A Platform as a Service for IoT Application Provisioning in Hybrid Cloud/Fog Environment

Fereshteh Ebrahimnezhad

A Thesis
in
the Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

June 2020
©Fereshteh Ebrahimnezhad, 2020

CONCORDIA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Fereshteh Ebrahimnezhad

Entitled: "A Platform as a Service for IoT Application Provisioning in Hybrid Cloud/Fog Environment"

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

Complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

——————————————— Chair
Dr. O. Ormandjieva

——————————————— Examiner
Dr. J. Rilling

——————————————— Examiner
Dr. O. Ormandjieva

——————————————— Supervisor
Dr. R. Glitho

Approved By: ———————————————

Dr. Lata Narayanan
Department of Computer Science and Software Engineering

————20———— ———————————————

Dr. Mourad Debbabi
Dean & Professor,
Faculty of Engineering and Computer Science

Abstract

A Platform as a Service for IoT Application Provisioning in Hybrid Cloud/Fog Environment

Fereshteh Ebrahimnezhad, M.Sc.
Concordia University, 2020

IoT ecosystem refers to web-enabled smart devices that use embedded processors, sensors and communication hardware to capture, send and act on data they obtain from their environments. IoT applications range from healthcare to autonomous vehicles. These Applications are composed of interacting components. Provisioning these applications is rather challenging, specially in hybrid cloud and fog settings.

Cloud computing with its three main services (i.e., Infrastructure-as-a-Service, Platform-as-a-Service, and Software-as-a-Service) still faces some challenges. Through the PaaS, application providers can provision (i.e., develop, deploy, manage, and orchestrate) applications in the cloud. However, the wide-area network used to connect the cloud to the end-users might cause high latency which may not be acceptable. Fog computing can reduce this latency induced by distant clouds by enabling the deployment of some application components at the edge of the network while keeping others in the cloud. Existing PaaS solutions (including IoT PaaS solutions) do not enable provisioning of applications with components spanning cloud and fog.

The main contribution of this thesis is twofold. First, a novel IoT PaaS architecture for hybrid cloud and fog environments is proposed. Second, the general architecture is prototyped and measurements are made to evaluate the feasibility of the architecture. The application components are implemented as Virtual Network Function(VNF) and Cloudify, an open-source NFV orchestration framework is used.

The essential PaaS functional entities, such as deployment engine, orchestrator, migration engine, and publish and discovery engine are identified. A concrete mechanism for publication and discovery for existing cloud and fog nodes (both stationary and mobile fog nodes) has been also proposed. In addition, a set of RESTful interfaces is proposed to enable interaction with components internally and externally. For the evaluation part, we have measured the system end-to-end and migration delay for both centralized and distributed PaaS architecture as well as different placement for application components on cloud and fog nodes.

Acknowledgements

First, and foremost I would like to express my sincere gratitude to my supervisor Dr. Roch Glitho for his patience, motivation, and immense knowledge. His guidance and direction made my research to stay on track and progress. His profound belief in my abilities and accepting me as his student changed my life.

I wish to thank the members of my thesis committee: Dr. Olga Ormandjieva, and Dr. Juergen Rilling for generously offering their time, and for their brilliant comments and suggestions. I also want to extend my thanks to Dr. Olga Ormandjieva for serving as the chair at my thesis defense.

I would like to express my deepest appreciation to Dr. Carla Mouradian for all the enlightening discussions, comments and collaboration during this journey. It is my honor to have her also as my friend.

Furthermore, I am thankful to all of my colleagues in the telecommunication service engineering lab at Concordia University. In particular, I would like to thank Yassine Jebbar, Dr. Narjes-Elaheh Tahghigh, and Dr. Abbas Soltanian for their companionship, support, ideas, and fruitful discussions.

Last but not least, I am forever indebted to my family and my boyfriend, Keyvan, for their encouragement, continuous support, and love. There are no words that can express my gratitude and love for them. This accomplishment would not have been possible without them.

Contents

List of Figures

List of Tables

Acronyms and abbreviations

5G	Fifth Generation of Mobile Networks
6LoWPAN	IPv6 over LoWPAN
API	Application Programming Interface
BLE	Bluetooth Low Energy
CLI	Command Line Interface
COAP	Constrained Application Protocol
CPU	Central Processing Unit
CRUD	Create Read Update Delete
DaaS	Database as a Service
DAG	Directed Acyclic Graph
DSL	Domain Specific Language
GUI	Graphical User Interface
HDTV	High Definition Television
HTTP	Hyper Text Transfer Protocol
IaaS	Infrastructure as a Service
IDE	Integrated Development Environment
IoT	Internet of Things

IP	Internet Protocol
IR	Infrared
ITU	The International Telecommunication Union
JPEG	Joint Photographic Experts Group
JSON	JavaScript Object Notation
LAN	Local Area Network
LED	Light-emitting Diode
LoWPAN	Low-Powered Personal Area Network
ML	Machine Learning
MPEG	Moving Picture Experts Group
NFV	Network Function Virtualization
NFVI	Network Function Virtualization Infrastructure
NIST	US National Institute of Standards and technology
O-FSP	Optimized Fog Service Provisioning
OS	Operating System
P2P	Peer-to-Peer
PaaS	Platform as a Service
PIR	Passive Infrared
POJO	Plain Old Java Object
QoS	Quality of Service
RAM	Random Access Memory
REST	REpresentational State Transfer

RFRD	Radio-frequency identification
RSU	Road Side Unit
SaaS	Software as a Service
SDM	Service Description Metadata
SDN	Software Defined Network
SLA	Service Level Agreement
SPMD	Single Program and Multiple Data
SSH	Secure Shell
TCP	Transmission Control Protocol
TOSCA	Topology and Orchestration Specification for Cloud Applications
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
V2G	Vehicle to Grid
VGA	Video Graphics Array
VM	Virtual Machine
VNF	Virtual Network Function
VNF-FG	Virtual Network Function Forwarding Graph
WDR	Wide Dynamic Rage
WHO	World Health Organization
WSGI	Web Server Gateway Interface
XML	Extensible Markup Language
YAML	YAML Aren’t Markup Language

Chapter 1

Introduction

1.1 Definition

This section starts with an overview of the key terms associated with our research such as IoT, Cloud Computing, PaaS, Fog Computing, and Publication and Discovery. Then, the motivation and problem statement are discussed. Finally, the summary of our contributions and organization of the thesis are presented.

1.1.1 Internet of Things

Internet of Things (IoT), according to the definition provided by the ITU, is "A global infrastructure for the information society enabling advanced services by interconnecting (physical and virtual) things based on existing interoperable information and communication technologies" [1]. As stated by Gartner, device connections to the internet will facilitate the used data to analyze, preplan, manage, and make intelligent decisions autonomously. In this context, we can see a wide range of sectors, like: logistics, e-education, smart city, healthcare, autonomous vehicles, and etc., are taking advantage of IoT architectures.

1.1.2 Cloud Computing

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [2]. It is composed of three service models, Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS). The lowest layer is the IaaS which enables consumers to provision all kinds of fundamental computing resources such as, network, storage, and processing. The middle layer, PaaS, provides a rapid development environment to create and deploy their SaaS applications onto the cloud infrastructure.

1.1.3 PaaS

PaaS enables application developers to provision (i.e., develop, deploy, manage, and orchestrate) consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider onto the cloud infrastructure. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment [3].

1.1.4 Fog Computing

Fog Computing is a highly virtualized platform that provides compute, storage, and networking services between end devices and traditional Cloud Computing Data Centers, typically, but not exclusively located at the edge of the network. Low latency and location awareness, wide-spread geographical distribution, mobility, a very large number of nodes, predominant role of wireless access, the strong presence of streaming and real-time applications, and heterogeneity are of the main characteristics of fog computing [4].

1.1.5 Publication and Discovery

Discovery of things as well as their resources, metadata, properties, and capabilities is a fundamental requirement in any Internet of Things (IoT) ecosystem [5]. There must be a mechanism to automatically discover existing cloud and fog resources as well as joining or leaving ones. To generate an efficient placement plan, PaaS needs to have the most updated information about available cloud and fog nodes with their specifications such as location, capacity. This task becomes more crucial when we have several domains in large scales like smart cities. When a new fog node joins a domain, it will broadcast its presence and from that moment, it will be available to host application components and share the workload.

1.2 Motivation and Problem Statement

Cloud computing comes with several inherent capabilities such as scalability, on-demand resource allocation, and easy application and services provisioning. However, the connectivity between the cloud and the end-users is set over the Internet, which may not be suitable for a large set of cloud-based applications such as the latency-sensitive Internet of Things (IoT) applications. Furthermore, an emerging wave of Internet deployments, most notably the Internet of Things (IoT), requires mobility support and geo-distribution in addition to location awareness and low latency [4]. Fog computing can reduce this latency by extending the traditional cloud architecture to the edge of the network and by enabling the deployment of some application components on fog nodes. This extension results in a hybrid cloud/fog system. Existing PaaS solutions (including IoT PaaS solutions) usually focus on cloud computing and do not enable the provisioning of applications with components spanning both cloud and fog. Designing such architecture for hybrid cloud/fog environments faces several challenges including development, deployment, execution and management, and orchestration phases related challenges. Discovering the cloud and the fog nodes by the PaaS is one of the deployment phase challenges. Current publication and discovery modules need to be significantly extended in order to fit in a hybrid cloud/fog environment. The PaaS should be aware of existing cloud and fog nodes (joining and leaving) with their specifications (e.g., capacity, cost, latency) in order to generate efficient placement plans. Another challenge is to ensure that there are appropriate control interfaces to enable interoperability at the level of providers and architectural modules, since PaaS interaction with cloud and fog nodes is a particular requirement in the processes of deployment and migration of application components between cloud and fog nodes.

1.3 Thesis Contribution

The thesis contributions are as follows:

- A set of requirements on the general architecture of an IoT PaaS for cloud and fog environment;
- A review on the state of the art solutions for PaaS based on our sets of our requirements;
- A High level IoT PaaS architecture for hybrid cloud and fog environments;
- A mechanism for publication and discovery of the existing cloud and fog nodes (both stationary and mobile fog nodes);
- A set of RESTful interface for interaction with components internally and externally;
- An implementation architecture, a proof of concept prototype, and performance evaluation.

1.4 Thesis Organization

The rest of the thesis is organized as follows:

- Chapter 2 presents the key concepts related to our research domain in detail.
- Chapter 3 introduces the motivating scenarios and the set of requirements derived from these scenarios. The state of the art is also evaluated against the requirements.

- Chapter 4 presents the proposed architecture for the IoT PaaS for hybrid cloud and fog environments. Functional entities and the proposed interfaces are discussed. A concrete mechanism for publication and discovery of the existing cloud and fog nodes is also introduced.
- Chapter 5 describes the implementation architecture and technologies used for the proof-of-concept prototype. Then the performance measurements evaluating the architecture are presented.
- Chapter 6 concludes the thesis by providing a summary of the overall contributions and identifying the future research directions.

Chapter 2

Background

This chapter presents the background concepts relevant to the research domain of this thesis. The following concepts are explained in the upcoming sections: Internet of Things (IoT), Cloud Computing with a specific emphasis on Platform as a Service (PaaS), and Fog Computing.

2.1 Internet of Things

This section presents a general overview of the internet of things (IoT). We start with a brief definition of IoT. Then, we present the IoT main architecture, followed by a subsection that discusses the main requirements of IoT.

2.1.1 General Definition of IoT

Internet of Things (IoT), according to the definition provided by the ITU, is “A global infrastructure for the information society enabling advanced services by interconnecting (physical and virtual) things based on existing interoperable information and communication technologies” [1]. Through the exploitation of identification, data capture, processing and communication capabilities, the IoT makes full use of things to offer services to all kinds of applications, whilst ensuring that security and privacy requirements are fulfilled. According to Jie Lin and Wei Yu [6], the Internet of Things can be defined as “IoT can connect ubiquitous devices and facilities with various networks to provide efficient and secure services for all applications anytime and anywhere." The basic idea of this concept is the pervasive presence of a variety of things or objects such as sensors, RFID tags, actuators, mobile phones that capture, send and act on data they obtain from their environments. IoT devices can be classified into two categories based on functionality: Sensor devices (e.g., sensors), and Actuation devices (e.g., robots). Sensors have the ability of sensing the state of an environment or an object, limited processing, and communicating to the other connected devices in the network. While the later, executes the commands received from other connected systems based on the gathered data by sensors [7]. Actuators are frequently used in combination with sensors to produce sensor-actuator networks [8].

2.1.2 IoT Applications

There are various application domains which will be affected by the emerging Internet of Things. The applications can be classified based on the type of network availability, coverage, scale, heterogeneity, repeatability, user involvement and impact [9]. Different categorizations for IoT applications are presented by the literature [8, 9, 10]. According to [11], IoT enterprise applications can be classified into three IoT categories: (1) monitoring and control, (2) big data and business analytics, and (3) information sharing and collaboration. A few typical applications in each domain are discussed below:

2.1.2.1 Monitoring and Control

Monitoring and control systems collect data on equipment performance, energy usage, and environmental conditions, and allow managers and automated controllers to constantly track performance in real time anywhere, anytime [11]. Smart home is the best example of monitoring and control systems.

2.1.2.2 Big Data and Business Analytics

IoT devices and machines with embedded sensors and actuators generate enormous amounts of data and transmit it to business intelligence and analytics tools for humans to make decisions. These data helps to discover and resolve business issues such as changes in customer behaviors and market conditions to increase customer satisfaction, and to provide value-added services to customers. Healthcare systems are one the examples of IoT applications in this domain. In this case, business analytics tools can be embedded into the IoT devices such as wearable health monitoring sensors to generate real time decisions [11].

2.1.2.3 Information Sharing and Collaboration

Information sharing and collaboration in the IoT can occur between people, between people and things, and between things. The first step of information sharing and collaboration is sensing a predefined event. Supply chain area is the best example of IoT application in this domain in which information sharing and collaboration enhance situational awareness and avoid information delay and distortion [11].

2.1.3 IoT Main Architecture

Typically, the architecture of IoT is composed of three basic layers which is shown in Figure 1. These layers are: perception, network, and application layer, representing the data source, data communication networks, and data processing, respectively [6].

1. Perception Layer: It is also known as the sensor/device layer. The perception layer interacts with physical devices and components through smart devices (RFID, sensors, actuators, etc.) [6]. All these sensors and end devices are interconnected in order to exchange data with each other and connected applications and provide additional services [12]. An IoT device communicates to other devices via several interfaces, both wired and wireless.

2. Network Layer: It is also known as the transmission layer, is implemented as the middle layer in IoT architecture [13]. The network layer is used to receive the processed information provided by the perception layer and specify the routes to transmit the data and information to the IoT hub, devices, and applications via integrated networks. This layer is the most important layer in IoT architecture because various devices (hub, switching, gateway, cloud computing perform, etc.), and various communication technologies (Bluetooth, Wi-Fi, long-term evolution, etc.) are integrated into this layer. It should transmit data to or from different things or applications, through interfaces or gateways among heterogeneous networks, and by using various communication technologies and protocols [6].

3. Application Layer: It is also known as the business layer, and is implemented as the top layer in IoT architecture. The application layer receives the data transmitted from the network layer and uses the data to provide required services or operations. A number of applications exist in this layer, each having different requirements. Examples include smart cities, health care, smart homes, etc [6].

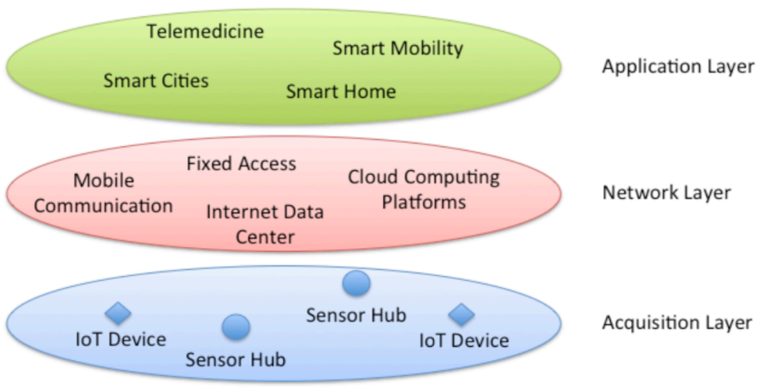


Figure 1: Basic architecture model for IoT [13]

2.1.4 Requirements of IoT

In this subsection some of the main requirements of IoT are discussed:

- Self-organizing Capability: Unlike computers which need users to configure, smart devices need to organize, configure, adapt to situations without involving humans. In order to meet the scale and complexity of IoT, devices need to manage themselves without external intervention. Automatic service discovery, automatic device discovery without requiring external trigger and the ability to adaptively tune to protocol behavior are the examples of Self-organization capabilities [10].

- Interoperable communication protocols: In IoT, different kinds of smart objects have different capabilities in terms of computation, communication, bandwidth, energy available, etc., [10]. To facilitate communication between these different types of devices, interoperable communication protocols are needed. These protocols enable communication between devices and also with the infrastructure. Typically, IoT communication protocols can be divided into two types: (1) lower layer communication protocols like RFID, BLE, etc., (2) higher layer communication protocols like COAP, REST, TCP, UDP, 6LoWPAN, etc.

