MYST: Automated DevOps for distributed applications across heterogeneous Cloud, Fog and Edge infrastructures.

Pieter Donkers
Pieterdonkers_3@hotmail.com

September 4, 2019, 70 pages

Research supervisor:	dr. Zhiming Zhao, Z.Zhao@uva.nl Senior Researcher SNE group, University of Amsterdam
Host/Daily supervisor:	Roger Salhani, Salhani.Roger@kpmg.nl Senior Consultant DevOps and Cloud Transformation
Host organisation/Research group:	KPMG - Digital Advisory, Amstelveen https://home.kpmg/nl/nl/home.html

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering
http://www.software-engineering-amsterdam.nl

Abstract

The amount of data gathered by humans is increasing [1] and more and more data is being generated at the edge of the network. To lower strain in the network, we should move processing closer to the data. Moving applications closer to the edge, distributes functionality geographically, increasing complexity. Distributed applications require smarter tools that allow a developer to maintain control over their applications. We enable developers to, not only move their applications closer to the Edge but also separate them over a distributed infrastructure across the Cloud, Fog and Edge. By adding a tool in the DevOps environment, current cloud engineers can expand their applications from cloud to the fog and edge. We combine literature research with practical research, providing insights from both academia and industry. Most DevOps tools provide specific solutions for one problem and not the whole process. Current solutions[2–5] automate the process on service level infrastructure provided by cloud providers, not on the Fog and Edge. We proposed a conceptual framework, called MYST, that can automate the process of orchestrating, configuring and deploying composite applications over Cloud, Fog, and Edge devices. MYST provides control, reconfiguration and quality maintenance during the lifetime of the application. In the future, we want to implement MYST and be able to run experiments on it.

Keywords: *DevOps, Cloud, Fog, Edge, TOSCA, Distributed Applications, Composite Applications, Orchestration, provisioning, Configuration, Deployment, Automation, Virtual Infrastructure, Heterogeneous Infrastructure*

Contents

Chapter 1

Introduction

While currently computing tends to be moving towards massive data centers [6], e.g. cloud computing More and more data is generated at the edge of the networks [1]. All this data can strain current infrastructure if the data has to be processed in these data centers. If the processing of this data can be brought closer to where the data is located, less data has to be transmitted to these central locations. If it was easier for developers to create their applications in a native distributed fashion, more processing can be done closer to the data.

Huge amounts of data require huge amounts of processing power to handle these loads. Currently, a popular way of processing large amounts of data is cloud computing. Cloud computing is fast and efficient and can process huge loads. However, cloud computing requires large central data centers, which mostly have to be scaled vertically. This results in data routing to geographically central hubs and back to the end devices. Horizontal scaling is much more suitable for expanding geographically, not only by splitting data centers but also by offloading to other devices. Putting the processing power closer to the end devices. Separating applications into multiple parts and running them on multiple machines is called distributed computing.

Cisco recognizes that moving processing closer to where the data is produced enables faster decision making and reducing the data that is sent over the network[1]. They sketch some futuristic scenarios relating distributed computing to the tv-series Westworld[2]. Linking fast decision making in the robots to fog computing while they are all connected to a central intelligence. In smart cities edge and fog computing could improve mobility. Allowing cars to talk to traffic lights and other cars, improving traffic flow and reducing human errors [7].

An online game, e.g. Fortnite[3], nowadays often involves multiple users. The number of users can be very large, in the scale of hundreds or higher, which is also called massive multi-player online(MMO) game. A MMO game has a large number of players connected to it simultaneously. Players are constantly entering and leaving games, creating a dynamic pool of machines connected to each other. During these games, the user experience has to be optimal. There are very strict requirements from the user to ensure the quality of the games. For instance, the quality of experience of many online games is sensitive to the latency users experience. Normally the players are distributed over the entire world, and the moments they are active constantly changes during the day. Not all these players have the same quality equipment and connectivity, making the types of connected devices very diverse. Depending on how the players interact with each other, large amounts of data could be sent between players. Because of the highly dynamic set of connected users, the system must be able to change the services during run-time, to ensure there is no down-time while they are playing.

Placement of gaming services closer to the players could minimize latency and offload data of the network, maintaining a high quality of experience for the players. Since cloud computing is centralized to large single data centers, we require infrastructure outside of cloud computing. Fog [8] and edge [9] computing have been proposed as extensions to the cloud. These paradigms move computing power closer to the edges of the network, closer to the users. Fog and edge devices are not as homogeneous as cloud, making the devices that are required to be supported much more heterogeneous. Effectively scaling gaming services across a distributed network, requires smart scaling strategies that are aware of the context of both the services and the players. Using smart automatic scaling, the gaming services

[1] https://blogs.cisco.com/perspectives/iot-from-cloud-to-fog-computing

[2] https://www.cisco.com/c/en/us/solutions/industries/government/us-government-solutions-services/fog-computing.html

[3] https://www.epicgames.com/fortnite

should be able to adapt to user demand dynamically, optimizing quality of experience.

It is important to regularly add new features to your games, to keep your game attractive for players. Pushing these updates quickly and regularly, while minimizing downtime, is essential to maintain Quality of Experience with the users. DevOps [10] enables the owners of such applications to achieve these fast and continues updates to their systems. As game servers are not in centralized locations anymore, the owner requires smarter ways to maintain his distributed application.

Continues integration and continuous deployment of applications allow developers to quickly get their application and updates to their users. Before this it often happened that updates got intentionally delayed to adhere to the release cycle or that unfinished products get pushed to release before they were ready [11]. By allowing the developers to create their application and deploy them whenever they want, you can accelerate the process of getting your product to your users.

1.1 Problem Statement

Currently, building distributed or composite applications for the Fog and Edge is significantly more complex than building applications for the Cloud. Tools like OpenTOSCA[2], Terraform[4], DRIP[3] and HEAT[5] are meant for automatically provisioning Cloud applications, but not across the Fog and Edge. Developers of Cloud based applications struggle to move their applications to the Fog and Edge. The lack of tooling requires them to invest more time and money to create such applications in comparison to Cloud based applications.

We want to enable developers to create distributed or composite applications for the Fog and Edge just as easily as for Cloud. Developers should be able to design an application accompanied with its dependencies and required resources. After a distributed or composite application is designed they should be able to easily roll out their application to a production environment.

Maintaining distributed applications across a combination of the Cloud, Fog and Edge is harder because of the separation and the multiple types of devices running the composite parts of the application. Even though of this separation over multiple paradigms and types of devices, we want to enable the developer to maintain their applications and the quality of them just as well as in the cloud. A relatively new paradigm that gives developers more control over applications during its lifetime and makes them easier to maintain is DevOps[10]. DevOps allows the developers to easily adept to changes in the environment and quickly update their applications. We add a tool to the DevOps tool-chain that is able to automate orchestration, configuration, deployment and quality assurance. By making the tool a part of the Cloud Native Landscape, a large group of cloud developers can scale their applications to the Fog and Edge.

It is important the quality of the applications over the Cloud, Fog and Edge is maintained during its lifetime, focusing on quality of experience. For example, the implications of lowering the latency only can be huge. Not only real time tasks require short responses times, consumers also require short response times. A recent study showed that Amazon noticed a 1% decrease in sales in their online store when an additional 100ms latency was introduced. While google saw a 20% decrease in revenue when the time it took to display search results was delayed with 500ms [12]. The size of these companies and their revenues already make it very relevant to ensure quality of service. Using fog or edge computing as an addition to the cloud provides a Software Defined Infrastructure(SDI). The flexibility this provides can be utilised by smart distribution of tasks, distributing the computing power to both high powered (vertical) nodes and between multiple (horizontal) lower powered nodes.

Some example applications for different Quality of Service constraints that could be optimized using our framework.

- **resource utilization** Match required workload with number of containers.
- **Latency** Collaborative working, game services and real time data processing.
- **Cost** Placement in cheaper VM's (e.g. spot instances).
- **Localisation** Privacy and providing localised information.

To summarize, we propose a framework that 1) fits in the Cloud Native DevOps landscape, 2) providing run-time application and infrastructure automation, 3) across the Cloud, Fog and Edge, 4) maintaining quality expectations during the lifetime of the application.

1.1.1 Research questions

To build a framework which solves these problems we have proposed several research questions. These research questions help us decompose the problem, and tackle the problems one by one.

- **Research Question 1** How to effectively model a distributed application and its underlying heterogeneous virtual infrastructure during the application lifecycle?
- **Research Question 2** How to automate the orchestration, configuration and deployment of distributed applications over heterogeneous infrastructures?
- **Research Question 3** How to autonomously maintain the quality of service of a distributed application over heterogeneous virtual infrastructure during run-time?

1.1.2 Approach

We have applied the following steps to be able to propose a solution to the problems we have mentioned earlier in this chapter.

1. Start by researching the subject, finding relevant literature and gaps in the current state of the art.
2. Find techniques that we can use to model or describe composite applications for heterogeneous infrastructures with their dependencies.
3. Find how the application model with its dependencies can be automatically created across the heterogeneous infrastructure.
4. Substantiate our academic research with literature from industry, and practical research to increase the relevancy within the field of DevOps.
5. Propose a framework that implements and solves the problems found for modeling and automation, using techniques found in academia and industry.
6. Verify the proposed solution using a real-world example.

1.2 Contributions

Our research makes the following contributions:

1. Extensions to the TOSCA language to better support heterogeneous infrastructures.
2. An extension to the TOSCA language that improves the quality of service specification.
3. Support for self decision making in tools implementing the TOSCA language.
4. A framework for orchestration, configuration and deployment of an application description across heterogeneous infrastructures.
5. A framework that implements automated run-time reconfiguration and QoS control.

The framework we propose extends current DevOps tools. Instead of only focusing on general purpose computing and cloud, we introduce the option to describe and automate applications across the Cloud, Fog and Edge. Our solution removes some limitations for Cloud developers and allows them to scale their applications to the Fog and Edge.

We extend Fog, Edge and IoT paradigms by providing a method of describing and automating composite applications across these devices. Not being limited to a single type of device, but the ability to do this across heterogeneous devices.

Note. We have not introduced TOSCA yet. However, some of the contributions are specific to TOSCA. TOSCA is a language that enables developers to describe their composite applications. More about this in chapter 4.

1.3 Outline

In chapter 2 we will discuss some background information and related work for this thesis. Chapter 3 gives a short introduction into the framework and its design. Chapter 4 describes the process of finding the right modeling and description language, the gaps and the extensions to fix these gaps. Chapter 5 discusses how the process from going to a deployment from a description can be automated. And in chapter 6 we propose a conceptual framework that implements solutions to the open problems we pointed out. Chapter 7 validates our solution and substantiates some of our decisions with experiments. Finally, in chapter 8 we conclude and address the future work.

Chapter 2

Background

The background ensures that the reader of this thesis has the relevant knowledge to understand the research. It introduces some concepts that are considered base knowledge. We start by explaining the concepts of cloud, fog and edge computing. Follow this up by distributed computing. And then talk about how applications and their infrastructure are created. After this we talk about context aware systems and how quality of service can be maintained.

