast build/rebuild of the container, and allows the same container to be reused for multiple different purposes in the same system. The major downside of this approach is that it may yield a large number of containers, and thus considerable overhead for interaction and management. The one-to-many solution, where each container houses multiple services, addresses this very concern, without however answering the question of which services to gather into which container.

Consequently, having multiple independent units of the containerized unit of microservice applications collaborate in responding to a user request most definitely incurs coordination overheads that must be addressed by *orchestration* [134]. Orchestration is defined as the automated arrangement, coordination, and management of complex computer system, middleware, and services, all of which helps to accelerate the delivery of IT services while reducing cost. It concerns services, workload, and resources in the system [134].

The entry level of of orchestration entails automating the deployment of the application by means of *Continuous Integration and Continuous Delivery* (CI/CD) solutions such as Chef [107], and Jenkins [70]. Despite, significant benefits provided by the DevOps approach into the process of orchestration, fluctuating user demand,

and on-demand Cloud service provisioning derives the need for dynamic orchestration across the Cloud environment. Supporting on-demand provisioning of the resource requires elastic resource allocation to minimize provisioning costs while meeting service level objectives (SLOs). Therefore, to align with elastic scalability, orchestration must be dynamic, that is, able to support the adjustment of resource provisioning and service deployment required to fluctuating user demand.

3.3 Implementation princincple within state-of-the-art technology

The complementary specification in the direction of designing and implementing elastic scalability for the definition of dynamic orchestration that we discussed earlier requires the technology solutions to support efficient resource allocation and coordination. Figure 3.5is shown the reference architecture for Orchestration process which can be understood in analogy to a conductor as an orchestrator, who conducts an orchestra so that performing all different instruments are in tune and in time and conductor (orchestrator) oversee the process by receiving responses.

Later, we will describe more in detail the main concerns and functionality of the container orchestration platform with considering current state-of-the-art in this technology.

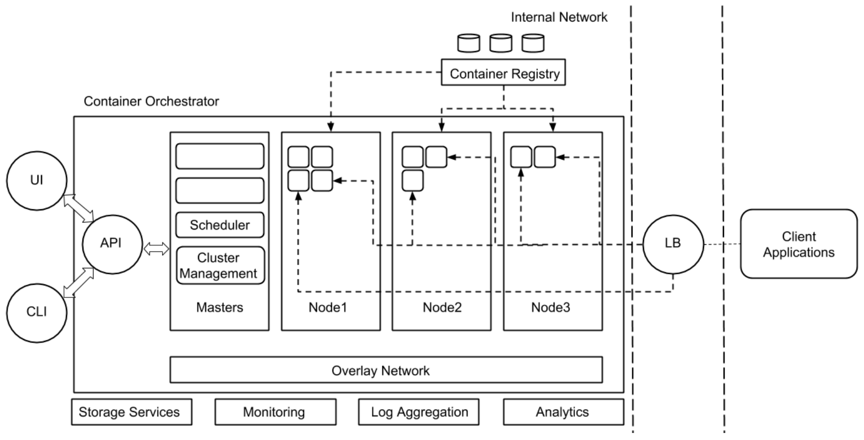


FIGURE 3.5: Refrence architecture for container cluster manager

In particular, it's primary objective is to provide alignment between application and resource so that it can address under-provisioning and over-provisioning issues. The scheduler is an essential component of an ideal dynamic orchestration solution toward elastic scalability issue, as it is the entity in charge of generating a dynamic execution plan that responds to changes in user demand. The scheduler (auto-scaling) must generate dynamic statistics on the state of the resources that it manages, to compute the best possible service-resource mapping, using either reactive or proactive strategies.

Reactive approaches are solutions that react to detected demand fluctuations before(periodic) demand prediction is available. Examples of this solution appear in most commercial solutions such as RightScale [114] and AWS [3].

Proactive approaches are solutions that use predicted demand to allocate resources before they are needed [58, 40, 34]. In using proactive approaches, the accuracy of demand prediction and provisioning is critical to saving costs with utility computing [131].

Accordingly, auto-scaling approaches (reactive, proactive) are introduced as a solution to allocate runtime resources according to workload dynamically. Therefore, a modern scheduler(autoscaling) should provide support for the following features:

- **Affinity / Anti-affinity**, to implement restrictions that enforce given entities to always be adjacent or separate to one another. An affinity rule simply means the ability to ensure that certain workload or virtual server always runs on the same host. Anti-affinity works in the opposite way, with the same impact as Affinity.
- **Taint and toleration**, to mark a particular (logical/physical) computing node as un-schedulable so that no container will be scheduled on it other than those that explicitly tolerate the taint. This requirement can also be used to keep away a node with a particular specification (e.g., a physical resource) from the others and dedicate it to given workload. It also allows rolling application upgrades of a cluster with almost no downtime.
- **Custom schedulers**, to delegate responsibility for scheduling an arbitrary subset of hosts to the user, alongside with the default scheduler.

To examine how current state-of-the-art technology meets the requirements for elasticity, we look in Kubernetes [82, 21], OpenShift [10], DockerSwarm [69], as the most widely used solutions for the dynamic orchestration.

3.3.1 Kubernetes

One of the first open source orchestration platform written in Google's Go programming language. Its architecture, as depicted in Figure 3.6, is based on a set of master and worker nodes configuration. Within each worker node, there are groups of containers called pods, which share resource and are deployed together. The basic unit of scheduling is called a "pod" which can be composed of one or more containers. Pods play a significant role in placing workload across a cluster of nodes. A set of pods that work together is called a Service in Kubernetes. Services are exposed within a cluster and are assigned an IP address and the Domain Name System (DNS) name so it can be discovered by other Kubernetes applications and service [9].

The Master uses an etcd key-value store to maintain information about the state of the entire cluster. An API server provides an HTTP interface for both internal and external access to the Kubernetes master. It processes REST or Protocol Buffers (Protobuf) requests and updates the etcd data store. The Scheduler selects the node that an unscheduled pod should run. The built-in scheduler knows about resource requirements, availability, affinity, etc. to make scheduling decisions. However, another specialized scheduler can be provided as an add-on plugin. In the worker, the Kubelet is responsible for discovering the resources and keeps track of running state on each node and relays that information to the Kubernetes master. Kubelet is also responsible for managing the pods and starting the containers [9].

Kubernetes uses architectural descriptions, written in JSON or YAML, to describe the desired state of services, which it feeds to the master node. The master node devolves all dynamic orchestration tasks onto worker nodes. Kubernetes has a controller that manages the state of the cluster and provides mechanisms to replicate and scale multiple copies of a pod across the cluster of server nodes.

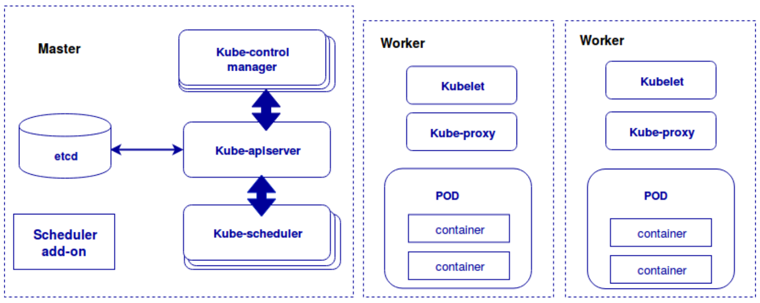


FIGURE 3.6: Kuberenets architecture

Kubernetes provides `Required` and `Preferred` rules. Required rules need to be met before a pod can be scheduled. Preferred rules do not guarantee enforcement. Kubernetes also uses a replication controller with a reactive approach to instantiate pods as required. Kubernetes provides pods and nodes with affinity /anti-affinity attributes. It defines rules as a set of labels for pods, which determines how nodes schedule them. Anti-affinity rules use negative operators. Kubernetes marks nodes to avoid them to be scheduled. The `Kubectl` command creates a taint on nodes marks them un-schedulable by any pod that does not have a toleration for taint with the corresponding key-value. Kuberenes at v1.6 supports customs schedulers in a way that allows users to provide a name for their custom scheduler and ignore the default one. Custom strategies are completely custom, meaning that the user must write an ad-hoc plugin to use them.

3.3.2 Docker Swarm

It is a native orchestration tool for the Docker Engine, which supports deploying and running multi-container applications on different hosts. The term swarm refers to the cluster of physical/virtual machines (Docker engines), called nodes, where services are deployed. Its architecture as shows in Figure 3.7 uses one master and multiple worker nodes.

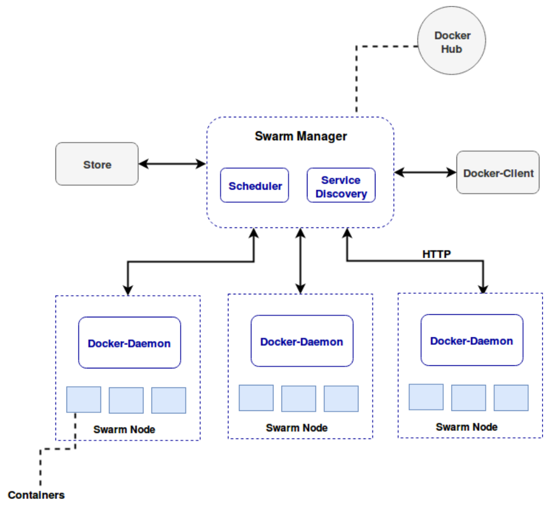


FIGURE 3.7: Docker Swarm platform schema

The key concepts in it are *services* and *tasks*. Service designates the tasks that need to execute on the worker nodes, as specified in the tasks' desired state. Tasks represent atomic scheduling units of work associated to a specific container. Docker

Swarm uses a declarative description, called Compose file, written in YAML to describe services. The swarm manager is responsible for scheduling the task (container) to worker nodes (hosts). Docker Swarm places a set of filters on nodes and containers to support the first two requirements mentioned earlier. Node filters `constraint` and `health` operate on Docker hosts to select a subset of nodes for scheduling. The node filter `containerslots` with a number value is used to prevent launching containers above a given threshold on the node.

Docker Swarm offers three strategies, called `spread, binpack, random`, to manage cluster assignment according to need. Under the spread strategy (which is the default one), it computes ranking according to the node's available CPU and RAM, attempting to balance load across nodes. The binpack strategy attempts to fill up the most used hosts, leaving spare capacity in less used ones. The random strategy (which is primarily intended for debugging) assigns computation randomly among the nodes that can schedule it.

3.4 Final Consideration

As mentioned earlier, scheduler is an essential component to achieve optimal provisioning that response to changes in user demand. Concentrating on the scheduler to achieve optimal provisioning in the mentioned technologies, we observe that all are using predictive approaches to achieve elasticity. As, using resources have boot-times that vary, depending on the application, from a couple of minutes to even 30-40 minutes to load all the components needed. Even in the container world, spawning a new large-image container on a new node while the network is under stress, will easily take 5 to 10 minutes. The same holds for shut-down times. Therefore, the longer the boot time will result in consuming more resource and less efficiency. So, the more important it becomes to be proactive and provide required resources in advance to achieve effective elasticity. This justifies investigating proactive algorithms in place / along with reactive ones to achieve effective elasticity.

Consequentlty, the technology exploration that we have discussed shows that there continues to be a gap between what state-of-the-art solutions have to offer in the way of support for elastic scalability and the full range of requirements that such a need entails. Accordingly, proactive approaches to resource scheduling are the most acute need that is not supported as yet other than in early research prototypes. To this end, our next research goal as describe in Chapter 4 is to design, implement and comparatively evaluate experimental proactive elastic resource allocation algorithms, which could be used by Cloud service provider to make a more sustainable use of DevOps.

Chapter 4

SCALABILITY IN SOFTWARE DEVELOPEMENT PROCESS

4.1 Background

Traditionally, software development has followed a linear Software Development Life Cycle (SDLC) that traverses planning, analysis, design, coding, and testing before enabling application deployment onto the production environment. In this style, post-deployment maintenance of the application was in charge of the IT/Operation professionals (IT/Ops), whose organization tile had separate objectives (and distinct key performance indicators) from developers. In particular, the development objective is to release application service augmented with new features using agile development practices, whereas the operation objective during maintenance is to ensure continued stability and reliability to the production environment.

Over the years, these two teams siloes by their respective separation of concerns has created barriers of practices and solutions to the problem of deploying service updates with new features quickly and frequently, without undermining the reliability and stability [80]. Besides, this hiatus broadened to encompass negative effects on the efficiency of the delivery cycle, as well as on the quality of the products and services provided.

In 2001, Agile methodology ultimately gave rise to new processes and technology breakthroughs aimed at streamlining and automating the entire software delivery lifecycle. Agile methodologies taught developers to break down software development into smaller chunks known as "user stories," which accelerate feedback loops and align product features with market need. Accordingly, the focus of the software life cycle moves downstream toward IT operation, to reduce the friction that arises with new software releases, under the label of DevOps [76, 14].

DevOps originated as a natural evolution of the agile methodology for software development, integrating traditional SDLC with operational support into one single methodology centered on the notion of continuous action [51]. In the conventional sense, DevOps is the integration of the process practices and associated tooling that drive continuous integration and delivery of business software applications [45].

Kim et al. [76] articulate five major points to describe the fundamental principles of DevOps framework, summed up in the acronym ”CALMS”: Culture, Automation, Lean, Measurement, Sharing. The core of these authors’ idea rests on building a culture of collaboration that bridges the gap between Development (Dev) and Operations (Ops) to design, deliver, manage and improve the way IT is used within the organization.

DevOps embodies those principles into a continual-improvement process called Continuous Integration and Continuous Delivery pipeline (CICD) in the intent to improve stability and performance of the organization's development and operation assets [136, 45, 13]. The CICD workflow is designed to ensure deployable and scalable product delivery that can be updated in real time in response to monitored evidence collected throughout the entire software life cycle.

Figure 4.1 depicted the Continuous Integration (CI) part of the CICD pipeline, whose prime benefit is to keep the developers synchronized with each other frequently. In fact, CI much reduces the delays related to integration issues as each step of integration involves highly automated test sessions aimed to detect latent errors as quickly as possible. In that respect, therefore, CI allows the rapid building of new features in the application while reducing the associated risks [54]

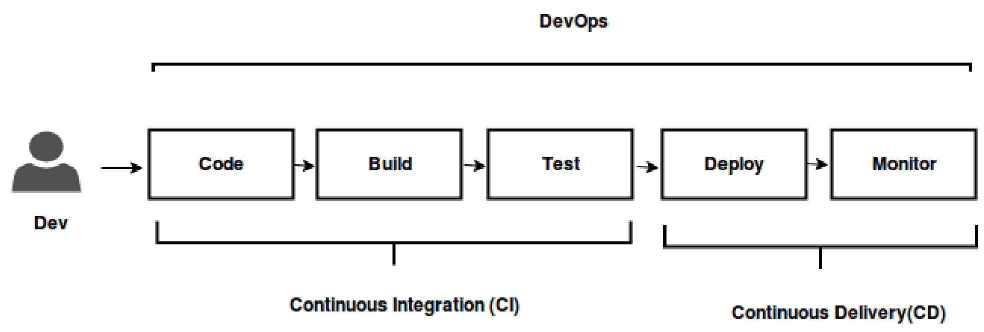


FIGURE 4.1: CI/CD pipeline for software development

The main focus of the DevOps culture is commonly associated with Continuous Delivery (CD) with more holistic view, originated from the goal of acquiring the ability that uses smart automation to create repeatable and reliable process for delivering software [53]. The CD practices (cf. the right part of Fig. 1) subject the changes applied a software part to automated verification tests performed on the so-called deployment pipeline, in a fashion that gets the application increasingly close to the eventual production environment [53]. In effect, Continuous Delivery expands the concepts behind Continuous integration. CI started as a development strategy and expanded to encompass production deployments, which meant bringing the operations team closer to development. When the principles of DevOps came into play, places emphasis on the operational issues such as deployability, scalability and monitoring.

4.2 Infrastructure Agility

The concept of DevOps, with its focus on quality building, requires a high degree of automation that extends from the application to the infrastructure itself, giving rise to the principle of *Infrastructure as Code (IaC)* [45, 64].

Infrastructure as code (programmable infrastructure) refers to machine-readable code to manage infrastructure resources, configurations and deployments while automating dynamic provisioning and releasing [136, 98]. IaC seeks to afford maximum velocity to the continuous integration and delivery pipeline, enabling the user to treat the infrastructure management as an agile discipline, thus helping balance time, resource and quality, which are the critical assets in any business environment [80].

As shown in Figure 4.2, the automation of IaC results in a workflow process that assures all software assets are portable, reusable, and subject to version control. It keeps the desired infrastructure state transparent while assuring the availability of IT infrastructure entire the whole software development process.

Therefore, the IaC paradigm is a natural fit for cloud infrastructure where a resource can be provisioned and configured through APIs thus helping balance *time*, *resource*, and *quality* which are the critical assets in elastic service delivery in the cloud environment [136].

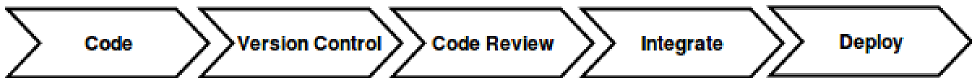


FIGURE 4.2: Infrastructure as code workflow

Traditional configuration management tools (such as Puppet [107] for example) have helped to achieve automation-based consistency in code-based infrastructures, but they fall short for scaling, where their single-server organization becomes a frustrating bottleneck. Terraform [127] is an example of an open source tool to provision and manage cloud infrastructure using declarative, version controlled configuration in an automated fashion. It generates an execution plan describing what it will do to reach the desired state and then executes it to build the described infrastructure. As the configuration changes, Terraform is able to determine what changed and create incremental execution plans which can be applied [127]. In fact, the IaC paradigm is able to reproduce and change the state of environments in an automated fashion from information in version control rather than configuring infrastructure manually as in the configuration management tool.

4.2.1 Operational Performance

Realizing software releases in terms of as-a-service offerings, which is natural for web and Cloud service development, naturally positions their providers as the early adopters of DevOps principles after the agility and speed that they promise [13]. One of the traits that forms a join point between the two perspectives of Cloud and DevOps is *rapid* elasticity, which, in the DevOps context, becomes the rapid delivery of scalable infrastructure as a service [56].

In this context, DevOps embraces the operational aspects of defining a service as a product item that needs to be deployed, scaled, maintained, monitored and supported through its life cycle. DevOps creates increased innovation opportunities and fostering a performance orientation with the highest priority to satisfy the customer through continuous delivery of valuable software. In such a way that releasing software happens rapidly, frequently, and more consistently, under the notion of continuous delivery and deployment pipeline.

Accordingly, one of the main challenges that DevOps methodology tries to address is adaptability to changes while maintaining high quality at low cost and high-speed [80].

Therefore, concerning the IT evolution depicted in Figure 4.3, to get the most value out of the rise of cloud, containers, microservices, enhancing agile and dynamic, given their complexity and operational costs must be organized and managed in the most optimal, and adaptable way. The adoption of Container technology as a lightweight and scalable solution to deployment challenges, which isolates the application and its dependencies within self-contained units agnostic of programming language and execution platforms, removes the need for runtime collaboration between Dev and Ops [103]. Precise allocation of resources for individual containers enables efficient usage of the underlying infrastructure. However, this practice also adds one level of abstraction (the container layer itself) to the problem of managing the deployment infrastructure of applications. Most important, this perspective changes the nature of scaling from handling relatively coarse-grained units (VMs) to a collection of small, lightweight coordinated units (containers). Therefore, containerization reduces over-dimensioning and under-utilisation, while increasing manageability to achieve optimal infrastructure scalability [103].