- Identification: In IoT, physical objects such as home appliances, vehicles, supply chain items, containers, etc., should have unique identities for interacting among themselves (such as IP address or URI) [14]. Unique identity enables users to query the devices, monitor their status, and control them. In addition, it enables the interaction between devices and also the applications. Having a unique identity in the IoT system makes it possible for devices to freely move within the networks. All these devices carry context information that can allow further track within the network [15].

2.2 Cloud Computing

This section presents a general overview of cloud computing. We start with a brief definition of cloud computing followed by a specific focus on the Platform as a Service (PaaS). Finally, the description is concluded giving the types of cloud and the advantages of using it.

2.2.1 Definition

According to definition provided by NIST, "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" [3]. Yet, in [2], after gathering definitions proposed by many experts, they provided an encompassing one as "Clouds are a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms and/or services). These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing also for optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the Infrastructure Provider by means of customized SLAs". It is composed of three service models, Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS). IaaS enables consumers to provide all kinds of fundamental computing resources such as network, storage, and processing. The middle layer, PaaS, is described in detail in the following subsection. The highest layer of the cloud, namely SaaS provides online services or applications running on cloud infrastructure.

2.2.2 Types of Cloud

Depending on different levels of management and security, we have different types of cloud which are described below.

2.2.2.1 Private Cloud

Private clouds are designed to be exclusively used by a single organization and it is not a shared property. This type of cloud offers the highest degree of control over performance, reliability, and security. However, it does not offer some benefits of cloud computing such as no up-front capital cost. Accordingly, it is criticized to be similar to traditional server farms [16]. It may be owned, managed, and operated by the organization itself or a third party company or some combination of the two [3].

2.2.2.2 Public Cloud

Service providers offer their resources as services for open use by the general public with some key features such as no initial capital investment on infrastructure [16]. It may be owned, managed, and operated by a business, academic, or government organization, or some combination of them [3]. All of the computing infrastructures are located on the service provided premises.

2.2.2.3 Hybrid Cloud

A hybrid cloud is the combination of public and private clouds with more flexibility. In this model, a part of the service infrastructure runs in a public cloud and the remaining part runs in a private cloud. Therefore, finding the best split between public and private clouds makes designing a hybrid cloud a careful task. Hybrid clouds provide tighter control and security over application data compared to public clouds, while still facilitating on-demand service expansion and contraction [16].

2.2.2.4 Community Cloud

A community cloud extends a private cloud for the exclusive use of a specific group or community of organizations with similar requirements. The ownership can be for one or more of the organizations in the community [3].

2.2.3 Benefits of Cloud Computing

Cloud computing offers several important advantages. Some of these features include:

- Reliability: Services running on the cloud should meet several desired requirements such as Quality of Service (QoS), availability, performance, fault tolerance, etc. These requirements should be according to the negotiated framework of the Service Level Agreement (SLA) between cloud service providers and customers. Hence, SLA assurance is a critical objective of every cloud provider [16].
- Geo-distribution and ubiquitous network access: Clouds are generally accessible via internet connection. Therefore, any device with an internet connection can access cloud services. In addition, there are several data centers around the world in order to increase the high network performance and locality. A service provider can easily leverage geo-diversity to achieve maximum service utility [16].
- Shared resource pooling: There is a pool of virtualized and physical computing resources offered by infrastructure providers that can be dynamically assigned to different consumers. This feature is one of the reasons for cloud services flexibility.
- Dynamic resource provisioning: This feature allows service providers to obtain and release resources based on the current demand and on the fly, which can considerably lower the operating cost.

2.3 Platform as a Service (PaaS)

In this section, we provide a definition of PaaS, then we discuss the benefits of using PaaS briefly. Finally, we conclude the description of PaaS by providing the evaluation of PaaS.

2.3.1 Definition

PaaS is a cloud computing offering that facilitates the provisioning of applications hosted in a cloud environment. By provisioning, we mean developing, deploying, managing, and orchestrating an application. Generally, these facilities operate as one or more virtual machines (VMs) running on top of a hypervisor in a host server. In fact, the purpose of the PaaS is to lessen the burden of deploying applications directly into VM containers. For example, Google App Engine operates at the platform layer to provide API support for implementing storage, database and business logic of typical web applications [16]. PaaS eliminates the need for buying and maintaining the underlying hardware and software and provisioning hosting capabilities by providing the infrastructure required including network, servers, operating systems, or storage to support the life cycle of the applications and services available from the Internet.

Similar to the way in which you might create macros in Excel, PaaS allows you to create applications using software components that are built into the PaaS (middleware). Applications using PaaS inherit cloud characteristics such as scalability, high-availability, multi-tenancy, SaaS enablement, and more. Enterprises benefit from PaaS because it reduces the amount of coding necessary, automates business policy, and helps migrate apps to a hybrid model. For the needs of enterprises and other organizations, Apprenda is one provider of a private cloud PaaS for .NET and Java [17].

The PaaS can be placed on multiple hosts, referred to as PaaS nodes, which may reside on one or more cloud-based systems provided by one or more cloud providers. PaaS consumers can be application developers who design and implement application software, application testers who run and test applications in various cloud-based environments, application deployers who publish applications into the cloud, and application administrators who configure and monitor application performance on a platform [3]. There are different strategies to bill the PaaS consumers, e.g., the number of consumers, the type of resources consumed by the platform, or the duration of platform usage.

PaaS can be deployed as public cloud services or as private cloud services. Public cloud PaaS mostly focuses on cloud of database, application middleware and Runtime management. The private cloud PaaS platform adds the concept of DaaS (database as a service) service layer at the database level, increasing the abilities of technical and business building upon the application of middleware [18]. However, using private PaaS may result in less efficiency than public ones, due to reduced chance of resource sharing which may cause cost increase.

2.3.2 PaaS Evolution

The evolution of PaaS is moving towards container based, interoperable PaaS.

- The first generation consists of classical fixed proprietary platforms such as Azure or Heroku [19].
- The second generation was built around open-source solutions such as Cloud Foundry or OpenShift that allow users to run their own PaaS (on-premise or in the cloud), already built around containers. OpenShift moves now from its own container model to the Docker container model, as does Cloud Foundry through its internal Diego solution.
- The third-generation includes platforms like Dawn, Deis, Octohost, and Tsuru, which are built on Docker from scratch and are deployable on their own servers or on public IaaS clouds.

OpenPaaS like Cloud Foundry and OpenShift treat containers differently. While Cloud Foundry supports state-less applications through containers, stateful services run in VMs. OpenShift does not distinguish these [19].

2.4 Fog computing

According to the definition provided by OpenFog Consortium, Fog computing is: “A horizontal, system-level architecture that distributes computing, storage, control, and networking functions closer to the users along a cloud-to-thing continuum” [20]. In other words, computation, storage, control, and networking functions are the building blocks of cloud computing which are extended by fog computing in a way that it preserves all the cloud advantages including containerization, virtualization, orchestration, manageability, and efficiency. The fog computing concept was first introduced by Cisco in 2012 to support IoT applications. IoT devices/sensors are highly distributed at the edge of the network along with real-time and latency-sensitive service requirements [21]. In the literature, it is widely acknowledged that cloud computing is not viable for most of the Internet of Things (IoT) applications [22]. Considering centralized architecture of cloud computing, it is not the best choice for IoT applications with massive data transferring and processing. Fog computing is not exclusively located at the edge of the network and can be implemented in multiple layers of a network’s topology. In this sense, fog computing can help by offering densely distributed points for gathering data generated by the end devices. Besides, fog computing can perform efficiently in terms of the requirement for mobility and geo-distribution, power consumption, network traffic, capital, and operational expenses, etc [21]. This is done through traditional networking components such as proxies, access points, base stations, and routers positioned at the network edge, near the sources [22]. It should be pointed out that fog is tightly linked to the existence of a cloud. It means that it cannot operate in a standalone mode. This has driven particular attention to the interactions between the cloud and the fog [22].

2.4.1 Fog Computing Architecture

The hierarchical structure of fog computing with three tires is shown in Figure 2. It has three strata: The cloud stratum, the fog stratum, and the IoT/end-users stratum [22]. The lowest layer which is IoT/end-user stratum includes two categories: IoT devices and end-user devices. It should be noted that there is no fixed rule about having both categories at the same time in this stratum. Although devices in this stratum generate a massive amount of data and can run critical real-time tasks, they have limited resources in terms of computation, memory, storage and battery power. Therefore, there is a need to send this data to the higher layers for processing.

The fog stratum features multiple fog domains from the same or different service providers. Each of these domains may contain many fog nodes including fixed or mobile ones such as routers, switches, gateways, access points, PCs, smartphones, etc. In fact, there is no specific rule or requirement for a device to be considered as a fog node. The highest layer which is cloud stratum has unlimited computational and storage capabilities. For that reason, resource-intensive tasks, such as machine learning or the persistence of large data sets, are transferred to this stratum. It should be noted that the IoT/end-user stratum can be directly connected to the cloud stratum.

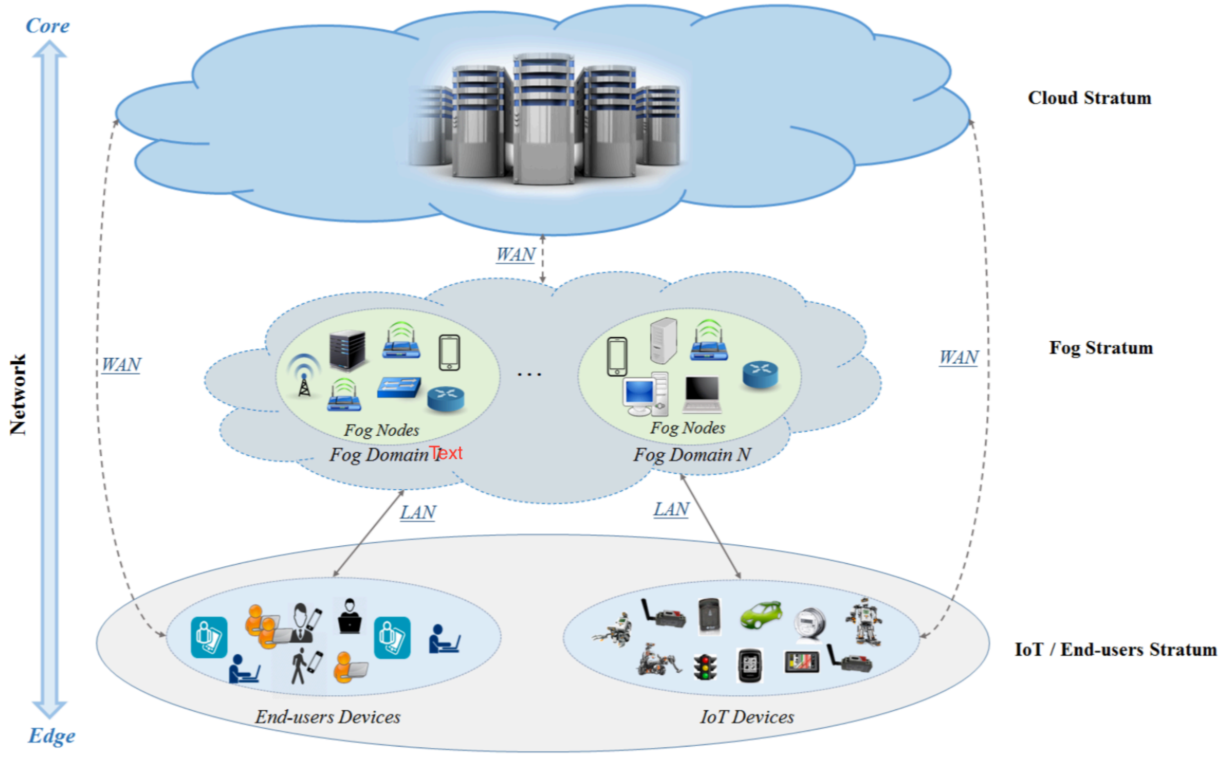


Figure 2: The fog system [22]

2.4.2 Fog computing in support of the IoT

Fog computing can serve as an optimal choice for the IoT designers for the following features:

- Location: Fog resources are placed between smart objects and the cloud data-centers, providing better delay performance.
- Distribution: Fog resources are limited in terms of storage, processing and communication capabilities compared to the cloud. Hence, it is much cheaper to deploy many “micro” centers of fog resources closer to the end-users.
- Scalability: It is possible to increase the number of fog “micro” centers as the number of end-users increases. Such an increase cannot be achieved by the cloud because the deployment of new data-centers is cost prohibitive.
- Density of devices: Fog helps to provide resilient and replicated services.
- Mobility support: Fog resources act as a “mobile” cloud as it is located close to the end-users and some of them are actually mobile nodes.
- Real-time: Fog has the potential to provide better performance for real-time interactive services.
- Standardization: Fog resources can interoperate with various cloud provides.
- On the fly analysis: Fog resources can perform data aggregation to send partially processed data as opposed to raw data to the cloud data-centers for further processing.

Therefore, fog computing has the potential to increase the overall performance of IoT applications as it tries to perform part of high level services which are offered by cloud inside the local resources [23].

2.5. Conclusion

In this chapter, we focused on the major concepts that are relevant to this thesis. The chapter began with a brief discussion on the IoT covering the definition, the main architecture, and the requirements of IoT. Then, we followed by discussing the concept of cloud computing, its different definitions, characteristics, benefits, and different types of cloud, followed by a section for the PaaS with its definition, and evolution. Fog computing and its architecture described in brief before concluding the chapter.

Chapter 3

Use cases and State of the art

In order to capture the requirements of an IoT PaaS, two motivating use cases are first presented, then the requirements are derived from them. Finally, we evaluate and summarize the current state of the arts against the requirements followed by the conclusion.

3.1 Use Cases

This section presents two illustrative motivating scenarios; a smart parade scenario and a smart accident management scenario. The motivating scenarios help to derive the requirements for an IoT PaaS solution that enables the provisioning of these type of applications with components spanning both cloud and fog. These types of applications are composed of multiple components that interact with each other based on different sub-structures such as sequence, parallel, selection, and loop structures [24]. Such applications must be modeled as graphs with these sub-structures, and chains need to be created between the components to define the relationship between them. These scenarios are presented in detail the application graphs depicted in Figure 2 and Figure 3.

3.1.1 Use Case 1: Smart Parade Scenario

We consider a smart parade application to illustrate the motivation behind our work. The idea is to get some information about the people participating in a parade and offer some real-time services based on captured data. The application captures parade footage and analyzes it to identify some patterns and/or security threats, such as an anomaly in the people or environment. This IoT application is composed of several components, as shown in Figure 3. For instance, the Capture Parade Footage component derives visible patterns from the parade footage and sends those patterns to the Parade Footage Analyzer for analysis. It can, for instance, identify the clothing brands of most of the people, and send advertisements of those brands more frequently to those people's phones.

The application uses Facial Recognition techniques to identify the ethnicities and the ages of the various parade participants. This allows advertising companies (through the Advertisement Issuer component) to release ads targeting those age groups and ethnicities. Analyzing the parade footage can also help in identifying security threats. For example, Visible Pattern Deriver can detect any sudden scattering of the crowd, which could be an indication of an altercation/ physical fight between a few individuals. Another example is being able to detect if parade participants enter any restricted areas. In such cases, the suspected patterns can be sent to the Warning Alert Issuer, where the latter notifies the respective authorities (Ambulance, police, etc.). In addition, all the derived patterns can be sent to a Historical Storage system for long term storage and to a Results Displayer component to display results relevant such as the total number of participants to the parade.

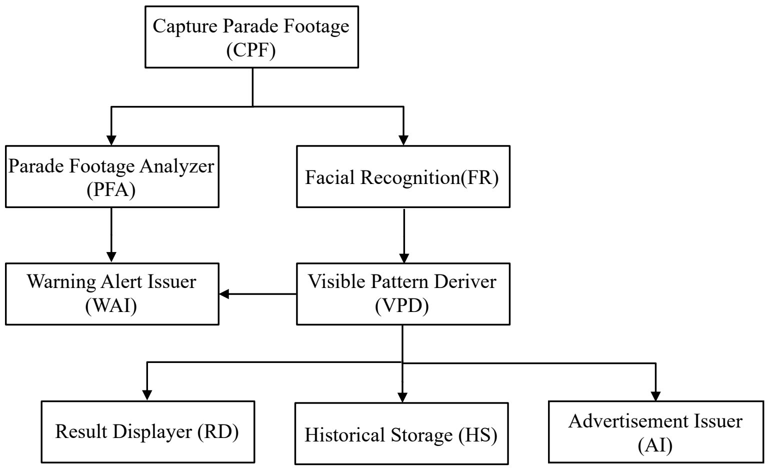

Capture Parade Footage (CPF)
Parade Footage Analyzer (PFA)
Facial Recognition(FR)
Warning Alert Issuer (WAI)
Visible Pattern Deriver (VPD)
Result Displayer (RD)
Historical Storage (HS)
Advertisement Issuer (AI)

Figure 3: The smart parade application

3.1.2 Use Case 2: Smart Accident Management Scenario

Smart transportation is an important pillar for the quality of life of citizens in a city. According to the World Health Organization (WHO) 2013, the total number of road traffic deaths is 1.24 million per year worldwide, while the number of injuries caused by crashes is more than 20 million [20]. Accordingly, we consider a smart accident management application running in a smart city environment that offers innovative services related to accident management. In the current settings, there is too much hassle to overcome in order to reach the scene of the accident both for an ambulance or fire engine and people in the streets and cars. This application enables innovative services related to accident management. To that end, it decreases the time needed for an ambulance to reach the scene of an accident. Also, it omits the sounds of sirens, which can be stressful for the elderly and for infants.