2.1 Terminology

QoS Quality of Service, Measurement of overall performance of a system.
QoE Quality of Experience, Measurement of quality based on user enjoyment.
OS Operating System, like Windows, Linux or MacOs.
GP General Purpose
IoT Internet of Things
IaaS, PaaS, SaaS {Infrastructure, Platform, Software} as a Service
Container Isolated applications with included dependencies. We are talking about application containers and not full os containers.
Provisioning Combination of orchestration and configuration.
DevOps Combination of "Development" and "Operations"
Heterogeneous Infrastructures Infrastructures consisting of Cloud, Fog and Edge devices.
Composite Applications Applications which are separated into multiple sub systems.
Distributed Applications Composite application which is scattered over multiple machines, often geographically separated.

2.2 Search Strategy

We used google scholar for finding relevant papers. The papers we found were mainly focused at the following keywords and topics.

- cloud {orchestration, provisioning, configuration, deployment}
- {cloud, fog, edge, IoT, infrastructure} automation
- configuration management
- infrastructure as code
- distributed computing
- context aware {provisioning, deployment, software distribution}
- {provisioning, deployment} heterogeneous infrastructure
- {workload, software} {estimation, characterization, profiling}
- adaptive software
- {Cloud, Fog, Edge, infrastructure, application} {description, modeling} language

With cloud computing and DevOps still gaining popularity we do not want to limit ourselves to scientific research. We also tried to gain knowledge in current trends and developments using blogs, whitepapers and forums. Also, to gain some experience with infrastructure automation and the configuration of it, we will experiment with the state of the art in industry and DevOps. This allows us to

better see the limitations of current technologies and the gaps we could fill.

2.3 Heterogeneous Infrastructure

For our interpretation of heterogeneous infrastructures we chose to use cloud computing paradigms, namely the cloud, the fog and the edge (figure 2.2). The heterogeneous devices can be all devices which operate with in these paradigms. Also because we see our framework as an extension for the DevOps community, which is mainly focused on cloud computing.

While cloud systems mostly rely on general purpose machines, Fog, Edge and IoT solutions are mostly domain specific and often rely on specialised hardware, communication protocols and data models [13].

Cloud

Cloud computing can be commonly described as large centralized data centers maintained by service providers. They provide on-demand compute resources like, processing power, storage and networking capabilities that are not managed by the user. This definition is not really the full image of what cloud is, but mainly the description of public cloud.

There is also the definition of private cloud which is comparable, but the cloud environment and physical infrastructure is mostly maintained by your own company but runs on a comparable cloud provider operating system. Both provide a layer of abstraction to the developers using them to create software defined infrastructure for their applications.

Next to public and private cloud there are some other definitions used in cloud computing. These definitions are just combinations or extensions on the public and private cloud. Multi-cloud, using multiple cloud providers to run a single application, picking the best provider for certain parts of the application. Hybrid-cloud, combining cloud technology with on-premise solutions, runs selected parts of the application on the cloud and other parts on-premise, mainly used for security sensitive purposes where the data stays on-premise.

Cloud providers provide different layers of resource abstraction. Infrastructure as a Service (Iaas), Platform as a Service (PaaS) and Software as a Service (SaaS). Each layer gives the developer more control over the infrastructure but also more responsibility (figure 2.1).

This flexible infrastructure is operated by a cloud operating system, often called a cloud stack. The operating system is responsible for creating and managing the software defined resources on the physical resources. Different providers use different stacks.

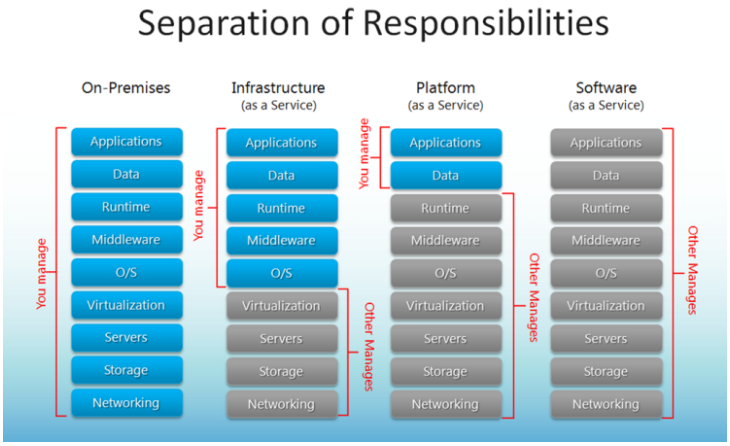

Figure 2.1: IaaS, PaaS, SaaS[1]

The main advantage of using cloud is the abstraction of resources. This allows the users to easily scale up or down their infrastructure, providing resources on-demand. The user only pays for what he uses. This abstraction has allowed start-ups to have access to enterprise grade infrastructure, and allows enterprises to cut costs when not requiring peak performance [14]. The versatility of the cloud has become a prime enabler for recent DevOps practices.

[1] http://robertgreiner.com/2014/03/windows-azure-iaas-paas-saas-overview/

Cloud computing is almost always based in large centralized data centers. Because there is a relatively small amount of these data centers, data often has to travel large distances to get from a device to the cloud. These large distances could introduce extra latency, and strain network infrastructure. Making it important to weigh the computation advantage against the disadvantages.

Edge

For the edge we consider devices that are on the edge of network. Think of IoT devices, sensors, but also personal computers, televisions and more. These devices are not always connected to the internet and can require other connection methods. The number of edge devices is far greater than the amount of cloud devices, and are more geographically scattered. Edge computing helps cloud computing by processing the data right where it is generated, preventing sending all data to the cloud to be processed.

The differences in edge devices are great and these devices are much more different from each other than cloud devices are. For example, embedded devices we can think of, micro-processors, digital signal processors(DSP), field programmable gate arrays(FPGA), application specific integrated circuits (ASIC) but also of more high level devices like devices able to run a Linux operating system, like smartphones and Raspberry Pi's. In comparison to cloud computing where everything runs on general purpose hardware and software, this is harder to unify and often requires project specific adjustments.

The advantages of computing at the edge is that the processing happens right where the data is generated. Mitigating the need to send the generated data over the network to another device processing the data. Removing strain on the network but also on the network interface of the device. Several IoT network providers like LoRa[2] base their price on the amount of data sent, possibly lowering cost.

Edge devices are more often than not devices with constraints on processing power and energy use. Depending on the amount and type of data, it could be very slow or drain the battery of such a device, if such data has to be processed locally. Making it important to weigh the advantages against the disadvantages.

As these devices are located at the edge of the network, their reachability is not as reliable as data centers are. In contrary to data centers having fast and reliable connections to an , edge nodes often have consumer grade connections. Or worse, have no continues or reliable connection to the internet.

Fog

For the fog we consider all the devices between the edge and the cloud. We can consider network switches, routers, internet service providers, domain name servers (DNS), micro data centers and more. The fog is an extension to the cloud as it places computing resources closer to the source of data, and sometimes the data in the fog and closer to the end-users. Fog devices are more likely to contain more processing power as edge devices and do not have the same power constraints. They are able to process data close to the edge without requiring extra work for the edge devices.

Since fog nodes are often connected to multiple edge devices, it allows systems to circumvent the cloud. Fog devices can enable connections between edge devices and communicate data between them without the need of cloud. An example of this could be, a traffic light telling a car it will go red in 10 seconds trough a shared fog gateway.

Depending on the fog device, they are not always accessible to put software on. For instance a company is really able to run software at a public DNS server. But they can set up a DNS server in their building which they are able to process information on.

Actually implementing fog applications is still a problem. For instance, according to Vogler *et al.* [15] their findings show that IoT gateways often posses limited hardware and rudimentary operating systems. And deploying software is often a manual and tedious task.

[2] https://www.simpoint.com/lora/tarieven/

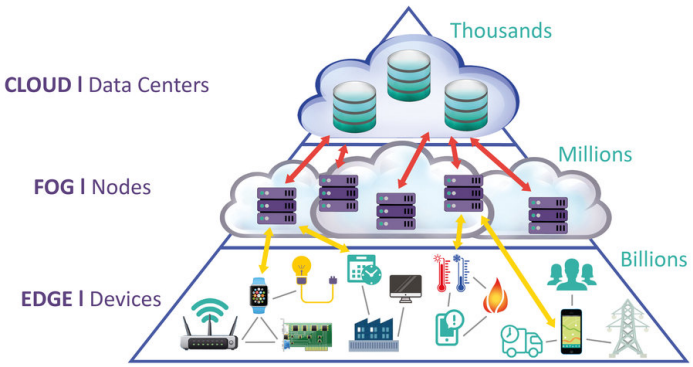

Figure 2.2: Cloud, Fog and Edge Architecture[3]

2.4 DevOps

DevOps is a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality. [16] DevOps allows developers to frequently change their applications while bothering the users as less as possible. A high degree of automation and short update cycles allow developers to quickly push their changes to their users. The practice is highly based on practices from agile and lean methods of working.

The DevOps pipeline includes several steps and tools. Ebert *et al.* [10] defines tools in three categories, build tools, continues integration tools and monitoring tools. The pipeline is considered circular, because the process is based on iterating the system to improve it (figure 2.3).

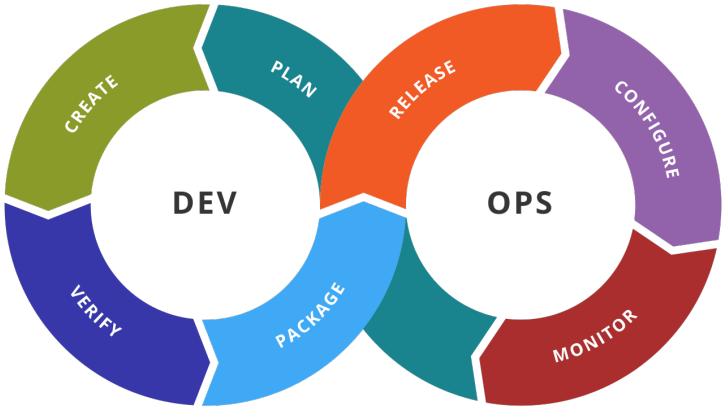

Figure 2.3: DevOps Cycle

A DevOps way of working utilizes experimentation, adaptation and learning. The iterative development allows the developers to make small changes, learn from them, and revert them if they negatively impact the system. The method is based around giving constant feedback to base the iterations on. Developers tend to be more productive when they quickly receive feedback[4].

Automation is a key enables of DevOps. Looking from the lean perspective, a lot of effort is wasted on errors and wait time[4]. Automating aspects like testing, deployment and monitoring, allows developers to focus on what is important, adding functionality.

Most companies require a culture shift to be able to adopt the DevOps way of working. Complex monoliths have to be decomposed into smaller parts, that can developed and deployed independently. The Operations environment has to be maintained instead of outsourcing it. Silod cultures in companies had to be broken up and combined, both from development and operations. During development, testing is important to ensure the quality of the system in the whole pipeline.

[3] https://leanbi.ch/en/blog/iot-and-predictive-analytics-fog-and-edge-computing-for-industries-versus-cloud-19-1-2018/
[4] https://devops.com/surprise-broad-agreement-on-the-definition-of-devops/

According to [10] it is challenging to combine embedded systems with DevOps. The reason for this is that they require legacy code and architecture. Since the internet of things contains a large amount of embedded devices, this could pose a problem.