DevOps encouragement for rapid integration and cross-team communication motivates abandoning siloed application architectures, and therefore favors the adoption of microservices (application architecture), which break down the application into a granular set of independent units of discrete functions that are built and deployed independently and communicate via APIs external to their code base and therefore accessible with technology-neutral solutions [99, 11]. Embracing microservice-based applications the developers may build more robust software and implement the extent of scalability and performance features that operations personnel much desire [41, 128]. Taken as part of the automated system, applications real- ized in terms of containerized microservices increase the agility and scalability dotation that serves the speed of deployment.

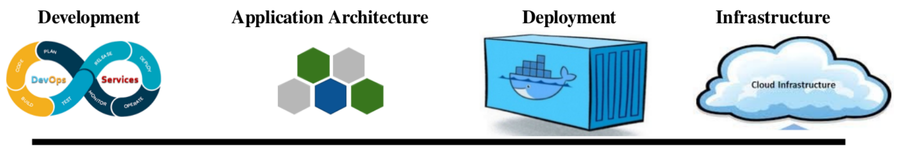


FIGURE 4.3: IT Evolution

To improve the delivery of value, Cloud offers DevOps an advantageous infrastructure (IaaS), which enables an effective team structure (topology) to be put in place between Dev and Ops personnel using Infrastructure-as-a-Service as its platform [91] as it shows in Figure 4.4.

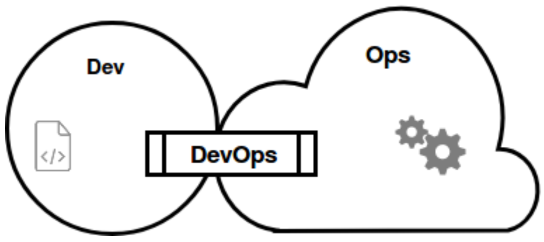


FIGURE 4.4: Dev and Ops united in an Infrastructure-as-a-Service (platform) topology)

In this topology, appropriate IaaS settings provide an elastic infrastructure to operation, to deploy and run applications on, assuring the low latency required to meet desired service level agreement (SLA), and facilitating the release of new software versions. Elastic infrastructure denotes the scaling of resource up and down in order to meet the quality of Service (QoS) requirements such as response time. However, applications deployed in a cloud environment require both performance and cost-efficient resource usage.

4.3 Experimental Evaluation

To get the most value out of the Cloud, containers, and microservices combined, requires innovative solutions for the coordination and management of infrastructure optimization. As noted in Chapter 3, the umbrella terms that encompasses those challenges is dynamic orchestration.

To this end, our contention here is that orchestrating containerized applications dynamically using automation in the faces of the continuous changes in the dynamics of service workloads, requires Artificial Intelligence techniques to learn, model and predict changes in system behavior in near-real time.

To explore the viability of this contention we looked at Auto-Regressive Integrated Moving Average (ARIMA) model [23] for forecasting a time series and Long-Short Term Memory (LSTM) [143] as a special kind of Recurrent Neural Networks (RNNs), which is capable of learning features and long-term dependencies in the modeling of dynamic systems.

The objective of RNNs is to map an input sequence, one step at a time, into a corresponding output sequence, using integration of information captured in the hidden layer (recurrent unit) to predict the input sequence ahead, while optimizing the weight parameters of the network. RNNs have been widely and successfully applied to many sequential tasks and time-series analysis [117]. The structure of RNNs is inspired on the biological neural network that is the brain. RNNs consist of multiple cascading layers (input, hidden, output) of non-linear artificial neurons that operate as basic cog- nitive units, and a feedback loop called a recurrent unit that provides persistent memory over time (through the input sequence) [117].

Most uses of RNNs to model long-term sequential dependencies in the state of the art report exposure to vanishing or exploding gradient during training [117]. Long-Short Term Memory (LSTM) [143] has been proposed to overcome those problems. To this end, the LSTM modifies the layers of the RNN adding an internal state variable, which keeps track of the already processed inputs and therefore eases the modeling of long-term sequential time-series data sets without requiring massive and costly updates to the recurrent unit.

The effectiveness of the prediction algorithm is evaluated by the Root Mean Squared Error (RMSE) and Mean Absolute Error (MAE) metrics, which are defined as follows:

$$RMSE = \sqrt{MSE} \text{ where } MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_{i_{ground}} - Y_{i_{pred}})^2 \tag{4.1}$$

$$MAE = \frac{\sum_{i=1}^{n} |Y_{i_{ground}} - Y_{i_{pred}}|}{n} \tag{4.2}$$

where $Y_{i_{ground}}$ is the actual output at time i, $Y_{i_{pred}}$ is the predicted output and n is the number of observations in the dataset. The smaller the RMSE or MAE values, the better the predictive quality of the model.

4.3.1 Experimental Environment

Having explained the context of interest to our work, we can now illustrate the strategy that we have adopted to embed our proposed method for orchestration of single-tier web application deployed on Amazon's Elastic Container Service.

The Elastic Container Service (ECS) of AWS provides a highly scalable container management that monitors, schedule and deploys containers across cluster of nodes corresponding to EC2 instance [8].

To create a simple, yet representative scenario for our experiments, we employed a cluster comprised of two EC2 instances of type t2.medium to run a container service composed of single tier web application (web tier) deployed on an Apache web server (Figure 4.5).

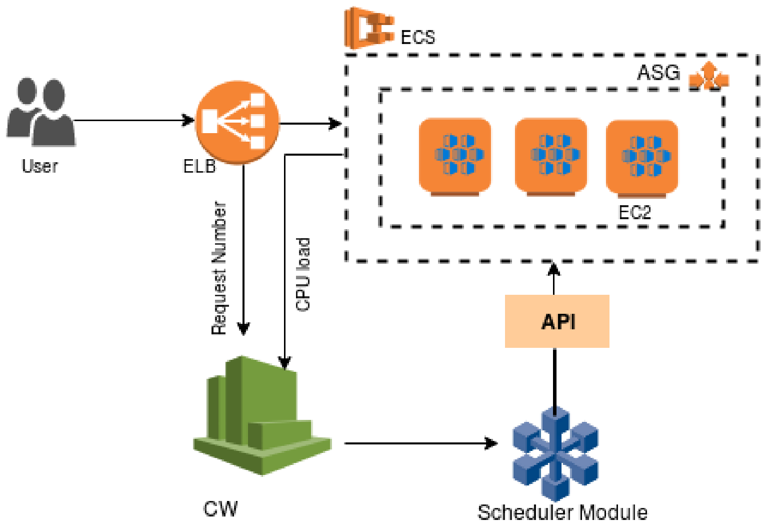


FIGURE 4.5: Experimental framework for orchestrated container on AWS

The implementation uses five parts:

1. Elastic Load Balancer, ELB,
2. Elastic Container Service, ECS,
3. Auto-scaling Group, ASG,
4. CloudWatch, CW, and
5. A Scheduler Module that bridges the ECS cluster with its associated CloudWatch.

The ELB, ECS, ASG and CW components are standard components of the AWS offering. The Scheduler Model was our own. To generated the historical data to train and test the LSTM forecasting model, we used the AWS server metrics (request per second, CPU utilization) as collected by the Amazon CloudWatch service available with the Numenta Anomaly Benchmark (NAB) [102]. Subsequently, we generated live workload using Hey (rakyll) [109] with the same probability distribution as used in the production of the training dataset, which we jittered with random noise. During the experiment, we used the CloudWatch service to monitor CPU utilization of ECS in the two dimensions of service level and cluster level, as well as the number of requests en-queued at the LB, to feed a prediction model and translate the output into the required number of containers. As part of that, we used the AWS API to access the provided key-value store to acquire the current state on each cluster node and integrated the resulting value in our custom scheduler. In that setting, we compared the behavior of our custom model with that of the default ECS. Overall, our experiments consisted of the following steps:

1. Designing and implementing automated scaling API for containers, which observes selected and configurable metrics from ECS;
2. Constructing an application performance model machine learning that predicts the number of container unit required to handle demand;
3. Periodically predicting future demand using time series and determining the application resource requirements using the performance model;
4. Automatically allocating resources using the predicted resource requirements;
5. Running experiments on real data to determine how fast the system can respond to emerging application needs.

4.3.2 Experimental Results

The first goal of our experiment was to compare the prediction of required number of containers obtained with the ECS default method against ours. Figure. 4.6 shows the results that we obtained, which shows a lower error rate corresponding to prediction time in applying the LSTM in comparison to the ARIMA model to proceed with the next step of the experiment

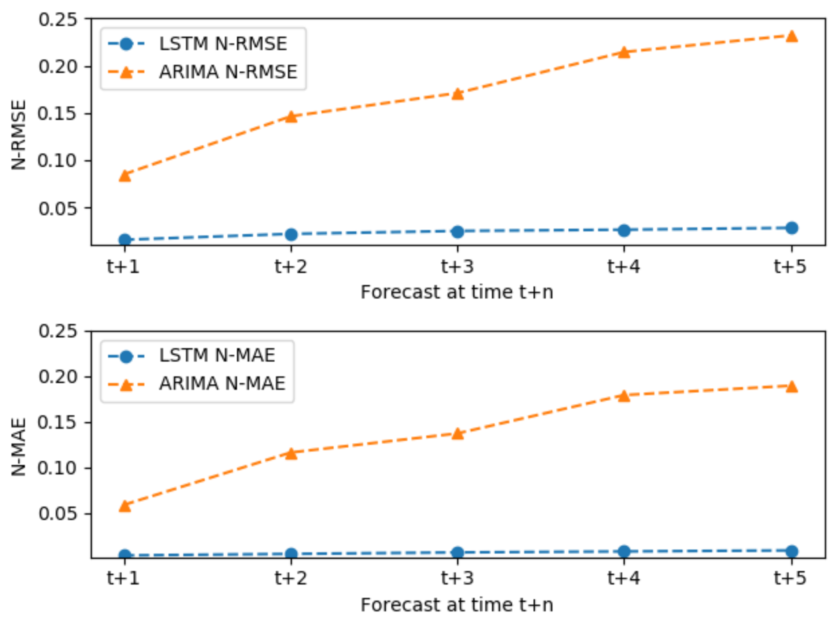


FIGURE 4.6: Prediction error.

Figure. 4.7 plots the results we obtained, which show better performance of our model, where the service time provided to the user stays quite stable throughout significant changes in the intensity of user requests.

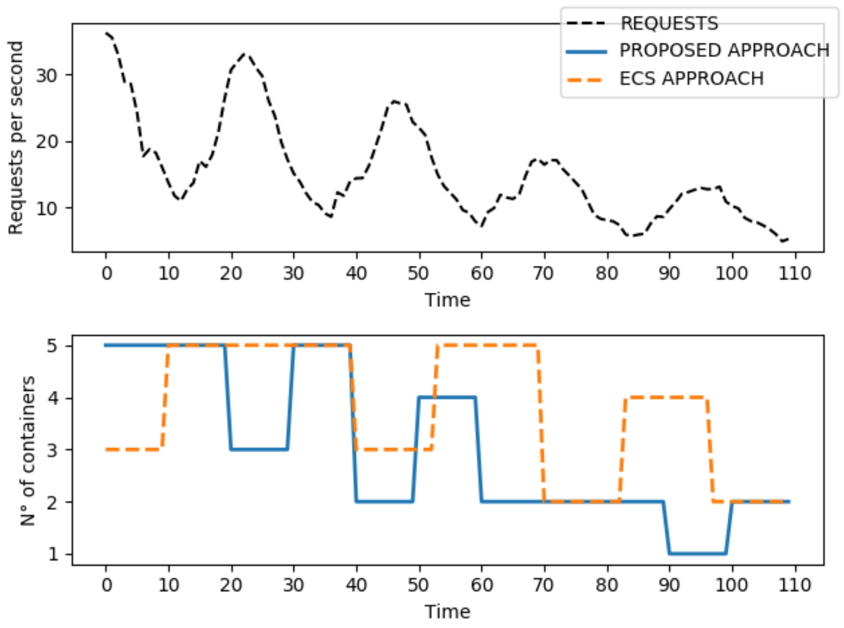


FIGURE 4.7: Container allocation in response to varying load.

In the second part of our experiment as shown on Fig. 4.8, we used response time as a performance indicator, that is, the time to serve a user request to completion. Once again, the compared the default auto-scaling mechanism, threshold based on CPU utilization per node, against our predictive model.

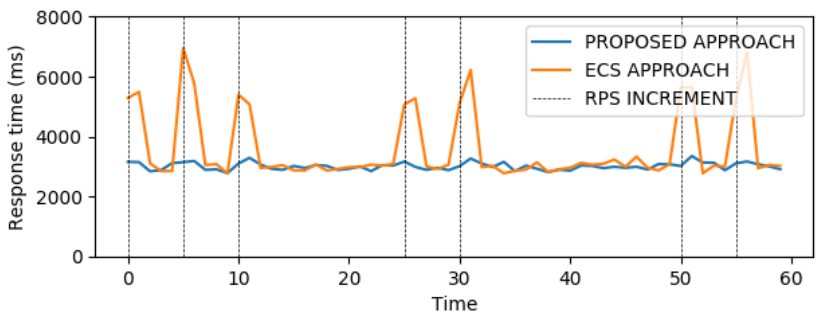


FIGURE 4.8: Variation of service response time under variable user load.

4.4 Final Consideration

In this work, we have presented and discussed our view of how the dynamic orchestration capabilities required for containerized microserviced-based applications responds to the DevOps demand for increased agility in the whole development cycle. We have argued that Machine Learning is a necessary ingredient to enable the realization of sound and effective automation rules. To prove our point we presented the results from an experimental implementation of an Elastic Container Service (ECS) hosted on AWS, augmented with our machine learning implements, to increase the agility in deployment of containerized application. We evaluated that prototype against a few experimental scenarios, which showed the lower latency achieved by our method for application deployment.

Chapter 5

SCALABILITY IN ANALYTIC TIER OF MACHINE LEARNING

The increasing amount of data generated and to be processed in the context of Smart Cities, along with the complexity of existing neural network algorithms to meet a decent accuracy on modeling, has led to bigger computational and memory resource requirements. Accelerating neural networks training to competitive accuracy while maintaining a short training time is a major challenge that entails greater computational resources demands. Consequently, handling this challenge due to the scalability and efficiency of the learning algorithms have given rise to the notion of distributed machine learning. Distributing machine learning towards scalable while efficient algorithms is a promising solution in a scenario where both data and resources are scattered along with the architecture, with the added challenge of the near impossibility of having all data in the same place, and the cost of constantly offloading computation to the Cloud.

In this Chapter, we present a distributed machine learning approach for road traffic modeling and prediction, designed for a city-wide scenario based on the *Fog computing* paradigm. We display an architecture for large scale Floating Car Data (FCD), data modeling on the Edge, leveraging parallelism on deep learning algorithms in a city-wide scenario towards prediction. FCD is collected in localized Edge nodes, and processed as time series indicating "volume of cars" and "average speed" on the surrounding area. This approach takes advantage of already active devices collecting data instead of purchasing additional Cloud resources, also reducing network communication and protecting services from network disruptions by keeping them autonomous on the Edge.

5.1 Introduction

The Internet of Things (IoT) has drawn significant interest from both academia and industry. The potential applications oriented towards Smart Cities and health-care services urge experts on designing new architectures for infrastructure, platforms and services, taking care of inherent limitations on connectivity and computing power found on edge devices and dynamic networks while leveraging near-data computing. Those IoT architectures are usually composed of real-time monitoring systems and actuators (sensors and acting devices) running in different locations, connected to data aggregation applications or data-warehouses through dynamic networks (such as 5G, Wi-Fi or wired Internet).

Cloud architectures provide "infinite" resources and scalability, whereas, scenarios like Smart Cities usually require proximity and quick reaction time, while generating big amounts of data that must be transmitted to the analytics applications.

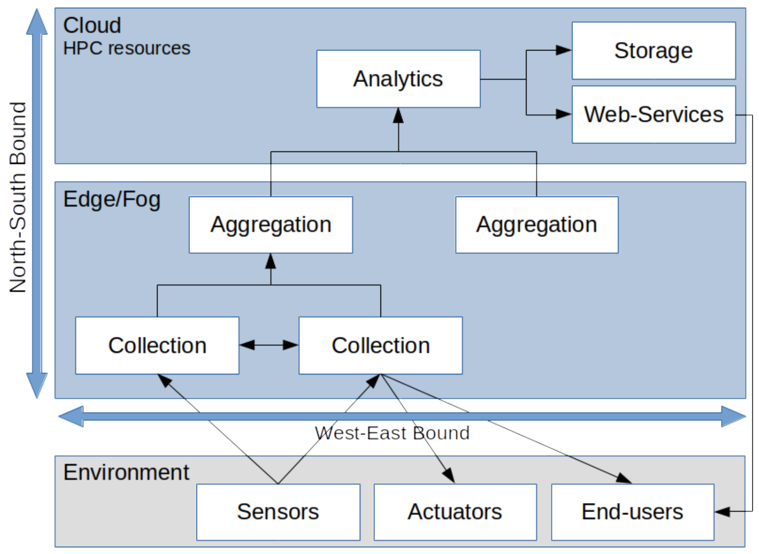


FIGURE 5.1: Edge-Cloud Aggregation Schema, with environment actors (Sensors, Actuators and Users)

Knowing that those analytics applications aggregate data, Fog computing pipelines contemplate aggregation at low and intermediate levels, moving data processing close to the sources, reducing data to synthesized volumes to be transmitted northbound to the Cloud, as shown in Figure 5.1. Additionally, when services provided in the Edge depend only on local data, collected and aggregated data among local nodes can provide solutions to the service without communicating with the Cloud. This is the case of traffic monitoring on cities, where street-placed nodes and antennas can collect data from the road and equipped car sensors, and analyze and predict traffic status locally while sending only relevant statistics to the Cloud.

Here we address the problem of modeling time-series data, particularly Floating Car Data (FCD), for predictive analytics on the Edge, where the Edge-node devices have resource limitations. Due to big volumes of geo-distributed and time-distributed data produced by FCD, we focus on:

1. Modeling data on the edge using deep learning methods.
2. Considering time to train models in scenarios with limited time and resources.
3. Producing accurate enough models with the given time.
4. Constrained number of CPUs/GPUs.
5. Leveraging availability of distributing load among CPUs/GPUs or neighbor devices.

Through this, we can distribute intelligence and load across the Edge, affording good-enough models with the available resources, without pushing all unnecessary data to the Cloud and using resources already available on the Edge, demonstrating the effectiveness and feasibility of Edge-based data analytics.

On this setting, we use Gate Recurrent Unit(GRU) neural networks to model the traffic behavior in order to produce short/medium-term prediction on such traffic. Additionally, we present a lesser-complex technique to parallelize and automate the training process, considering that models will be created on the Edge, fitting to local behaviors. From here on, we study different levels of data aggregation, different levels of data processing parallelism, time requirements for achieving suitable accuracy levels for models, and suitability for real-time applications in the Edge. To validate this approach, we used real traffic logs from one week of Floating Car Data (FCD) in the city of Barcelona, provided by one of the largest road-assistance companies in Spain, comprising thousands of vehicles from their fleet only in the city of Barcelona. The data-set comprises data collected over one week between 10/27/2014 and 11/01/2014 across the Barcelona metropolitan area. Deep Learning modeling is done through Keras (R + TensorFlow), comparing train time, obtained accuracy, used resources (i.e. 1 CPU/GPU per process, as edge devices are not High-Performance designed), and GRU configuration (provided hidden units). Finally, we provide a set of criteria to automate the learning process by selecting training time in function of observed behaviors on model training and validation.