This application can be composed of several components, as shown in Figure 4. For instance, a Collision Detector can detect collisions/crashes and share the location of the crash to an Alert Issuer on the nearest Road Side Unit (RSU). The Alert Issuer informs the Emergency Planner for real-time emergency response management. The application can also find the shortest path between the accident scene and the emergency vehicle through a Road Planner component. This component shares the real-time location of the ambulance with a Car Detector & Notifier component, which is originally hosted on the RSU closest to the ambulance's initial location. The Car Detector & Notifier keeps migrating to RSUs one step ahead of the ambulance in order to detect all the cars on the same street and direction as the ambulance. It sends a message to cars to move to the right so that the ambulance can pass easily. The Car Detector & Notifier can also coordinate with a Traffic Light Manager component to facilitate and accelerate the ambulance movement. In addition, all the accident data can be sent to a Diagnostics & Prognostics component for further analysis and long-term storage.

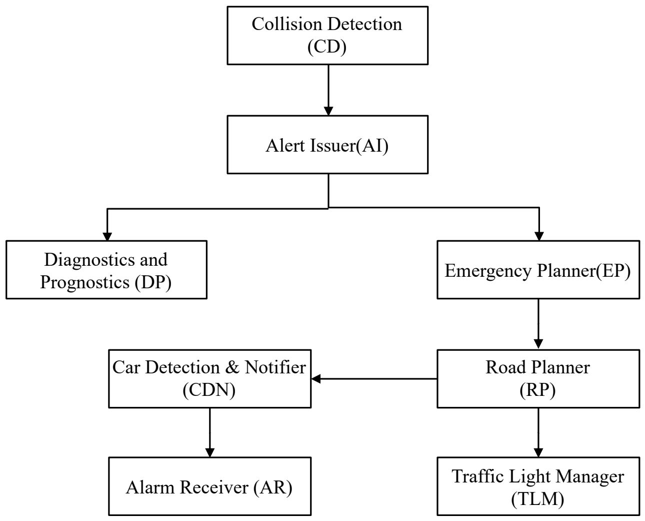


Figure 4: The smart accident management

3.2 Requirements

According to the motivating scenarios, the identified requirements cover the whole IoT application's lifecycle, i.e., development, deployment, execution, management, and orchestration which is discussed in the next subsections.

3.2.1 Development Layer

- *The PaaS should enable the development of application components that can be hosted and executed in either cloud or fog.*

This is needed because for instance, in smart accident management use case, the collision detection component is a latency sensitive component and might be better to run it in fog while the diagnostics and prognostics is computationally intensive and delay tolerant component and might be better to run it in the cloud.

- *The PaaS should be able to generate application graphs*

In smart accident management use case, the application is composed of a set of interacting components that can be executed in sequence, parallel, selection, and loop. Accordingly, they are modeled as graphs with these substructures.

- *The PaaS should be able to specify the application QoS requirements*

Each of these applications, i.e., smart parade and smart accident management, have specific QoS requirements. The PaaS should be able to specify the QoS requirement for each application such as the deadline threshold for each application so that the Deployment Engine finds the best deployment plan accordingly.

3.2.2 Deployment Layer

- *The PaaS needs a discovery mechanism that enables it to be aware of existing cloud and fog nodes.*

In both scenarios, the PaaS needs to be aware of existing (joining and leaving) cloud and fog nodes along with their specifications (e.g., capacity, cost) in order to be able to generate a placement plan and decide where to place each component.

- *The PaaS should be able to dynamically determine the optimal placement plan for application components given a specific set of objectives and constraints, and deploy these components over cloud and/or fog nodes as instructed by the placement plan.*

In the smart accident management scenario, there is need to make decisions whether to place components on cloud and/or fog such that the application requirements and the constraints are met (e.g., latency, cost). For instance, one may envision running the collision detection component in fog while the diagnostics and prognostics running in the cloud.

3.2.3 Execution and Management Layer

- *The PaaS should be able to interact with both cloud and fog nodes. This implies the need for appropriate control interfaces to enable interoperability at the level of providers and architectural modules:*

Considering the smart parade application, in order to support the application's lifecycle, this is needed when the PaaS wants to deploy application's components over cloud and fog nodes, when the PaaS wants to migrate the video analytics component from fog node to another, etc.

- *The PaaS should be able to find the best migration plan and migrate components form cloud to fog and vice versa, also from fog to fog according to the migration plan:*

As an example, in the smart accident management scenario, there is need to migrate the car detection and notifier components to the RSU in the road from which the ambulance will pass, to ensure the road is clear when the ambulance arrives. Also, in the smart parade scenario, the component performing video analytics should be migrated between the fog nodes along with the parade movement.

- *The PaaS should be able to create or update chains between the application components.*

For instance if an application component such as the video analytics in the smart parade scenario is migrated to another fog node, there is a need to update the chain to keep the application working properly.

3.2.4 Orchestration Layer

The PaaS needs an orchestrator for coordination purposes, this orchestrator needs to:

- *Orchestrates the cloud/fog resources and manages the application lifecycle including deployment, chaining, execution, monitoring, and migration*
- *Responsible of deciding whether to execute the deployment orchestration plan or the migration orchestration plan*

Considering the smart parade application, the application lifecycle provisioning process needs to be fully automated to enable the dynamic incorporation of the required changes. This is critical for instance in case of security threat. If a drone hosting a components that captures the parade footage in the security threat location moves to another location, the component needs to be migrated to another fog node in the security threat location immediately.

3.3 State of the Art

In the subsequent sections, we first evaluate the state of arts and draw summary from it focusing on architectures for PaaS for hybrid cloud/fog environments. Then, we evaluate and summarize the proposed architectures for PaaS for fog systems. Table 1 and Table 2 provide a summary of the papers reviewed in this section, in which we outline the requirements addressed by each paper.

3.3.1. Architectures for PaaS for Hybrid Cloud/Fog Systems

In this subsection, we first discuss the proposed architectures for PaaS that span the cloud and the fog, then in the subsequent subsection, we discuss the proposed fog architecture. In addition we evaluate each work against our set of requirements.

Relatively few works have proposed PaaS architecture for hybrid cloud/fog systems. Yangui *et al*. [25] propose a PaaS architecture for a hybrid cloud/fog system composed of four layers: development, deployment, hosting and execution, and management. The authors try to design new architecture as an extension to the high-level existing PaaS architectures and reuse existing modules as much as possible. Their proposed architecture is able to specify the applications' QoS requirements using the SLA Manager module. It also has appropriate control interfaces to enable interoperability between the PaaS and the fog. The deployed application components can also interact and exchange messages through the REST APIs. For the development phase, their proposed architecture provides an Integrated Development Environment (IDE) to enable the development of application components that can be hosted on either the cloud or the fog. However, this IDE uses existing application development frameworks to provide developers with the tool facilitating such development. Moreover, the proposed PaaS does not enable the discovery of newly joining or leaving cloud and fog nodes. The existing cloud/fog nodes are pre-configured. As of the deployment phase, they use service containers for deploying each application component. Accordingly, application isolation can be obtained if they place multiple application components on one host node. It also does not enable the optimal placement plan for these components to be determined. Finally, few details about orchestration is mentioned. In the case of having mobility in the usecase, more work should be done in the scope of orchestration.

Pahl *et al*. [26] present a container-based edge cloud PaaS architecture. The authors proposed using clusters of single-board computers like Raspberry Pis instead of gateways in order to reach IoT integration. This approach would benefit us in terms of cost, energy consumption, flexible placement of containers, and robustness against power failure. This architecture enables fog nodes to run their applications in containers as well as the orchestration of the deployment of those containers. Their proposed architecture for the cluster management has the following components: the service node (cluster), an API, a platform service manager, a lifecycle management agent, and a cluster head node service. However, this paper mostly talks about the concerns and technologies for a PaaS rather than proposing a concrete architecture. While their proposed architecture includes a development layer to provision and manage applications over cloud/fog nodes, it does not support the discovery of existing cloud/fog nodes or the generation of the best deployment plan. However, they claim that the proposed architecture has the ability to monitor joining/leaving nodes to the clusters. The master node in each cluster is responsible for (de)registration of nodes. Additionally, it is mentioned that since just the necessary components in the form of containers remain on the local resources, this architecture is dynamic. The proposed architecture enables the migrating of containers, but it does not enable the best migration plans to be generated.

In contrast to Yangui *et al*. [25] and Pahl *et al*. [26], the PaaS architecture proposed by Liyanage *et al*. [27] enables generating the best deployment plan by proposing a component distribution scheme. Their main contribution is proposing a service-oriented mobile-embedded PaaS architecture that allows users to deploy and execute their own applications on cloud and mist resources. Mist was proposed to reduce the burden on the fog. The proposed architecture supports resource-aware autonomous service configuration and takes the QoS requirements of an application into consideration. In addition, they incorporate a publication/discovery mechanism for the underlying Mist node's specifications and available service modules supported by the Mist node using Service Description Metadata (SDM). Since hardware usage of the mobile devices are dynamic, the activities on the device performing affects their service provisioning availability. Hence, the author proposes the service scheduling scheme to obtain dynamic updates on SDM. Although the publication mechanism is based on RESTful services, the interfaces between the remaining architectural modules are not discussed. They support task distribution among a group of Mist nodes and have some predefined workflow patterns. Accordingly, the proposed framework can create chains between different application components, however the author does not discuss the graph patterns and chaining in details. Although the authors claimed that they support migration, it does not support migration to/from cloud nodes and it is just defined between mist nodes. Finally, none of the remaining challenges discussed in our paper are addressed by this proposed architecture.

3.3.2 Summary of the State of the Art of the Architectures for Hybrid Cloud/ Fog Systems

Table 1 shows the summary of meeting the requirements against the proposed frameworks for hybrid cloud/fog systems. A "✓" means that the corresponding requirement was met by the proposed framework. A "x" means that the framework failed to meet the requirement.

Table 1: Summary of the related works for hybrid cloud/fog systems

Scope	**Papers**	**Requirements**									
		Development			**Deployment**		**Management**			**Orchestration**	
		Host apps in cloud/fog	**App graphs**	**Specify QoS**	**Discover cloud/fog nodes**	**Deployment plan**	**Control interface**	**Migration plan**	**Create chains**	**Orchestrate cloud/fog**	**Parse graphs**
Hybrid Cloud/Fog	Yangui *et al.* [25]	✓	x	✓	x	x	✓	✓	x	✓	x
	Pahl *et al.* [26]	✓	x	x	x	x	x	x	x	✓	x
	Liyanage et al. [27]	✓	✓	✓	✓	✓	x	x	✓	x	x

3.3.3 Architectures for PaaS for Fog Systems

In this subsection, we discuss the state of the art that proposed architectures for fog systems and evaluate them against our set of requirements.

Several works have proposed architectures for fog systems. Most of these architectures are designed to span the cloud and the fog, such as the architectures proposed by Yigitoglu *et al*. [28] and Saurez *et al*. [29]; only one architecture is strictly fog architecture, the architecture proposed by Donassolo *et al*. [32]. Yigitoglu *et al*. [28] provides a framework called Foggy that facilitates dynamic resource provisioning and automates application deployment in fog computing architectures. Foggy assumes that IoT devices can host Docker containers. It has three-tier architecture: edge devices (e.g., fog nodes), a network infrastructure to connect the edge devices to the cloud, and cloud services. The focus of the proposed framework is on the orchestration server which is responsible of the scalable IoT application deployment. For example, it enables determining an optimal deployment plan to run containers on the devices for optimal quality service. In addition, the deployment planner is designed to be pluggable that allows users to bring their own deployment planning strategies. It also proposes some predefined deployment plans in order to meet divers user's needs. However, it does not enable the fog nodes to be discovered dynamically and instead assumes a pre-configured list of the nodes. Moreover, creating chains between different application components and migrating application components among different nodes are not discussed. In the development phase, the developers push their containerized application packages and their specifications to the orchestrator to ensure the QoS for each application. The orchestrator is a central entity and is also in charge of monitoring the nodes' resources. They propose policy-based resource management procedure in order to have fast provisioning mechanism to meet the instant update requirements. All of this is possible through an orchestration client code (agent) running on top of the operating system on each IoT device. It is responsible of pulling the container image from the container registry, running/ stoping the application, as well as sending the monitoring related data to the central orchestration server.

Saurez *et al*. [29] propose a high-level programming model called Foglet that facilitates large scale distributed programming across the heterogeneous resources from IoT devices to the cloud. Besides, they provide an execution environment that enables the incremental provisioning of resources in the whole system. Their proposed framework provides communication APIs for discovering fog resources. Additionally, running application components can communicate to each other through the hierarchical and point-to-point API. Hence, control interface requirement is satisfied completely. In the proposed programming model, there are four entities in the Foglet runtime system: discovery server, docker registry server, entry point daemon, and worker process. Through this entities, foglet enables the developers to have distributed computing across the network. For example, entry point daemon is a process that contributes to the discovery and migration processes and responsible to keep the discovery server to be updated regarding the available fog nodes. Also, worker process is the one who executes the functionality contained in a specific application component. Additionally, worker proceeds monitors its host fog node . Hence, by having multiple worker processes and containerized application components, they can place multi-application collocation on any computational node. It also enables QoS-aware incremental deployment over different fog nodes via containerization. Foglet first places application components at the lowest layer, and gradually finds the best candidates in upper layers; hence it does not enable determining the optimal deployment plan. The main focus of this work is on the different approaches for migration. The migration algorithm is triggered by the worker process running on the fog node who reaches a threshold for its end-to-end latency to find the best candidate for the migration. Therefore, migration process is not initiating from a central module and it is distributed on the fog nodes. Accordingly, the migration is only supported between fog nodes. The orchestrator is responsible for the deployment and migration of the application components. However, it is not capable of parsing application graphs. In addition, there is no discussion on how to create or update chains among different application components.

Tao *et al*. [30] propose an architecture called Foud that can facilitate the growth of Vehicle to Grid (V2G) services and applications. Powerful Computing and Standardized Data Storage, Balanced Power Management Services and Applications, Resource-Sensitive Services and Applications, Location-Aware Services and Applications, Business Services and Applications are the examples of the applications which Foud helps to extend. Their proposed architecture is inspired by cross-layer design and organized over three layers: the user layer, which is composed of different types of end-users in V2G systems, the service layer, which is made up of two sub-models, permanent cloud and temporary fog, and the network layer. The network layer provides an interconnection between the cloud and the fog by taking advantage of 5G communications. It basically provides protocol, interface, and security (e.g., certification and authorization) techniques. Accordingly, global control for the V2G system and interoperability between the two sub-models is achieved. By using OMNET++, the author claims that the transmission performance improves significantly when using 5G technologies. However, most of the execution and management layer challenges are not discussed, such as migrating applications/ components between cloud/fog nodes (which is critical to support the mobility of end-users and fog nodes), and chaining application components. In addition, orchestrating the cloud/fog resources and managing applications' lifecycles are not discussed in this proposed architecture. Finally, due to electric vehicle mobility and the dynamic participation of mobile computing resources in temporary fog, the proposed architecture does not ensure the applications' desired level of QoS.

Tuli *et al*. [31] propose a lightweight framework called FogBus to integrate IoT-enabled systems, fog, and cloud infrastructures. Their proposed framework uses blockchain mechanisms to provide secure and authenticated data transfer between IoT devices, fog nodes, and cloud data centers which consequently increases the reliability of the framework. It also enables implementing resource management and scheduling policies for executing different concurrent/ application programming models such as SPMD (Single Program and Multiple Data), workflows, and streams spanning the cloud and the fog. This framework functions as a Platform-as-a-Service (PaaS) model for integrated fog/cloud environment. Hence, the author claims that this framework can generate optimal deployment plans using the resource manager module. This module identifies the requirements of different applications and selects the suitable resources to execute the applications accordingly, thereby determining optimal application placement plans. The author proposes both a high level view of integrating hardware and software components. Additionally, network structure is discussed in terms of topology, scalability, reliability, security, and performance. For instance, the procedure which is used to overcome a master fog node outage is discussed in detail. Although the authors claimed that the framework supports scalability, we argue that, it is not an automatic procedure and it should be done by the service provides by scaling up the number of fog nodes. In addition, it does not enable the discovery of the underlying joining and leaving fog nodes. This framework is also capable of monitoring the applications to ensure the QoS requirements are met. In the case of QoS violation, the framework initiates application migration. However, creating and updating chains between the components is not discussed. The applications' details are maintained in a catalog that contains information about different applications, including their operations, resource requirements, and dependencies. However, it is not mentioned if this catalog supports application graphs with interacting components using different substructures. The proposed FogBus framework is developed in cross-platform programming languages and REST-based interfaces that helps to overcome the OS and P2P communication-level heterogeneity of different fog nodes. Hence, it enables interoperability. Finally, it is mentioned that in order to ensure application-level consistency, a fog node can run at most one application at a time which is a drawback of this framework. Using container technology to host the applications on the fog nodes can tackle this problem.