2.5 Distributed Computing

Distributed computing is the process of dividing large monolithic applications into smaller individual parts, also called micro-services or composite applications. The smaller parts can be run individually and communicate with each other to implement the same functionality as the monolith. This way the application can be distributed on multiple machines and different geographic locations. Distributed computing can be used to offload required processing from central machines to multiple smaller machines. Or to easily create applications with decomposed building blocks.

Dividing large monolithic applications into distributed applications of smaller parts can have several advantages like, theoretically infinite scaling, allowing parallel processing, reducing single point of failure errors and smaller individual blocks allow individual updates. But it also has negative aspects like, harder to implement, it requires extra synchronisation and coordination (scheduling) effort and it is harder to achieve a secure platform due to multiple entry points.

Distributed data processing, allows an application to process very large amounts of data that no single machine would be able to process. Data will be sent to a machine to be processed and the machines will only communicate thinned data or events to a master machine. Which requires less compute power for an analytics application. This could be realised using two ways, 1) the processing lays between the date acquisition and the master, thinning the data before it reaches the master, or 2) the master could send unprocessed data to the distributed system, essentially combining all the compute power of the machines.

The flexibility of the cloud allows for flexible infrastructures. Separating applications into smaller blocks and placing them in separate compute units, allows easy scaling of blocks that require more processing power. Dividing applications into micro services allows for greater maintainability of separate services in the application, and also allows developers to easily update or replace a part of the application without replacing the whole monolith. Different versions of a part could be hosted for different regions allowing flexibility for locality.

Peer-to-peer enable to processing of data without a central master to control the process. The peers in the distributed system are responsible for communicating the data and the findings between them themselves. If one of the peers goes down the entire system will not notice the difference, in comparison to a master going down. This method combines nicely with mesh networking, where there is also no single master in the network. A problem with this method is finding where the data is, because you cannot ask a master for the data.

Multiple papers [17–20] research optimal scheduling algorithms for placement of distributed applications in the edge, fog and cloud. We want algorithms like that to easily be implemented in our framework. Such that the developers does not have to build the supporting infrastructure. There is research about optimal scheduling with already provisioned nodes and how to optimally provision these nodes. But there is a lack of scheduling using both types. A developer should be able to take the provisioning of a node into account when scheduling tasks.

Stefan Nastic *et al.* [21] They propose to extend software defined infrastructure definitions to IoT devices. By adding IoT gateways to the cloud, they propose a standardized method of communicating with IoT devices. Their proposed extension for IoT systems, do not really enable the modeling and automation of the IoT devices themselves. This means their framework does not really configure or deploy to these heterogeneous devices, but deploys management units in the cloud. We want to enable a developer to model cloud, fog, edge and IoT devices in the same description. In the future work they also mention how they want to implement policy based automation. Allowing them to optimize data quality, security and safety aspects.

2.6 Orchestration, Configuration and Deployment

The creation of the infrastructure and setting up the application to run on it can be divided into three steps. Orchestration, creating the infrastructure. Configuration, facilitating the right dependencies. And Deployment, releasing the application to the infrastructure and running it. In this chapter we will explain these concepts.

Orchestration

To run an application a machine is required, which can either be set up manually(physical orchestration) or can be provided by a service provider(software based orchestration). Cloud providers allow a user to easily create a software defined infrastructure, using either a user interface or a description language. A developer needs to determine the requirements for the application before hand and needs to create the appropriate infrastructure for this. These days, this process is often automated with cloud getting more popular, simplifying the process of creating the infrastructure. In this thesis we will largely ignore on premise orchestration since this is mainly manual labor.

Infrastructure as Code (IaC) describes the infrastructure (VM's, Network, Security, etc.) in a scripting like language. The advantage of using IaC as code is that your entire infrastructure is declared in a format that is maintainable and is able to be kept in a version management system. Also because everything is scripted it is very repeatable and able to be automated while almost no knowledge is required when setting up the infrastructure, since it only requires running a script. It should be mentioned that normally this does not include configuring, but some of the tools implementing IaC implemented some of it in their IaC tools. The tools can be divided into declarative or imperative infrastructure generation. Examples of tools providing automated orchestration using IaC are Terraform[4], Cloudformation[22], ARM[23] and HEAT[5].

Contrary to physical orchestration, automated orchestration using service providers provide optimal flexibility for user to scale their infrastructure up and down. While also allowing developers to easily realize enterprise grade infrastructure. Processing load could easily vary during the day or over a larger period, this flexibility allows the user to just have enough processing capacity for that moment instead of always having the maximum capacity available.

In comparison with using an user interface to create a software defined infrastructure, automated orchestration requires more initial effort for setting up the infrastructure. The advantages come when the infrastructure should be created more than once. The scripted infrastructure often requires debugging, and multiple runs to get right. It also is harder to learn in comparison to using a user interface.

Some tools like Terraform[4] are considered cloud agnostic. The tool itself is cloud agnostic, as it allow orchestration across multiple service providers. However, the language used is only partly agnostic, as different providers require different code for creating infrastructure. A developer should only be required to specify a VM with certain properties, allowing our solution to find the machine at multiple providers.

Configuration

Configuration Management (CM) describes the applications and its dependencies that should be ran on the infrastructure, describing the process that sets up the infrastructure enabling the application to run. The process can be executed manually, or can be automated. In this thesis we will mainly consider automated configuration because the other one obviously requires manual labor. They mostly work by defining a set of instructions that should be executed on the infrastructure. This set of instructions will provision the correct dependencies and the application. The downside of this is that it not really describes how certain dependencies link to the application or parts of it.

Most configuration management tools define a list of steps (imperative) the tool has to execute to make a system ready for the application. The steps are mostly comparable to commands you would execute manually on a system but now its in a script. Because of the imperative nature, a developer has to build in checks for the state of the machine himself. Examples of tools providing CM are Ansible, Chef and Puppet.

By scripting the configuration, there is no uncertainty in the configuration process anymore. A developers knows which steps have been executed and this is very repeatable across different machine.

Automating the configuration process allows developers to easily scale up the configuration process. Once the configuration script is written, it is easy to execute it on a large amount of machines. Just like orchestration, the initial development is harder to learn and requires more time.

Deployment

During this thesis we will define deployment as the process of building an application, moving it to a machine that is supposed to execute it and start the application. Officially deployment is considered much wider and considers more aspects. Carzaniga *et al.* [24] define deployment as 8 different activities.

- **Release** Prepare and package the application for transfer.

- **Install** Transfer the application from host to consumer and configure the consumer machine.
- **Activate** Start the dependencies and the application on the consumer machine.
- **Deactivate** Inverse of activation.
- **Update** Application change gets triggered by developer. Includes triggering all previous actions.
- **Adapt** Application itself changes itself to adapt to changes in the environment.
- **Deinstall** Inverse of Install, removes all the dependencies and the application from the machine.
- **Derelease** Executes De-install and retires the machine. Includes advertising the retirement of the system.

Some of the configuration tools already enable to get the application to the machine but do not really support this during run time or continuously. There are tools available that will automate the deployment process. They often offer extra services like automated testing and rollback on error. These tools often only work with general purpose operating systems and devices. Making them suited for (micro)data centers but often unsuited for some fog and edge devices.

Many devices are not of the general purpose type, requiring custom methods for application deployment. The supported software can differ and the connections they support. Depending on the frequency of deployments, manual deployment can be better suited. For instance, sensors are often only provisioned once in their lifetime, making continues deployment unneeded. Often software comes packaged with the hardware. Manufacturers often do this for embedded devices or application specific hardware. Requiring no deployment on the devices since this is already done during manufacturing.

Carzaniga *et al.* [24] already address the problem of deploying across heterogeneous devices. But their definition of heterogeneous devices still adheres to general purpose computing with different operating systems, like mainframes, servers, workstations and personal computers. They fail to address the problem with heterogeneity in other devices like sensors, running specialised hardware.

Continuous deployment is and extension that helps team deploy their applications and updates faster. Such that users can more quickly use the newest features developed. Updates do not have to be delayed for scheduled updates of the entire system. The core functionality is still the same as it just moves the newest version of the application to the production environment. Continuous deployment is however closely tied in with other parts of the development pipeline, automating large parts of it. For example, updates in the code base are automatically tested and if they pass, are automatically built and deployed to the production environment.

2.7 Context Aware Systems

"Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves" [25] This roughly translates to, how the environment influences the interaction between the user and the application. A context aware system will use the environment to make decisions [25]. For example, a navigational system making decisions based on your location. In our case the environment variables would mainly consist of the computing resource availability, user latency and service reliability. The context will be used to maintain quality of service constraints set by the designer of the application.

Ichiro Satoh [26] implemented a context aware deployment application. The application tracks users during a guided tour in a museum, and displays information on screens close to them. The system binds information agents to users and moves these agents between systems. The agents are aware of the context, mostly user location, and use this to migrate themselves between systems. Because the agents themselves move between systems this is less of a context aware deployment and more of a self migrating agent application. The system also assumes that the infrastructure and the machines in there are pre-configured. Meaning the application lacks orchestration and configuration, only implementing deployment.

Chantal Taconet *et al.* [27] presented a framework for the provisioning and deployment of applications for mobile users called Smart Deployment Infrastructure (SDI). A component based application description is used with their types and connections between the instances. A self defined description language is used which is comparable to TOSCA but not as well defined. The language describes the assembly of the components that describe the application. The purpose of their system is not to share the resources in a network in a smart way, but to only fetch packages on devices when they are needed. The context selection is not about deciding where to place certain workloads but about selecting what workloads to pull based on for instance geography. By using the description language it should be portable

for heterogeneous devices but currently the scope of the devices they are able to deploy to is limited to terminals. The application requires an agent to run on the target device for fetching packages. They do talk about just-in-time deployment, which is deploying your application to the device just before the user needs it. I would summarize their service as, devices only pulling the parts of an application they require when they are required.

2.8 Dynamically Scaling Software

For developers the computing resources in cloud are virtually unlimited. When running your own servers there is a static amount of computing resources available. Since you own the hardware you can not easily scale your servers up or down. Making your instances underutilised most of the time or not able to handle the load during peak times. The cloud allows developers to increase the computing power of their services with ease. This process could even be automated, increasing computing power during peak times and reducing it when not required.

Chieu *et al.* [28] propose a method to mitigate load surges on IT resources. They ensure an application is spread across multiple machine and create duplicates of applications to handle extra requests. By scaling up and down the number of machines running the application, they are able to match the processing capabilities to the processing needs. Their method is based on introducing a single machine that reroutes traffic to multiple other machines. This allows their application to have a single point of entry instead of multiple.

Calheiros *et al.* [29] proposed an automated application provisioner for cloud based applications, more specifically for SaaS and PaaS based applications. By analyzing workloads and predicting performance the framework itself ensures quality of the applications in the cloud. The system is able to estimate and place a given workload in a virtual machine. Not really modeling the infrastructure that belongs to an application. Their system is limited to linking single applications to single virtual machines, with no support for multi-tier applications. In the future work section, they mention that they want to extend the framework to composite applications and combine it with more QoS parameters. We also see that the framework lacks support for devices not managed by service providers, which we want to extend, to better the fog and edge.