Through experimentation we observe that the presented methodology allows to bring this kind of predictive analytics to the Edge, including complex learning mechanisms like GRUs, allowing cost/effective compromises between model accuracy and training times at low powered or restricted devices, and can be enhanced by distributing load among extra CPU/GPU resources when available. Additionally, compared to previous prediction methods, GRUs achieve good accuracy results with constrained training time in front of the previously used methods, even if the modeling process is split to reduce training time.

5.1.1 Contribution

The contributions of this work are summarized as follows:

- An architecture for distributed modeling Floating Car Data based on the Fog computing paradigm and Edge Analytics.
- A simplistic method to distribute Deep Learning data modeling processes on restricted resource scenarios, using multiple CPUs/GPUs.
- A comparison and study of resource usage vs. accuracy on training models for real Floating Car Data, also comparing to baseline methods.
- A comparison of required time on modeling when performed on High-Performance environments (e.g. the Cloud) versus Low-Power environments, using real standard devices on the Edge.
- A mechanism to automate decisions on-the-run for stopping training processes when accuracy levels are considered reliable.

From these contributions on, further advances can be done by reducing not only data transmission, but also model transmissions on scenarios where models were computed on the Cloud and pushed southbound, also increase the quality of service on Edge-provided end-user services by not depending on Edge-Cloud communications and being resilient to back-haul interruptions.

5.2 State of the Art

5.2.1 Background and Motivation

In the past years, the "pay-as-you-go" model from Cloud Computing has become an alternative to private data-centers, due to its scalability paradigm. The Cloud has been widely used to address the emerging challenges of big data analysis in many smart city ecosystems such as smart houses, smart lighting, and video surveillance [38, 125, 24]. However, IoT scenarios usually require low latency between sensors/actuators and computing power, while attempting to avoid unnecessary north-south bound communication of data that can be processed on the edge or intermediate nodes. Location awareness is also a must on many Smart Cities IoT architectures providing immediate in-place services. As IoT services in Smart Cities are getting more involved in the daily life of people, Cloud services alone can hardly satisfy the mentioned requirements of this ecosystem.

Keeping data processing near the actuators and end-users on the Edge, to offer localized and low latency, is required by the explosive growth data generated by the ubiquitous deployments of sensors. Within this context, intelligent data analysis is playing an important role in predictive analytics related to city-wide scenarios, facing large data-sets to be processed, most of them localized [104]. Submitting the enormous data generated by massive sensors to the Cloud, to later receive results to the user, requires communication bandwidth and power consumption.

Consequently, the inherent distributed architecture of the Edge has significant advantages over the Cloud to provide localized services to a user and is well positioned for real-time big data analytics close to the source for latency-sensitive applications, such as complex event processing, gaming, and video streaming [66]. Fog computing, the paradigm combining the Edge and Cloud capabilities, can handle the significant data treatment, including acquisition, aggregation, analytics and preprocessing, while reducing transportation and storage, even balancing computation power among intermediate nodes. In addition, transforming this data into actionable knowledge and adapting to changing dynamics of modern cities, requires *intelligent* modeling techniques not only accurate but adaptive. Machine learning techniques enable *smartness* in Smart Cities by modeling, predicting and extracting useful information from collected data, through advanced statistics and artificial intelligence algorithms. Deep Learning, a machine learning technique based in multilayer neuronal networks, is becoming an important tool in City modeling paradigms across many areas such as forecasting [], , image processing [81, 28], speech recognition [32, 85, 4] or object recognition [29], useful to manage public services, detect hazardous scenarios or aid emergency services among others.

5.2.2 Related Work

The exponential growth of the IoT, caused by the opportunity of leveraging smart devices in generalized enterprise settings, motivate the quest for novel approaches to developing deep learning system that can scale to very large models and large data-set to which suit the computational resource and time requirements. However, training to competitive accuracy while maintaining a short training time with large and complex networks together with huge data-sets results in a major challenge in taking advantage of Fog computing paradigm for city wide scenario (longer training time).

A significant amount of effort and research has been put into tackling training huge data-sets through building large models with more parameters and parallelization or distribution methods based on the Cloud computing infrastructure [17, 37, 122, 83, 94, 144]. For example, Google implemented a distributed framework for training neural networks over CPUs based on the DistBelief framework [35, 84] which makes use of both model parallelism, and data parallelism.This model has also proved useful for computer vision problems, achieving state-of-the-art performance on a computer vision benchmark with 14 million.

Within the context of scale up learning in the direction of training , researchers utilize accelerators such as a single or cluster of GPUs to achieve scale through computational power [124, 30]. Recently, Facebook [59] present the results of 90% scaling efficiency on training visual recognition model using data parallelism combining with the use of GPUs. R.Raina et al. [108] present one of the earliest research based on data parallelization using GPU. [28] used GPUs to train a collection of many small models and subsequently averaging their predictions. The authors of Hovord [119] presented an open source library that employ efficient inter-GPU communication and enable faster, easier distributed training in TensorFlow.

K. Hong et al. [66] propose a fog-based opportunistic spatio-temporal event processing system to meet the latency requirement. Their system predicts future query region for moving consumers and starts the event processing early to make timely information available when consumers reaches the future locations.

5.3 Architecture

The current approach presents a data processing schema to be implemented on distributed architectures for Smart Cities, focusing on low capacity Edge devices processing data before using it directly or submitting it to the Cloud. In this section, we present the different parts of such architecture, detailing the main reasons and challenges, the relation among components between the Edge and the Cloud, and how data flows across the system from source to usage.

5.3.1 Edge Computing Networks

Edge computing networks are based on architectures where sensors collect data (e.g. near-by cars, users, events...) and send it to Edge nodes, close to the captured events, to be aggregated and pushed to the Cloud, where data is stored in data warehouses, analytics are made, and services are provided from such analytics. The "Fog" is that part of the architecture embracing Edge nodes receiving data from sensors, Intermediate nodes performing intermediate data aggregation, and Cloud APIs receiving data to be processed and stored. In fact, the *Fog Computing* paradigm, proposed by Cisco [20] towards Smart Cities scenarios, extends the Cloud by moving computation of data (analytics and services) between the Edge and the Cloud.

Fog architectures can be leveraged by distributing the load between the Edge, with low computing power enough to receive data from sensors and minimal processing towards pushing it to the Cloud, and the Cloud itself, with high computing power but far from the sensors, actuators and service end-users [104]. Figure 5.2 shows a Fog infrastructure, with near-data nodes on the Edge, intermediate nodes with medium power to pre-process aggregate or localized data, and the Cloud.

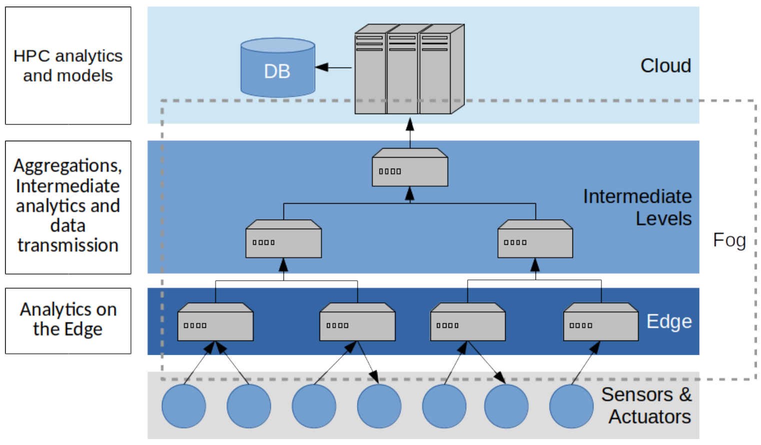


FIGURE 5.2: Schema of the Fog Infrastructure, from Edge to Cloud

Current devices on the Edge are specially designed to consume low power, hence producing low throughput and offering low capabilities (e.g. "Raspberry Pi" or "Arduino" devices [111]). But given the recent interest on Edge computing by public administrations towards Smart Cities, the industry is moving forward computing devices that offer a compromise between low power and high capabilities, oriented towards complementing those Edge devices. Examples are the Intel Movidius "Tensor Processing Units" [106], that provide advanced matrix multiplication operations, focusing on Computer Vision and Deep Learning for Edge devices, or its competitor NVIDIA Jetson [47] incorporating an NVIDIA GPU and CUDA framework.

5.3.2 Edge-Cloud Hierarchies

The two reasons to run load in the Edge are 1) user proximity: when the local edge nodes collect data required by nearby end-users, and that data do not require high performance processing or non-local data. This can also be applied for actuators, devices that need to perform actions according to the information collected from local sensors. Also 2) big volumes of data: when the aggregate of collected data to be transmitted to the Cloud escalates, but as only aggregates are relevant, these operations can be performed at Edge levels.

While Edge nodes are low-powered, intermediate nodes are expected to have more power but also be localized, aggregating sets of Edge (or other intermediate) nodes close in space, and finally Cloud nodes designed for high performance. The decision of where to place data processing, aggregation and analytics depends on the trade-off between "computing and network capacity" and "proximity to user and volume of data to be aggregated". Here we focus on policies that push load to the Edge, taking advantage of already active devices collecting data instead of purchasing additional Cloud resources, also reducing network communication and protecting services from network disruptions by keeping them autonomous on the Edge.

5.3.3 Data Pipelines

Floating Car Data is produced by equipped vehicles that indicate their speed by radio means to listening stations, that is processed into traffic information. Antennas act as sensors transmitting information to their closest Edge nodes, that collect information of local traffic, that can refer to a street, a neighborhood or an urban area, depending on the required localization of analytics. More details on the format and treatment of the FCD is explained on next Section 5.4.

Once data is collected by sensors in an irregular pace (there is no synchronous reporting from antennas to the Edge node), then performing a first aggregation in defined time windows (e.g. minutes). The aggregated data-set contains now a regular time-series, also with default values on empty windows. This regular time-series is used first as training data-set for creating a local model on the node, then new incoming series are passed through the model towards forecasting future windows in short term. Figure 5.3 shows the modeling process.

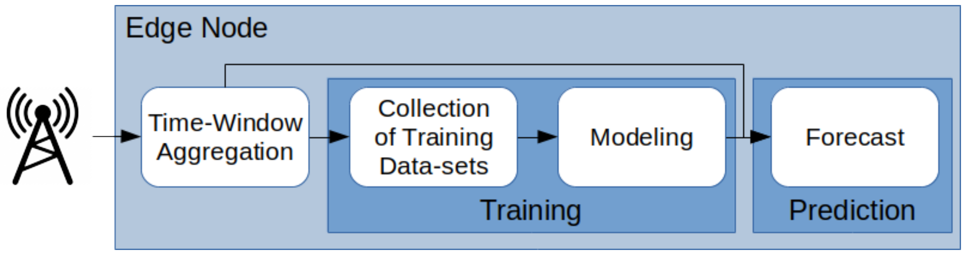


FIGURE 5.3: Pipeline of Time-Window Aggregation, Learning and Prediction

Aggregated data and created models can be pushed up to the Cloud for storage and further analysis and global-scale prediction, or pushed to intermediate nodes for further aggregation with neighbor nodes and create more generalist models, in the previously shown architecture of Figure 5.2. As we observed in previous works, local models can "fit" to local scenarios better than general models in the Cloud [105], avoiding intense communication and network interruption problems.

5.3.4 Floating Car Data

As previously mentioned, FCD is received by antennas covering the City, and can represent a large urban zone, a localized neigborhood, a street or a street segment, depending on which granularity we require. That data is provided to the Edge analytics indicating the received time-stamp for each vehicle transmission, plus the speed of the vehicle (in this case). Data like vehicle position (e.g. GPS) is not provided for privacy and security reasons, e.g. public transportation in Barcelona provides information about buses E.T.A. to stops (adding a minute delay) but never their real position, to prevent vehicles to be closely tracked. The GPS position corresponds to the sensor capturing the data.

This data, arriving asynchronously to our Edge nodes, is the aggregated to produce periodic summaries of traffic information, i.e. Average Speed of vehicles surrounding the node (in Km/hour), and count of vehicles considering that vehicles will report once in the aggregation window time. Considering a first aggregation of 1 minute information as lower bound, we aggregate data reaching the node obtaining data-sets like the presented in Table 5.1.

latitude	longitude	n.cars	μ.speed	timestamp
41.36196	2.09515	4	17	1414365180

TABLE 5.1: Example of Minute-Aggregated Data-set. Notice that position is from the receiving antenna/Edge node, not the vehicle

One-minute aggregation data is only a base for larger aggregations, as traffic time-series can be aggregated from minutes to hours to days, as traffic usually presents a periodic pattern. While large aggregations can be easily predicted due to this periodicity, smaller aggregations can be more challenging. For the validation experiments, in the next Section 5.5, we test different levels of time aggregation (from 5 minutes to 1 hour) to observe the capabilities of models depending on how much precise in time we want to be.

5.4 Methodology

Here we introduce the different elements of the methodology towards Edge aggregation, modeling and forecasting, and also how these methods are distributed, leveraging resource parallelism. We focus on the algorithms used for modeling traffic (training machine learning techniques) and distribution of modeling among resources on the Edge.

5.4.1 Recurrent Neural Networks

The proposed technique used for traffic modeling is Recurrent Neural Networks (RNNs), specifically a configuration known as *Gated Recurrent Unit (GRU)* network. This network is formed by one or more layers of *GRU* and other neural network layers, e.g. densely connected layer for prediction.

This kind of neural network layer is able to deal with time dependencies using a gating mechanism. This gating mechanism is composed by two different gates for each *GRU* layer: the reset gate and the update gate. These two gates affect to the value that the next hidden status S_t will have. This S_t is used for internal calculations as the state of the memory and it is also used as output of the layer. A graphical representation of this kind of layer can be seen in Figure 5.4. The flow of data represented there is a simplification of the following equations. Biases and weights are omitted for the sake of clarity. The box containing r_t is the reset gate, the box containing u_t is the update gate and the one containing $\widetilde{S}_t$ is the proposed new state. In the following equations, x_t represents the input at time t.

First, the reset gate controls the access to the previous hidden state S_{t-1} and is used to compute the new proposed state $\widetilde{S}_t$ as can be seen in Equation 5.3. The reset gate is computed as shows Equation 5.1:

$$r_t = \sigma(x_t W^{xr} + S_{t-1} W^{sr} + b_r) \tag{5.1}$$

Second, the update gate is in charge of how much the state is updated, i.e. which interpolation between S_{t-1} and $\widetilde{S}_t$ we want to do, being the extremes maintaining the previous state or updating it completely. This interaction can be seen in Equation 5.4 The update gate is computed as shows Equation 5.2:

$$u_t = \sigma(x_t W^{xu} + S_{t-1} W^{su} + b_u) \tag{5.2}$$

Finally, the proposed state $\widetilde{S}_t$ is the new hidden state calculated regarding the previous state and the current input. This status is calculated as follows:

$$\widetilde{S}_t = tanh(x_t W^{xs} + (r_t \odot S_{t-1}) W^{ss} + b_s) \tag{5.3}$$

After the gates and the proposed state is computed, we do an interpolation between the current state S_{t-1} and the proposed next state $\widetilde{S}_t$, which represents how much do we change our memory given the current input x_t:

$$S_t = (1 - u_t) \odot S_{t-1} + u_t \odot \widetilde{S}_t \tag{5.4}$$

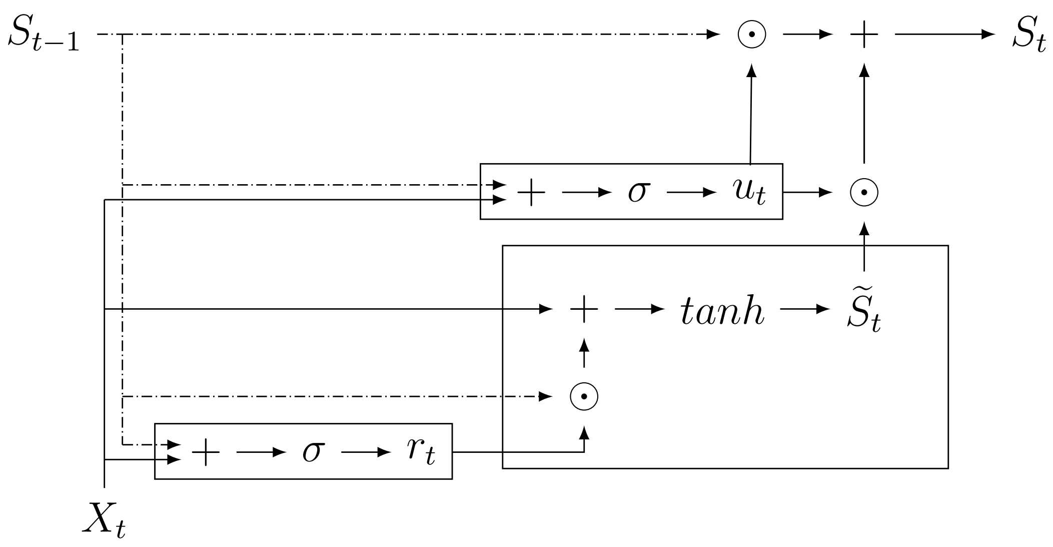


FIGURE 5.4: Schema of a unit of a GRU Recurrent Neural Network layer

Each layer has a given number of units as the one represented in Figure5.4 that have memory and the more units we add, the more different things the layer will be able to remember.

5.4.2 Traffic Forecasting

Having the FCD Time-Series data-set as a matrix of $[time.window \times input.features]$, being the input features $\langle antenna.lat, antenna.long, num.cars, avg.speed\rangle$ and ordered by *time.window*, the forecasting problem targets predicting variables *num.cars* and *avg.speed* of $time.window = t + 1$ from the same variables from $time.window \in [t - d, t]$, being d our *delay* or memory window.

As *time.window* is a whole aggregation period, the problem here addressed has as objective predict the next period of traffic information. Given the capabilities of the GRUs, it could be possible to forecast far from $t + 1$, as GRUs are shown to be capable of medium-term forecasting in many scenarios. GRUs are *generative*, and can generate predictions by using their last prediction and status as input/memory for the next prediction. As for our problem, we are predicting from $t + 1$ up to $t + N$, being N the size of testing data-set in the experiments (approx 1 day in the following experiments).

5.4.3 Training Process Automation

Training models in Machine Learning are usually controlled by a "Training vs. Validation" process, and in Neural Networks (NNs) this is used to decide when to stop the iterative process. Training data is divided in two batches: "training" and "validation" data-sets. The more time a NN trains (the more *epochs* training data-set is visited), the more fitted the model is expected to be, but to the training data! To prevent such *over-fitting*, the validation data-set is predicted at the same time, checking how the model behaves with "non-training" data. While error in training data decreases at each epoch, error in validation data decreases until the point of *over-fitting* and increases from there, as seen in Figure 5.8. That point is considered the "bouncing point", and data-scientists would manually stop iterating at that point. But for non-stable data as we face with FCD, validation can differ enough from training data in certain occasions, and those expected behaviors can not be found during training, and a decision on when to stop iterating must be made.