Donassolo *et al.* [32] propose an orchestration framework for the automation of the deployment, the scalability management, and the migration of component-based IoT applications in the fog environment called FITOR. Additionally, they propose an Optimized Fog Service Provisioning solution named O-FSP which optimizes the placement and composition of the IoT application components and deals with the heterogeneity of underlying fog infrastructure. The proposed strategy for the provisioning problem is validated and compared to several classical approaches. The authors claim that O-FSP is better than the other approaches in terms of the acceptance rate of IoT apps, provisioning cost, and CPU usage. Their proposed architecture offers a general centralized framework for holistic fog resource orchestration and application orchestration. Using this framework, application components can be deployed on either the end devices or the fog nodes. It is not discussed if the components can be deployed over the cloud nodes. The framework also includes a module called a service descriptor that describes the application, its components, and the components' requirements. However, it is not clear if this module can describe application graphs with interacting components using substructures. Additionally, they mention using a directed acyclic graph (DAG), which does not include loops, for modeling IoT application components used for the provisioning problem. Also, to make sure of having guarantied QoS in the service descriptor module, the information of network-related requirements for the links between actors, is described. The service manager module of the proposed framework can deal with dynamic applications with continuous monitoring of application containers and use scale up/down actions to manage resources or trigger migration actions when necessary. The proposed architecture contains an infrastructure monitor module, which extracts real-time information about both host-related and network-related metrics. The first one concerns all the information about the host and containers in which the application is running. The later is to meet the latency and bandwidth requirements. However, generating and updating chains between the components is not discussed. In addition, having appropriate control interfaces to enable interoperability is not supported. Finally, the proposed architecture is not capable of discovering the underlying fog nodes (joining/leaving) when generating the placement plan.

Liu *et al*. [33] propose a fog computing architecture for resource allocation. The main objective of this work is to investigate the model of resource allocation, subtask scheduling, and optimization for the perps of latency reduction in fog computing. It considers latency reduction combined with reliability, fault tolerance, and privacy. The authors propose the architecture of fog computing which shows the procedures from original data to processed data and also the design of a fog node which is also based on the architecture of fog computing. This fog computing architecture is elaborated in two parts: computing and networking. Four layers are considered for the computing part which in fact represent the design of a fog node: a hardware platform, a software and virtualization platform, functional components, and a fog computing applications interface. The networking side is composed of three layers: wireless technology, single-hop/ad-hoc communications, and a software-defined network concept. The authors formulate the resource optimization problem considering the QoS in terms of latency and use a genetic algorithm to solve it. Hence, this approach supports generating the best deployment plan. In terms of fault tolerance, there is a tradeoff between latency minimization and task protection. One solution proposed by the authors is using virtual machine mechanisms for the backup of subtasks instead of simply replicating the fragment of a task on physical nodes. Since, the latter would degrade the overall performance if no failure occurs in the fog nodes. However, fog nodes have low capacity to host multiple VMs. In addition, they consider ~~both~~ the fog as well as the cloud for hosting application components. The proposed architecture includes an orchestration that is responsible for analyzing, planning, and executing a task. However, none of the remaining challenges presented in our paper is addressed by the proposed fog computing architecture. It should be noted that none of the works presented here enable generating or parsing application graphs to model the interactions between different components of an application. In contrast, we introduce a novel module that can generate application graphs as well as model the interactions between the different application components. In addition, none of the presented papers enable the creating or updating of chains between the application components. These chains are necessary when, for example, a component is migrated from one node to another, where the chain needs to be updated such that the application works properly. Moreover, several functions offered by existing PaaS need to be significantly extended in order to fit in a hybrid cloud/fog system, such as the publication/discovery function proposed in [27] [29] [30].

Sami *et al.* [34] propose an on-demand fog framework which benefits from containerization and micro-service technologies. The main objective of this architecture is to obtain optimized and on the fly deployment of services on the fog nodes with the opportunity of using volunteering devices. This framework has the ability to discover volunteer fog nodes on demand. The fog nodes can be part of a cluster orchestrating by one master node or directly managed by the central orchestration engine running on the cloud, in case there is not enough resources to initial a new orchestrator near the end-user. In case of a change in the fog node list, the orchestrator publishes the information to all other users in the same cluster. Moreover, they have proposed a container placement scheme that selects the best candidates for hosting the services. To reach this goal, they have used an evolutionary Memetic Algorithm to solve the multi objective container placement problem. However, services can only be deployed on the fog nodes. Accordingly, there is neither migration between fog and cloud nodes nor orchestration of cloud nodes. In addition, having multi-component applications or any relation between running services is not discussed. Hence, it does not meet the generating application graph requirement. The proposed architecture includes an orchestration mechanism that can be distributed among the fog nodes by leveraging Kubernetes functionalities. The orchestrator is responsible to trigger the decision module to run the placement plan and to push the services to the fog nodes. Also, it can monitor the fog nodes' status and services to ensure the QoS requirements are met. In case of having an overloaded fog node, the orchestrator can function as load balancer and decide wether to run another container on the same node or distribute the running services on the other available fog nodes. Additionally, orchestrator has the ability to get a back-up from a fog node as they want to leave the cluster in a scheduled time. Therefore, migration due to both overloaded fog node and leaving fog node to another fog node is supported. However, generating and updating chains between the services is not discussed. In addition, having appropriate control interfaces to enable interoperability is missing.

3.3.4 Summary of the State of the Art of the Architectures for Fog Systems

Table 2 shows the summary of meeting the requirements against the proposed frameworks for the fog systems. A "**✓**" means that the corresponding requirement was met by the proposed framework. A "x" means that the framework failed to met the requirement.

Table 2: Summary of the related works for fog systems

Scope	**Papers**	**Requirements**									
		Development			**Deployment**		**Management**			**Orchestration**	
		Host apps in cloud/fog	**App graphs**	**Specify QoS**	**Discover cloud/fog nodes**	**Deployment plan**	**Control interface**	**Migration plan**	**Create chains**	**Orchestrate cloud/fog**	**Parse graphs**
Fog Systems	Yigitoglu *et al.* [28]	✓	x	✓	x	✓	x	x	x	✓	x
	Saurez *et al.* [29]	✓	x	✓	✓	x	✓	✓	x	✓	x
	Tao *et al.* [30]	x	x	x	✓	x	✓	x	x	x	x
	Tuli *et al.* [31]	✓	x	✓	x	✓	✓	✓	x	x	x
	Donassolo *et al.* [32]	x	x	✓	x	✓	x	✓	x	x	x
	Liu et al. [33]	✓	x	✓	x	✓	x	x	x	✓	x
	Sami et al. [34]	x	x	✓	x	✓	x	x	x	x	x

3.4 Conclusion

In this chapter, we first presented two motivating use cases. Using these use cases, we derived the requirements for the IoT PaaS for hybrid cloud/fog environments. Then, we reviewed the current state of the arts against our derived requirements. For each paper, we showed the requirements which are met and the ones which are not met. Apparently, none of the states of art was able to fulfill all of the requirements. Finally, we presented two summary tables showing which of the requirements were fulfilled by which state of the art.

In the next chapter, we present our proposed architecture, describe the associated components, and functionalities in detail.

Chapter 4

The Architecture of the Platform

In this chapter, we focus on our high-level architecture of the proposed IoT PaaS for hybrid cloud/fog systems. First, we provide a general description of our designed architecture. Then, we focus on layer by layer module description based on application life-cycle phases (development, deployment, execution and management, orchestration). The two sets of the interface are also discussed in detail along the way. Then, the two main procedures about deployment and migration are explained with the help of presenting of the illustrative sequence diagrams for the interaction between involved modules within the IoT PaaS. Finally, we conclude this chapter by a brief evaluation of proposed architecture against the requirements followed by a conclusion section.

4.1 General Overview

A high-level view of the proposed architecture is shown in Figure 5. This architecture is consist of the PaaS, the cloud domain(s), and the fog domain(s). It should be noted that the PaaS could be running in the cloud domain, in the fog domain, or be provided by a third party. It can also be distributed across several domains. For instance, if we take the smart parade application, cameras could be distributed along the roads of the parade route to capture parade footage. Accordingly, some of the application components (e.g., Capture Parade Footage) will be distributed to improve its effectiveness. In such cases, it is better to distribute the PaaS across several domains to ease the development, the deployment, the management, and the orchestration of the application. The proposed architecture consists of different modules covering all of the application life-cycle phases. Accordingly, it is distributed over four layers: An

Application Development layer, an Application Deployment layer, an Application Execution, and Management layer, and an application Orchestration layer. It should be pointed out that some of the modules of the proposed architecture are novel, such as the App. Graph Generator and the Infrastructure Repository. These modules are depicted in yellow in Figure 5. Additionally, some other modules are extended modules from traditional PaaS architecture, shown in blue.

In the next section, we describe the modules in each layer of the PaaS and the modules in the cloud and fog domains along with their intended functionalities.

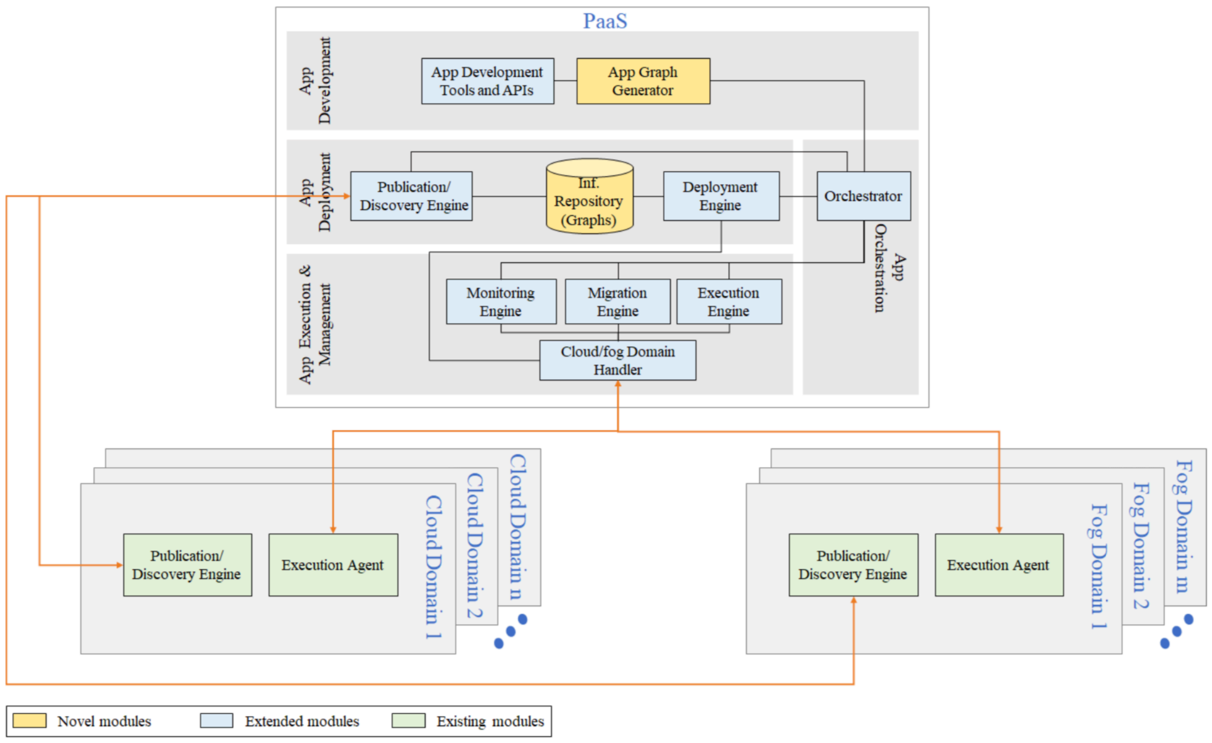


Figure 5: High-level architecture of IoT PaaS for hybrid cloud/fog

4.2 Application Development Layer

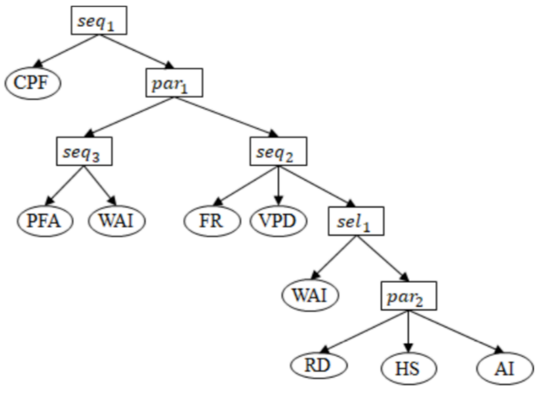
This layer contains two modules: The App Development Tools and APIs and the App Graph Generator. In the following subsections we explain each of them.

4.2.1 Application Development Tools and APIs

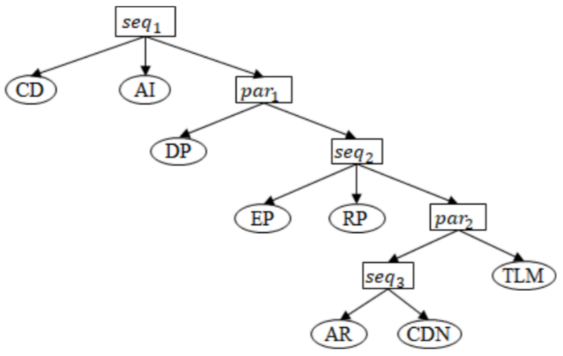
The Application Development Tools and APIs module includes different tools and APIs to give developers an environment for developing IoT applications. It is an extended module of traditional PaaS, as it provides the actual environment inside the platform to facilitate the development of applications that span both the cloud and the fog. The extension comes in two parts. First, it provides the actual IDE environment inside the platform to make developers needless of using tools like Git for pushing the application codes. Second, it supports various standards and protocols to program the connections between different entities as fog nodes have different requirements from clouds.

4.2.2 Application Graph Generator

The Application Graph Generator is a novel module. It is responsible for generating a graph for an application and a description of each component, like resource requirement or the license cost of each VNF. As we discussed before, the IoT applications can be modeled as structural graphs with sub-structures including: parallel, loop, selection, and sequence. These sub-structures can show the different situations in an application. For example, when there are two possible transitions, we can use the selection sub-structure. Hence, the application graph shows how application components can interact with each other. The application graphs for the two proposed scenarios are depicted in Figure 6.



(a)



(b)

Figure 6: Structured VNF-FG representation
(a) Smart parade application
(b) Smart accident management application

4.3 Application Deployment Layer

This layer contains three modules, the Infrastructure Repository, the Deployment Engine, and the Publication/Discovery Engine. Each of the modules are described in detail below.

4.3.1 Infrastructure Repository

The Infrastructure Repository is a novel module that allows the storage of graph-like data. It uses a graph structure with nodes, edges, and properties to represent and store data. This data includes information about the cloud and the fog nodes, such as their capacity, their location, the battery level, the hardware address of the device and relationships between them. This repository is updated whenever any change happens in the infrastructures.

4.3.2 Deployment Engine

The Deployment Engine is an extended module of regular PaaS in terms of considering the fog infrastructure. Considering fog nodes in the deployment process needs different protocols and mechanisms rather than having only cloud nodes used in traditional PaaS. This module is responsible for finding the optimal deployment plan of IoT application components over the cloud and fog infrastructures. The Deployment Engine is triggered by the orchestrator where it gets the infrastructure graph and application graph information from the orchestrator. It runs a placement algorithm to find the best candidates between fog and cloud nodes to place the VNFs, such as the one presented in [25]. For instance, let us consider both application scenarios: the smart parade application and the smart accident management application. They consist of a set of interacting components that represent a VNF-FG. The placement algorithm presented in [25] finds the near-optimal placement of this VNF-FG over the cloud and fog infrastructures (i.e., NFVI) in a way that the application execution time and cost are minimized. The Deployment Engine instantiates the cloud/fog resources required for hosting and executing the applications' components (e.g., service containers) and processes the deployment of the application's components over these resources. The placement problem is a multi-objective optimization problem due to having several criteria and measures which we didn't go through solving this problem in this thesis.

4.3.3 Publication/Discovery Engine

The Publication/Discovery Engine another extended module, is responsible for the publication and discovery functions that locate the cloud nodes/resources as well as the fog nodes/resources. The mobile fog infrastructure management capability is also added to the traditional Discovery module. In other words, it discovers all resources available in the area and stores this meta information into the infrastructure repository. Accordingly, it constructs a graph structure representing the relations among the cloud and the fog nodes which is only done by one of the state of the arts. To that end, we assume that the city is divided into multiple domains. After joining process completed, the nodes are ready to host the images and run the application components. There might be some mobile nodes as well as the fixed ones. To that end, we need a system that can manage joining/leaving nodes as well. Hence, mobile nodes notify the zone-head of the domain which they are leaving/entering. There is two ways that the list kept in the infrastructure repository being updated. First, whenever any changes happen to the list of resources, the publication/discovery engine updates the infrastructure repository. Second, orchestrator triggers the publication/discovery to get the most recent info and updates the infrastructure repository.

4.4 Application Execution and Management Layer

Four modules are included in this layer: The Monitoring Engine, the Migration Engine, the Execution Engine, and the Cloud/Fog Domain Handler. It should be noted that all the modules in this layer are extended modules from a traditional PaaS in terms of handling the fog infrastructure.

4.4.1 Monitoring Engine

The Monitoring Engine monitors the cloud/fog resources to detect mobility, bottlenecks, QoS, SLAs, etc. This module is also an extended one in which the ability of connecting, monitoring and managing the mobile or fixed fog nodes are added to the traditional monitoring engines. One specific capability is mobility detection which was not existed in the traditional PaaS because of immobility of cloud resources. It can monitor the status and services running on a cloud/fog node. In occurrence of any failure, it tries to restart service and report the situation to the orchestrator. It is also responsible to detect mobility among the resources. In such cases, monitoring engine informs the orchestrator to trigger the migration engine to find the best alternative according to the new situation.