2.9 Adaptive Software

W. Luk *et al.* [30] implemented a tool chain that is able to transform a single code base to target code for heterogeneous processing elements and distributes the platform specific code to these. Their definitions of heterogeneous systems is limited to embedded systems with processing elements like micro processors, field programmable gate arrays (FPGA) and digital signal processors (DSP). While their tool chain is not suited for transforming applications for general purpose computers it could be suitable for applications running in the edge. They describe the application in a high level language which is transformed to an intermediate representation (IR). A set of tools use the IR to determine which processing element is most appropriate for the task. The IR is transformed to platform specific code and is executed on the processing element. In their tests they achieve a speedup of around 11 times compared too running the original code on a micro processor.

Tore Fjellheim [31] describes how an application for a multi-platform environment should be programmed to enable packaging and deployment to heterogeneous platforms. They advice to make applications component based for iterative deployment. The granularity of the components is based on the activity it performs, a single component should fulfill a single activity. By creating a separation between data and behaviour the component should be updateable without harming the data. Smaller components should also be easier to transform for specific platforms. The components should hold metadata or context information with information of run time requirements. The information can be used by a selection algorithm that determines on what machine the component should be ran. Components should decouple the processing from communication to enable incremental delivery. Event based decoupled communication would be most suitable between components.

Joanna Sendorek *et al.* [32] propose a framework for building data processing pipelines across heterogeneous infrastructures. The framework implements an API which can be used to build the data pipeline where processing components are encapsulated by their API components. Their framework decides per component where it can be placed in the heterogeneous infrastructure. They generate platform specific source code to implement the actual components. Their framework lets the users specify the data pro-

cessing pipeline in such an abstract way, that it is relatively easy to port it to heterogeneous platforms. The applications are limited to data flow applications and to the java language. The java language is itself portable since it uses a java virtual machine and the inputs and outputs of their components are limited to streams. Meaning the framework is mainly a packaging and placement tool which forces the user to build his application in a certain way.

Containers (e.g. Docker [33]) allow developers to bundle applications with their dependencies and run them in an isolated environment. They are not really adaptive but do run on most operating systems. If a container is built on the machine of a developer and it runs, it will also run on another machine, without installing all dependencies. Containers are comparable to virtual machines as they allow running multiple bundled applications. However, virtual machines require the entire operating system to be packaged, requiring more resources and less flexibility. Virtual machines make use of an hypervisor which allows every virtual operating system to run on another one, as the hypervisor translates instructions to the kernel. Containers built for one kernel are not able to run on another kernel, the host operating system will take care of the virtualisation. The advantage of containers is that they are easy to build, small in size and quickly able to deploy.

2.10 Summary

Short introductions were given for different run-time environments like the cloud, the fog and the edge. We covered how applications could be distributed too many devices and what the advantages are. And we spoke about how the realization of such a system could be automated.

To maintain the quality of applications we discussed how applications are able to use their environment to make decisions. We introduced how application are able to change to support multiple types of platforms.

The discussed automation techniques can describe the infrastructure and the dependencies of an application and correctly provision it. However they still not describe the connections between different parts of the application and how the application communicates, this would require another model. Meaning from these descriptions we would be unable to reason the purpose of the application.

Chapter 3

MYST

In this chapter we will give a short introduction about the design of the MYST(no abbreviation) framework to give the reader some context while reading the upcoming chapters.

3.1 Overview

MYST should allow developers to design composite applications across heterogeneous infrastructures in conjunction with the resources required for his application. We want to unify designing these, and automate the process of realizing a running application. Developers should be able to easily design composite applications around heterogeneous devices. MYST should be responsible for deciding how the composite application is distributed across different types of devices and with different service providers, abstracting infrastructure organization from the developer. A developer should be able to set what performance he expects of his application during run-time. MYST should then ensure that these expectations are uphold, and if not take actions to mitigate that. When a developer wants to update their application or make changes in the design of his application or infrastructure, it should be easy to do so.

3.2 User stories

To clarify requirements across stakeholders we propose some user stories. The user stories help us formalize requirements of the framework. The format of our user story is as follows: As a <stakeholder>, I want <action>, <reason>.

- As a developer, I want to be able to describe my composite application across cloud, fog and edge, so that I can describe entire applications on heterogeneous devices.
- As a developer, I want to be able to describe my composite application, infrastructure and requirements in a single description, to abstract the design and unify steps in the development pipeline
- As a developer, I want the framework to automatically orchestrate, configure and deploy my application, to lower effort and prevent errors in repetitive tasks.
- As a developer, I want the framework to find the correct resources for me, so that load matches the resources.
- As a developer, I want to be able to define performance constraints of my composite application during run-time, to ensure the application works for my users as I intended.
- As a developer, I want the framework to automatically adjust the infrastructure when performance constraints are not met, so that I do not have to monitor it full time.
- As a developer, I want to be able to change the description during run-time and have the application reconfigured, so that my users will not encounter downtime during updates.

3.3 Requirements

Based on the previous description we have defined some requirements of the system.

3.3.1 Functional Requirements

Functional requirements should describe the behaviour of the system, without saying how it should be achieved and without the performance measurement.

- The framework has to be able to translate an application description to an orchestration, provisioning and deployment description.
- The framework must be able to estimate the workload and required resources for different types of applications for heterogeneous devices and infrastructures.
- The framework has to be able to use a QoS description to maintain the QoS of the application and infrastructure. Both during initial orchestration and during run-time.
- The framework has to be able to generate orchestration, provisioning and deployment instructions for heterogeneous devices, infrastructures and connections.
- The framework has to be able to communicate and execute instructions to heterogeneous devices.
- The framework must allow the developer to easily reconfigure his composite application.

3.3.2 Non-Functional Requirements

Non-functional requirements are requirements that specify how something should be achieved or describe the performance of functional requirements. As our framework is still conceptual we did not set many non-functional requirements yet.

- The framework should be easy to use. We want to make it easier to create composite applications, not harder.
- Once the framework is actually implemented it should be open source. This would allow other people to improve it. This means the code quality and maintainability of the framework should be high.
- The framework should make use of existing DevOps tools to ensure we are not reinventing the wheel.
- The framework should keep track of the state of the current infrastructure to be able to compare it to a new description.
- The framework should be able to request compute resources from cloud providers to ensure virtually unlimited scalability.
- The framework should automate is much as possible and not require developer interference.

In the future we would like to specify more Non-functional constraints. Other examples of possible future non-functional are.

- The framework should maintain QoS constraints minimally 98% of the time.
- The framework should be extensible to support new technologies.
- Once QoS constraints are broken, the framework should be able to recover to a state where they are met within 10 seconds.
- When the developer provides an updated application description, the framework should have it deployed in under 1 minute.

3.4 Design

In figure 3.1 we propose a simple overview of the MYST framework. Starting with the description provided by the developer, we describe what actions the framework will perform on the description to actualize the infrastructure and the application.

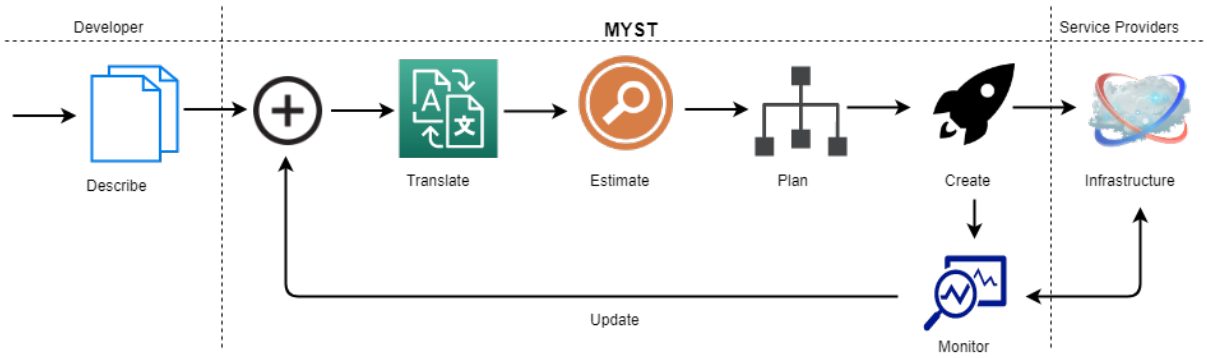

Figure 3.1: Initial design of framework

Description The user has to describe his application in combination with the infrastructure. The description should contain all the information that is required to actually get the application running. It should include the logic of the application, the infrastructure required to run the application, how the infrastructure interacts with the application logic and how the application should be maintained during lifetime. The description allows a unification across different steps in the development pipeline.

Translate The description provided by the developer should be transformed into a representation that is usable by the framework. The translator defines a common language between the developer and the framework. The common language allows the communication between them. The representation should just contain the translation of the description, while the rest of the framework is able to update and add information to this representation.

Estimate The composite application requires different types of devices and different performance. By estimating the workload of parts of the application, we are able to make smart decisions on how parts of the application are distributed over the resources.

Plan After we have gained knowledge about the requirements of the applications, we are able to plan the application across the infrastructure. We are able to bind components to actual computing resources. This step should also generate the plan that is needed to realize the infrastructure, which steps should be taken and how should they be executed and in what order.

Create To create the infrastructure several tools are required. Depending on the device requested, we need to request cloud resources, or configure edge or fog devices. This block is responsible for creating the infrastructure, configuring the infrastructure for the application, and deploying the composite application to the infrastructure.

Monitor This block will monitor the infrastructure that is created by the framework. It ensures that the quality of the infrastructure stays as expected, even during run-time changes. The block is also responsible for updating the description and triggering a reconfiguration, if the infrastructure does not meet the quality standards.

Chapter 4

Describing Distributed Applications

In this chapter we will introduce modeling and why it is needed for our framework. We will discuss present techniques and compare them. The most fitting technique will be chosen for our framework, and will be explained. To support our framework and the requirements, we are proposing some extensions.

4.1 Introduction

To create abstract descriptions of the application, developers are going to model their applications and their infrastructure. We strive to combine modeling for cloud applications with more heterogeneous devices, like in the fog and the edge. In search for the right language to enable developers to correctly describe their applications we are mainly focusing on cloud modeling languages. Since fog and edge are both cloud related paradigms, a cloud modeling language should be most suitable. At the end of the chapter a suitable modeling language should be proposed, whether it being a single one, a combination of multiple, or an existing one with proposed extensions.

The description is a common language across the development pipeline. It allows different professions, who are responsible for different parts of a system, to communicate the design of the application. A software architect is able to build the initial design; a developer creates the implementation and attaches it to the design; and an operator ensures the design and its infrastructure are upheld during run-time of the system. They all use the same design of the system, but are responsible for different parts of it. The description enables different levels of abstraction to be in the same design. For instance, a software architect does not care how the implementation is built, as long as it adheres to the specifications.

Not only does it provide a common language between different people in the development pipeline, it also provides a common language between the people and the framework. This allows for the automation, which increases consistency and reliability during the development process, and abstracts much of the manual labor normally required. The description allows for the decoupling of the implementation, design and operations of the system. It provides a level of abstraction which makes it easier to get an overview of the entire system. A formal description of a system preserves knowledge, which is normally held by people in a maintainable method.