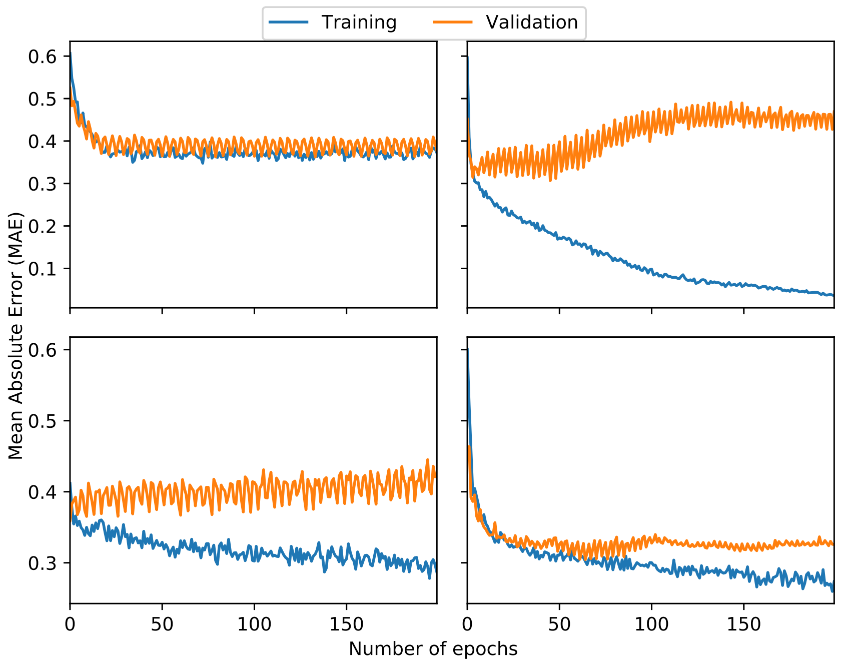


FIGURE 5.5: Different Training NNs Scenarios

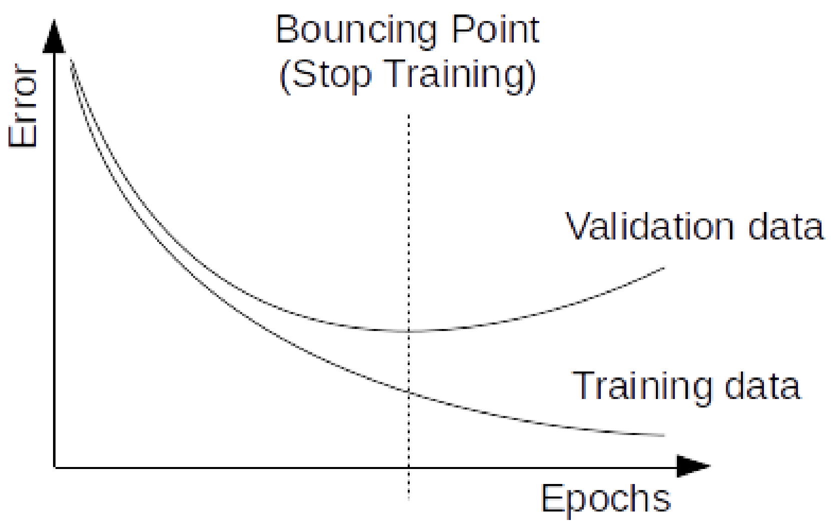


FIGURE 5.6: Training NNs Scenario, where a bouncing point in Validation can be detected and used to prevent over-fitting

When modeling on laboratories with fixed data, finding the optimal stop point can be made manually through several training experiments, or using complex rules to be tested. In our scenario, each Edge node has a model to be trained and manual or complex decisions are not available. For that reason we implemented a set of simplistic rules, to be followed to stop the training from iterating, in case a trivial bouncing point on validation can not be detected. Those cases can be scenarios where validation error does not decrease and the bouncing point is at $Epoch = 1$, the validation error starts below training error, or validation (and training) decrease asymptotically horizontal.

To detect those trends is required to train first, so after several experiments we concluded on a *Max.Epoch*. The GRU trains until that point, detecting the point p of minimal error on training (the "bounce point"). But in case $p = 1$ or $p = Max.Epoch$, meaning that the validation error does not decrease or decreases monotonically, we attempt to detect if 1) there is a crossing point between validation and training error, 2) there is a point where errors (training or validation) decrease below a $delta$ threshold, 3) previous decisions occur before a minimal number iterations considered necessary after observing experiments. Finally, in case previous rules fail, a maximum number of iterations is fixed. Algorithm 1 shows the decision algorithm for fixing p dynamically, using the error on training and validation, previously smoothing both sequences to facilitate treating *wavy* sequences as seen in Figure 5.8, using a LOESS regression method.

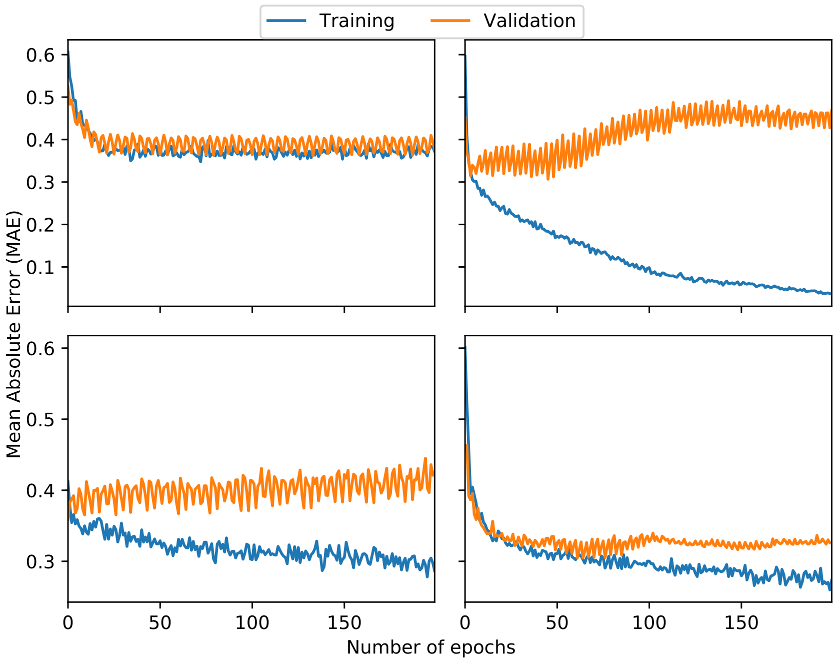


FIGURE 5.7: Different Training NNs Scenarios

When modeling on laboratories with fixed data, finding the optimal stop point can be made manually through several training experiments, or using complex rules to be tested. In our scenario, each Edge node has a model to be trained and manual or complex decisions are not available. For that reason we implemented a set of simplistic rules, to be followed to stop the training from iterating, in case a trivial bouncing point on validation can not be detected. Those cases can be scenarios where validation error does not decrease and the bouncing point is at $Epoch = 1$, the validation error starts below training error, or validation (and training) decrease asymptotically horizontal.

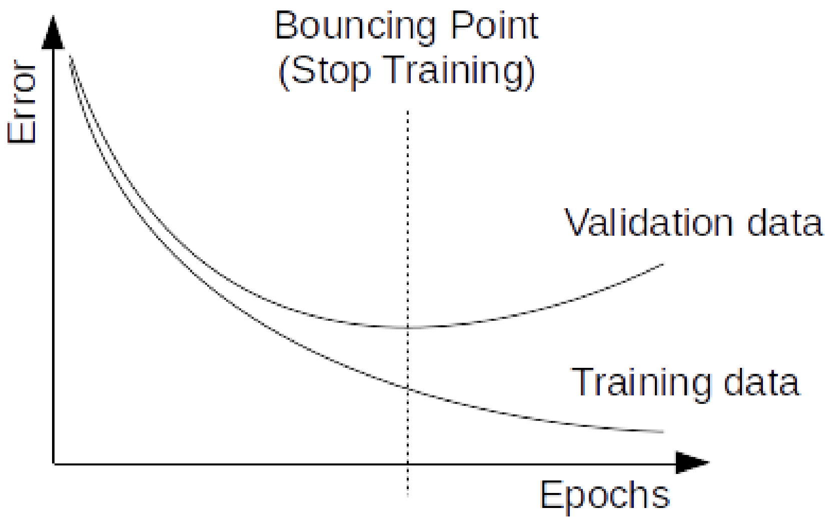


FIGURE 5.8: Training NNs Scenario, where a bouncing point in Validation can be detected and used to prevent over-fitting

To detect those trends is required to train first, so after several experiments we concluded on a *Max.Epoch*. The GRU trains until that point, detecting the point p of minimal error on training (the "bounce point"). But in case $p = 1$ or $p = Max.Epoch$, meaning that the validation error does not decrease or decreases monotonically, we attempt to detect if 1) there is a crossing point between validation and training error, 2) there is a point where errors (training or validation) decrease below a $delta$ threshold, 3) previous decisions occur before a minimal number iterations considered necessary after observing experiments. Finally, in case previous rules fail, a maximum number of iterations is fixed. Algorithm 1 shows the decision algorithm for fixing p dynamically, using the error on training and validation, previously smoothing both sequences to facilitate treating *wavy* sequences as seen in Figure 5.8, using a LOESS regression method.

Algorithm 1 Detecting GRU training cutting-point (epoch) p from training and validation error

```
Result: p point of bounce/intersect/convergence/minimum, prioritizing error on validation over error on training
smooth_tr, smooth_val ← loess_smooth(error_tr, error_val)
 1: if exists_bounce(smooth_val) then
 2:     return bounce_p(smooth_val)
 3: else
 4:     if exists_intersect(smooth_tr, smooth_val) then
 5:         return intersect_p(smooth_tr, smooth_val)
 6:     else
 7:         if exists_bounce(smooth_tr) then
 8:             return bounce_p(smooth_tr)
 9:         else
10:             if converges(smooth_tr, min_threshold) then
11:                 p ← converging_p(smooth_tr, min_threshold)
12:                 return min(p, minimum_p)
13:             else
14:                 return minimum_p
15:             end if
16:         end if
17:     end if
18: end if
```

The process of finding p implies running a certain amount of epochs, to find the trend and detect the bouncing, intersection or convergence point. This process can be substituted by more sophisticated methods that can be done online, although the presented set of rules can be used once to find p given a certain amount of data, and keep that p for future models.

Finally, but not less important, a third data-set ("test") is always kept aside from "training" and 'validation", and used as final evaluation of the model. This step is important as "training" also "validation" are used in the process of training, refining and selecting the final model, that needs to be evaluated with new data not used in any part of the previous process. Then, data is divided in training/validation for creating the model, and testing for checking experiment results. This split is done here by $80\% - 20\%$ as training/validation vs. testing data-sets. As data is a time series, the split is done by cutting data instead of random sampling.

5.4.4 Distributed Model Training

The computational resources available in the Edge are not High-Performance oriented as in the Cloud. Machines tend to be low-power processors with reduced number of cores and threads, aside of limited amount of Memory. While the limited number of processors are mostly used for receiving and transmitting data from sensors to Cloud, spare CPUs can perform aggregation and modeling processes, but never counting with 100% of availability of those CPUs.

For the modeling process we assume single CPU processes, different from modeling on the Cloud where training processes can be spread along large amounts of CPUs without a problem. Parallelization can occur when more than one CPU is available, or when neighbor nodes have spare CPUs available, but not necessary in a complete synchronous environment. Considering the parallel workers as independent agents, we must use a load distribution method that allow independent modeling for later joining separate outcomes.

Breaking down computation on modeling can be done by *Data Parallelism* and *Model Parallelism*. To distribute training over workers (local or remote CPUs and devices) we partition the training-validation data-set, and send it over the available workers. Each worker creates a model from its subset and validates it, then all sub-models are reunited in the original Edge node, and merged. Merging sub-models on neural networks is easier than in other methods, as NNs are composed by a matrix of weights and a vector of biases, that can be aggregated (e.g. averaged) to obtain the "average model". Aggregated models do not need to be exact as a "unified" model created from a single training process, but usually approximate the results of the unified one, as ensembles do with other methods (e.g. Random Forests vs Decision Trees). The resulting aggregated model can either work better for the dilution of noise among sub-models, or either work worse due to over-fitting of each sub-model to its sub-set. For this reason and also good practice, the aggregation model is subject to the testing data-set in the original Edge node, to evaluate the aggregation. Figure 5.9 shows the process of distributed training, merging and evaluation

Some machine learning and NN frameworks include methods to automatically split computation on training, like TensorFlow does, by creating a single model in different stages or data segments distributed among multiple CPUS, useful when all computing is well defined and close (i.e. same machine/cluster). Here we focus better on scenarios where CPUs/GPUs are "disaggregated", meaning that those resources do not need to be in the same machine, but physically distributed in common or uncommon architectures or networks (i.e. an IoT network).

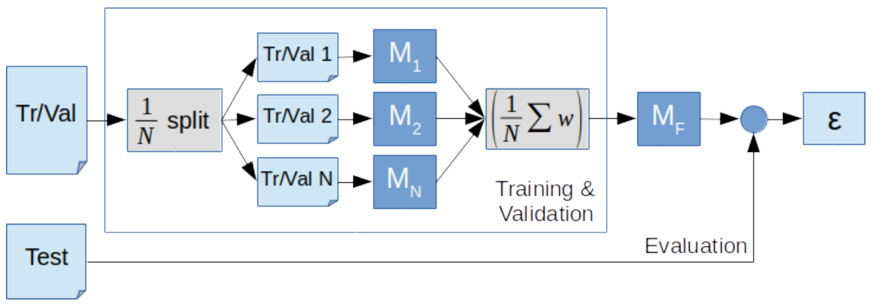


FIGURE 5.9: Schema of Distributed Modeling in Training and Testing Phases. Data is split among N workers, creating N models to be merged creating the *Final* model to be evaluated

Distributing data among nodes is trivial as we do not introduce load balancing for the current case of use. For data distribution among processes (workers), training/validation data is split in N parts, where N is the number of workers to be deployed. The split of training vs. validation is done following the criteria $80\% - 20\%$ as training vs. validation, for every subset. Each worker splits its data to train and validate its model. The test set (a 20% of the total data) is kept for the aggregated model for evaluating the final model.

Finally, there is an optimization step that can be performed on distributing the learning process, by reducing the number of epochs per worker. With less data to model and same amount of epochs than the original problem, workers are expected to overfit, and an option is to reduce the amount of epochs at each worker. We expect that each worker will require less iterations to model fewer data, reducing training time while maintaining the validation and test error. Then, when distributing data, modeling algorithms receive $(epochs \leftarrow 1/N \times epochs)$ for training.

5.5 Evaluation

The presented approach is evaluated with a series of experiments to test the capacity of learning the proposed traffic data, distributing the modeling task among processes, automatically tuning part of the training process, and comparing with other valid techniques.

5.5.1 Evaluation infrastructure

Implementation and evaluation have been performed using Tensorflow and Keras frameworks for Deep Learning, using R as interface and Python TF + Keras as a learning engine. The infrastructure to run and measure training times corresponded to a single thread Xeon processor for comparison experiments among different training configurations, an NVIDIA Jetson Nano (ARM processor + NVIDIA GPU) for testing IoT+GPU environments.

The outcomes of the presented problem depend principally on the trade-offs between the model error, the model training time, the number of epochs for training, the NN hyper-parameters like the number of hidden units, the level of aggregation of data in time, and the distribution of load among processes/processors. The model error depends on the amount of data and number of training epochs; the training time depends on the number of epochs and the amount of data; the number of epochs determine the error and training time; the number of hidden units affects the training time; the level of aggregation determines the amount of data also may affect the achievable accuracy and error when modeling; and the distribution of load defines the amount of data per process.

The evaluation will focus on the following experiments:

1. The effects of training the RNN with different number of Hidden Units {2, 4, 8, 16, 32} and different number of epochs, and check the usefulness of determining a stop-point p dynamically using the presented set of rules versus fixing a large enough p *a-priori*.
2. The comparison and trade-off between training epochs vs. hidden units vs. resulting error vs. level of time aggregation ({5, 10, 15, 20, 30, 60 min.}).
3. The effects of distributing the training process among N different processors {1, 2, 3}, considering a low range for N matching the dimensions of common low-power devices.

The goal of studying such trade-off is to find a point where we meet good results (error), in reasonably few time (training time/epochs), using the appropriate available resources (parallelism).

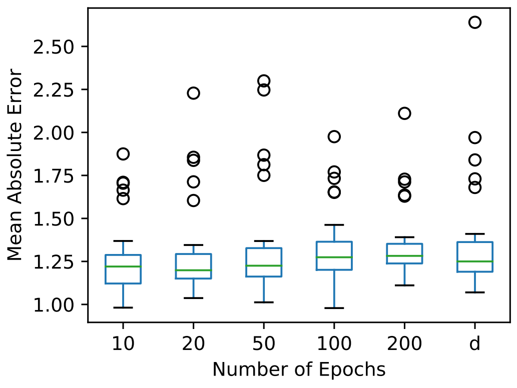
5.5.2 Detecting the Required Training Epochs

First of all, we must evaluate the capability of the neural network to learn the target time-series, concerning *Volume of Traffic (Cars)* and Average Speed of Traffic (Speed). The following experiments evaluate in a grid search-like the Mean Absolute Error (MAE) of the model when trained using a different number of hidden units in the GRU, and different levels of aggregation in time (from 5 to 60 minutes). As previously explained, the model is trained using the *Training* data-set, the *Validation* data-set is used for selecting the best p value (stop-point for training) on the dynamic/automatic scenario, and the *Test* data-set is used to obtain the evaluation MAE on a range of different number of training epochs plus the selected dynamic stop-point.

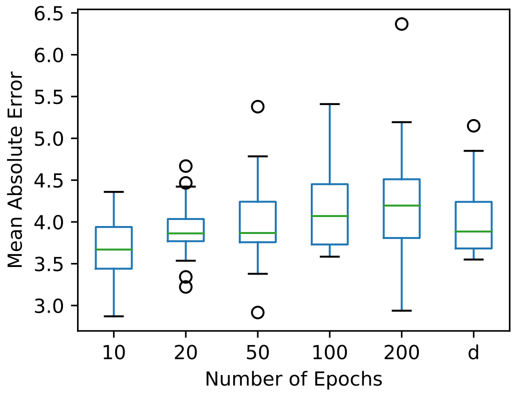
Figure 5.10 shows the error distribution for different training epochs over the *Test* data-set and the dynamic stop-training point. While for the volume of cars, learning seems to be easy with a MAE around $1 - 1.3$, predicting the speed of cars becomes more complex with a more volatile error $3 - 5.5$, (the variability of the traffic speed on those data-sets is already known from previous works [105]). While most of training is done on the first epochs of the different tested NN configurations, selecting automatically a stop-training point becomes conservative respect the best option, but performs almost as good as the optimal having into account the automation.

Observing the different results of individual experiments we realized the difficulty to establish a set of rules that match every single training-validation scenario. Selecting the best number of epochs is still an open problem that can be automated with more complex mechanisms, but for the current scenario where quick decisions must be made, the current set of rules becomes an adequate solution. We plan to focus on the selection of training time trade-off *error* vs *method complexity* for future work following the current contributions.

Finally, this set of experiments show the capability of learning both traffic attributes on different ranges of time aggregation and NN *power*. The next experiments will study in detail the effect of different time aggregations and NN configurations.