4.4.2 Migration Engine

The Migration Engine is an extended module which runs a migration algorithm, similar to the one presented in [36]. The ability to perform migration among fog nodes or from fog nodes to cloud or vice versa makes this module distinguished from the one existed in traditional PaaS. Considering the smart parade application, when the Capture Parade Footage component needs to be migrated between the fog nodes, the algorithm finds the best node to migrate to, and in an acceptable time.

Another possible situation to start the migration procedure is when a mobile fog node leave its current domain. Hence, the distance between that node and the end-user device causes some delay which may violate the SLAs. The Migration Engine also performs the actual migration of application components.

4.4.3 Execution Engine

The Execution Engine is responsible for creating or updating chains between application components as well as for executing the application components. For this reason, it allocates required PaaS resources and prepares execution environment before hosting the real application components. This module also is an extended module in terms of involving fog infrastructure as well as cloud resources.

Regarding creating or updating chains, the orchestrator informs the execution engine to create the chains between application components for the first deployment. Subsequently, it will be triggered to update the chains, every-time a migration happens.

4.4.4 The Cloud/Fog Domain Handler

The Cloud/Fog Domain Handler is an extension of the IaaS communication component in conventional PaaS architectures. It handles all of the communications between the PaaS and the cloud and fog infrastructures. Handling heterogeneous fog resources makes this module distinct from the previous versions, Since it needs to perform device independent protocols and communications.

4.5 Application Orchestration Layer

This layer includes the Orchestrator which is in charge of orchestrating the cloud/fog resources. It is also an extended module of traditional PaaS architecture in terms of involving fog resources into all the orchestration processes. It is responsible for managing the lifecycle of the application, including deployment, chaining, execution, monitoring, and migration. It can execute different orchestration plans according to the requests it receives, such as a Deployment Orchestration Plan and a Migration Orchestration Plan. In the mentioned plans, the detailed procedures regarding deployment and migration which both includes interaction with publish/ discovery engine are discussed.

The placing order of one application components is determined based on the VNF-FG. However, in the case of having multiple applications to be orchestrated at the same time by the proposed PaaS, each application has priority parameters mentioned in the application graph file. Hence, the orchestrator can prioritize the placement of the application components ad handles multiple applications at the same time.

4.6 Modules in the Cloud/Fog Domains

The Publication/Discovery Engine is responsible for the publication and discovery function of the nodes inside the domain they are locating. The Execution Engine provides the necessary execution environment (e.g., containers) for the cloud and fog nodes to execute the application components. In fact, these two modules are the hub between the fixed/mobile nodes inside the domain and the PaaS components.

4.7 Interfaces

The general principle for designing the interactions between the different modules and the different domains is the use of the REpresentational State Transfer (REST) architectural style. The reason REST is selected is that it is lightweight, standard-based and flexible for multiple data representations(e.g., plain text, XML, and JSON), allowing us to describe the APIs in a generic way. All of the interfaces expose CRUD (i.e., Create, Read, Update, and Delete) operations.

- The Fog/Cloud Resource Management Interface allows the Publication/Discovery Engine in each domain (zone-head) to discover fog/cloud resources inside its domain and publish and update the information of its resources to the main Publication/Discovery Engine inside the PaaS. Tables 3-5 detail a list of interfaces for accessing Fog/Cloud resources and publishing the resource information to the Infrastructure Repository inside the PaaS.

Table 3: Example of API exposed by each Zone-Head to nodes inside the same domain

REST Resource	Operation	Http action & resource URI	Request body parameters	Most important info in response
List of Nodes	Create: Publish a node	POST: /nodes	Node description: ID, status, CPU, Storage	ID & URI of the published node resource
Node	Update: update a node info in the zone head list	PUT:/nodes/{nodeURI}	Node description: ID, status, CPU, Storage	None
Node	Delete: remove a node from a domain	DELETE:/nodes/{nodeURI}	None	None

Table 4: Example of API exposed by the Publication/Discovery Engine to the Zone-Heads

REST Resource	Operation	Http action & resource URI	Request body parameters	Most important info in response
List of Domains	Create: Publish domain information	POST: /domains	Domain description: ID, list of fog nodes inside it	ID & URI of published domain resource
List of Domains	Update: re-Publish domain information, update an already created resource.	PUT: /domains/{domainId}	Domain description: ID, new information about the list	None
List of Domains	Delete: remove a domain from the list of domains	DELETE: /domains/{domainId}	None	None

Table 5: Example of API exposed by the Infrastructure Repository to the Publication/Discovery Engine

REST Resource	Operation	Http action & resource URI	Request body parameters	Most important info in response
List of Domains	Read: Get list of domains	GET: /domains	None	List of all of domain resources with their nodes
List of Nodes	Read: Get list of nodes in specific domain	GET: /domains/{domainId}	None	List of IDs of nodes resource
List of Domains	Create: Publish domain information	POST: /domains	Domain description: ID, list of nodes inside the domain	ID & URI of published domain resource
List of Domains	Update: re-Publish domain information, update an already created resource.	PUT: /domains/{domainId}	Domain description: ID, new information about the list	None
List of Domains	Delete: remove a domain from the list of domains	DELETE: /domains/{domainId}	None	None

- The Internal PaaS Modules Interaction Interface which enables interaction between different modules inside the PaaS and transferring requests and data between these modules. Hence, it is both control and data interfaces. Table 6-11 below, gives a non-exhaustive list of the Interaction Interface between orchestrator and other modules inside the PaaS.

Table 6: Example of API exposed by the Orchestrator to the App Graph Generator

REST Resource	Operation	Http action & resource URI	Request body parameters	Most important info in response
DeployApp	Deploy application	POST: /DeployApp	Application graph, Constructed tree, Descriptor of each component	None
	Retrieve a specific deployed app	GET: /DeployApp/{Id}	None	Info of a specific deployed app
	List all deployed apps	GET: /DeployApp/all	None	Info of all the deployed apps
	Remove a specific deployed app	DELETE: /DeployApp/{Id}	None	None
	Update a specific deployed app	PUT: /DeployApp/{Id}	Application graph, Constructed tree, Descriptor of each component	None

Table 7: Example of API exposed by the Publication/Discovery Engine to the Orchestrator

REST Resource	Operation	Http action & resource URI	Request body parameters	Most important info in response
List of Domains	Read: Get list of domains	GET: /domains	None	List of all of domain resources with their fog nodes
List of Fog Nodes	Read: Get list of fog nodes in specific domain	GET: /domains/{domainId}	None	List of IDs of fog nodes resource
Fog node	Read: SUBSCRIBE for information of a fog node	POST:/fognodes/{fogId}?fromuri={subscriberuri}	None	Subscription ID
List of Fog Nodes	Read: SUBSCRIBE to a list of fog nodes	POST:/fognodes?fromuri={ subscriberuri}	None	Subscription ID
Fog node	Delete: un-SUBSCRIBE from info of a fog node	DELETE:/fognodes/{fogId }/{subscribeId}	None	None

Table 8: Example of API exposed by the Deployment Engine to the Orchestrator

REST Resource	Operation	Http action & resource URI	Request body parameters	Most important info in response
Deployment Plan	Generate a deployment plan	POST: /DeploymentPlan	App descriptor, Infrastructure Info	None
	Retrieve a specific deployment plan	GET: /DeploymentPlan/{Id}	None	Info of a specific deployment plan
	List all deployment plans	GET: /DeploymentPlan/all	None	Info of all the deployment plans
	Remove a deployment plan	DELETE: /DeploymentPlan/{Id}	None	None
	Update a deployment plan	PUT: /DeploymentPlan/{Id}	App descriptor, Infrastructure Info	None

Table 9: Example of API exposed by the Execution Engine to the Orchestrator

REST Resource	Operation	Http action & resource URI	Request body parameters	Most important info in response
ChainingPlan	Generate a chaining plan	POST: /ChainingPlan	Chaining Plan Info	None
	Retrieve a specific chaining plan	GET: /ChainingPlan/{Id}	None	Info of a specific chaining plan
	List all chaining plans	GET: /ChainingPlan/all	None	Info of all the chaining plans
	Remove a chaining plan	DELETE: /ChainingPlan/{Id}	None	None
	Update a chaining plan	PUT: /ChainingPlan/{Id}	Chaining Plan Info	None
ExecutionPlan	Generate an execution plan	POST: /ExecutionPlan	Information of nodes hosting app components	None
	Retrieve a specific execution plan	GET: /ExecutionPlan/{Id}	None	Info of a specific execution plan
	List all execution plans	GET: /ExecutionPlan/all	None	Info of all the execution plans
	Remove a execution plan	DELETE: /ExecutionPlan/{Id}	None	None
	Update a execution plan	PUT: /ExecutionPlan/{Id}	Information of nodes hosting app components	None

Table 10: Example of API exposed by the Migration Engine to the Orchestrator

REST Resource	Operation	Http action & resource URI	Request body parameters	Most important info in response
MigrationPlan	Generate a migration plan	POST: /MigrationPlan	Info of the node needs to migrated	Info of the selected node for the migration
	Retrieve a specific migration plan	GET: /MigrationPlan/{Id}	None	Info of a specific migration plan: source and destination info, component migrated
	List all migration plans	GET: /MigrationPlan/all	None	Info of all the migration plans
	Remove a migration plan	DELETE: /MigrationPlan/{Id}	None	None
	Update a migration plan	PUT: /MigrationPlan/{Id}	Info of the node needs to migrated	Info of the selected node for the migration

Table 11: Example of API exposed by the Monitoring Engine to the Orchestrator

REST Resource	Operation	Http action & resource URI	Request body parameters	Most important info in response
MonitoringPlan	Execute a monitoring plan	POST: /MonitoringPlan	Information of nodes hosting app components	None
	Retrieve a specific monitoring plan	GET: /MonitoringPlan/{Id}	None	Info of a specific monitoring plan
	List all monitoring plans	GET: /MonitoringPlan/all	None	Info of all the monitoring plans
	Remove a monitoring plan	DELETE: /MonitoringPlan/{Id}	None	None
	Update a monitoring plan	PUT: /MonitoringPlan/{Id}	Information of nodes hosting app components	None

4.8 Procedures

The proposed architecture includes the following procedures: application development, application deployment, and application migration. We describe the application deployment and migration procedures in the subsequent subsections below. Furthermore, in order to give a mental picture of how the different components within the PaaS interact together, we present the deployment orchestration plan, migration orchestration plans in separate sequence diagrams on which the interaction between components is shown.

4.8.1 Application Deployment

Figure 7 illustrates a sequence diagram of the interactions of different architectural modules during the application deployment phase. The process is initiated when the Orchestrator receives a request from the Application Development layer to deploy an application. This request includes the IoT application descriptor (i.e., the application graph and the descriptor of each component). The Orchestrator, as a part of the Deployment Orchestration Plan, first gets the cloud/fog infrastructure information from the Infrastructure Repository if we assume it already contains the most updated information. Otherwise, the Orchestrator sends a request to Publication/Discovery Engine to perform the discovery procedure and obtain the cloud/fog resources information. It then sends the infrastructure information along with the application descriptor to the Deployment Engine. The latter runs a placement algorithm to generate a deployment plan. According to the deployment plan, the Deployment Engine instantiates the cloud/fog resources required for hosting and executing the application's components (e.g., service containers) and processes the actual deployment of the application's components over these resources. The Orchestrator then asks the Execution Engine to generate a chaining plan. The latter chains the application components according to the chaining plan and begins executing the components. Execution of the application components is then started by the Execution Engine. Finally, the Orchestrator sends a request to the Monitoring Engine to monitor the application components. It should be mentioned that the return messages are omitted in the figure for the purpose of simplification.

It is worth noting that the proposed IoT PaaS architecture supports the on-demand discovery of the cloud/fog resources. This process is initiated when the Orchestrator receives a request from the App Development layer to deploy an application. In response, the Orchestrator asks the Publication/Discovery Engine to discover the cloud/fog resources and then gives that information to the Deployment Engine along with the application descriptor to generate a deployment plan. Besides, Publication/Discovery Engine sends this information to the Infrastructure Repository for further usage. After first on-demand discovery, Publication/ Discovery Engine updates the Infrastructure Repository whenever new notifications come from the cloud/fog resources. Hence, Infrastructure Repository always keeps the most updated information about the cloud/fog resources.

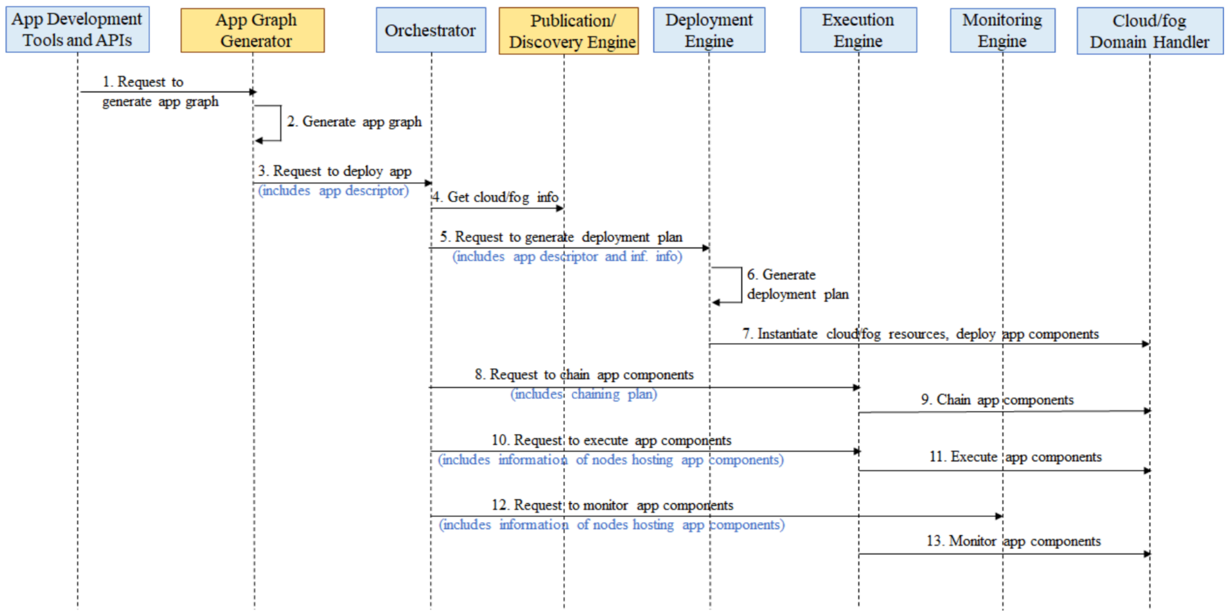

App Development Tools and APIs
App Graph Generator
Orchestrator
Publication/ Discovery Engine
Deployment Engine
Execution Engine
Monitoring Engine
Cloud/fog Domain Handler
1. Request to generate app graph
2. Generate app graph
3. Request to deploy app (includes app descriptor)
4. Get cloud/fog info
5. Request to generate deployment plan (includes app descriptor and inf. info)
6. Generate deployment plan
7. Instantiate cloud/fog resources, deploy app components
8. Request to chain app components (includes chaining plan)
9. Chain app components
10. Request to execute app components (includes information of nodes hosting app components)
11. Execute app components
12. Request to monitor app components (includes information of nodes hosting app components)
13. Monitor app components

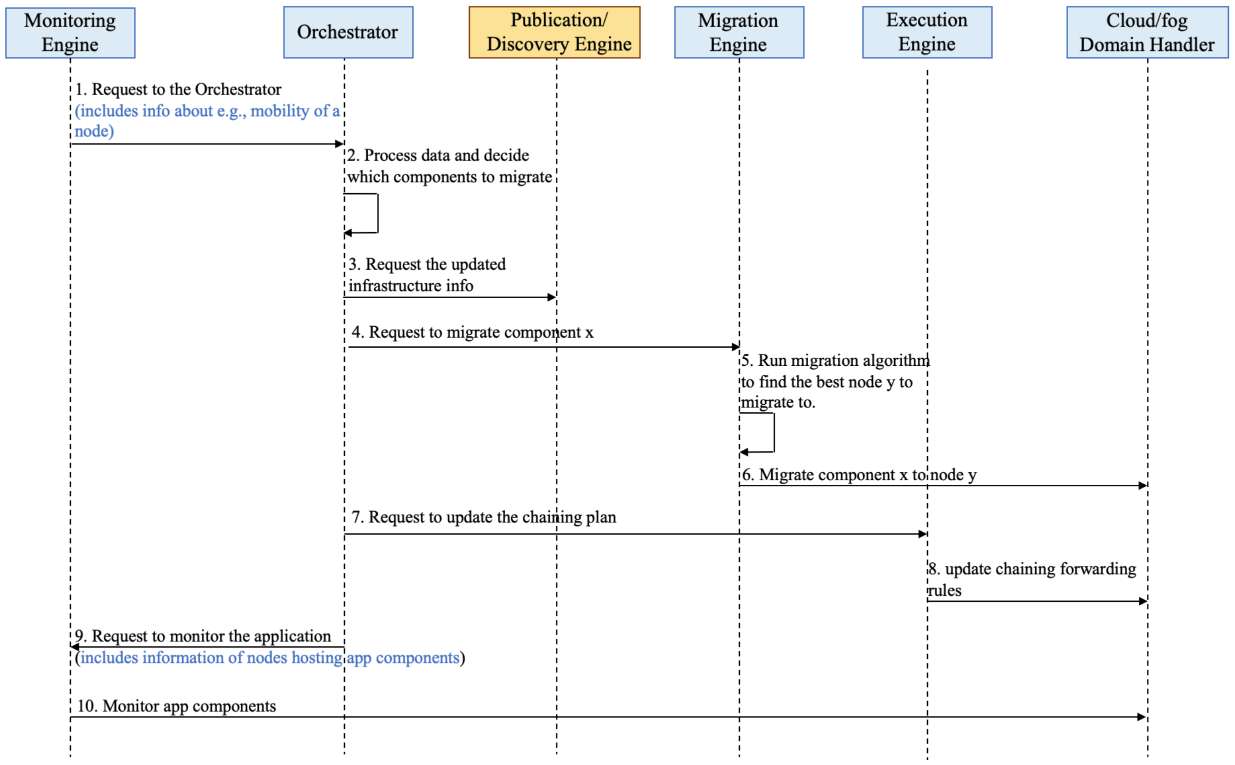
Figure 7: Sequence diagram for the Orchestrator Deployment Plan (Application deployment procedure)

4.8.2 Application Migration

Figure 8 illustrates a sequence diagram of the interactions of different architectural modules during the application migration phase. This process is initiated when the Orchestrator receives a request from the Monitoring Engine. It is again assumed that an initial discovery of cloud/fog nodes has already been done and that their information is in the Infrastructure Repository. However, when Orchestrator gets a notification from Monitoring Engine leading a need to initiate migration process, the Orchestrator gets the most updated information about cloud and fog nodes and sends it to the Migration Engine to generate a migration plan.