Requirements of the Description Language

From the use-case and the problem specification we have formed some requirements for the modeling language.

- **Heterogeneity** The language should support multiple types of devices, across cloud, fog and the edge.
- **Extensibility** With the constantly changing technologies, the language should be able to be extended to support these.
- **Automation** The language should not only support describing the infrastructure, but also the software configured on top of it.
- **Tool implementation** Some tools should be able to use to language to actually realize the infrastructure.
- **Quality specification** The developer should be able to describe the hardware requirements or the required quality of service for his application.

- **Live adaptation** The language should support the run-time adaptation of the description, the infrastructure and the configured software.

More specific requirements from the gaming example are:

- Run-time orchestration, configuration and deployment
- Run-time selection of host for application components
- Edge based rerouting and load balancing
- Run-time resource monitoring and decision making

4.2 State of the Art

This section investigates state of the art in modeling and description languages. First, we specify some characteristics that are important to us. Second, we discuss languages that we found and how we found them. Third, we analyze what is missing in these languages. And fourth we will talk more about the chosen language.

4.2.1 Key Performance Indicators

Current description languages have different purposes and features. To review different description languages we set a few interest areas. These areas are defined by what we find the required characteristics for describing the application infrastructures. In table 4.1 these key performance indicators are used to compare the different languages.

Service layer The modeling language should have support for multiple layers of orchestration. To support the heterogeneous infrastructure, ideally the language should support IaaS, PaaS and SaaS. When including fog, edge and IoT devices the support should even be extended to also support more hardware oriented approaches, like networking. Or support newly proposed services like Fog as a Service (FaaS) [34].

Extensible If the language only supports the cloud paradigm, is it extensible to devices in the fog and edge? How easy is it to extend the language to support new paradigms and technologies?

Heterogeneous Does the language natively support heterogeneous infrastructures (cloud, fog, edge), not only devices but also connection protocols, provisioning and deployment.

QoS Does the language allow a developer to define non-functional constraints in their design. And how easy is it to implement and uphold them?

Automation Does the language include a set of automation tools? How easy is it to go from the description to a provisioned and deployed application.

Syntax A language should be readable and easy to interpret by the developer, while also allowing software to read the description to automate the process.

Run-time support Does the language support run-time adaptation and reconfiguration?

4.2.2 Language collection

A. Bergmayr *et al.* have done a systematic review of multiple cloud modeling languages. They state that multiple languages have different purposes. For instance, CLOUDMIG is for migrating on premise solutions to optimal cloud based solution and CloudNaaS emphasizes on networking [35]. The whole list of 19 cloud modeling languages and its pros and cons can be found in [35]. Since their paper is only a review, they do not conclude what the best language is. The paper does describe the purpose of every language.

We have found a number of description languages that could be interesting for modeling the infrastructure of the cloud applications. The selection of the languages was mostly manual. The university supervisor suggested looking at the TOSCA modeling language, which is also interesting since it is a standard recognized in the industry[1]. Based on this we found several papers comparing their language to the TOSCA language, CAMEL [36] and HOT [37]. After reading Bergmayr's systematic review we also selected some languages we deemed interesting for our use. Namely, GENTL[38], CloudML[39], Blueprint[40] and the language they propose CAML [41]. The development community seems to recognize Terraform[2] as the standard for cloud agnostic orchestration. And most cloud providers have their

[1] http://www.oasis-open.org/committees/tosca/
[2] https://www.terraform.io

own orchestration languages[3][4][5], which we compare too. A short comparison between the languages using the key performance indicators can be found in table 4.1.

CAML, models architecture of cloud applications. Bergmayr *et al.* [41] recognize that UML is used for modeling applications in 86% of organizations, and they want to add cloud modeling support to the UML language. Their purpose is to combine the architecture modeling with cloud provisioning. Building on the existing language does make it easier for developers to adopt the language, and supports most features from the established language. Tooling depends on applications that implement tools for UML. Which require them to implement custom support for CAML. A tool was proposed called CAML2TOSCA in the same paper, but this is conceptual. Concluding that actual practical support is not present. *Ext.* The language can be extended to support heterogeneous devices but this would require the change of the language code base.

GENTL[38] aims to act as a bridge between different cloud modeling languages. Their purpose is to be generic enough to translate most cloud modeling languages to it and back. The GENTL environment provides conceptual mapping and also includes a topology builder. It also includes a cost model for calculating operational expenses. The current purpose is to optimize the infrastructure for cost and then map back to the original language. Being this generic could mean that is would be easily extensible for heterogeneous devices. Although their lack of support for practical tools make it unpractical to use at the moment. *Ext.* Since the GENTL language is so abstract, it should be relatively easy to extend for heterogeneous infrastructure. The problem with this would be the translation to the actual infrastructure, since this would have to be implemented by the language where the GENTL description is translated into. Requiring double the work, since you have to extend both languages.

CAMEL[36] is based on TOSCA but strives to combine the design and deployment with operating and adjusting. Their framework will support changes in the infrastructure during run-time and will automatically update the desired entities. The language has its own DSL and is not based on the UML language. It also support non-functional constraints like, maximum latency. The language comes with an execution environment which makes it practical. It has been chosen as the language for the Melodic[6] project, which is an EU funded research project. Sadly the language only support IaaS. *Ext.* The current CAMEL implementation does not really allow for easy extension, requiring the developer to rewrite the code for the language. The language is really aimed at cloud systems.

Eirik Brandtzaeg *et al.* propose the **CloudML** language [39], which is purposed for modeling cloud application and automating provisioning and configuration. They also support run-time adaptation of the infrastructure and comes with a provisioning engine. The engine is extensible and support for multiple cloud providers can be added. The engine uses metadata that is gathered during run-time to make provisioning decisions. *Ext.* The language can be extended to support heterogeneous devices but this would require the change of the language code base.

Dinh Khoa Nguyen *et al.* proposed **Blueprint**[40] as the common foundation for cloud modeling across different service levels. The language is for describing and not provisioning or configuring. The language uses the concept of a blueprint which is an abstract description of a service that a cloud provider could offer. This makes the language extensible for heterogeneous infrastructures as a service provider could easily add support for fog nodes. The language also supports adding policies to their model, which can be used for setting quality of service constraints. Their is support for mapping to GENTL and TOSCA but the tooling for the language itself is limited. *Ext.* Theoretically the language is extensible, but the specification of the extension is in cloud services. Using the extensions to describe non-services not in the cloud would completely change the scope of the language.

OASIS has chosen **TOSCA** [42] to be the standard modeling language for cloud applications. With TOSCA we are able to describe the topology of the application and how it relates internally. It also enables us to have event driven configuration using the management plans. TOSCA would be the obvious choice to model our application. Since the language is a standard, seems to suit our needs and has good amount of tools supporting it, we will continue with this language. *Ext.* The language supports extensions natively with a sort of templating form. Adding support for heterogeneous devices would be relatively simple.

[3] https://docs.microsoft.com/en-us/azure/azure-resource-manager/
[4] https://aws.amazon.com/cloudformation/
[5] https://cloud.google.com/deployment-manager/
[6] http://melodic.cloud

Language	Service	Ext.	Het.	QoS	Aut.	Syntax	RT	Purpose
GENTL	XaaS	++	0	+	- -	0,Graphical	- -	Bridging cloud modeling languages, optimizing configurations
TOSCA	XaaS	++	+	+	+	++,Both	-	Modeling, provisioning & configuring infrastructure for cloud applications
Blueprint	XaaS	++	0	+	- -	++, Both	- -	Describing deployment configurations
CAMEL	IaaS	-	0	0	0	0, Textual	++	Modeling infrastructure for cloud applications with run-time support
CloudML	XaaS	+	0	0	0	++, Both	++	Run time provisioning of multi-cloud
CAML	XaaS	+	0	0	0	++, Both	- -	Implement cloud modeling support for UML
Terraform	IaaS	- -	- -	-	++	0, Textual	- -	Cloud agnostic infrastructure orchestration
HOT, ARM, CloudFormation	IaaS	- -	- -	-	++	0, Textual	- -	Provider specific infrastructure orchestration

Table 4.1: Infrastructure description languages {- -,...,++}

4.2.3 Gap Analysis

- While some of the languages do support the specification of QoS constraints, they do not allow the specification of the application workload. Specification is mostly linked directly to the compute instance where the developer has to request certain hardware, like cores and RAM. Allowing the specification on application level would allow the framework to make decisions about placement and orchestration itself. The open problem with this would of course be that the estimation of workload is still somewhat of an open problem in the research community, making it hard for developers to specify the workload.
- Some of the languages allow the specification of run-time QoS constraints, but the implementation of this is not complete. They also do not specify how the information is gathered and how the decision is made based on this information.
- The language descriptions do not leave much room for interpretations for the tools implementing the language. Especially in cloud systems, some ambiguity in the description could let tools make their own decisions. Tools could make these choices based on run-time information or on workload estimations.
- Since all the languages are purposed towards cloud systems, they assume compute instances are mostly general purpose devices. Some languages are easily extensible for describing non-general purpose devices. But they lack tool support for these devices.

4.2.4 TOSCA

TOSCA is an effort by the community to standardize the modeling language for deploying cloud applications. The TOSCA language describes the infrastructure/topology, the components, the relationships between them and the processes of composite cloud applications. A good and more in depth explanation of the TOSCA language is given by Tobias Binz *et al.* [42, 43]. To represent the language a YAML representation maintained by OASIS is used [44]. The language is divided into two parts, the topology description and the management plans. In figure 4.1 we show the definition of a TOSCA serive template.

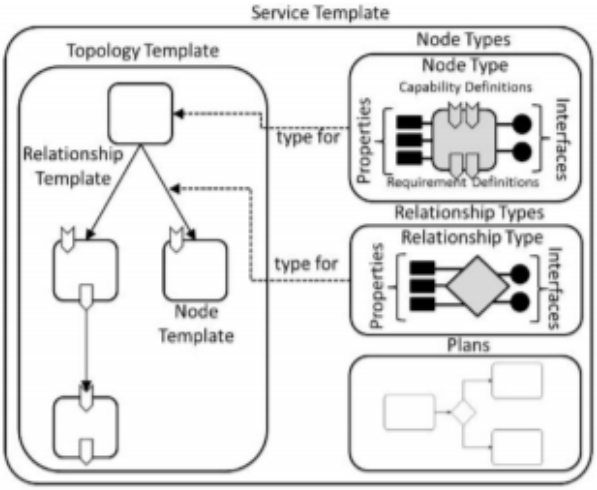

Figure 4.1: TOSCA description [42]

Topology Description defines the infrastructure of the application. It represent the infrastructure by using nodes and relations, which types are unspecified. For example, nodes can be hardware like machines but also software like applications and more. And connections can be connections, like database connections, or dependencies, like libraries, etc. The flexibility of the language allows us to not be limited by pre-defined nodes. We would have no problems with modeling newer technologies like containers.

Management Plans resemble a series of actions that should be taken for nodes. They can be triggered on certain moments in a node's lifetime, like configuring dependencies on creation of a virtual machine. The management plans are mostly comparable with configuration management tools like Ansible [45]. It should be noted that these plans are imperative in contrary to the topology being declarative.