(A) MAE for different epochs and dynamic *d* on Volume of Cars



(B) MAE for different epochs and dynamic *d* on Average Speed

FIGURE 5.10: Comparison of Mean Absolute Error (MAE) for a various static number of epochs with a dynamic number of epochs for estimating the number of cars and average speed. Here *d* represents a dynamic number of epochs

5.5.3 Model Performance and Tuning

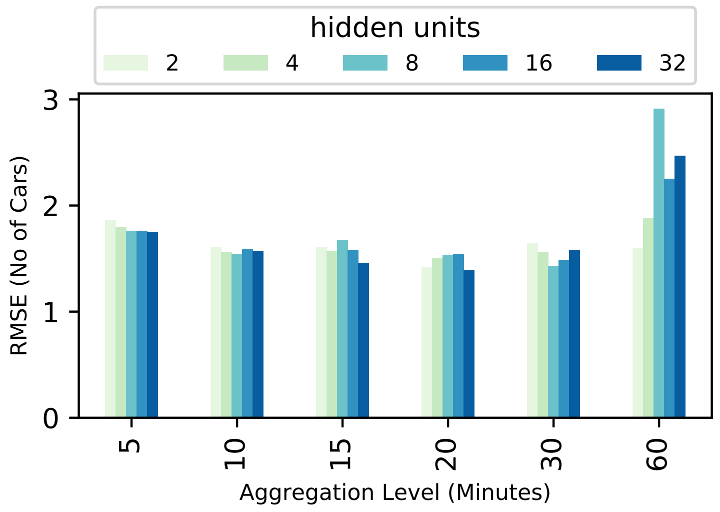
The following experiments focus on the evaluation of the different configurations for time aggregation and NN hyper-parameters (here hidden units). Previous experiments were done to find a proper value for most of NN hyper-parameters (learning rate, GRU delay, etc.), leaving as free parameter the number of hidden units as the one affecting the most, also affecting the computing requirements, crucial for our low-power scenario. We also include the decisions on *Number of epochs* for training process and their effect on the model error.

As previously indicated we consider parallelism as a viable solution to cut training time, thus we divide experimentation in two approaches: a single-model method ($N1$) and a parallel-model method ($N2, N3$).

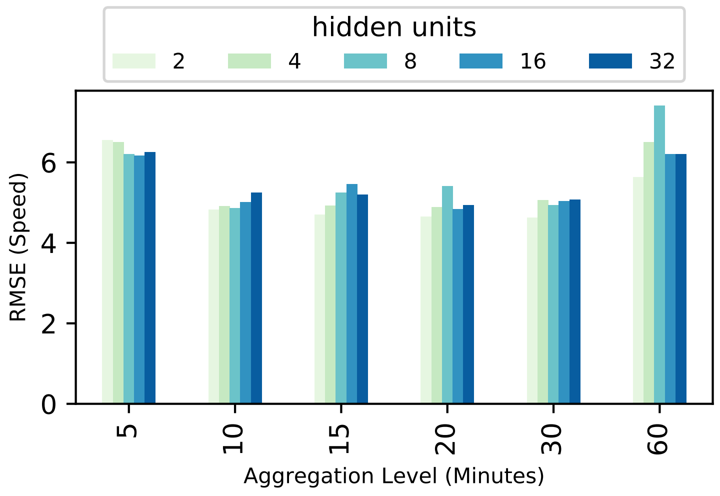
Single Model (N1) Evaluation

Single-model (N1) method is evaluated by running the different configurations for hidden units and time aggregation levels, checking for the Relative Mean Squared Error (RMSE) in this case. For the selection of the number of epochs, the dynamic stop-point p is chosen. The baseline system is the single-CPU Xeon machine for comparison between configurations, to later be compared with the corresponding low-power devices.

Figure 5.11a shows the RMSE for estimating the number of cars on test data for 2, 4, 8, 16 and 32 hidden units with 5, 10, 15, 20, 30 and 60 minutes aggregation levels. We observed the aggregation yields stable behavior until 30 minutes aggregations as the RMSE remain under 2. However, with 60 minutes aggregation level, we observed a significant decrease in accuracy. This is because of a higher level of data aggregation, the underlying fine-grained details are hidden, and the model cannot learn from data accurately. We observed the effect of changing the number of hidden units does not have any significant effect on accuracy. The average RMSE of estimating the number of cars remain in between 1 to 2 except aggregation level of 60 minutes.



(A) Error vs. Hidden units vs. Time aggregation for number of cars estimation



(B) Error vs. Hidden units vs. Time aggregation for speed estimation

FIGURE 5.11: Effect of using different number of hidden units with various aggregation levels for number speed of cars estimation

Figure 5.11b shows the RMSE for estimating the speed of cars on test data for 2, 4, 8, 16 and 32 hidden units with 5, 10, 15, 20, 30 and 60 minutes aggregation level. We observed the high error for aggregation levels 5 and 60; however, for other aggregation levels, it remains similar. We do not observe any noticeable accuracy gain for using a different number of hidden units. The average RMSE of estimating the speed of cars remain between 4.5 to 5.5 except aggregation level of 5 and 60 minutes.

From this set of experiments, we identify the appropriate aggregation level and hidden units to be used in the final model, and we compute average RMSE of speed and number of cars on test data for $(2, 4, 8, 16, 32)$ hidden units with $(5, 10, 15, 20, 30, 60)$ minutes aggregation levels. Figure 5.12 shows the average RMSE for estimating the speed and number of cars. Each level of aggregation has its optimal number of hidden units, meaning that there is no optimal configuration able to deal with all levels of aggregation, a desirable state allowing us to decide precision of the time interval. Making the decision of selecting a time interval where we can trust predictions the most, we observe that the 2 hidden units with 20 minutes aggregation level yield the minimum error as compared to the other configurations. Therefore, in the rest of the experiments, we used 2 hidden units and 20 minutes of aggregation level.

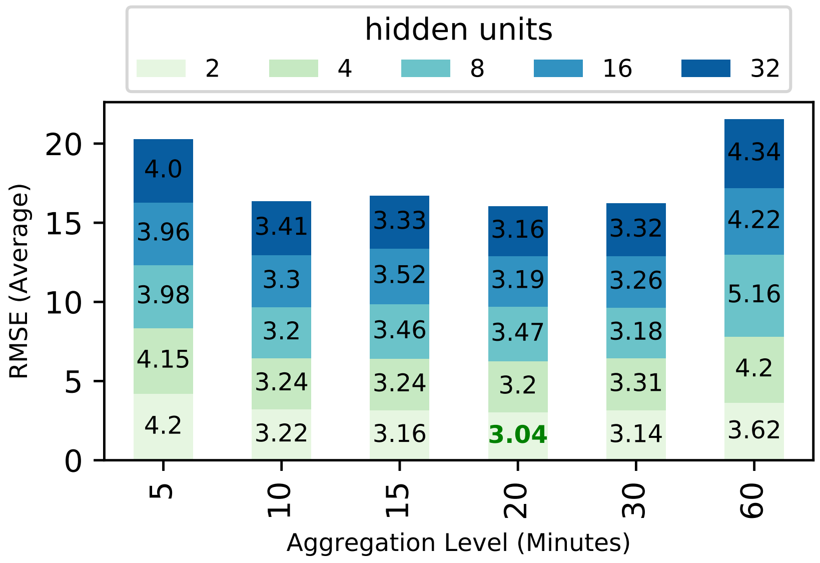
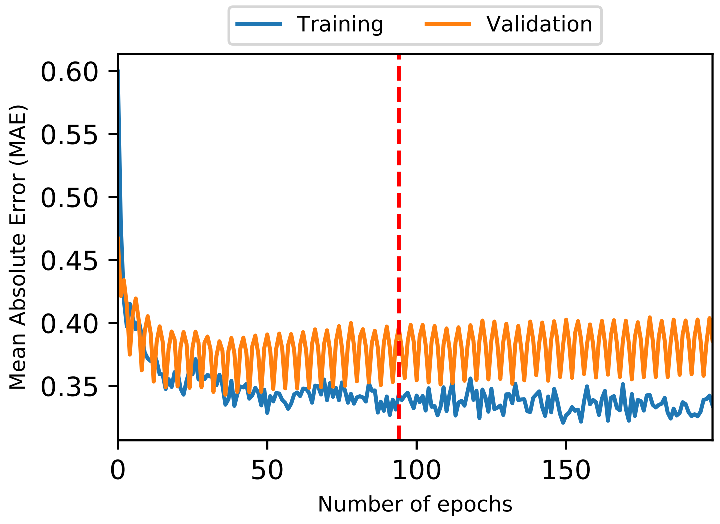
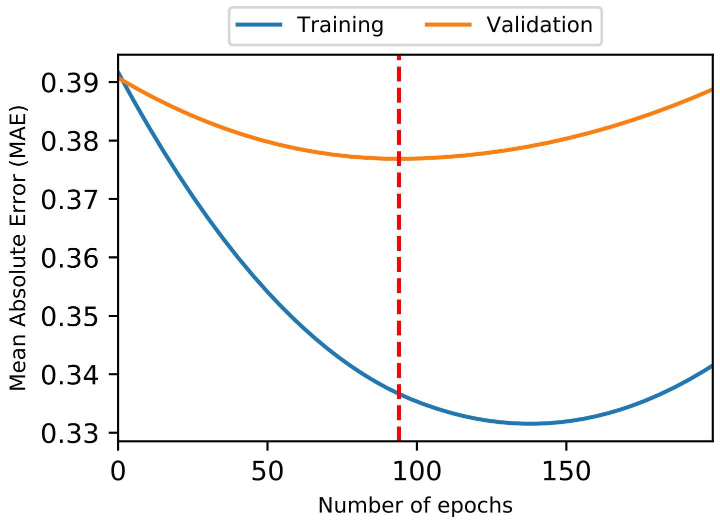


FIGURE 5.12: Comparison of Average RMSE using different number of hidden units with various aggregation levels

The number of epochs is selected using the dynamic mechanism, and in the case of 2 hidden units and 20 minute aggregation, this is 94 epochs as a bouncing point in validation smoothed MAE as shown in Figure 5.13a. Additional experiments were made manually tuning the epochs from 10 to 200 to validate the automatic decision. Figure 5.13b zooms into the smoothed fitting regression from Figure 5.13a behavior on validation. Remember that while training goes below the optimal validation error, we use validation as non-training data to make non-overfitted decisions. As an example scenario, the experiments on parallelism $(N2, N3)$ in the following set of experiments will take the parameters on the best case for $N1$, 2 hidden units, 20 minute aggregation, 94 training epochs.



(A) Effect of number of epochs on the MAE for training and validation



(B) Zoomed in version with respect to y-axis and show the interval $0.33 - 0.39$. It can be observed that at 94 number of epochs the validation data bounced back, selecting it as training stop-point

FIGURE 5.13: Selection of the Training number of Epochs

Comparison of Single Model (N1) with Parallel Models (N2 and N3)

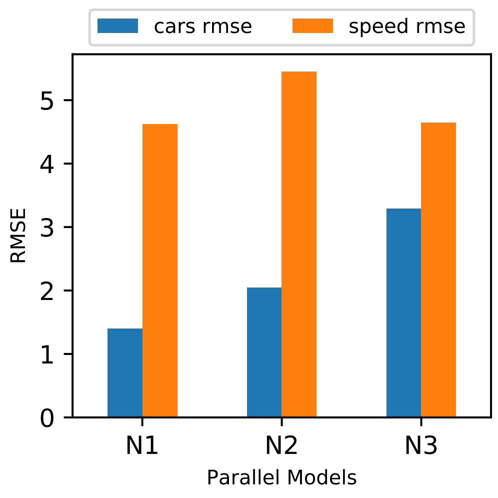
Next experiments focus on the capacity of breaking down the training model between computing units (CPUs and GPUs). We focus two different scenarios here, where the processes can work in multiple CPUs in the same place (e.g. Tensorflow working with multiple CPUs in the same machine), or a scenario where CPUs are disaggregated and become independent between each other (e.g. different edge nodes cooperating). The best configuration from the previous $N1$ exploration are used for the parallelism comparison, with the addition that when distributing the data to be modelled we are applying the $1/N$ factor to the number of training epochs as the “proposed epochs”.

Table 5.2 shows the comparison of RMSE for number of cars and speed with fixed and proposed number of epochs for different parallel models (N1, N2 and N3). We observed that for N2 and N3, using fixed number of epochs results in increase of RMSE due to over-fitting of the model. We also observed that training time did not change even we distribute the input data to be processed by more than one model. This is because of number of epochs are same for each configuration. However, we observe a significant decrease in training time when we used proposed number of epochs which are obtained by dividing the optimal fixed epochs for $N1$ by number of parallel models. We observed that RMSE is slightly increased for estimating number of cars and it remains almost stable for estimating speed of cars.

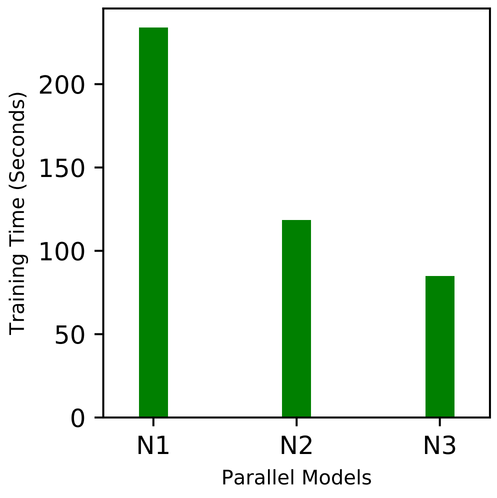
Model	**RMSE (Cars)**		**RMSE (Speed)**		**Time (Sec)**	
	Fixed Ep	**Prop. Ep**	**Fixed Ep**	**Prop. Ep**	**Fixed Ep**	**Prop. Ep**
N1	1.40	1.40	4.62	4.62	233.71	233.71
N2	3.25	2.05	7.84	5.45	237.79	118.32
N3	6.25	3.29	6.83	4.64	229.53	84.69

TABLE 5.2: Comparison of error and training time with fixed number of epochs ($FixedEp$) and proposed number of epochs ($PropEp$)

The effect of using parallel models (N1, N2 and N3) on the number of cars and speed accuracy is shown in Figure 5.14. We observed in Figure 5.14a that accuracy to estimate the number of cars slightly decrease with the increase of number of parallel models used to train the input data. This is because of models are trained on less data and are more specific to particular input set due to which when we combined them and use the final model to predict the number of cars on test data, we observe this behaviour. However this does not effect the accuracy of predicting speed of cars and it somehow remain stable regardless of number of models used to train the input data set. We also observed that on average we are loosing some accuracy but we are saving more than 50% of training time when we used parallel models as shown in Figure 5.14b.



(A) Effect of parallel models on accuracy



(B) Effect of parallel models on training time

FIGURE 5.14: Comparison of RMSE vs Training time for different parallel models(Nx) with number of epochs divide by x where x=1,2,3

In respect to the speed-up comparison for parallel models (N2, N3) with respect to N1, Table 5.3 shows the improvement factor for error and time. We observed that when we distribute the input data set to be processed by 2 models. there is decrease of 115.39 seconds in training time with the increase of 0.65 and 0.83 for number of cars and speed accuracy. Similarly we observed that 149.02 of training time decrease when use use three parallel models with the increase of 1.89 and 0.02 of RMSE for number of cars and speed.

Model	RMSE (Cars)	RMSE (Speed)	Time (Sec)
N2	0.65	0.83	-115.39
N3	1.89	0.02	-149.02

TABLE 5.3: Comparison of change factor in error and training time for N2 and N3 w.r.t N1

Finally, Table 5.4 shows the RMSE and training time for single model (N1) with the different configurations of number of cores which are used to train the input data set. We observed that changing number of cores does not effect the training time however it almost maintain the accuracy. This is because of creating single model which is used to make the predictions.

Cores	RMSE (Cars)	RMSE (Speed)	Time (Sec)
1	1.40	4.65	233.71
2	1.43	4.65	257.41
3	1.55	4.61	219.76

TABLE 5.4: Comparison of using multiple cores with single model (N1)

5.6 Final Consideration

Through experimentation we observe that the presented methodology allows to bring this kind of predictive analytics to the Edge, including complex learning mechanisms like GRUs, allowing cost/effective compromises between model accuracy and training times at low powered or restricted devices, and can be enhanced by distributing load among extra CPU/GPU resources when available. Additionally, compared to previous prediction methods, GRUs achieve good accuracy results with constrained training time in front of the previously used methods, even if the modeling process is split to reduce training time.

Chapter 6

SECURITY

The Internet of Things (IoT) is a new computing paradigm that integrates traditional vertically embedded and horizontally distributed systems into a continuum that extends from the physical word, near the edge, to the resource-intensive computing, toward the Cloud. The development of the IoT requires advancements in a variety of neighboring technologies, including near-field communication (NFC), 5G wireless systems, and radio-frequency identification (RFID), which play central roles in data sensing as well as in near-equipment processing. As the resource capacity of near-edge IoT devices is naturally insufficient to cope with the influx of sensed data and its processing requirements, integration of the IoT system with cloud services allows opportune offloading of excess work. In several domains, the data must be protected before being offloaded to the cloud. However, for the sake of privacy, the data must be protected before being offloaded to the cloud, as required for example in health care, transportation, and security-sensitive applications. Moreover, the resource limitations of IoT devices require fitting the encryption algorithms of interest to the corresponding constraints. Addressing this requirement opens a new front of challenges in the field of security and privacy. In this work, we propose a novel lightweight attribute-based encryption scheme based on both dynamic and static attributes suited for IoT systems. In particular, we consider the revocation of a user's access permissions, and present solutions to this problem. Our proposal provides for higher security guarantees than the state-of-the-art protocols thanks to the use of elliptic curve cryptography (ECC).

6.1 Motivation and Background

From the communications perspective, the Internet of Things (IoT) is a new technology in which connectivity is provided by radio waves near the edge, where the system borders the physical world, and then through communication networks, such as the Internet, as we move away from the edge toward the system core.

The Internet of Things is an orchestrated ensemble of networked near-physical objects, each of which is provided with a variety of sensors and actuators. This ensemble interacts intensively with the physical environment, capturing data from it and transferring them to computing devices capable of drawing intelligence from them and returning consequent actuation commands. The computational intelligence of many such devices resides in the Cloud and is reached via the Internet.

The connectivity among objects is wireless. Each such object is a small physical unit (e.g., RFID tag), with important limitations in power, memory, security and access control capacity. It is this limitation that makes the Cloud an essential companion to the Internet of Things (IoT) architecture. Figure 6.1 depicts the IoT architecture as divided into three stacked layers [71]. At the bottom we have the perception layer, which is the core of the IoT. This layer is responsible for identifying objects and collecting the data sensed from them. Examples of such objects include RFID, WSN, GPRS. The middle layer is called the network layer, which provides transparent transmission capabilities to transfer the collected data to specialized application software through the Internet. At the top we have the service or application layer, where value-added enterprise applications operate [140].

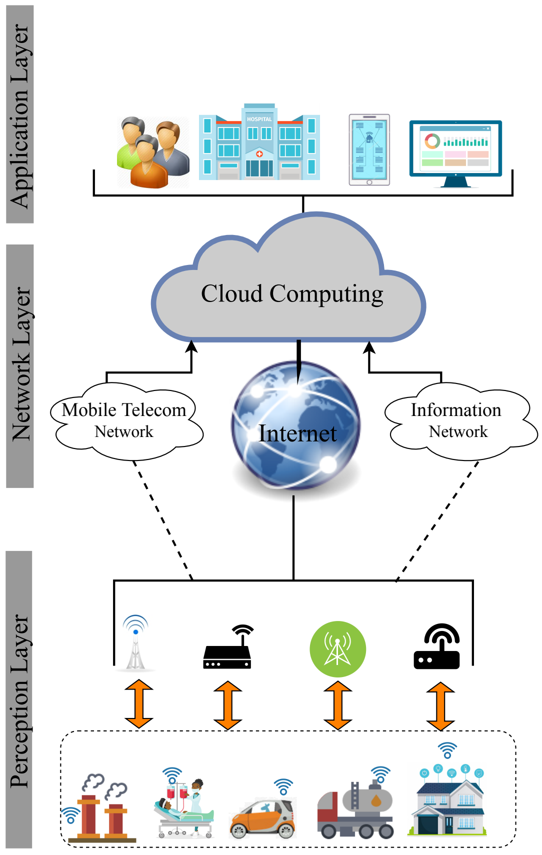


FIGURE 6.1: A conceptualized view of the IoT architecture.