The Orchestrator, as part of the Migration Orchestration Plan, first processes the request and decides which component needs to be migrated. It then sends a request to the Migration Engine to generate the best migration plan. Upon receiving the updated Cloud/Fog resources information, the Migration Engine runs a migration algorithm [36] and finds the best node, instantiates the cloud and fog resources, and performs the actual migration of the concerned application component. Once the component has migrated to the chosen destination, the

Orchestrator sends a request the Execution Engine to update the existing chaining plan and forwarding rules. The latter then updates the chains. Finally, the Orchestrator sends a request that includes the new node hosting the application component to the Monitoring Engine so that it can monitor all the application components. The return messages are also omitted in this figure for the purpose of simplification.



Monitoring Engine
Orchestrator
Publication/ Discovery Engine
Migration Engine
Execution Engine
Cloud/fog Domain Handler
1. Request to the Orchestrator (includes info about e.g., mobility of a node)
2. Process data and decide which components to migrate
3. Request the updated infrastructure info
4. Request to migrate component x
5. Run migration algorithm to find the best node y to migrate to.
6. Migrate component x to node y
7. Request to update the chaining plan
8. update chaining forwarding rules
9. Request to monitor the application (includes information of nodes hosting app components)
10. Monitor app components

Figure 8: Sequence diagram for the Orchestrator Migration Plan (Application migration procedure)

4.9 Evaluation of Proposed Architecture against the Requirements

The requirements derived from the motivating use-cases should be satisfied by the proposed architecture. In fact, the proposed architecture fully meets the requirement.

The proposed PaaS architecture has a development layer including App Development APIs and Tools and App Graph Generator modules which creates an environment to develop applications and generate the application graph along with specifying the application QoS requirements. Hence, the first group of requirements regarding development layer are fully met.

The architecture includes the appropriate modules (Publication/Discovery) placed both inside the PaaS and each domain to discover the existing cloud and fog resources. Moreover, the proposed publication/discovery mechanism in this thesis supports mobility of fog nodes and keeps the info stored in the Infrastructure Repository updated with any change in the nodes status.

The proposed Deployment Engine module is responsible for dynamically determining the optimal placement plan for application components considering the set of requirements and objectives. Besides, it instantiates cloud/fog resources and deploys the application components on the selected resources based on the derived placement plan. Therefore, the proposed architecture fulfills requirements regarding deployment layer.

Two sets of interfaces introduced regarding communicating between different modules inside the PaaS architecture, along side accessing underlying fog/cloud resource uniformly which enables the interoperability at the level of providers and architectural modules.

The proposed Migration Engine module is able to find the best migration plan and migrate components form cloud to fog and vice versa, also from fog to fog according to the migration plan.

The proposed PaaS can create or update the chains between application components using the Execution Engine. Hence, the requirements regarding execution and management layer is fully satisfied.

Finally, the Orchestrator module is comprehensively designed to manage the application lifecycle including deployment, chaining, execution, monitoring, and migration. Besides, it can automatically distinguish whether to execute the deployment orchestration plan or the migration orchestration plan when needed resulting orchestration layer requirements to be fully satisfied.

4.10 Conclusion

In this chapter, we presented our proposed architecture, explained the module's functionality in each layer along with interfaces for communication between PaaS modules, and also publication/discovery interfaces used inside each domain. Afterward, we provided two illustrative sequence diagrams, showing how different components of the proposed architecture communicate with each other in the deployment and migration procedures. Finally, we justified how the proposed architecture was able to meet the previously derived requirements from the motivating use-cases.

In the following chapter, we will present an implemented prototype of the proposed architecture followed by the results and conclusion from it.

Chapter 5

Validation of the Architecture

In this chapter, we start by an overview on prototype architecture including a brief description on implemented scenario, a general description of implemented prototype, followed by hardware and software we used. Then, we moved over to the prototyped architecture in more details for the IoT prototyped PaaS and prototyped application. We describe the experimental setup in the last part of same section. Finally, we discuss the results of various experiments and analyze them accordingly. We conclude the chapter by summarizing it.

5.1 Prototype Architecture Overview

In this section we first present the implemented scenario followed by a high level description of how prototype operates. At the end, a brief description of the hardware and software used for implementing the prototype is given.

5.1.1 Implemented Scenario

The implemented scenario is the first use-case, smart parade application, presented in section 3.3.1. The application captures parade footage and sends it for analysis, in which facial recognition techniques are utilized to identify and display each person's age and gender. In the fog domains, only the information received from the cameras in the same fog domain is displayed. However, the footage received from all the fog domains is displayed in the cloud. In other words, the cloud acts as a centralized displayer for the information displayed at each fog domain. Besides, the captured data will be stored on the cloud for further analysis. It should be noted that the identification of genders and ages could trigger several value-added services. As the parade moves, the PaaS migrates the application components residing in the fog, namely the machine learning module that is responsible for facial recognition analysis of the people captured in the video footage, and the results displayer that displays the results of the machine learning module.

Accordingly, the following application components are implemented as VNFs:

1. Capture Parade Footage - where the camera manager resides; it starts/stops/pulls out footage from the camera;
2. Parade Footage Analyzer - includes a machine learning (ML) module that can determine the gender, and the age of the participants; and
3. Results Displayer - displays the ID, age, and gender for each face captured by the cameras.

5.1.2 Description of Implemented Prototype

The application programming interface (API) for Cloud/fog resource management and also for interaction between the PaaS internal modules are sets of REST API. This facilitates a programming interface independent of the languages or the type of platforms. We developed a Graphical User Interface (GUI) which utilizes this programmable interface and enables the developers to enter the input regarding the application graph information and request a deployment of the IoT application. The applications are deployed on top of an open source PaaS, namely the Cloudify. An extension of the PaaS captures the request from the GUI and performs the application modeling and lifecycle automation, like deploying, orchestration and management. The Cloudify REST plugin has been used to integrate the extended modules to the core PaaS. This plugin utilizes the JINJA templates that will be evaluated as the content of separate REST calls.

The application components are implemented as VNFs packaged in Docker containers which facilitates the deployment and migration processes. Two IP cameras are used to capture the parade footage. We used a python application that can directly access the IP cameras. Having python application for the age and gender recognition purpose has led us to use Flask framework to implement the API for interaction between different the different IoT application.

Six different settings are used for the PaaS modules placement and application component placements to validate the prototype. We both considered the centralized and decentralized PaaS, and changed the application component placements to measure the delays and compare the system performance in different settings.

5.1.3 Used Hardware and Software

In this subsection, we describe the software and hardware used for implementing the prototype IoT PaaS and the prototyped application.

5.1.3.1 Cloudify

Cloudify is an open-source cloud orchestration framework that enables modeling applications and services and automates their entire life cycle. An application in Cloudify is described in a blueprint and its DSL (Domain Specific Language) and is based on the TOSCA standard. The blueprints are YAML documents and are used to describe how the application should be deployed, managed, and automated. Nodes representing the services can be defined in the blueprints. Each node has its own properties as well as unique features. One of the key features of the Cloudify is that we can install the Cloudify CLI on a separate host to control the Cloudify Manager remotely. Cloudify Manager is the computing host where Cloudify components running.

5.1.3.2 Restlet

Restlet is a framework for writing both server and client web applications in the Java platform. It is an open source, comprehensive, and lightweight framework. It supports all HTTP actions and data formats like XML and JSON which we used the latter. As we developed the extended modules of the IoT PaaS in Java, we used Restlet to build both APIs for resource management and communication between modules in the PaaS.

5.1.3.3 Flask

Flask is a lightweight WSGI web application framework written in python and is considered a micro-framework since it does not require any specific libraries. However, it is easy to add functionalities to the applications using extensions supported by the Flask. Hence, Flask-REST is an extension for Flask that adds support for building REST APIs quickly.

5.1.3.4 Jackson

Jackson is an open source library for Java to process JSON data format. It offers three methods for processing the data: Streaming API, Tree Model, and Data bindings. We used the Data Binding method which converts JSON to and from POJOs in order to parse and generate our JSON data representations.

5.1.3.5 Axis M1031 network camera

The Axis M1031 camera, showed in Figure 9, offers superior video quality compared to its competitors at 30 frames per second in VGA resolution. This camera has the ability to detect movements even in the dark using a passive infrared (PIR) sensor. It includes a white LED for illumination upon request or occurring an event. It also offers two-way audio communication through built-in microphone and speaker. Axis M1031 can provide configurable video streams in

H.264, Motion JPEG, and MPEG-4 Part 2. The H.264 compression is more optimized for bandwidth and storage usage by significantly reducing the bit rate.

Figure 9: Axis M1031 Network Camera

5.1.3.6 Axis M1065 LW Network Camera

The Axis M1065 LW camera, showed in Figure 10, delivers HDTV 1080p quality and Dynamic Range (WDR) technology to ensure having details in the picture anytime. It offers IR illumination which enables you to record videos even in the dark. Furthermore, it uses Axis Zip-stream technology to lower bandwidth and storage usage by an average of 50% while providing high-quality images. It comes with a built-in motion sensor triggering the camera to record only when it detects movement.

Figure 10: Axis M1065 LW Network Camera

5.1.4 Programming Language and IDE Used

For the extended modules to the Cloudify and all the APIs designed for PaaS and resource management, and the GUI we used Java. Eclipse was used as the editor. For the application, Python programming language was used.

5.2 Prototype Architecture

The PaaS prototype architecture is shown in Figure 11. The software architecture of Cloudify is reused for our PaaS implementation. As shown in Figure 11, the Application Graph Generator, the Publication/Discovery Engine, the Deployment Engine, the Orchestrator, and the Migration Engine are implemented, but the Monitoring Engine and the Execution Engine modules are not implemented. An extension for Cloudify was developed so that it can also discover the real time cloud/fog resources. Nodes can be defined in the blueprints. These nodes represent the services. Each node has its own properties and some unique features. In this prototype, we define the following nodes in the blueprints: the graph generator node, the deployment node, the publication/discovery node, the orchestrator node, and the migration node. Accordingly, using the blueprints, Cloudify orchestrates the execution of the App. Graph Generator, the Deployment

Engine, the Publication/Discovery Engine, and the Migration Engine. These nodes act as REST clients using Cloudify REST plugin in order to communicate with different architectural modules and nodes. In the prototype architecture, the smart parade application is implemented using the Parade Footage Analyzer component and the Results Displayer component.

In the following subsections, the details of the IoT PaaS and the covered functionality within the prototyped implementation followed by a brief review on the deployed application components will be discussed. Following the end, we summarize the validation of the prototype in short.

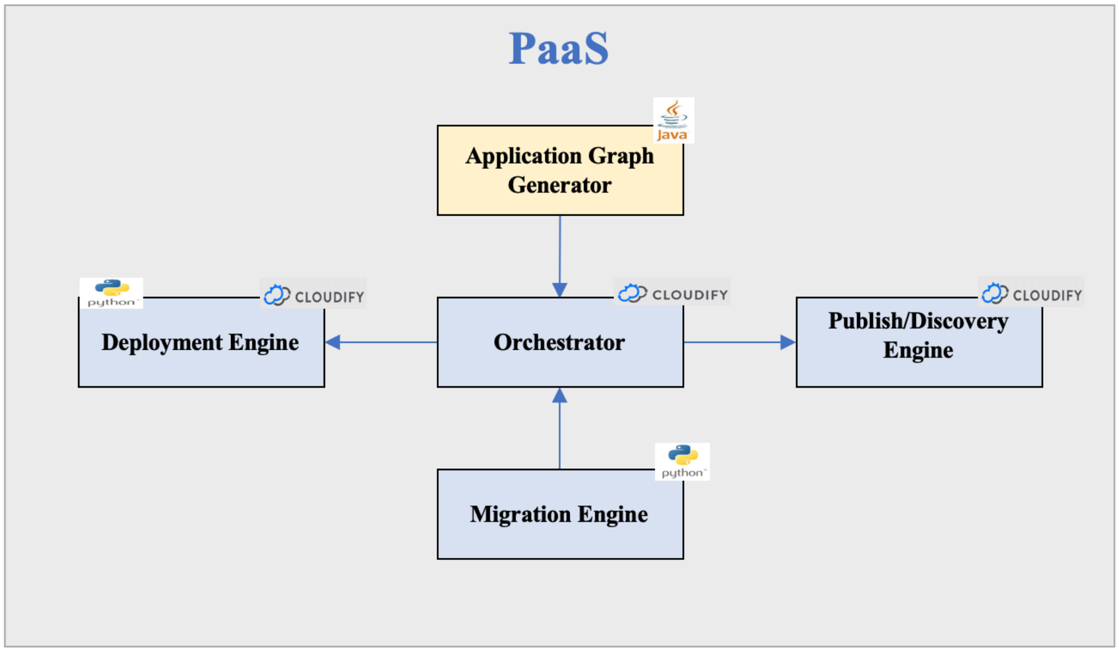


Figure 11: The prototype architecture of IoT PaaS

5.2.1 Prototyped IoT PaaS

In this subsection, we will cover the implemented modules inside the PaaS. The two sets of interfaces are also covered. We will briefly describe to what extent the functionality was covered, and how we implemented it.

5.2.1.1 Application Graph Generator

The Application Graph Generator is implemented using Java Swing libraries. We implemented it as a simple Java desktop application that generates description files based on the user input. This input contains various information about the application and the relationship between its components, including information about performance requirements of the application (e.g., required traffic, memory size, disk size, etc.)

5.2.1.2 Publication/Discovery Engine

For the Publication/Discovery Engine, a publication node in a Cloudify blueprint acts as a REST client using the Cloudify REST plugin. It sends a request to the Publication/Discovery Engine in each fog domain in order to get the most updated fog nodes' information. It then stores this information in a runtime property inside the Cloudify framework. In the prototype, we assumed that we have one cloud node, hence no need to discover it.

5.2.1.3 Orchestrator

For the Orchestrator, the orchestrator node (a Cloudify blueprint) uses the Cloudify REST plugin to communicate with different architectural modules and to cooperate among them. It receives the application description from the App Graph Generator. This information includes the interaction between different components and the description of each component. It also uses the same plugin to receive the underlying cloud/fog nodes' information from the Publication/ Discovery Engine. It then merges this information in the blueprint, and, using the REST plugin of Cloudify, sends this information to the Deployment Engine.

5.2.1.4 Deployment Engine

The Deployment Engine is implemented as a python-based web application using the python web framework Flask. The deployment process relies on a couple of Docker containers to launch both the results displayer and the Machine Learning module on the target fog node. This node, described in a Cloudify blueprint, uses the Cloudify REST plugin to receive data (i.e., the application descriptor and the cloud/fog nodes information) from the Orchestrator. It then sends a request to the Deployment Engine to deploy the application components (implemented as VNFs) on the cloud/fog nodes. This blueprint uses the Cloudify Fabric plugin to communicate with the Deployment Engine. The Fabric plugin enables Cloudify to SSH into the respective fog node in order to deploy the application component on it. In addition, this blueprint contains additional details about the nodes and the scripts needed during the deployment process.

5.2.1.5 Migration Engine

For the Migration Engine, the migration node in the Cloudify blueprint sends a request to the Migration Engine using the Cloudify REST plugin to start migrating the Capture Parade Footage and the Results Displayer components from one fog node to another. The Migration Engine is implemented using the python web framework Flask, relying on Docker containers to migrate both components residing on fog nodes (i.e., Results Displayer and Parade Footage Analyzer) from one fog node to another.

5.2.1.6 Uniform Interface For Accessing Underlying Cloud/Fog nodes

For this interface, all the CRUD operation needed for registering/deregistering a node as well sharing the info of the nodes inside a specific domain were implemented. This interface was implemented in two layers: First, the interaction between nodes inside a domain and its the zone-head . Second, the interaction between zone-heads and the publication/discovery engine inside the PaaS. Hence, all the operations needed to manage the underlying fog/cloud resources were implemented with support of mobile fog nodes due to having join/leave notifications to the zone-heads.