Advantages

TOSCA is able to represent the structure of the application. Not only does it describe the (cyber)physical infrastructure such as virtual machines, databases and connections. It is also able to describe which applications run where and what their underlying requirements are, both software and hardware. As we can see in the simple example in figure 4.2. A database application requires a specific database manager, some configuration properties and an open endpoint. The database manager is hosted in a container, which in turn runs on a virtual machine, where we could even specify the hardware requirements.

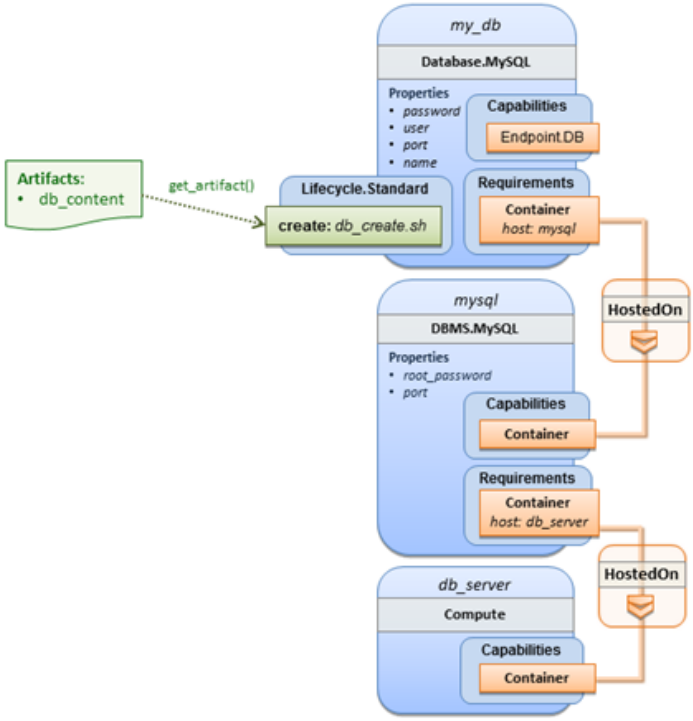

Figure 4.2: TOSCA topology example [44]

For configuration purposes TOSCA allows adding artifacts to nodes. Artifacts are simply files that are executed once a certain life-cycle event triggers. This could be after the node is created or once it has been destroyed or other. There is no specification on what the files can be, and can be implemented in any programming language. The execution environment of the artifacts can be defined by the developer. The environment can be an operating system, like Linux, or a software environment, like python or Ansible.

TOSCA allows the developer to build custom types for nodes and relations. Since the node and relation type is unspecified, a large scale of devices, software or relations could be represented. The language is not directly limited to general purpose computing, e.g. for embedded systems SPI connections could also be added as relations. The original purpose of the language is for cloud orchestration but the extensible nature of the language allows describing applications across more platforms.

Node and relation types allow the developer to easily reuse the building blocks. For example, once the developer described an Ubuntu node on one VM, he can easily reuse the node to describe the operating system on another VM. The description of these nodes can easily be shared across projects. A platform could even be created in the public domain where TOSCA node description could be shared between developers.

Compared to other description languages, TOSCA comes with a large set of tools. The OpenTOSCA[7] project provides multiple tools, like winery which allows graphical modeling and interaction with the language. OpenTOSCA container which provides a TOSCA run time. And more like:TOSCA UI, Vinothek and TOSCA planning engine. Companies like Cloudify have built orchestration and deployment tools built on top of the TOSCA language. The research community also published tools making use of the language. For example, DRIP [46] which automates the orchestration across the multi-cloud platform.

TOSCA uses a declarative representation for their topology in combination with an imperative representation for their build plans. For orchestration and provisioning purposes a declarative representation makes more sense since it defines the end state of the infrastructure. An imperative representation would result in a list of entities being declared, where the relation to each other is not as clear. When updating the infrastructure, the orchestration engine could compare the desired end state with the current state and just update the affected infrastructure. The declarative nature of the language allows for easy live adaptation and reconfiguration.

Tosca implements policies, which are non-functional constraints of entities. Policies implement cus-

[7] https://www.opentosca.org/

tomization over placement/scaling, quality of service and access control. In the current description the policies do not seem to be implemented completely. The Cloudify implementation does offer support for polices, both pre-defined and custom policies using Riemann engine [8] in combination with Diamond [9]

Disadvantages

TOSCA gives the developers a lot of freedom. While this freedom allows the developers to describe a large scala of applications, it also requires the framework to support a large scala of platforms. This requires the developers, when they are designing the topology, to also build the implementations for the specific platforms. When a block defines the Linux operating system, all the logic for instantiating this on the VM should be created.

The artifacts are not by definition heterogeneous, they require a run-time. If the artifacts is a python script it requires the nodes to have installed this environment. Even a bash script requires a linux machine and will not work on windows. A problem with the artifacts is also internal dependencies. If the script requires some software or hardware that is not available, this problem is only clear during run-time.

The downside of the artifacts is that configuration logic is hidden in these files. When analyzing the topology, the contents of the artifacts are not directly visible, unlike the dependencies that are represented as nodes. For instance, in the example we could remove the database manager node and say that the database is hosted on the VM, moving the installation and configuration of the database manager to the configuration artifact of the database.

The TOSCA standard defined by OASIS does not fully describe the policy system and how to implement it. For instance, if you want to create custom policies, how is this linked to scripts or where does the data it acts on come from.

The language does not enable application to be defined ambiguously. If an application is able to be ran on both Linux and Windows machines, there is no way of defining a forked *hostedOn* property.

According to the CAMEL paper [36], TOSCA does not provide an instance model and because of this does not support models at run-time. Not allowing the model to be self adaptive during run-time.

Relation types do allow the binding of properties. This could be used to define authentication or connection settings. For instance the property could say that an SSL certificate is required. Or it could define the frequency of a wireless data protocol. The relation type lacks the ability to define quality of service constraints. E.g. defining that the connection can only be routed in the Netherlands, or that the connection can only have a maximum of 10ms delay.

Implementation To deploy TOSCA descriptions, the OpenTOSCA team has created a build plan generator (figure 4.3). The topology will get translated to a set of abstract actions required to create the infrastructure. It will then add artifacts to the created skeleton and after this will attach the platform specific code to the build plan. The output is a plan that is able to be executed such that the infrastructure at the specific platform is created.

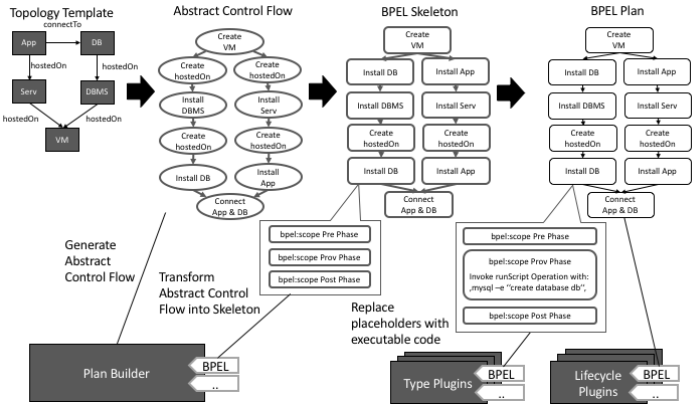

Figure 4.3: TOSCA plan builder [10]

[8] http://riemann.io/
[9] https://github.com/python-diamond/Diamond
[10] http://opentosca.github.io/container/PlanBuilder.html

4.3 Extensions

To enable modeling across heterogeneous infrastructures, we propose some extensions on the TOSCA language. These extensions should enable a developer to describe their composite applications.

Kritikos *et al.* [47] Created an extension on the CAMEL language to support serverless devices. They show us what are some things to take into consideration when extending a description language. The extension proposed is purposed towards SaaS, which is just more abstraction in the cloud. In comparison, we try to include devices from paradigms other than cloud. Their method for verifying their extension could also be usefull while verifying our framework.

4.3.1 Ambiguous Hosts

During orchestration the tool is not able to decide where to place components. For instance, in TOSCA you can specify where a component is hosted on, but the component can then only be hosted on that type of node. If there was a possibility to specify that a component can be hosted on multiple components (figure 4.4) and attach platform specific artifacts for configuration, the provisioner can make decisions on where to deploy components. Triggering different artifacts for different platforms.

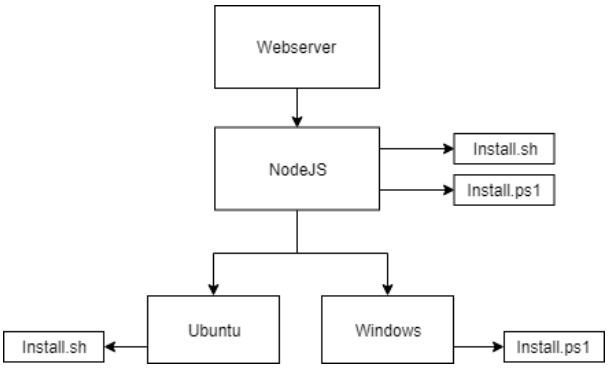

Figure 4.4: Ambiguous Host Example

An existing network of fog nodes could be running different operating systems that are already provisioned. With the extension, the existing infrastructure could be utilized by only configuring parts of the topology. Instead of creating the VM, configuring it, and creating the application. We now just have to configure the machine and create the application.

TOSCA supports scale-out plans, of which the purpose is to replace a single node. E.g. if the application has to be replaced, it will create a plan to update that node and use the relations, like *HostedOn*, to update related nodes. This functionality could be utilized to push applications to existing machines.

Current tools, e.g. TOSCA-parser[11], only support single hosts. Meaning the *HostedOn* relation can only be a single-to-sinlge relationhip instead of a single-to-multiple. The attribute could support an array of possible hosts, instead of the unary one. In listing 4.1 we've created a simplified TOSCA example of the proposed extension.

```
tosca.nodes.WebServer:
  derived_from: tosca.nodes.SoftwareComponent
  capabilities:
    - hosts:
      - host: linux_server
      - host: windows_server
    - interfaces:
      - linux_server:
        create: installServer.sh
      - windows_server
        create: installServer.ps1
windows_server:
  type: tosca.nodes.Compute
  capabilities:
    - host:
```

[11] https://github.com/openstack/tosca-parser

```
      properties:
        num_cpus: 2
        mem_size: 4096 MB
    - os:
      properties:
        type: windows
        distribution: Windows_10
    - interfaces:
      Standard:
        create: install.ps1

linux_server:
  type: tosca.nodes.Compute
  capabilities:
    - host:
      properties:
        num_cpus: 1
        mem_size: 2048 MB
    - os:
      properties:
        type: Linux
        distribution: Ubuntu
    - interfaces:
      Standard:
        create: install.sh
```

Listing 4.1: TOSCA ambigous host

SummaryWe have proposed an extension to the TOSCA language such that the orchestration, configuration and deployment tools could make their own decisions on where to place certain components of the application. Also a developer could now deploy his applications while not having to know the exact infrastructure his application runs on. If existing infrastructure is provided he can now deploy his application on top of that.

4.3.2 QoS Specification

TOSCA theoretically supports defining QoS constraints, it is defined under the policy section (chapter 12) of the simple profile [44]. The specification for this is not implemented yet and lacks a good explanation.