In these systems, the information is transferred through wireless networks, which are intrinsically exposed to security threats that include malicious interference with or unauthorized access to the transmitted data. It therefore follows that opportune data encryption techniques and appropriate access control policies have to be contemplated to provide acceptable security for these systems.

Attribute-Based Encryption (ABE) has been widely used to cope with encryption scheme with access control capability to address these needs, in the two variants of Key-Policy ABE (KP-ABE) [60] and Ciphertext-Policy ABE (CP-ABE) [18]. In ABE, every user has the ability to decrypt the encrypted data which has its required attributes at that time. The schemes that use ABE employ bilinear pairing operations, which are heavy weight and costly [141]. Hence, the need arises for solutions that can assure security and confidentiality while being lightweight in terms of resource requirements. Some of the proposed schemes in this regard employ the Elliptic Curve Cryptography (ECC) [97, 77], which are more convenient than RSA-based schemes in implementation and security aspects. However, the schemes currently proposed to this end only consider static attributes and do not support dynamic ones, such as location, time, temperature, and noise, in the implementation of access control policies.

The exponential growth of the IoT, caused by the opportunity of leveraging smart devices in factories, plants and generalized enterprise settings, motivates the quest for novel approaches to provide privacy and security that must be lightweight enough to suit the resource constraints of edge devises as well as providing access control to support dynamic attributes (e.g., location, temperature, time) in addition to static ones (e.g., role, degree, identity).

Attribute based encryption was first introduced by Sahai and Waters in 2005 [116]. In 2006, Goyal *et al.* [60] and Bethencourt *et al.* [18] defined the concept of ABE, which may use a private key or encrypted text solutions. The two options of the cited model gave rise to KP-ABE and CP-ABE schemes, based on bilinear pairing. In KP-ABE, the access policies are in the private key, and the text is encrypted with a set of attributes. In CP-ABE the access policies are encrypted in the text, and the attributes are in the key of each user. All of those works, however, solely focus on fulfilling access control policies, and ignore the resource constraints of the IoT.

In [141], the authors propose a lightweight scheme that is appropriate for use in the IoT systems. This scheme is based on elliptic curve cryptography (ECC) that does not use bilinear pairing, and does not support dynamic attributes. Li *et al.* [86] presented a new scheme that includes dynamic attributes in addition to static ones. The authors show that their proposed scheme is more secure than the previous schemes that are only based on static attributes. However, this solution is not lightweight. In [130], the author propose a lightweight KP-ABE scheme called C-KP-ABE for cloud-based IoT applications. More recently, the authors in [16] propose a policy update ABE (PU-ABE) scheme that supports access policy update. In these two works, however, the authors use bilinear pairing which is not lightweight and cannot be used for IoT systems. This work addresses this limitation by proposing a lightweight scheme based on dynamic and static attributes that warrants high security through the use of elliptic curve cryptography (ECC).

6.2 Preliminaries

This section introduces the basic concepts needed to understand the core contents of this paper. It does so using the notation summarized in Table 6.1.

TABLE 6.1: The symbols used in this paper.

Notation	Description
p, q	Large prime numbers
Z_q	A finite integer field with elements set $\{0, 1, \ldots, q\text{-}1\}$
$x \in_R Z_q^*$	x randomly chosen from Z_q^*, where $Z_q^* = Z_q\text{-}\{0\}$
G	A generator of elliptic curve group
$HMAC(a, k)$	A hash function which gets the value a and key k to generate an authentication code for a
$h(\cdot)$	A one-way hash function
$ENC(m, k)$	A symmetric encryption algorithm which encrypts m using k
$DEC(C, k)$	A symmetric decryption algorithm which decrypts the cipher-text C using k
AA	An attribute authority center
$KDF(\cdot)$	A key derivation function
$M_T(a, b)$	A mapping function, if a-b is less than threshold T then the output is 1; otherwise the output is 0
$params$	Parameters of the public key in ABE
γ	The static set of attributes
σ	The dynamic set of attributes
Γ_r	The access tree with root r

6.2.1 Elliptic Curve Cryptography

ECC was first introduced by Miller [97] and Koblitz in 1985 [77]. Although RSA [115] and ElGamal [46] are regarded as the highest-security algorithms, their security is achieved at the expense of using large keys and heavyweight operations. ECC-based algorithms achieve the same level of security with a much shorter key, and therefore attain equivalent reliability with faster processing [61, 62].

In most ECC implementations, encoding is based on Elliptic Curve Discrete Logarithm Problem (ECDLP), in which, with points P and $Q = k \cdot P$, computing k has exponential complexity and it is a hard problem.

Moreover, ECDH is the Diffie-Hellman key exchange protocol over elliptic curves (EC), a shared key between two entities with pairs $(s_A, P_A = s_A.G)$ and $(s_B, P_B = s_B.G)$, which are using same EC with the parameters (q, a, b, G, p) are calculated as $K_{A,B} = s_A.P_B = s_B.P_A = s_A.s_B.G$. This shared secret key can be used to transmit symmetric encryption keys safely.

6.2.2 Elliptic Curve Integrated Encryption Scheme

In this paper, we use the Elliptic Curve Integrated Encryption Scheme (ECIES) to provide for confidentiality and data integrity. The scheme presented in [57] uses the (ECDH) and the authentication code, which is based on the MAC function to authenticate data. The encryption and decryption process of ECIES are presented in Algorithms 2 and 3 in which KDF is a key derivation function that is responsible to produce a set of keys from keying material and some optional parameters (see [57] for more detail definitions).

Algorithm 2 ECIES encryption algorithm.

```
Input: Message $m$
Output: $R, C, MAC_C$
1: Randomly select integer $r$ from [1,$p$-1]
2: $R \leftarrow r \cdot G$
3: $K \leftarrow r \cdot P_B = (K_X, K_Y)$
4: if $K = O$ then go to Step 1
5: else
6:     $k_{ENC} \| k_{MAC} \leftarrow KDF(K_X)$
7:     $C \leftarrow ENC(m, k_{ENC})$
8:     $MAC_C \leftarrow HMAC(C, k_{MAC})$
9:     return $R \| C \| MAC_C$
10: end if
```

6.2.3 Secret Sharing Scheme

We use Shamir's (t, w)-*threshold* secret sharing scheme proposed in 1979 [121]. Shamir's (t, w)-*threshold* scheme is based on interpolated polynomial and progresses across the three phases as described below.

- *Initialization phase*: the special entity as a dealer chooses $s \in Z_p^*$ as a secret and chooses w distinct non-zero elements of $x_i \in Z_p$, $1 \leq i \leq w$. Then, the attribute-authority center AA sends these elements to the w entities.

- *Share distribution phase*: subsequently, the dealer chooses $t-1$ elements $a_j \in Z_p$, $1 \leq j \leq t-1$ and computes $y_i = a(x_i) = s + \sum_{j=1}^{t-1} a_j x_i^j \ mod\ p$. Eventually, the dealer gives $v_i = x_i \| y_i$ to the *i-th* entity.

Algorithm 3 ECIES decryption algorithm

```
Input: $R, C, MAC_C$
Output: $m$ or $\perp$
1:  if $R$ is not on the elliptic curve then return $\perp$
2:  else
3:      $K \leftarrow s_B \cdot R = (K_X, K_Y)$
4:      if $K = O$ then return $\perp$
5:      else
6:          $k_{ENC} \| k_{MAC} \leftarrow KDF(K_X)$
7:          if $MAC_C \neq HMAC(C, k_{MAC})$ return $\perp$
8:          else then
9:              $m \leftarrow DEC(C, k_{ENC})$
10:             return $m$
11:         end if
12:     end if
13: end if
```

- *Secret reconstruction phase*: at this point, a group of at least t entities can reconstruct s by executing the Lagrange interpolation equation. $s' = f(0) = \sum_{i=1}^{t} y_i \prod_{k=1,k\neq i}^{t} \frac{x_k}{x_k - x_i} \; mod \; p$.

6.2.4 Access Tree

Our solution uses an access tree [60] to describe access policies. Let us now briefly describe how that works.

For an access tree Γ, each non-leaf node is a threshold gate described by its children and a threshold value d_x. For each node x with num_x children, the value of d_x is within the range $0 < d_x \leq num_x$; in particular, for $d_x = 1$ the threshold gate is an OR gate, whereas for $d_x = num_x > 0$, the threshold gate is an AND gate. Moreover, each leaf node x of this access tree has a threshold value $d_x = 1$ and is assigned an attribute. In accordance with [116], the parent of each node x in the access tree is denoted by the $parent(x)$ function, whereas the attribute of a leaf node e is returned by the $att(e)$ function. In this tree, each children is assigned a node number between 1 and num_x, and the $index(x)$ function returns this value for node x.

Assume that Γ_r is an access tree with root r and Γ_x is the sub-tree of Γ_r with root x with children x'. The value of the $\Gamma_r(\gamma)$ is recursively calculated for Γ_r as below. First, for the leaf node e, $\Gamma_e(\gamma) = 1$ if and only if $att(e) \in \gamma$. Then, the value $\Gamma_{x'}(\gamma)$ is calculated for each children x' of x and $\Gamma_x(\gamma) = 1$ if and only if the minimum number of d_x children nodes returns the value of 1. So, if the attribute set of γ satisfies the Γ_x tree, then the value of the $\Gamma_r(\gamma)$ is equal to 1.

6.2.5 Formal Key-policy attribute-based encryption (KP-ABE)

Our solution uses attribute-based encryption with the key policy [60]. The KP-ABE scheme is composed of four sub-algorithms, namely Setup, Encryption, Key-Generation, and Decryption. In the following we briefly explain the working of each such sub-algorithm.

Setup is a random algorithm run by attribute authority AA Center. This algorithm generates public key PK parameters and master key MK. It publishes PK parameters as an output and holds the value of MK to itself.

Encryption$(m, \gamma, params)$ is a random algorithm run by a sender. In this algorithm, a public key PK parameters, message m, and attributes set γ are defined as an inputs and an encrypted message C as output.

Key-Generation (Γ, MK) is a random algorithm run by AA. This algorithm receives the master key MK, and access structure Γ as an input and returns the decryption key D as an output.

Decryption $(C, D, params)$ is an algorithm executed by a receiver, in such that it receives encrypted message C, a public key PK parameters, and a set of attributes γ, to which a text has been encrypted as an input. It receives the decryption key D that is encrypted for the access structure Γ and, if the condition $\Gamma(\gamma) = 1$ holds, then it returns the message m as an output.

6.2.6 Dynamic Attributes

As noted earlier, in this work we consider not only static attributes, but also dynamic ones, such as location, temperature, time, noise. Using dynamic attributes may require adding another security layer in the internal architecture of the IoT node.

The benefit of using dynamic attributes is easily explained as follows. Assume, for the sake of the example, that a message receiver is a mobile device that uses only static-based policies. In the event that an unauthorized user compromises this entity (e.g., by stealing the device), the attacker can decrypt the data encrypted with access policies based on static attributes at any location and time. Conversely, if dynamic attributes were used in addition to static ones, to decrypt the encrypted data the unauthorized user would need to access further dynamic information besides the decryption key, which is a much more difficult attack.

For a mobile set, this functionality could be provided with a "behavior-profiling" app, as discussed in [87], together with a security and privacy analysis of it.

As shown in Figure 6.2, the app is installed within a secure container using software platforms such as KNOX or BES12 [118, 19], and updated periodically after installation. The values to be assigned to such dynamic attributes are obtained from the raw sensors available in the smartphone (e.g., GPS, RFID).

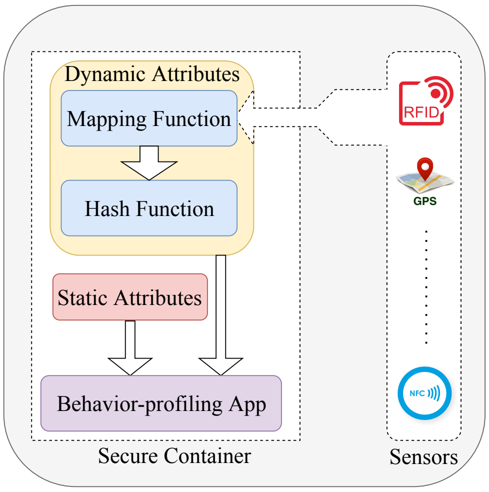


FIGURE 6.2: High-level architecture of the proposed algorithm for use on smartphones.

The app includes a Mapping function $M_T(\cdot)$ that performs a linear comparison. $M_T(a, b)$ takes two inputs: one from a selected sensor in the smart device (i.e, location); the other from the data owner embedded securely within any smartphone app. $M_T(a, b)$ compares those inputs, returning the yield of the Boolean expression. So, if *a*-*b* is less than threshold T then the output is 1; otherwise the output is 0. As an example, for the location, if the implemented location by the data owner is $Italy$ and the location collected from GPS is $Spain$. $M_T(Italy, Spain)$ evaluates to 0 and if the location collected from GPS is $Italy$ the output of the $M_T(Italy, Italy)$ evaluates to 1.

6.3 The Proposed Scheme

This section describes our proposed solution, which employs lightweight cryptographic scheme based on dynamic attributes for IoT systems. As shown in Figure 6.3, our solution is comprised of the following components:

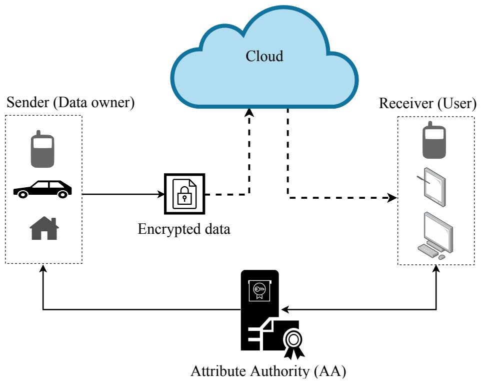


FIGURE 6.3: The main components of the proposed scheme.

Attribute Authorities (AA): this is a trusted authority that has the task of establishing the Setup and Encryption phases in a secure environment. In this scheme, a user needs to prove her attributes to AA to receive the decryption key for each attribute from the corresponding authority.

Sender (data owner): this is assumed to be a smart device such as a smartphone. It encrypts outgoing data using the access policies based on static attributes obtained from AA together with its dynamic attributes. In that manner, the sender can dispatch encrypted data to the receiver, using appropriate Cloud services.

Receiver (user): this is assumed to be a smart device, too, which accesses encrypted data incoming through the Internet, and uses the decryption key obtained from AA together with its own dynamic attributes for decryption.

The proposed scheme is based on the ECC, whose secure parameters are (q, a, b, G, p).

In this scheme, the secret key for the static attributes set γ, is obtained by Shamir's secret sharing scheme presented in Section 6.2, in which the Lagrange coefficient $\Delta_{i,\gamma}$ is computed by executing equation $\Delta_{i,\gamma} = f(x_i) = \prod_{j \in \gamma, j \neq i} \frac{x_j}{x_i - x_j}$. where, γ is the attributes set and $i \in Z_q^*$.

The solution uses the four algorithms described in the following.

Setup: assume that the space of static attributes is $V = \{1, 2, \ldots, n\}$; AA selects one $s_i \in_R Z_q^*$ for each static attribute $i \in V$, and calculates the corresponding public key equal to $P_i = s_i \cdot G$. Then, uniformly, AA selects $s \in_R Z_q^*$ as MK and chooses $PK = s \cdot G$ corresponding to it. Therefore, the public parameters are set as $params = \{PK, P_1, \ldots, P_n\}$.

Encryption $(m, \omega, params)$**:** this step uses an algorithm that differs from the conventional algorithms in this field. It derives from the ECIES algorithm (Algorithm 2) described in Section 6.2, and uses dynamic attributes in addition to static ones. We also use the message m as an input for $HMAC$: see Algorithm 4 for a specification of it.

Algorithm 4 Proposed encryption algorithm.

```
Input: $m, \omega, Params$
Output: $CM$
1: Randomly generates $k \in Z_q^*$
2: $C' \leftarrow k \cdot PK = (K_X, K_Y) \neq O$
3: $C_i \leftarrow k \cdot P_i$, $i \in \gamma$
4: $H \leftarrow h(M_T(\sigma_1, \sigma_1) \| M_T(\sigma_2, \sigma_2) \| \ldots \| M_T(\sigma_n, \sigma_n))$
5: $C \leftarrow ENC(m, K_X \| H)$
6: $MAC_M \leftarrow HMAC(m \| H, K_Y)$
7: return $CM = \gamma \| C \| MAC_M \| C' \| C_i$, $i \in \gamma$
```

The algorithm encrypts a message m using attribute sets $\omega = \gamma \cup \sigma$ in which the static set of attributes $\gamma = (\gamma_1, \gamma_2, \ldots, \gamma_n)$ and the dynamic set of attributes $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_n)$. Algorithm 4 runs as follows:

- Initially, a sender randomly selects a value k from Z_q^* and computes $C' = k \cdot PK = (K_X, K_Y) \neq O$.
- Subsequently, the values of C_i, $i \in \gamma$ are computed as $C_i = k \cdot P_i$.
- Then, the sender first computes the value $H = h(M_T(\sigma_1, \sigma_1) \| M_T(\sigma_2, \sigma_2) \| \ldots \| M_T(\sigma_n, \sigma_n))$, and then uses $K_X \| H$ and K_y as an encryption key and HMAC-key respectively, to construct the encrypted text CM as presented at Lines 5∼7 of Algorithm 4.

Key-Generation (Γ, MK)**:** in this algorithm, the decryption key generated by AA, the master key and a series of static attributes γ are received in input, and on the condition that $\Gamma(\gamma) = 1$, a decryption key is returned under the static attributes set γ as follows. At first, the polynomial $q_{node}(x) = q_{node}(0) + \sum_{j=1}^{d_{node}-1} a_j x_i^j$ of degree d_{node}-1 for each node $node$ is determined from the access tree Γ from top to bottom, where d_{node} is the threshold value of this node. Therefore, in this algorithm, for a root r of the access tree Γ, a value $q_r(0) = s$, and for other d_r-1 points a_j of polynomial $q_r(x)$, random variables are considered, where d_r is the threshold value of root r. Similarly, for each node v (including the leaf node e), the value $q_{v,e}(0) = q_{parent(v,e)}(index(v,e))$ and the other $d_{v,e}$-1 points a_j of the polynomial $q_{v,e}(x)$ are considered to be random. After determining the polynomial of a leaf node e, the decryption key for leaf node e is obtained from $D_e = \frac{q_{v,e}(0)}{s_i}$, where $i = attr(e)$ and $s_i \in_R Z_q^*$. All the polynomials and the secret keys corresponding to them are generated for all leaf nodes in this manner, and their collection is obtained as the decryption key $D = (D_e = \frac{q_{v,e}(0)}{s_i}, i = attr(e)$ and $i \in \gamma)$.

Decryption $(CM, D, Params)$**:** this algorithm runs at the receiver side (see Algorithm 5).