5.2.1.7 Interface For Interacting between Internal PaaS Modules

The internal PaaS modules interaction interfaces that are required for the scenario were implemented while others were excluded. Basically it contains the interaction between Publication/Discovery Engine, Deployment Engine, Migration Engine and Graph generator with Orchestrator.

5.2.2 Prototyped Application

The Smart Parade application was chosen for the implementation. Figure 12 demonstrates a prototype view of the implementation scenario. The application components are implemented as VNFs. The VNFs are packaged in Docker containers and are pushed to the DockerHub repository. Whenever a VNF needs to be migrated from one fog node to another, the Migration Engine sends a request to the first fog node to stop the container. The fog node then pushes the container image to the DockerHub repository, from which the second fog node pulls the container image and runs the container.

For the Parade Footage Analyzer (ML Module) component, we used a python application that can directly access the IP camera by specifying the camera's URL and thus obtains real-time video streams. This ML application recognizes the age and the gender of the people in front of the camera and tags each face with the detected age and gender. The photo is taken from the live camera stream by the cv2 module (a python library designed to solve computer vision problems), which then converts the image to grayscale to detect faces. The cropped faces are used later to feed the neural network model for prediction purposes. These results are then sent from the Parade Footage Analyzer to the Results Displayer via a REST API (Flask-REST app). The

Results Displayer component is implemented using Flask. It exposes a REST API implemented as a Flask web app to the Results Displayer (on the cloud) and to the Parade Footage Analyzer.

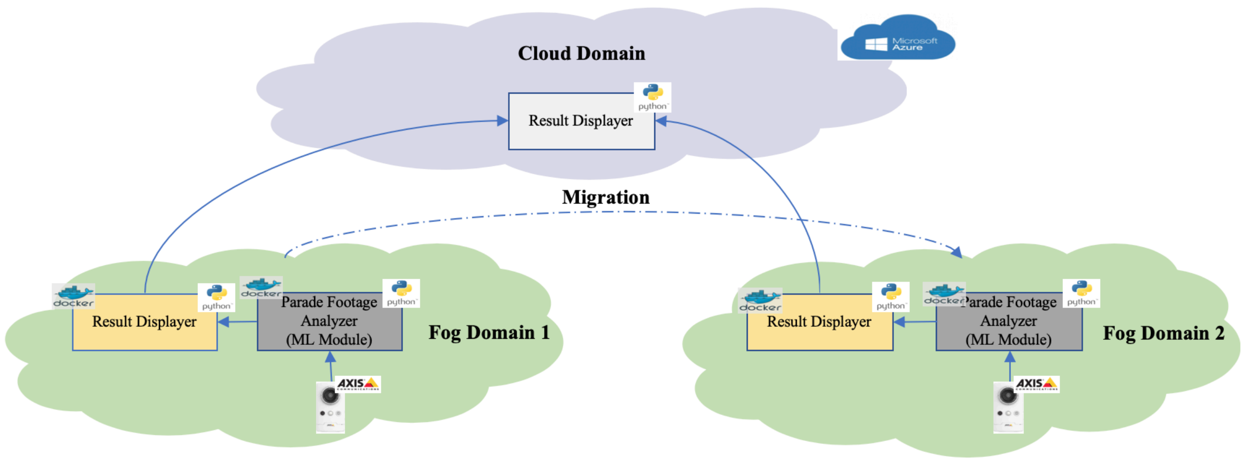


Figure 12: A prototype view of the implemented scenario

5.2.3 setup

The PaaS runs on a machine with dual 2X8-Core 2.50GHz Intel Xeon CPU E5-2450v2 and 40GB of memory in one setting and it is distributed between the local machine and Microsoft Azure cloud in another setting. In a distributed PaaS setting, the machines used (in Virginia and Iowa) in Microsoft Azure have 4Go of RAM with 2 vCPUs Intel® Xeon® CPU E5-2660 0 @ 2.20GHz and Ubuntu Server 18.04. The prototype includes one cloud node and two fog nodes. The cloud node is a Virtual Machine (VM) on the Microsoft Azure cloud. The VM has an Intel® Xeon® CPU E5-2660 0 @ 2.20GHz (2 CPUs) with Windows 10 Pro 64-bit. The first fog node (i.e., fog node 1) is a laptop with an Intel® Core i7- 2620M 2.70GHZ CPU with 8GB of RAM running Ubuntu 18.04.2, and the second (i.e., fog node 2) is another laptop with an Intel® Core i5-2540M 2.60GHz CPU with 4GB of RAM running Ubuntu 18.04.2.

5.3 Performance Evaluation

In this section we start by describing the performance metrics, followed by different test cases for evaluating the performance metrics. Finally, we conclude the section by presenting the result obtained and analyzing them.

5.3.1 Performance Metrics

Two performance metrics utilized to evaluate the performance of the proposed architecture: orchestration latency, end-to-end delay.

- **Orchestration latency:** It is measured from the time a request to deploy an application to the orchestrator is initiated to the time the acknowledgment of orchestration is received. Orchestration latency is measured for executing both the deployment plan and the migration plan. In addition, the orchestration latencies for centralized and distributed PaaS are also measured considering different distributions of the PaaS modules. Different test cases (i.e., test cases 4, 5, and 6) have been used to execute the deployment plan and the migration plan. This metric will help us decide which settings and distributions are the best to get the lowest orchestration delay.
- **End-to-end (E2E) delay:** It is measured from the time the cameras send footage to the time the cloud Results Displayer displays the final results. We vary the placement of the components and show the effect of changing the placement.

5.3.2 Test Cases

The first three test cases consider the PaaS as a centralized entity, where all its modules are deployed on a local machine in our lab in Montreal. However, they consider different distribution of application components. The remaining test cases consider a distributed PaaS with a different distribution of its modules (mainly the Deployment Engine and the Migration Engine). However, they consider application components running on the same node.

Test Case 1 – This test case considers an environment composed of two fog nodes and one cloud node. Similar to the description of the prototype architecture, the Parade Footage Analyzer (ML Module) and the fog's Results Displayer are each deployed on a fog node (i.e., a laptop), while the cloud Results Displayer is deployed in the cloud.

Test Case 2 – This test case considers an environment with only two fog nodes. All the components are deployed on the fog nodes. The first fog node runs the fog Results Displayer while the second fog node runs the cloud Results Displayer and the Parade Footage Analyzer.

Test Case 3 – This test case considers an environment with one fog node and one cloud node. The Parade Footage Analyzer runs on the fog node, while both Results Displayers (the one designed for the fog and the one designed for the cloud) run on the cloud.

Test Case 4 – This test case considers the fact that the Migration Engine and the Deployment Engine are deployed on Microsoft Azure in Virginia while Cloudify and the remaining PaaS modules are deployed on our local machine in Montreal. In addition, it considers all the application components that are initially hosted on Microsoft Azure in Iowa and need to be migrated to Microsoft Azure in Virginia.

Test Case 5 – This test case considers the fact that the Migration Engine and the Deployment Engine are deployed on our local machines in our lab in Montreal, while Cloudify and the remaining PaaS modules are deployed on another machine in our local network in Montreal. Application components are initially running on Microsoft Azure in Iowa and need to be migrated to Microsoft Azure in Virginia.

Test Case 6 – This last test case considers the fact that the Migration Engine is deployed on Microsoft Azure in Virginia while Cloudify and the remaining PaaS modules are deployed on our local machines in our lab in Montreal. The application components are initially hosted on Microsoft Azure in Iowa and need to be migrated to Microsoft Azure in Virginia.

5.3.3 Results and Analysis

This subsection discusses the performance results obtained based on the specified performance metrics. First, we start by analyzing the orchestration latency for executing both migration and deployment plan. Then, we look through the end to end latency analysis.

5.3.3.1 Orchestration Latency for Executing the Migration Plan

The average latency for executing the migration plan in a centralized PaaS over 15 consecutive experiments conducted for test case 1 is presented in Figure 13. We assume that the fog Results Displayer and the Parade Footage Analyzer are migrated from fog node 1 to fog node 2. The Linux built-in tool time is used again, this time to get the time required to execute the migration plan. The average latency for executing the migration plan is 36.26 seconds.

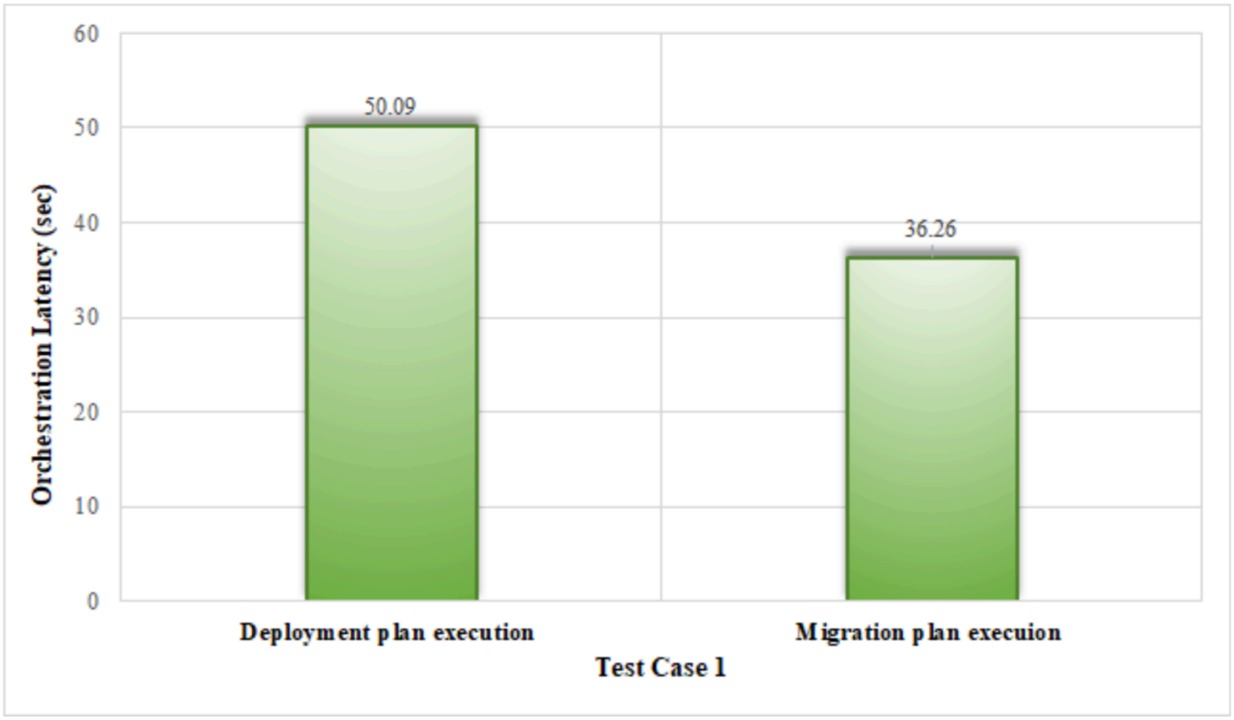


Figure 13: Orchestration latencies for executing the deployment plan and the migration plan for the parade application considering a centralized PaaS

Figure 14 indicates the average latency for executing the migration plan in a distributed PaaS over 15 consecutive experiments for test cases 4, 5, and 6. In test case 4, the Migration Engine is close to the destination node (where we want to migrate the application components) and far from the remaining PaaS modules and the source node hosting the application components. In test case 5, the Migration Engine is closer to the other PaaS modules and far from the source and destination nodes. Finally, in test case 6, the Migration Engine is close to the source node and far from the other PaaS modules and the destination node. From performance results, we can conclude that the placement of the Migration Engine close to the destination node results in lower latency. Although the difference with the measurements made for the other test cases (Test Cases 5 and 6) was not very big, the PaaS architecture still needs to be combined with a placement algorithm for its modules as well as the application components in order to obtain optimal results in terms of latency.

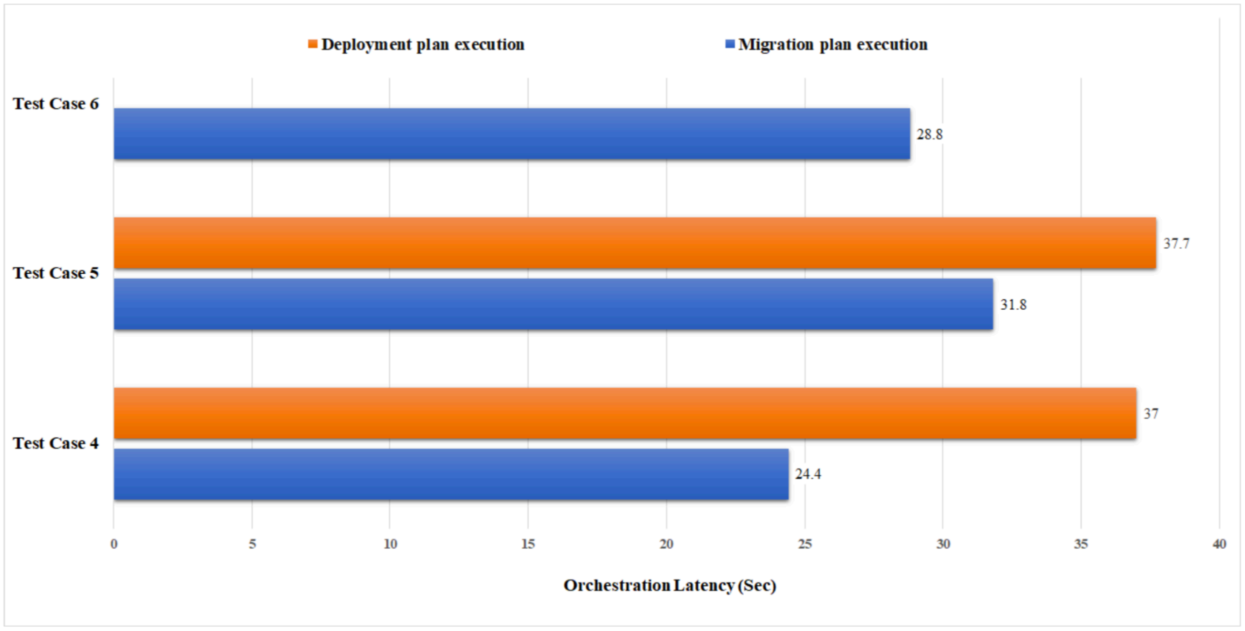


Figure 14: Orchestration latency for executing the deployment plan and the migration plan for the parade application considering a distributed PaaS

5.3.3.2 Orchestration Latency for Executing the Deployment Plan

Figure 13 also shows the average latency for executing the deployment plan in a centralized PaaS for test case 1, the only test case conducted for this experiment. We used the built-in Linux tool time to get the time required to execute the deployment plan. The results are provided for 15 consecutive experiments. The average latency for executing the deployment plan was 50.09 seconds.

Figure 14 presents the average latency for executing the deployment plan in a distributed PaaS over 15 consecutive experiments for test cases 4 and 5 only. The same approach for migration was followed for deployment, where the Deployment Engine was first placed closer to the application than the PaaS (test case 4) and then closer to the PaaS modules than the application (test case 5). The results obtained were similar, which shows that the placement of the deployment engine does not affect the execution of the deployment plan for our proposed PaaS architecture.

The procedure for executing the deployment plan involves two additional modules in comparison to the procedure for executing the migration plan. Therefore, having the longer average latencies is reasonable. More specifically, for deployment, the orchestrator has to first communicate with the Publication/Discovery Engine and the App. Graph Generator before sending a request to the Deployment Engine to deploy the application components. However, executing the migration plan only involves sending a request from the Orchestrator to the Migration Engine, which proceeds to migrate both components (i.e., Capture Parade Footage and fog Results Displayer) from one fog node to another. It is worth notting that for the deployment plan execution, Cloudify orchestrator needs to install two blueprints (one for Publication/ Discovery Engine and the App. Graph Generator, and another one for the Deployment Engine). Meanwhile, Cloudify only installs one blueprint for the migration process, since only a single request to the Migration Engine component is needed for this process to take place.

One key note to be taken from the results is that the latency for the execution of the migration plan considering a distributed PaaS is lower than the latency in a centralized PaaS. This lower latency is particularly due to the increased networking capabilities of Microsoft Azure compared to the local machines in our lab. The same conclusion can also be made for the difference between the latency for the execution of the deployment plan in a centralized and distributed PaaS.

5.3.3.3 End to End Latency

The result of the end to end latency experiment is shown in Figure 15 in which implementation scenario for the smart parade application has been executed. This experiment was conducted over three test cases (i.e., test case 1, test case 2, and test case 3). The bar represents the results for 15 consecutive experiments. The latency is measured via timestamps in the ML module of the Parade Footage Analyzer and in the cloud's Results Displayer components. The end to end latency can thus be obtained by calculating the time difference between these two timestamps.

As expected, the lowest latency is obtained in test case 2 where all the components are deployed on the fog nodes. All of the fog nodes are in the same LAN and hence there is very low latency (9.87 ms). Test case 1 shows a relatively low latency which is 67.73 ms, and can be due to the fact that two of the 3 components are deployed on the same machine, while only the cloud's Results Displayer is placed in the cloud. Finally, while in test case 3, two of the components are deployed on the same node, the fact that the ML module (i.e., Parade Footage Analyzer) is the only component on the fog node resulted in a very high latency (~ 1s). These results could imply that the original test case chosen for this work (i.e., test case 1) is a reasonable compromise to reduce the end-to-end latency. In particular, test case 1 is suitable even for more complicated scenarios, where computationally intensive components (compared to our simple results displayer) must be placed in the cloud.