QoS constraints have two objectives, 1) choosing the right infrastructure during initial orchestration and 2) ensure the infrastructure adheres to QoS constraints during run-time of the application. Examples of applications that could benefit of QoS constraints are: geographical location affinity, optimizing cost, minimizing latency and clustering comparable applications

Policies are able to define certain QoS constraints. The TOSCA implementation itself does not offer an actual implementation, but Cloudify does (example project with QoS policy [12]). The TOSCA specification does not mention how the policies should be practically implemented. So gathering the information for the policies and acting on it is unspecified. Policies are attached to certain nodes by adding the node to the member property of a group, and adding the policy to the group.

Side note. We do not agree with the way that the current TOSCA language links policies to nodes. Currently, if you want to attach a policy to a node, you have to go to the policy and add the node to an array of attached nodes. The method we would prefer is that you attach policies to nodes in the definition of the node itself. In listing 4.2 we have created an object oriented example to illustrate the problem.

```
# current implementation
group.nodes = [node_1, node_2, ...]
group.policy = policy_1

# Preferred implementation
node_1.policy = policy_1                    # single policy or,
node_1.policies = [policy_1, policy_2, ...]   # multiple policies
```

Listing 4.2: TOSCA policy definition

Rui Han *et al.* [48] proposed an extension on the TOSCA language called Elastic-TOSCA. The extension implements support for managing elastic applications. Their interpretation is mainly dynamically

[12] https://github.com/cloudify-examples/nodecellar-auto-scale-auto-heal-blueprint

scaling instances. To make the scaling possible, they define a *constraintstemplate* where limits can be defined for certain QoS constraints. While they do mention the scaling constraint extension, they do not mention how this information is gathered and processed. Constraints that are used seem static, meaning they have to be pre-defined in the language. If we want to create a new type of constraint, like maximum users connected, it is not explained how to extend the language for it. Theoretically the extension does support run-time QoS constraints, accompanied by an example of the monitoring needed for this.

Static properties can be used while making placement decisions for certain nodes. A QoS cost property could be added to a node in the TOSCA description. Using metadata that is distributed by the service level provider, the application could choose where to place the node, while adhering to the QoS property.

Dynamic or run-time properties are not really used while initially provisioning the infrastructure. These are more appropriate for scaling nodes within the infrastructure or updating the infrastructure if QoS constraints are not met. E.g. you could define a maximum number of connected users, if this is exceeded the infrastructure has to be scaled up. How the scaling is handled depends on the framework.

Custom constraints also require some custom actions.

- Data has to be gathered to make these decisions on. This could either be data from the service provider, it could be fetched every time a decision is made, or it could be data from the infrastructure.
- Data has to be compared, and how it should be compared should be configurable by the developer. The outcome of this comparison should be available and readable by the framework that is orchestrating the application.
- An action has to be defined that the framework should execute once the policy is broken. E.g. if the CPU utilization breaks the lower threshold the framework should scale down, and if it breaks the upper threshold it should scale up.

```
qos_policy_latency_cpu:
  type: tosca.policy.QoS
  description: My QoS policy for maximum latency and cpu utilization
  properties:
    cpu_util_max: 60
    latency_ms_max: 3

latency_ms_max:
  type: tosca.policy.QoS.property
  description: Create a new QoS property for maximum latency
  action: "python ~/get_latency.py"

webserver:
  derived_from: tosca.nodes.SoftwareComponent
  # ..... Removed for brevity
  policies: [qos_policy_latency_cpu]
```

Listing 4.3: QoS policy run-time

Cloudify offers an implementation in combination with some tools to offer dynamic QoS constraints. They run a daemon on all of the machines that constantly gathers metadata about the machine, like CPU utilization. The daemon sends this information to a central point where all the metadata is used to make scaling decisions based on the policy. This method is really similar to one of the scaling methods we mentioned in this document.

The problem is letting the user define what data has to be collected at the target system. A method where the user is able to provide custom logic with pre-defined outputs could solve this problem. But it would also require the user to provide custom logic in the entity that makes the decision. This would mean that in order to run this custom logic at both sides, extra configuration may be required. The process of adding this custom logic should be defined in both the daemon and the decision maker, such that the logic can be easily injected by the developer.

The language currently has no support for attaching QoS constraints or policies to relations. A connection could have properties that are not really part of the nodes it connects. Examples of QoS constraints for relations could be: percentage of packets dropped or maximum latency. Monitoring and making scaling decisions based on the connection could enhance the control the developer has over his application.

Since the connection is not really a piece of software or hardware, we are unable to directly monitor this. The monitoring of the connection still has to be abstracted to one or both of the connected nodes. Meaning the same result could be achieved with the current language.

Summary In this section we defined how developers can define their quality of service constraints in the description of their composite application. We proposed how policies can be used to maintain the quality of service, and what adjustments should be made to the policies description. The extension is discussed, and what aspects should be considered while implementing it.

4.3.3 Workload Specification

If the developer does not want to bother with the exact specification of the machine, but does know the requirements of his software, he should not have to specify all the hardware in his description. This would allow the framework to make its own decisions on placement of software or requesting resources. The framework should check the requirements of all the software that runs on a piece of hardware, and be able to determine itself the requirements of the hardware. We propose to extend the TOSCA node types related to software components with workload specifications.

The extension has the same problems as workload estimation has. It is very hard to know the exact workload a piece of software will have. We will talk more about workload estimation in chapter 6.

4.3.4 Modeling IoT

Fei Li *et al.* [13] proposed an extension on the TOSCA language that adds support for IoT devices. This is not an extension we propose, but we would like to add it to the extensions that should be implemented for the framework. They implement extra properties like binary artifacts and the ability to add a compiler. This allows us to describe software for the fog and edge more easily, by including build tools for the application specific platform in the description.

4.3.5 Modeling Logic

To describe the actions in the system we have found two modeling techniques that are interesting. Sequence diagrams and activity diagrams. Both are part of UML [49] which means they are universally recognized and standardized. Sequence diagrams enable us to model the interactions in the systems. How actions and data will flow between different parts of our architecture. In our case the sequence diagram is ideal to describe the abstract workings of our system.

Activity diagrams are great for modeling actions in a single system, but they lack the ability to see where the actions takes place in the system. It essentially models a monolith with the flow from action to action. Making the Activity diagram not an optimal solution to model our framework. However it could be useful when modeling actions in the core of our system.

Modeling the logic is not within the scope of the composite application description language. Because of that we are not currently going to implement it as an extension.

4.4 Summary

We were able to find multiple modeling or description languages for composite applications. After comparing the different languages we were able to choose one that best fits our requirements. All of the languages had some gaps in relation to our requirements. To be able to fully support our implementation we had to propose some extensions on the TOSCA language.

We have proposed an extension on the language that allow for better control of the run-time environment. The quality of service description, allows the developers to describe run-time constraints both on individual parts of the composite application and the relations between them.

The ambiguous host extension allows our automation tooling to make smart decisions on the placement of parts of the composite application. Allowing the framework to find the best suiting infrastructure for it.

We propose the ability to describe workloads of software nodes in the topology. This allows the framework to match the hardware to the software that is running on top of it. This allows the framework to correct under-specified or over-specified hardware. The extension is not formalized yet, and the grammar currently undefined.

By merging our extension with the extension proposed by Fei Li [13], we would be able to support embedded devices, allowing for a more heterogeneous infrastructure.

Chapter 5

Automation

In this chapter we discuss how we will automate the orchestration, configuration and deployment of distributed application across the heterogeneous infrastructure.

We will discuss why there is a need to automate the process, and what the requirements are for the automation. After this we will compare some of the latest technologies in both industry and scientific and talk about some of the gaps. Finally, we talk about how we are solving the proposed problems.

5.1 Introduction

During the previous chapter we explored how a developer is able to describe their distributed applications over the Cloud, Fog and Edge. The description is required to automate the orchestration, configuration and deployment of the application, since it provides a formal definition of the desired state of the system. Based on the description we are able to orchestrate the infrastructure across Cloud, Fog and Edge; configure the infrastructure with the correct software running on top of it; and maintain quality of service of the system during run-time.

Automating the process has several advantages. Some of which are:

- Automation makes the process really easy to *scale*. When all actions have to be executed manually the process will scale linearly. Once a script is written, it is relatively easy to add extra resources to it. The process requires more effort initially but this will pay off the more times the process is ran.
- Scripting or automating orchestration, configuration and deployment make the process *reproducible*. Every infrastructure that is created should be the same if the description is the same. This also allows for easy automated testing of the infrastructure.
- The knowledge of the orchestration, configuration and deployment should not be in a human's/employee's head. By scripting the process the knowledge can be *portable*. This allows the knowledge to be saved and easy to update (iterate on). Also this allows the knowledge to be easily transferable.
- *Complexity* should be *abstracted* away from a developer such that description only describes relevant parts of the system. This would also allow a developer to easily interpret the described infrastructure.
- Automation reduces *complexity* during the *life-cycle* of software products. It facilitates *collaboration* among developers, operators, system administrators, etc. and improves efficiency.

5.1.1 Requirements for Automation

To better understand the open problems for automation we propose some requirements we need to implement while choosing automation tools.

1. Orchestration must be automated across distributed virtual infrastructure, in particular when the infrastructure consists of heterogeneous resources like Cloud Fog and Edge devices.
2. Automated orchestration should consider QoS constraints and contextual information while selecting resources.
3. Configuration of the orchestrated infrastructure and existing devices must be automated across devices in the Cloud, Fog and Edge.

4. Applications must be automatically deployed to the heterogeneous infrastructure, both during initial deployment and during run-time.
5. MYST should be able to update software components on the heterogeneous infrastructure during run-time.
6. MYST must be able to maintain quality of experience during run-time across Cloud, Fog and Edge, using quality of service constraints described by the developer.

5.1.2 Challenges

We have recognized some challenges when deploying to heterogeneous infrastructures. The major ones that we will address are:

Communication Different devices support different types of communication. Devices in the edge or in the fog or much less probable to have an internet connection in comparison to general purpose or cloud machines. Devices could require deployment via 4g connections instead of via SSH for example. Also devices in the edge and the fog are more likely to not always be available, only connecting on a certain part of day.

Difference in hardware Not all machines have the same hardware. Different machines could require different libraries or different libraries for correctly configuring the machines. Also machines could miss hardware that could be required like a video card or a floating point processor. The performance between machines could also differ greatly with different processors.

Tool support Tools often only support specific platforms. For example, many tools only support Unix based platforms for automated deployment, making them unusable for a windows based system.

5.2 State of the Art

In this section we will discuss the latest technologies that are used to orchestrate, configure and deploy composite application. We will compare technologies from both industry and the scientific community.

5.2.1 Industry

In the industry, more particular, in the DevOps community there are a wide set of tools available mostly performing specific goals. We are looking for orchestration, configuration and deployment tooling. There are many tools available which enables a developer to automate his infrastructure orchestration.

Cloud providers use a cloud stack as a sort of operating system for their entire cloud. All of the cloud stacks provide tooling that lets a developer write their infrastructure as code. For example, Azure offers Azure Resource Manager(ARM)[23], Amazon uses Cloudformation[22], Google uses Google Deployment Manager[50] and Openstack uses Heat[5].