An algorithm $DecryptNode(CM, D, e)$ for the leaf node e is done by the access tree as follows:

If $i = attr(e) \in \gamma$, then $DecryptNode(CM, D, e) = D_e \cdot C_i = (\frac{q_{v,e}(0)}{s_i}) \cdot k \cdot P_i = (\frac{q_{v,e}(0)}{s_i}) \cdot k \cdot s_i \cdot G = q_{v,e}(0) \cdot k \cdot G$. Otherwise $DecryptNode(CM, D, e) = \perp$. For the non-leaf node u if $i = attr(u) \notin \gamma$, we have $DecryptNode(CM, D, u) = \perp$. Otherwise, for all v children nodes of non-leaf node u, the recursive algorithm $DecryptNode(CM, D, u)$ is expressed as follows:

Algorithm 5 Proposed decryption algorithm.

```
Input: $CM, D_v(i), Params$
Output: $m$ or $\perp$
 1: if $i = attr(v) \notin \gamma$ then $\perp$
 2: else
 3:     Run $DecryptNode(CM, D, r)$ and obtain $(K'_X, K'_Y)$
 4:     $H' \leftarrow h(M_T(\sigma_1, \sigma'_1) \| M_T(\sigma_2, \sigma'_2) \| \dots \| M_T(\sigma_n, \sigma'_n))$
 5:     $m' \leftarrow DEC(C, K'_X \| H')$
 6:     if $HMAC(m' \| H', K'_Y) \neq MAC_M$ then $\perp$
 7:     else
 8:         $m \leftarrow m'$
 9:         return $m$
10:     end if
11: end if
```

Suppose γ_u is an arbitrary attribute set with size d_u of non-leaf node u. For $DecryptNode(CM, D, v) \neq \perp$, if there is no such set γ_u, then $DecryptNode(CM, D, u) = \perp$; Otherwise, we have:

$DecryptNode(CM, D, u) =$
$\sum_{v \in \gamma_u} \Delta_{i,\gamma'_u}(0) \cdot DecryptNode(CM, D, v) =$
$\sum_{v \in \gamma_u} \Delta_{i,\gamma'_u}(0) \cdot q_v(0) \cdot k \cdot G =$
$\sum_{v \in \gamma_u} \Delta_{i,\gamma'_u}(0) \cdot q_{parent(v)}(index(v)) \cdot k \cdot G =$
$\sum_{v \in \gamma_u} \Delta_{i,\gamma'_u}(0) \cdot q_v(i) \cdot k \cdot G = q_v(0) \cdot k \cdot G$
where $\gamma_{u'} = \{index(v), v \in \gamma_u\}$ and $i = index(v)$.

Given the recursive function $DecryptNode(CM, D, u)$ described above, for a root r from the access tree, we have: $DecryptNode(CM, D, r) = q_r(0) \cdot k \cdot G = s \cdot k \cdot G = k \cdot PK = (K'_X, K'_Y)$.

In this step, using the values of the dynamic attributes collected by selected sensors through the receiver's mobile app, the value H' is calculated as $H' = h(M_T(\sigma_1, \sigma'_1) \| M_T(\sigma_2, \sigma'_2) \| \dots \| M_T$
Using the values of H' and K'_X computed in the previous steps, the value of m' computed as $m' = DEC(C, K'_X \| H')$ and if the equality $HMAC(m' \| H', K'_Y) = MAC_m$ holds, then the integrity of the data is accepted and it can be concluded that $m' = m$.

6.3.1 Revocation of Access Permission

As stipulated in Section 6.2.6, dynamic attributes are installed by the employer using software platforms such as KNOX or BES12 [118, 19] and updated periodically. Those apps have been shown able to detect unauthorized users [44, 88] who attempt deception by presenting false dynamic attributes. Concerning the above functionality, whenever it is necessary to revoke a particular user's access to a series of information, it is sufficient for the software to be updated and loaded into a new mapping function.

6.4 Security and Performance Analysis

We now analyze the security and efficiency of the proposed scheme.

6.4.1 Preserving data confidentiality

One of the critical issues in the IoT is to maintain the confidentiality of data and access control in a manner consistent with AA's defined access policies. These systems require users not to have unauthorized access to data. Our proposed scheme, which uses secure cryptographic algorithms and access policies based on the access tree as well as secure apps, can be seen to maintain complete confidentiality of the data.

6.4.2 Preserving data integrity

In the proposed scheme, the $HMAC$ function is used to prevent the data from being altered by unauthorized users. This provision in turn provides for the authenticity of the data.

6.4.3 Possibility to change access policies

In the proposed scheme, applying dynamic attributes along with static ones, together with the use of secure software apps such as KNOX and BES12, provides the ability to change the access policy of a particular user at any time. All that it takes to that end is a software update, which causes the loading of a new mapping function.

6.4.4 Efficiency and lightweightness

The scheme proposed in this work uses elliptic curve cryptography that is known to be more lightweight and cheap than state-of-the-art alternatives [87, 60, 7], whose authors use bilinear pairing techniques.

Table 6.2 compares the communication cost of our proposed scheme against other KP-ABE schemes. In this table, the term l is equal to 160 bits. Thus, in 160-bit ECC, the size of a point is $2l$ and the size of private and public keys is l and $2l$, respectively.

Similarly, for 1024-bit RSA, which is supposed to have the same security level as 160 bit, the size of public and private keys are both $6.4l$ and the size of output element is $12.8l$. The lengths of MAC is also assumed l bits.

Note that in our comparison we use l_γ to denote the size (cardinality) of the static attribute set. So, the length of the encrypted message CM in our proposed protocol is computed as $|CM| = |\gamma| + |C| + |MAC_M| + |C'| + |C_i| = l_\gamma + 4l$.

TABLE 6.2: Communication cost comparison between our scheme and KP-ABE.

Scheme	CM size (bit)	Attributes
Ours	$l_\gamma + 4l$	Static + Dynamic
[87]	$l_\gamma + (3 \times 6.4)l$	Static + Dynamic
[60]	$l_\gamma + (2 \times 6.4)l$	Static
[65]	$l_\gamma + (3 \times 6.4)l$	Static
[141]	$l_\gamma + 4l$	Static

The results presented in Table 6.2 show that our scheme is more lightweight than other KP-ABE schemes in terms of communication overhead. The results reported in Figure 6.4 support this claim, assuming $l = l_\gamma = 160$ bits, as a typifying example. Although the CM size of our proposed solution is the same as in the scheme proposed in [141], our solution is superior because it supports dynamic attributes.

Interestingly, for our scheme and [87], decreasing the number of static attributes and increasing the number of dynamic ones, reduces significantly the bit-length of l_γ, as shown in Figure 6.5. In this figure, the total number of attributes are constant and is equal to $nl_\sigma + (m\text{-}n)l_\gamma$, $m > n$. In which n, m, and l_σ are number of the dynamic attributes, number of the static attributes and the size (cardinality) of the

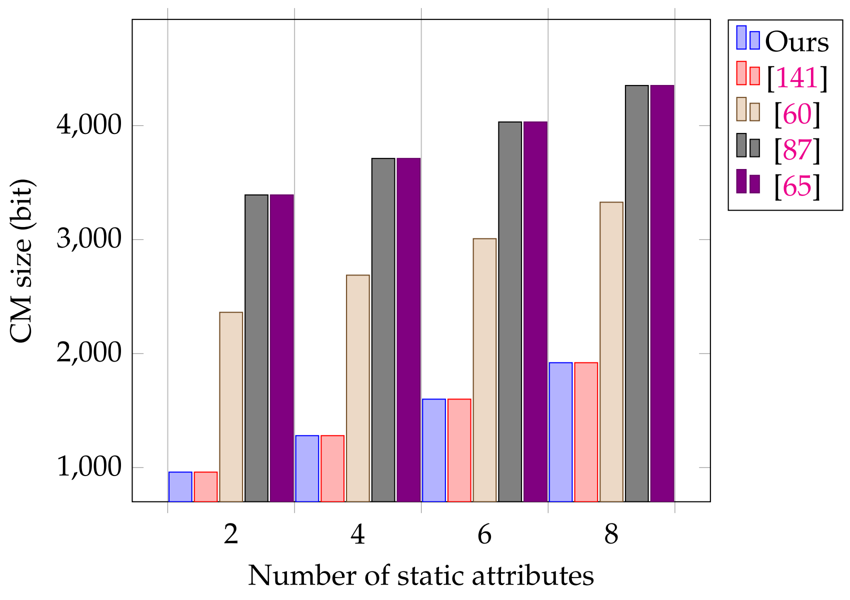


FIGURE 6.4: Transmission cost of ABE schemes

dynamic attribute set, respectively. For example, if $l = l_\gamma =$ 160 bits, for the scheme which should consider ten attributes (i.e., $n + m = 10$), if we reduce the number of the static attributes (m) and evenly increase the number of dynamic attributes (n), we can keep number of attributes but make the scheme more efficient than the other schemes which do not support the dynamic attributes.

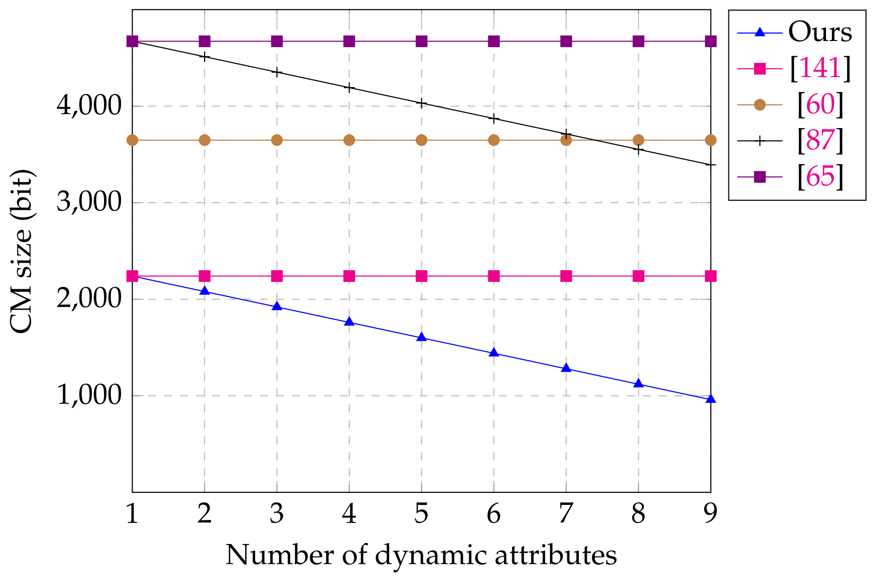


FIGURE 6.5: The communication cost can be reduced by considering the dynamic attributes

Furthermore, applying dynamic attributes in addition to static attributes, computational time can be significantly reduced. For example, assuming that ten attributes are required to be considered, If all these attributes are static, the time to use these attributes is far more than when five static attributes and five dynamic attributes are required to be considered. Due to the fact that, the use of dynamic attributes requires only a new map function, it can significantly reduce the computational time.

6.5 Conclusions and Future Work

Owing to its intrinsic flexibility and its ability to draw value added from its immersion in the physical world, the Internet of Things paradigm is finding rich and attractive opportunities of use in various application scenarios such as smart cities, health care, and transportation. However, the limited resource capacity of IoT devices present serious challenges to those who want to achieve adequate levels of security in their system.

In this paper, we have proposed a novel lightweight scheme based on dynamic attributes in combination with the elliptic curve encryption technique to provide high security at low resource cost, fit for use in IoT environments.

In addition to incorporating dynamic attributes to overcome attacks from compromised entity, our proposed scheme uses a smaller key size to provide the same level of security as the RSA and other modular exponentiation-based techniques. Furthermore, applying point scalar multiplication operation makes our scheme faster than the conventional schemes, which use modular exponentiation and bilinear mapping operations.

The Fog has recently emerged as a new computational layer where data collection and processing can be performed closer to the edge, where the physical world begins, sensed and actuated by such "things" as sensors, smart devices, hence with assurance of lesser latency than with full offloading to the Cloud. Adding a Fog layer between the Edge and the Cloud facilitates the apportionment of the computational load and improves efficiency of future IoT system. An obvious extension to our work explores assuming and leveraging the presence of a Fog layer, while also validates our proposed scheme using real case scenarios.

Chapter 7

CONCLUSION AND FUTURE WORKS

7.1 Conclusion

In this thesis, we presented three complementary steps toward the creation of practical systems to achieve higher resource efficiency for service in Clod environment. First, we drew a trajectory that, starting from a better understanding of the essential service design features, related them to the microservice architectural style and its implications on elastic scalability, most notably dynamic orchestration, and concludes reviewing how well state-of-the-art technology fares for their implementation. Secondly, we use these principles to uncover the potential to improve the resource-efficiency of dynamic orchestration of service in technology level. Moreover, then, we propose an intelligent system making informed decisions on auto-scaling in dynamic orchestration platform. We further exploit the resource-efficiency potential via distributed machine learning in Fog computing paradigm. In this sense, we present a distributed machine learning approach for road traffic modelling and prediction, designed for a city-wide scenario based on the *Fog computing* paradigm. We display architecture for large scale Floating Car Data (FCD), data modelling on the Edge, leveraging parallelism on deep learning algorithms in a city-wide scenario towards prediction. Finally, we explore another major challenge that typically discourages users to use a cloud-like environment is the security concerns. As the resource capacity of near-edge IoT devices is naturally insufficient to cope with the influx of sensed data and its processing requirements, integration of the IoT system with cloud services allows opportune offloading of excess work. In several domains, the data must be protected before being offloaded to the cloud. However, for the sake of privacy, the data must be protected before being offloaded to the cloud. In this work, we also describe a security challenge concerning resource limitations of IoT devices and presents light-weight security schema for IoT ecosystem. We have proven that all the presented techniques achieve their performance objectives.

7.2 Future Works

We believe that the contributions described above open many interesting paths for future research. Therefore, in this section, we present some promising future directions for the work done in this thesis.

- SECURITY IN CLOUD: Future research is needed to address these concerns to enable a more secure cloud environment. Nowadays, integration (combination) of the Internet of Things (IoT) and Cloud computing have been widely used in several domains of application. However, the limited resource capacity in the Internet of Things (IoT) has arisen various challenges in the authentication technologies that have been investigated for secure data retrieval and robust access control on large-scale IoT networks. Within this context, recently, a protocol has been proposed for these systems that claims to resists the common security threats on large-scale IoT networks. In this work, we analyze the security of the mentioned protocol. We prove how the mentioned proposed protocol is not immune to users, cloud server impersonation, and man-in-the-middle attacks. To overcome these shortcomings, firstly, we have advanced(enhanced, improved) the Zhou *et al.*'s protocol and informally show (demonstrate, explain, describe) that our advanced(enhanced, improved) scheme is secure against the most common attacks in these systems. Then, we verify formally our proposed protocol using the Proverif language. Consequently, we prove that our scheme provides better efficiency and security in comparison to the Zhou *et al.*'s scheme, applicable to large-scale IoT networks.

  c5. Seyed Farhad Aghili, Kiyana Bahadori, Mohammad Shojafar, Hamid Mala, Tullio Vardanega, " Breaking and fixing an IoT-based authentication and key agreement scheme in Cloud computing". Transaction Conference on Computer Communications and Networks (under-submission)

- In future work, we plan to extend the line of work as resource efficiency in service design looking into combined auto-scaling policies that operate at both container and node level, integrating business-level and service-level objectives (e.g., performance, cost, etc.) by converting them to utility functions that can be fed as additional input parameters such as online learning method into our models, so as to facilitate dynamic rule generation for a container orchestration platform for optimal resource provisioning in the Cloud.

Bibliography

[1] Martin L. Abbott and Michael T. Fisher. *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise*. 1st. Addison-Wesley Professional, 2009. ISBN: 0137030428, 9780137030422.

[2] Martin L Abbott and Michael T Fisher. *The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise*. Pearson Education, 2009.

[3] Amazon. *Amazon Web Service*. https://aws.amazon.com/. 2017.

[4] Dario Amodei et al. "End to end speech recognition in English and Mandarin". In: (2016).

[5] Vasilios Andrikopoulos et al. "How to adapt applications for the cloud environment". In: *Computing* 95.6 (2013), pp. 493–535.

[6] Michael Armbrust et al. "A view of cloud computing". In: *Communications of the ACM* 53.4 (2010), pp. 50–58.

[7] Nuttapong Attrapadung, Benoît Libert, and Elie De Panafieu. "Expressive key-policy attribute-based encryption with constant-size ciphertexts". In: *International Workshop on Public Key Cryptography*. Springer. 2011, pp. 90–108.

[8] The Amazon Authors. *Automated container deployment, scaling, and management*. https://aws.amazon.com/ecs/. 2018.

[9] The Kubernetes Authors. *Automated container deployment, scaling, and management*. https://kubernetes.io/. 2018.

[10] The OpenShift Authors. *Automated container deployment, scaling, and management*. https://www.openshift.com/. 2018.

[11] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. "Microservices architecture enables DevOps: migration to a cloud-native architecture". In: *IEEE Software* 33.3 (2016), pp. 42–52.

[12] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. "Migrating to cloud-native architectures using microservices: an experience report". In: *European Conference on Service-Oriented and Cloud Computing*. Springer. 2015, pp. 201–215.

[13] Soon K Bang et al. "A grounded theory analysis of modern web applications: knowledge, skills, and abilities for DevOps". In: *Proceedings of the 2nd annual conference on Research in information technology*. ACM. 2013, pp. 61–62.

[14] Maximilien de Bayser, Leonardo G Azevedo, and Renato Cerqueira. "ResearchOps: The case for DevOps in scientific applications". In: *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*. IEEE. 2015, pp. 1398–1404.

[15] D. Beaumont. *How to explain vertical and horizontal scaling in the cloud*. https://www.ibm.com/blogs/cloud-computing/2014/04/explain-vertical-horizontal-scaling-cloud/. 2017.

[16] Sana Belguith, Nesrine Kaaniche, and Giovanni Russello. "PU-ABE: Lightweight Attribute-Based Encryption Supporting Access Policy Update for Cloud Assisted IoT". In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE. 2018, pp. 924–927.

[17] Tal Ben-Nun and Torsten Hoefler. "Demystifying parallel and distributed deep learning: An in-depth concurrency analysis". In: *arXiv preprint arXiv:1802.09941* (2018).

[18] John Bethencourt, Amit Sahai, and Brent Waters. "Ciphertext-policy attribute-based encryption". In: *Security and Privacy, 2007. SP'07. IEEE Symposium on*. IEEE. 2007, pp. 321–334.

[19] BlackBerry. *BES12*. `https://global.blackberry.com/en/bes-support`. 2005.

[20] Flavio Bonomi et al. "Fog computing and its role in the internet of things". In: *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM. 2012, pp. 13–16.

[21] Eric A. Brewer. "Kubernetes and the Path to Cloud Native". In: *Proceedings of the Sixth ACM Symposium on Cloud Computing*. SoCC '15. Kohala Coast, Hawaii: ACM, 2015, pp. 167–167. ISBN: 978-1-4503-3651-2. DOI: `10.1145/2806777.2809955`. URL: `http://doi.acm.org/10.1145/2806777.2809955`.

[22] Rajkumar Buyya et al. "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility". In: *Future Generation computer systems* 25.6 (2009), pp. 599–616.

[23] Rodrigo N Calheiros et al. "Workload prediction using ARIMA model and its impact on cloud applications' QoS". In: *IEEE Transactions on Cloud Computing* 3.4 (2015), pp. 449–458.