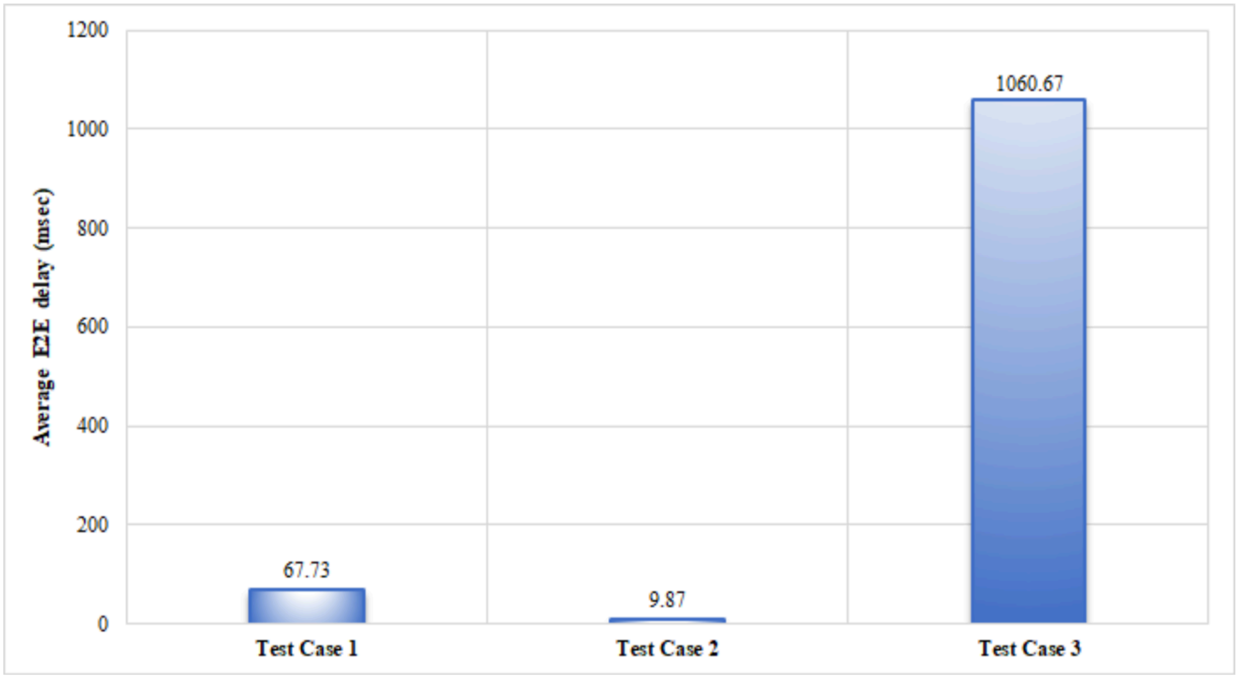


Figure 15: End to end latency for executing the smart parade application

5.4 Conclusion

The prototyped architecture and implemented scenario was discussed in brief along with the software and hardware used to develop it. Afterward, the detailed prototype architecture and application were discussed to summarize the prototype validation. Finally, the performance measurement was made, the result was shown, and analyzed.

We conclude our thesis in the next chapter by focusing on the summary of the thesis and the future work to be done.

Chapter 6

Conclusion

In this chapter, we will first summarize the contributions of this thesis and then focus on the possible future research direction.

6.1 Contributions Summary

Internet of Things includes a vast amount of applications in various domains from healthcare to autonomous vehicles. The IoT applications are usually composed of different interacting components which make their provisioning rather challenging, especially in hybrid cloud and fog settings. In the early days of IoT, the cloud was used to manage the provisioning of the IoT application. The distance between cloud and the end-users in IoT ecosystem could cause big latency, which is in contrast with some IoT applications' requirements. Leveraging fog computing can tackle this issue by extending the traditional cloud architecture to the edge of the network and enabling the deployment of some application components on fog nodes. There are some solutions trying to utilize either the fog computing or cloud computing in order to provision IoT applications. However, a comprehensive PaaS for provisioning of IoT applications with components spanning cloud and fog was never addressed.

In order to understand the requirements, we presented two motivating use-cases; a smart parade scenario and a smart accident management scenario. The two use-cases had overlapping requirements. We determined several requirements for an IoT PaaS in order to automate the IoT application provisioning over cloud and fog resources. The IoT applications' graphs generating was one of the novel requirements. The publishing and discovery mechanism, along with the orchestration mechanism plays an essential role in realizing the IoT PaaS. Enabling dynamic deployment on the cloud and fog resources as well as migration between fog and cloud nodes were key requirements. The need to create chains between VNFs and update these chains in case of application component migration was identified as the management layer requirement. However, additional requirements were identified to provision IoT applications in a uniform manner. Based on the requirements we evaluated the existing full-fledged IoT PaaS architectures and the frameworks that can be used within an IoT PaaS. None of the state-of-the-art was able to fulfill all the requirements.

We then proceeded to propose an IoT PaaS architecture that can fulfill the derived requirements. In this regard, we proposed a novel IoT PaaS architecture for NFV-based hybrid cloud/fog systems. In contrast to the existing IoT PaaS solutions, the proposed solution enables the discovery of existing cloud and fog nodes as well as the generation of application graphs with different sub-structures (e.g., selection, parallel). For all the interfaces, we used REST paradigm and used IoT friendly description model. In the end, we show how the different components within the IoT PaaS interact with each other both in the deployment orchestration plan and migration orchestration plan.

Then, a prototype for validating the IoT PaaS was implemented using the Cloudify open source. A subset of the proposed architecture was implemented, and the smart parade scenario as a SaaS was developed and deployed on top of the Cloudify. Some extensions of the Cloudify was made to enable the fulfillment of all the requirements. Namely, the Application Graph Generator which is a novel module and provides the IoT application graphs to the Orchestrator of the PaaS. As an another example, the uniform interface was designed for managing the cloud and fog resources and supporting the mobile fog nodes.

Finally, two performance metric and experimental setup were defined. A set of experiments are conducted to evaluate the feasibility of the architecture and explore possible options for placing the PaaS and IoT applications different components. For example, both centralized and distributed PaaS have been explored considering different distributions of the PaaS modules. The results from the experiment were shown and analyzed. The results show the higher latency of executing the deployment plan compared to the migration plan. In addition, the end-to-end latency was analyzed over three different test cases with a different distribution of the application components over the cloud and the fog nodes. These experiments conducted that provisioning IoT applications over a hybrid cloud/fog setting is the best choice. Placing latency sensitive components on the fog while having computationally intensive components placed in the cloud could decrease the end-to-end latency significantly, and at the same time preserves the QoS requirements of the application. The performance of distributed and centralized PaaS was also analyzed considering the placement of PaaS modules in clouds and fogs in different geographical locations. It was apparent that the PaaS needs an efficient placement algorithm for its modules as well as for the application components in order to obtain optimal results in terms of latency.

6.2 Future Research Direction

In this work, we did not include the design of VNF deployment algorithm on cloud and fog resources. Such algorithms need to be adapted to the context of hybrid cloud/fog environment, taking into consideration the heterogeneity of the fog nodes hosting the VNFs while ensuring the QoS. Another similar example is the design of VNF migration algorithm among fog/cloud resources. In real-world scenarios, there could be a variety of incentives for migrating of the application components. For instance, a fog node communicating with an IoT devices might move far from the device or the fog node fails or becomes unavailable. Hence, designing such algorithms that enable finding the efficient deployment plan and migration plan could be considered as a direction.

Some of the critical aspects of an PaaS were omitted in the proposed IoT paaS. Security is one of such. An automated asymmetric public/private key exchange mechanism can be included in order to secure the fog and cloud resources. Furthermore, a user authentication mechanism could be added to the PaaS to allow end-to-end security.

The ability to provision multiple applications at the same time is one of the possible features that could be added to the proposed PaaS. In this regard, we can design a mechanism to prioritize the different application components based on the latency and usage requirements, regardless of which application they are part of.

Employing SDN along with NFV is one of the interesting directions that can be investigated to utilize chaining process of the application components. SDN aims at splitting the control plane and the data plane into network elements in order to provide a flexible management of the forwarding behavior of those network elements. It can enable the easy on-the-fly chaining of these VNFs. In future work, SDN can be used to provision the paths between VNFs dynamically once they are deployed. Furthermore, NFV allows deploying the application components anywhere anytime, and SDN enables reusing the same components in different flows for different applications. Hence, if various applications are using the same service, taking advantage of SDN technology can be beneficial.

In this thesis we have investigated the possibility of geographically distributing the PaaS modules. However, designing a distributed PaaS still faces several challenges. Latency requirements, security, heterogeneity, failure handling are some examples of these challenges which can be investigated further each in details. Furthermore, designing efficient placement algorithms for PaaS modules is also a potential research direction, considering the application component placement possibilities and QoS requirements.

References

[1] *Recommendation ITU-T Y.2060, Overview of the Internet of things*, International Telecommunications Union, Switzerland, 2013.

[2] Vaquero, Luis M., Luis Rodero-Merino, Juan Caceres, and Maik Lindner. "A break in the clouds: towards a cloud definition." *ACM SIGCOMM Computer Communication Review* 39, no. 1 (2008): 50-55.

[3] Hogan, Michael, Fang Liu, Annie Sokol, and Jin Tong. "Nist cloud computing standards roadmap." *NIST Special Publication* 35 (2011): 6-11.

[4] Bonomi, Flavio, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. "Fog computing and its role in the internet of things." In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pp. 13-16. ACM, 2012.

[5] Bröring, Arne, Soumya Kanti Datta, and Christian Bonnet. "A categorization of discovery technologies for the internet of things." In *Proceedings of the 6th International Conference on the Internet of Things*, pp. 131-139. ACM, 2016.

[6] Lin, Jie, Wei Yu, Nan Zhang, Xinyu Yang, Hanlin Zhang, and Wei Zhao. "A survey on internet of things: Architecture, enabling technologies, security and privacy, and applications." *IEEE Internet of Things Journal* 4, no. 5 (2017): 1125-1142.

[7] Guth, Jasmin, Uwe Breitenbücher, Michael Falkenthal, Frank Leymann, and Lukas Reinfurt. "Comparison of IoT platform architectures: A field study based on a reference architecture." In *2016 Cloudification of the Internet of Things (CIoT)*, pp. 1-6. IEEE, 2016.

[8] Whitmore, Andrew, Anurag Agarwal, and Li Da Xu. "The Internet of Things—A survey of topics and trends." *Information Systems Frontiers* 17, no. 2 (2015): 261-274.

[9] Gubbi, Jayavardhana, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. "Internet of Things (IoT): A vision, architectural elements, and future directions." *Future generation computer systems* 29, no. 7 (2013): 1645-1660.

[10] Sobin, C. C. “A Survey on Architecture, Protocols and Challenges in IoT.” *Wireless Personal Communications*: 1-47.

[11] Lee, In, and Kyoochun Lee. “The Internet of Things (IoT): Applications, investments, and challenges for enterprises.” *Business Horizons* 58, no. 4 (2015): 431-440.

[12] Yu, Wei, Fan Liang, Xiaofei He, William Grant Hatcher, Chao Lu, Jie Lin, and Xinyu Yang. "A survey on the edge computing for the Internet of Things." *IEEE access* 6 (2017): 6900-6919.

[13] Leo, Marco, Federica Battisti, Marco Carli, and Alessandro Neri. "A federated architecture approach for Internet of Things security." In *2014 Euro Med Telco Conference (EMTC)*, pp. 1-5. IEEE, 2014.

[14] Aggarwal, Renu, and Manik Lal Das. “RFID Security in the Context of “Internet of Things”.” In *Proceedings of the First International Conference on Security of Internet of Things*, pp. 51-56. 2012.

[15] Sadique, Kazi Masum, Rahim Rahmani, and Paul Johannesson. “Trust in Internet of Things: An architecture for the future IoT network.” In *2018 International Conference on Innovation in Engineering and Technology (ICIET)*, pp. 1-5. IEEE, 2018.

[16] Zhang, Qi, Lu Cheng, and Raouf Boutaba. “Cloud computing: state-of-the-art and research challenges.” *Journal of internet services and applications* 1, no. 1 (2010): 7-18.

[17] “PaaS (Platform-as-a-Service) Comparison.” Apprenda, September 19, 2017. Https:// apprenda.com/library/paas/conducting-a-paas-comparison (accessed February 1, 2020).

[18] Jinzhou, Yang, He Jin, Zhang Kai, and Wang Zhijun. “Discussion on private cloud PaaS construction of large scale enterprise.” In *2016 IEEE International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*, pp. 273-278. IEEE, 2016.

[19] Pahl, Claus. “Containerization and the paas cloud.” *IEEE Cloud Computing* 2, no. 3 (2015): 24-31.

[20] OpenFog Consortium Architecture Working Group. “OpenFog reference architecture for fog computing.” *OPFRA001* 20817 (2017): 162.

[21] Mahmud, Redowan, Ramamohanarao Kotagiri, and Rajkumar Buyya. “Fog computing: A taxonomy, survey and future directions.” In *Internet of everything*, pp. 103-130. Springer, Singapore, 2018.

[22] Mouradian, Carla, Diala Naboulsi, Sami Yangui, Roch H. Glitho, Monique J. Morrow, and Paul A. Polakos. "A comprehensive survey on fog computing: State-of-the-art and research challenges." *IEEE Communications Surveys & Tutorials* 20, no. 1 (2017): 416-464.

[23] Al-Fuqaha, Ala, Mohsen Guizani, Mehdi Mohammadi, Mohammed Aledhari, and Moussa Ayyash. "Internet of things: A survey on enabling technologies, protocols, and applications." *IEEE communications surveys & tutorials* 17, no. 4 (2015): 2347-2376.

[24] Mouradian, Carla, Somayeh Kianpisheh, Mohammad Abu-Lebdeh, Fereshteh Ebrahimnezhad, Narjes Tahghigh Jahromi, and Roch H. Glitho. “Application component placement in NFV-based hybrid cloud/fog systems with mobile fog nodes.” *IEEE Journal on Selected Areas in Communications* 37, no. 5 (2019): 1130-1143.

[25] Yangui, Sami, Pradeep Ravindran, Ons Bibani, Roch H. Glitho, Nejib Ben Hadj-Alouane, Monique J. Morrow, and Paul A. Polakos. “A platform as-a-service for hybrid cloud/fog environments.” In *2016 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, pp. 1-7. IEEE, 2016.

[26] Pahl, Claus, Sven Helmer, Lorenzo Miori, Julian Sanin, and Brian Lee. "A container-based edge cloud paas architecture based on raspberry pi clusters." In *2016 IEEE 4th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW)*, pp. 117-124. IEEE, 2016.

[27] Liyanage, Mohan, Chii Chang, and Satish Narayana Srirama. “mePaaS: mobile-embedded platform as a service for distributing fog computing to edge nodes.” In *2016 17th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pp. 73-80. IEEE, 2016.

[28] Yigitoglu, Emre, Mohamed Mohamed, Ling Liu, and Heiko Ludwig. “Foggy: A framework for continuous automated iot application deployment in fog computing.”

In *2017 IEEE International Conference on AI & Mobile Services (AIMS)*, pp. 38-45. IEEE, 2017.

[29] Saurez, Enrique, Kirak Hong, Dave Lillethun, Umakishore Ramachandran, and Beate Ottenwälder. “Incremental deployment and migration of geo-distributed situation awareness applications in the fog.” In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pp. 258-269. 2016.

[30] Tao, Ming, Kaoru Ota, and Mianxiong Dong. “Foud: Integrating fog and cloud for 5G-enabled V2G networks.” *IEEE Network* 31, no. 2 (2017): 8-13.

[31] Tuli, Shreshth, Redowan Mahmud, Shikhar Tuli, and Rajkumar Buyya. “Fogbus: A blockchain-based lightweight framework for edge and fog computing.” *Journal of Systems and Software* (2019).

[32] Donassolo, Bruno, Ilhem Fajjari, Arnaud Legrand, and Panayotis Mertikopoulos. “Fog based framework for iot service provisioning.” In *2019 16th IEEE Annual Consumer Communications & Networking Conference (CCNC)*, pp. 1-6. IEEE, 2019.

[33] Liu, Yang, Jonathan E. Fieldsend, and Geyong Min. “A framework of fog computing: Architecture, challenges, and optimization.” *IEEE Access* 5 (2017): 25445-25454.

[34] Sami, Hani, and Azzam Mourad. “Dynamic On-Demand Fog Formation Offering On-The-Fly IoT Service Deployment.” *IEEE Transactions on Network and Service Management* (2020).

[35] Belqasmi, Fatna, Roch Glitho, and Chunyan Fu. “RESTful web services for service provisioning in next-generation networks: a survey.” *IEEE Communications Magazine* 49, no. 12 (2011): 66-73.

[36] Afrasiabi, Seyedeh Negar, Somayeh Kianpisheh, Carla Mouradian, Roch H. Glitho, and Ashok Moghe. “Application Components Migration in NFV-based Hybrid Cloud/Fog Systems.” In *2019 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, pp. 1-6. IEEE, 2019.