[24] Miguel Castro, Antonio J Jara, and Antonio FG Skarmeta. "Smart lighting solutions for smart cities". In: *2013 27th International Conference on Advanced Information Networking and Applications Workshops*. IEEE. 2013, pp. 1374–1379.

[25] David Chapman. *Introduction to DevOps on AWS*. `https://aws.amazon.com/whitepapers/introduction-to-devops-on-aws/`. 2014.

[26] Erik Christensen et al. *Web Service Description Language (WSDL) 1.1 W3C Note*. Tech. rep. World Wide Web Consortium (W3C), 2001.

[27] Pablo Cibraro et al. *Professional WCF 4: Windows communication foundation with. NET 4*. John Wiley & Sons, 2010.

[28] Dan Cireşan, Ueli Meier, and Jürgen Schmidhuber. "Multi-column deep neural networks for image classification". In: *arXiv preprint arXiv:1202.2745* (2012).

[29] Adam Coates, Andrew Ng, and Honglak Lee. "An analysis of single-layer networks in unsupervised feature learning". In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. 2011, pp. 215–223.

[30] Adam Coates et al. "Deep learning with COTS HPC systems". In: *International conference on machine learning*. 2013, pp. 1337–1345.

[31] Frank Cohen. "Understanding Web service interoperability". In: *IBM Technical Library* (2002).

[32] George E Dahl et al. "Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition". In: *IEEE Transactions on audio, speech, and language processing* 20.1 (2012), pp. 30–42.

[33] Hugo Haas Francis McCabe Eric Newcomer Iona Michael Champion Chris Ferris David Orchard David Booth Hewlett-Packard. "Web Services Architecture, W3C Working Group Note". In: (11 February2004). URL: https://www.w3.org/TR/ws-arch/.

[34] Wesam Dawoud, Ibrahim Takouna, and Christoph Meinel. "Elastic vm for cloud resources provisioning optimization". In: *Advances in Computing and Communications* (2011), pp. 431–445.

[35] Jeffrey Dean et al. "Large scale distributed deep networks". In: *Advances in neural information processing systems*. 2012, pp. 1223–1231.

[36] Patrick Debois. "Agile infrastructure and operations: how infra-gile are you?" In: *Agile 2008 Conference*. IEEE. 2008, pp. 202–207.

[37] Li Deng, Dong Yu, and John Platt. "Scalable stacking and learning for building deep architectures". In: *2012 IEEE International conference on Acoustics, speech and signal processing (ICASSP)*. IEEE. 2012, pp. 2133–2136.

[38] Swarnava Dey et al. "Smart city surveillance: Leveraging benefits of cloud data stores". In: *37th Annual IEEE Conference on Local Computer Networks-Workshops*. IEEE. 2012, pp. 868–876.

[39] Tharam Dillon, Chen Wu, and Elizabeth Chang. "Cloud computing: issues and challenges". In: *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*. Ieee. 2010, pp. 27–33.

[40] Torgeir Dingsøyr et al. *A decade of agile methodologies: Towards explaining agile software development*. 2012.

[41] Nicola Dragoni et al. "Microservices: How to make your application scale". In: *arXiv preprint arXiv:1702.07149* (2017).

[42] Nicola Dragoni et al. "Microservices: yesterday, today, and tomorrow". In: *Present and Ulterior Software Engineering*. Springer, 2017, pp. 195–216.

[43] Rajdeep Dua, A Reddy Raja, and Dharmesh Kakadia. "Virtualization vs containerization to support paas". In: *Cloud Engineering (IC2E), 2014 IEEE International Conference on*. IEEE. 2014, pp. 610–614.

[44] Nathan Eagle and Alex Pentland. "Reality mining: sensing complex social systems". In: *Personal and ubiquitous computing* 10.4 (2006), pp. 255–268.

[45] Christof Ebert et al. "DevOps". In: *IEEE Software* 33.3 (2016), pp. 94–100.

[46] Taher ElGamal. "A public key cryptosystem and a signature scheme based on discrete logarithms". In: *IEEE transactions on information theory* 31.4 (1985), pp. 469–472.

[47] *Embedded Systems for Next-Generation Autonomous Machines*. 2019. URL: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/.

[48] Thomas Erl. *Soa: principles of service design*. Prentice Hall Press, 2007.

[49] Christoph Fehling et al. *Cloud computing patterns: fundamentals to design, build, and manage cloud applications*. Springer Science & Business Media, 2014.

[50] Wes Felter et al. "An updated performance comparison of virtual machines and linux containers". In: *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*. IEEE. 2015, pp. 171–172.

[51] Brian Fitzgerald and Klaas-Jan Stol. "Continuous software engineering and beyond: trends and challenges". In: *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*. ACM. 2014, pp. 1–9.

[52] Nicole Forsgren, Jez Humble, and Gene Kim. "Accelerate: The Science of Lean Software and DevOps Building and Scaling High Performing Technology Organizations". In: (2018).

[53] Martin Fowler. *Continuous Delivery*. `https://martinfowler.com/books/continuousDelivery.html`. 2006.

[54] Martin Fowler. *Continuous Integration*. `https://www.martinfowler.com/articles/continuousIntegration.html`. 2006.

[55] Martin Fowler. *Microservices, a definition of this new architectural term*. `https://www.terraform.io/`. 2017.

[56] Armando Fox et al. "Above the clouds: A berkeley view of cloud computing". In: *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS* 28.13 (2009), p. 2009.

[57] Víctor Gayoso Martínez, Luis Hernández Encinas, and Carmen Sánchez Ávila. "A survey of the elliptic curve integrated encryption scheme". In: (2010).

[58] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. "Press: Predictive elastic resource scaling for cloud systems". In: *Network and Service Management (CNSM), 2010 International Conference on*. Ieee. 2010, pp. 9–16.

[59] Priya Goyal et al. "Accurate, large minibatch sgd: Training imagenet in 1 hour". In: *arXiv preprint arXiv:1706.02677* (2017).

[60] Vipul Goyal et al. "Attribute-based encryption for fine-grained access control of encrypted data". In: *Proceedings of the 13th ACM conference on Computer and communications security*. Acm. 2006, pp. 89–98.

[61] Nils Gura et al. "Comparing elliptic curve cryptography and RSA on 8-bit CPUs". In: *International workshop on cryptographic hardware and embedded systems*. Springer. 2004, pp. 119–132.

[62] Darrel Hankerson, Julio López Hernandez, and Alfred Menezes. "Software implementation of elliptic curve cryptography over binary fields". In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2000, pp. 1–24.

[63] Nikolas Roman Herbst, Samuel Kounev, and Ralf H Reussner. "Elasticity in Cloud Computing: What It Is, and What It Is Not." In: *ICAC*. Vol. 13. 2013, pp. 23–27.

[64] Vanessa Ho. *Bringing DevOps to the masses with Microsoft's Donovan Brown*. `https://blogs.microsoft.com/firehose/2016/11/29/bringing-devops-to-the-masses-with-microsofts-donovan-brown/`. 2016.

[65] Susan Hohenberger and Brent Waters. "Attribute-based encryption with fast decryption". In: *Public-Key Cryptography–PKC 2013*. Springer, 2013, pp. 162–179.

[66] Kirak Hong et al. "Opportunistic spatio-temporal event processing for mobile situation awareness". In: *Proceedings of the 7th ACM international conference on Distributed event-based systems*. ACM. 2013, pp. 195–206.

[67] Jez Humble and Joanne Molesky. "Why enterprises must adopt devops to enable continuous delivery". In: *Cutter IT Journal* 24.8 (2011), p. 6.

[68] Steven Ihde. *InfoQ | From a Monolith to Microservices + REST: the Evolution of LinkedIn's Service Architecture. [Online].* `http://www.infoq.com/presentations/linkedin-microservices-urn/`. 2015.

[69] Docker Inc. *Docker Datacenter Enables DevOps*. `https://www.docker.com/use-cases/devops`. 2018.

[70] Jenkins. *Jenkins*. `https://jenkins.io/`. 2017.

[71] Xiaolin Jia et al. "RFID technology and its applications in Internet of Things (IoT)". In: *Consumer Electronics, Communications and Networks (CECNet), 2012 2nd International Conference on*. IEEE. 2012, pp. 1282–1285.

[72] Staci Kamer. *GIGAOM, The Biggest Thing Amazon Got Right: The Platform. [Online].* `https://gigaom.com/2011/10/12/419-the-biggest-thing-amazon-got-right-the-platform/`. 2011.

[73] Jeffrey O Kephart and David M Chess. "The vision of autonomic computing". In: *Computer* 1 (2003), pp. 41–50.

[74] Ali Khajeh-Hosseini et al. "The cloud adoption toolkit: supporting cloud adoption decisions in the enterprise". In: *Software: Practice and Experience* 42.4 (2012), pp. 447–465.

[75] Asif Khan. "Key Characteristics of a Container Orchestration Platform to Enable a Modern Application". In: *IEEE Cloud Computing* 5 (2017), pp. 42–48.

[76] Gene Kim et al. *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. IT Revolution, 2016.

[77] Neal Koblitz. "Elliptic curve cryptosystems". In: *Mathematics of computation* 48.177 (1987), pp. 203–209.

[78] Nane Kratzke and René Peinl. "ClouNS - A Cloud-native Application Reference Model for Enterprise Architects". In: *CoRR* abs/1709.04883 (2017). arXiv: `1709.04883`. URL: `http://arxiv.org/abs/1709.04883`.

[79] Nane Kratzke and René Peinl. "Clouns-a cloud-native application reference model for enterprise architects". In: *Enterprise Distributed Object Computing Workshop (EDOCW), 2016 IEEE 20th International*. IEEE. 2016, pp. 1–10.

[80] Nane Kratzke and Peter-Christian Quint. "Understanding cloud-native applications after 10 years of cloud computing-a systematic mapping study". In: *Journal of Systems and Software* 126 (2017), pp. 1–16.

[81] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[82] Kuberenetes. *Container Orchestration*. `https://kubernetes.io/`.

[83] John Langford, Alexander Smola, and Martin Zinkevich. "Slow learners are fast". In: *arXiv preprint arXiv:0911.0491* (2009).

[84] Quoc V Le et al. "Building high-level features using large scale unsupervised learning". In: *arXiv preprint arXiv:1112.6209* (2011).

[85] Honglak Lee et al. "Unsupervised feature learning for audio classification using convolutional deep belief networks". In: *Advances in neural information processing systems*. 2009, pp. 1096–1104.

[86] Fei Li et al. "Robust access control framework for mobile cloud computing network". In: *Computer Communications* 68 (2015), pp. 61–72.

[87] Fei Li et al. "Robust access control framework for mobile cloud computing network". In: *Computer Communications* 68 (2015), pp. 61–72.

[88] Fudong Li et al. "Behaviour profiling on mobile devices". In: *Emerging Security Technologies (EST), 2010 International Conference on*. IEEE. 2010, pp. 77–82.

[89] C Matthew MacKenzie et al. "Reference model for service oriented architecture 1.0". In: *OASIS standard* 12 (2006), p. 18.

[90] Mark Masse. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. " O'Reilly Media, Inc.", 2011.

[91] Manuel Pais Matthew Skelton. *DevOps Topologies*. https://web.devopstopologies.com/#anti-types.

[92] Tony Mauro. *Nginx | Adopting Microservices at Netflix: Lessons for Architectural Design.[Online]*. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/. 2015.

[93] Dimitri Mazmanov et al. "Handling Performance Sensitive Native Cloud Applications with Distributed Cloud Computing and SLA Management". In: *Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*. UCC '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 470–475. ISBN: 978-0-7695-5152-4. DOI: 10.1109/UCC.2013.92. URL: http://dx.doi.org/10.1109/UCC.2013.92.

[94] Ryan Mcdonald et al. "Efficient large-scale distributed training of conditional maximum entropy models". In: *Advances in Neural Information Processing Systems*. 2009, pp. 1231–1239.

[95] Peter Mell, Tim Grance, et al. "The NIST definition of cloud computing". In: (2011).

[96] Dirk Merkel. "Docker: lightweight linux containers for consistent development and deployment". In: *Linux Journal* 2014.239 (2014), p. 2.

[97] Victor S Miller. "Use of elliptic curves in cryptography". In: *Conference on the theory and application of cryptographic techniques*. Springer. 1985, pp. 417–426.

[98] Kief Morris. *Infrastructure as code: managing servers in the cloud*. O'Reilly Media, Inc., 2016.

[99] Dmitry Namiot and Manfred Sneps-Sneppe. "On micro-services architecture". In: *International Journal of Open Information Technologies* 2.9 (2014), pp. 24–27.

[100] Netflix. *Netflix*. https://www.netflix.com.

[101] Sam Newman. *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.", 2015.

[102] Numenta. *The Numenta Anomaly Benchmark*. https://github.com/numenta/NAB. 2018.

[103] Claus Pahl. "Containerization and the paas cloud". In: *IEEE Cloud Computing* 2.3 (2015), pp. 24–31.

[104] Juan Luis Pérez et al. "A resilient and distributed near real-time traffic forecasting application for Fog computing environments". In: *Future Generation Computer Systems* 87 (2018), pp. 198–212.

[105] Juan Luis Pérez et al. "A resilient and distributed near real-time traffic forecasting application for Fog computing environments". In: *Future Generation Computer Systems* 87 (2018), pp. 198 –212. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2018.05.013.

[106] *Powering the Machine Intelligence Revolution*. 2019. URL: https://www.movidius.com/.

[107] Puppet. *Configuration Management tool*. https://www.puppet.io/chef/.

[108] Rajat Raina, Anand Madhavan, and Andrew Y Ng. "Large-scale deep unsupervised learning using graphics processors". In: *Proceedings of the 26th annual international conference on machine learning*. ACM. 2009, pp. 873–880.

[109] rakyll/hey. *rakyll/hey*. https://github.com/rakyll/hey. 2018.

[110] Jinghai Rao and Xiaomeng Su. "A survey of automated web service composition methods". In: *SWSWPC*. Vol. 3387. Springer. 2004, pp. 43–54.

[111] *Raspberry Pi*. 2019. URL: httpsbs://www.raspberrypi.org/products/raspberry-pi-3-model-b/.

[112] Chris Richardson. *Choosing a Microservices Deployment Strategy*. https://www.nginx.com/blog/deploying-microservices/. February 10, 2016.

[113] Chris Richardson. *Monolithic architecture*. http://microservices.io/patterns/monolithic.html. 2017.

[114] RightScale. *RightScale*. https://www.rightscale.com/. 2017.

[115] Ronald L Rivest, Adi Shamir, and Leonard Adleman. "A method for obtaining digital signatures and public-key cryptosystems". In: *Communications of the ACM* 21.2 (1978), pp. 120–126.

[116] Amit Sahai and Brent Waters. "Fuzzy identity-based encryption". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2005, pp. 457–473.

[117] Hojjat Salehinejad et al. "Recent Advances in Recurrent Neural Networks". In: *arXiv preprint arXiv:1801.01078* (2017).

[118] Samsung. *KNOX*. https://www.samsungknox.com/. 2005.

[119] Alexander Sergeev and Mike Del Balso. "Horovod: fast and easy distributed deep learning in TensorFlow". In: *arXiv preprint arXiv:1802.05799* (2018).

[120] Martín Serrano et al. "Cloud services composition support by using semantic annotation and linked data". In: *International Joint Conference on Knowledge Discovery, Knowledge Engineering, and Knowledge Management*. Springer. 2011, pp. 278–293.

[121] Adi Shamir. "How to share a secret". In: *Communications of the ACM* 22.11 (1979), pp. 612–613.

[122] Qinfeng Shi et al. "Hash kernels". In: *Artificial intelligence and statistics*. 2009, pp. 496–503.

[123] Matt Stine. *Migrating to Cloud-Native Application Architectures*. 2015.

[124] Nikko Strom. "Scalable distributed DNN training using commodity GPU cloud computing". In: *Sixteenth Annual Conference of the International Speech Communication Association*. 2015.

[125] Kehua Su, Jie Li, and Hongbo Fu. "Smart city and the applications". In: *2011 international conference on electronics, communications and control (ICECC)*. IEEE. 2011, pp. 1028–1031.

[126] Fei Tao et al. “FC-PACO-RM: a parallel method for service composition optimal-selection in cloud manufacturing system”. In: *IEEE Transactions on Industrial Informatics* 9.4 (2013), pp. 2023–2033.

[127] Terraform. *Write, Plan, and Create Infrastructure as Code*. https://www.terraform.io/. 2017.

[128] Johannes Thönes. “Microservices”. In: *IEEE software* 32.1 (2015), pp. 116–116.

[129] Giovanni Toffetti et al. “Self-managing cloud-native applications: Design, implementation, and experience”. In: *Future Generation Computer Systems* 72 (2017), pp. 165–179.

[130] Lyes Touati and Yacine Challal. “Collaborative KP-ABE for cloud-based internet of things applications”. In: *Communications (ICC), 2016 IEEE International Conference on*. IEEE. 2016, pp. 1–7.

[131] Wei-Tek Tsai, Xin Sun, and Janaka Balasooriya. “Service-oriented cloud computing architecture”. In: *Information Technology: New Generations (ITNG), 2010 Seventh International Conference on*. IEEE. 2010, pp. 684–689.

[132] Luis M Vaquero, Luis Rodero-Merino, and Rajkumar Buyya. “Dynamically scaling applications in the cloud”. In: *ACM SIGCOMM Computer Communication Re* 41.1 (2011), pp. 45–52.

[133] Jinesh Varia. “Architecting for the cloud: Best practices”. In: *Amazon Web Services* 1 (2010), pp. 1–21.

[134] Suchitra Venugopal. *Cloud orchestration technologies,IBM*. https://www.ibm.com/developerworks/cloud/library/cl-cloud-orchestration-technologies-trs/index.html. 2016.

[135] Mario Villamizar et al. “Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud”. In: *Computing Colombian Conference (10CCC), 2015 10th*. IEEE. 2015, pp. 583–590.

[136] Manish Virmani. “Understanding DevOps & bridging the gap from continuous integration to continuous delivery”. In: *Innovative Computing Technology (INTECH), 2015 Fifth International Conference on*. IEEE. 2015, pp. 78–82.

[137] Xingwei Wang, Hong Zhao, and Jiakeng Zhu. “GRPC: A Communication Cooperation Mechanism in Distributed Systems”. In: *SIGOPS Oper. Syst. Rev.* 27.3 (July 1993), pp. 75–86. ISSN: 0163-5980. DOI: 10.1145/155870.155881. URL: http://doi.acm.org/10.1145/155870.155881.

[138] Charles B Weinstock and John B Goodenough. *On system scalability*. Tech. rep. CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2006.

[139] Matt Werner. *Executive Summary, Cloud, Serverless, Functions, Multicloud*. https://dzone.com/cloud-computing. 2018.

[140] Xue Yang et al. “A multi-layer security model for internet of things”. In: *Internet of things*. Springer, 2012, pp. 388–393.

[141] Xuanxia Yao, Zhi Chen, and Ye Tian. “A lightweight attribute-based encryption scheme for the Internet of Things”. In: *Future Generation Computer Systems* 49 (2015), pp. 104–112.

[142] Qi Zhang, Lu Cheng, and Raouf Boutaba. “Cloud computing: state-of-the-art and research challenges”. In: *Journal of internet services and applications* 1.1 (2010), pp. 7–18.

[143] Zheng Zhao et al. "LSTM network: a deep learning approach for short-term traffic forecast". In: *IET Intelligent Transport Systems* 11.2 (2017), pp. 68–75.

[144] Martin Zinkevich et al. "Parallelized stochastic gradient descent". In: *Advances in neural information processing systems*. 2010, pp. 2595–2603.