These languages are not platform agnostic. They are developed for the specific cloud stack and do not work on other stacks. Also, depending on the provider the language is not always open source. When a developer builds his infrastructure for one stack, it is hard to switch between providers. But since these are offered by the specific provider, they always support the generation of the actual infrastructure.

Non-cloud specific tools often are open source tools which support multiple cloud providers and cloud stacks. DevOps is still a relatively young paradigm and the most popular tools are constantly changing. For building the infrastructure, the tool that is most often used is called Terraform[4]. According to some developers on a large DevOps community forum, there are currently no non-stack specific good alternatives to Terraform[1]. A developer is able to script their infrastructure and deploy this to a large number of cloud stacks using the accompanied tooling.

Terraform[4] is a cloud agnostic infrastructure provisioner, and support a large list of providers[2]. But it should be noted that a Terraform solution for one provider does not work for another provider. Instead of declaring a VM with certain constraints you still declare a specific type of VM for the specific provider, making it not fully provider agnostic.

Most of these languages are only meant for orchestrating the infrastructure. But provide limited support for dependency installation and management. They do not model the application that is supposed to run on the infrastructure. Deploying the application to the infrastructure is supposed to be done by other tools, like Amazon CodeDeploy [3] or Ansible [45].

[1] https://www.reddit.com/r/devops/comments/934jib/best_alternative_to_terraform/
[2] https://www.terraform.io/docs/providers/
[3] https://aws.amazon.com/codedeploy

When the infrastructure is created we still need to tools to configure it. The community has a number of favourites for configuration management. Ansible offers an agent-less type of configuration. Meaning there is no need to install tools on the target machine to be able to configure them. Other tools require an agent to run on the to be configured machine, meaning you need to configure them to be able to configure them. CFEngine[4], Puppet[5], Salt[6] and Chef[7] are more for managing running machines. While Ansible is purposes more toward the initial configuration. But the flexibility of these tools also allow us to do initial configuration.

Since fog instances are normally not managed by cloud providers, it would be helpful if some of these tools do not require the cloud infrastructure. Although some of these tools have been built for the cloud, there is support for on premise and other devices. For instance CFEngine, Chef and Ansible can easily be used to configure non-cloud instances. But setting up the infrastructure is in this case mostly manual labour.

Unlike these languages TOSCA models the infrastructure, application structure and application dependencies, while these languages mostly describe the infrastructure or their configuration, without linking it to the application. Although these scripts are used for the orchestration of the infrastructure and the configuration of it, they cannot be seen as models.

5.2.2 Research

Tobias Binz *et al.* [2] proposed a run-time environment for TOSCA-based cloud applications. They support translating the TOSCA description to into management plans and combine this with plugins to create the implementation. Initially the topology gets translated to an abstract control flow graph, containing the abstract instructions for realizing the infrastructure. This gets combined with the artifacts to create a more specific set of instructions. After this plugins define how each instruction is executed for the specific platform. These plugins could be very useful for heterogeneous devices, since the plugin could define a custom communication protocol with custom deployment for instance. Using scale-out plans the environment is already able to provision specific parts of the infrastructure without changing the rest. The code is publicly available on [8].

Spiros Koulouzis *et al.* [3] proposed an infrastructure optimization suite that is able to build optimized cloud infrastructure. Their framework, Dynamic Real-Time Infrastructure Planner (DRIP), implements planning, orchestration, provisioning, deployment and run-time control of the infrastructure. It uses a TOSCA topology, provided by the developer, to identify application components. It combines time critical constraints from the description with information from the service provider to plan an optimized infrastructure. The framework is able to make use of current DevOps tools like Ansible to be as extensible as possible. It would be nice if this could be extended such that custom tools can be used for heterogeneous devices. The framework uses agents to support run-time scaling, but this is mainly for containerized scaling. The code is publicly available on [9].

5.2.3 Gaps

All of the discussed industry solutions only support general purpose computers. Depending on the heterogeneity of the infrastructure this could be good enough. If the edge and fog devices run supported operating systems and support the connection methods, this is no problem. They are however unable to provision and deploy too other types of devices. Because the OpenTOSCA environment supports plugins for the build plan, custom deployment methods could be implemented. This would allow very specialised methods, like sensing devices would probably require. Depending on if DRIP supports custom deploy agents, extending the solution to other types of devices would be relatively easy. This would require the developer to write a custom deployment agent to support their application, which is comparable to writing a plug in.

The basis of live reconfiguration would be the declarative description of the infrastructure. This way the tool we would use is able see what changed between the current and desired state and will only update the necessary parts of the infrastructure. Some orchestration tools, like Terraform and CloudFormation, support reconfiguration of the infrastructure by keeping a state file. This allows them to compare the

[4] https://cfengine.com/
[5] https://puppet.com/
[6] https://www.saltstack.com/
[7] https://www.chef.io/products/chef-infra/
[8] https://github.com/OpenTOSCA
[9] https://github.com/QCAPI-DRIP/

previously created infrastructure with the newly desired one. As they are not responsible for configuring the infrastructure, they will only be responsible for the orchestration part of the reconfiguration. Configuration tools support live reconfiguration partly, by creating a set of instructions that installs the new application. Since these tools are imperative, the developer has to clean up the machine before he is able to install the new application on it, manually or scripted. This would require the developer to have knowledge of the old application to configure the new one. This could be solved by completely reinstalling the machine, but this would probably lead to data loss. OpenTOSCA's support for scale-out plans support partially overwriting the infrastructure. Only replacing the TOSCA nodes that changed in the design. The declarative design allows the comparison of the end state. Since the old state is saved, it is relatively easy to clean up the old application. Some tools inherently support live reconfiguration by creating containerized application. Examples of these tools are Kubernetes and Docker. Since these containerized applications contain their own configuration, the host machine requires almost no configuration to run them. Meaning there is almost no reconfiguration required to run these applications. The drawback is that to run these applications a specialised hypervisor is required, and not all machines are able to run this.

The tools from industry have no support for custom decision making on what instances to request and where the placement should be. Developers are able to specify certain hard requirements for the instances, like in what zone to place and the number of CPU cores, but no quality of service constraints. The only orchestration tools that have implemented this are virtualized orchestration tools, like Kubernetes. But these do not really create machines but orchestrate application on top of the machines. DRIP supports an infrastructure planner which uses constraints to plan the infrastructure passing it to the provisioner. In the OpenTOSCA plan builder documentation we could not find a hook for the policies. Meaning there is probably no support for using the QoS constraints while creating the orchestration plan.

All of the tools that we have described miss a mechanism that describes the workload or required resources of application components. If the workload is not described, the frameworks are unable to estimate the requirements of a machine that would run the application. Estimating the workload of an application would allow the framework to make its own orchestration decisions without wasting unused resources.

5.3 Translation

Parsing TOSCA

The textual description has to be translated into a model that is usable by a computer, such that it can be orchestrated. A number of TOSCA parsers are available for exactly this purpose. After parsing the description all entities have to be linked together correctly. Ideally the model should be translated to an abstract syntax tree or an parse tree. Such a representation will make the topology easy to traverse and manipulate it. The topology that is created by the parser could then be used by the orchestrator to execute the necessary steps for creating the infrastructure.

Building Internal Representation

An internal representation allows us to optimize the use of the description throughout our framework. Instead of constantly parsing and adding data to the description, we parse it one and then add data to the internal objects. This allows a uniform method of communication within the framework.

The internal representation is comparable to the TOSCA topology. It contains the topology with references to the connected nodes. We propose an internal representation that contains multiple tree structures for the individual parts of the application. This is comparable to the *HostedOn* property in TOSCA. An object oriented structure would allow us to easily update or add data to the nodes in the tree structures. It also allows is to easily connect them by reference, and inject new object in between.

Compare with Existing Infrastructure

To allow reconfiguration and scaling we should be aware of the current state of the infrastructure. The current state of the infrastructure should be compared with the newly desired infrastructure. This can be provided by the developer, or the changes are triggered from the monitoring part of the framework.

In the field of software evolution there is the concept of clone detection. Otherwise describes as, the amount of duplicate code in a project. One method of finding clones makes use of abstract syntax trees

[51]. A comparable method could be used to detect if a tree in the topology has changed or not. And if so what the changes are.

When the differences between the old state and the new description are known, the framework should take the new changes as leading. So if something is removed in the new description, the framework should tag that the part should be deleted and what changes in related parts.

Current Translation Tools

The OpenTOSCA development group from the University of Stuttgart, the same group that proposed the TOSCA standard[42], is building a framework that contains a set of tools to work with TOSCA. This framework includes a TOSCA parser both for XML and YAML. One of the things they create is the topology in memory, which is used in the rest of the framework. They create a syntax tree representation of the model in memory and use this in the rest of their framework.

OpenTOSCA implements a Plan builder[10] which is responsible for generating an abstract high level control flow. Next to creating a build plan, it also generates a termination plan and a scale out plan. The control flow is just a set of steps required to implement the orchestration, but does not offer the actual orchestration itself. The control flow that is generated can be combined with a set of plug-ins to execute the implementation steps with the service level providers.

Cloudify implements a framework which was later made open-source called ARIATOSCA[11]. The framework is a TOSCA orchestration engine which also implements a parser. The parser translates TOSCA to a deployment graph. The deployment graph describes the required orchestration steps without the actual implementation. To execute the orchestration, some plug-ins are provided. ARIATOSCA itself does not supply the plug-ins but Cloudify has a set of plug-ins which they also published on Github.

Openstack has created a TOSCA parser library[12] for python that is also open source. The library translates TOSCA files to an in memory graph of the model, only supporting the YAML representation. It is meant to be used as a library in your own programs. It does not offer any orchestration or plug-ins. If we would create or own orchestration this would be the best solution since it can be easily used in our own application, while not requiring us to write our own parser. It will be extensible so that we can attach our own orchestration logic.

The same developer from ARIATOSCA started a new project called Puccini[13] which is not backed by Cloudify, has a different architecture and different goals.They also offer an independent parser which can be standalone from the rest of the framework. In comparison to ARIA, Puccini offers a new compiler and a new intermediate representation with CLOUT. They already provide tools to build and deploy containerized applications.

5.4 Orchestration

For orchestration we define some steps in figure 5.1. These do not take into account other parts of automation. This is just to illustrate how the description is translated into a running application.

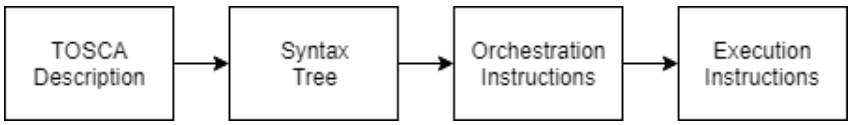

Figure 5.1: Orchestration steps

Orchestration Models

In Orchestration and configuration there are two methods for describing and realizing infrastructure[52], declarative and imperative. In short we could describe the methods as *WHAT* versus *HOW*.

Declarative orchestration requires the developer to define an end state of the required infrastructure. He will not specify how it should be created, the orchestration tool should do this. Since the developer has no direct control over how the infrastructure is created this could be harder to debug. It also allows the

[10] http://opentosca.github.io/container/PlanBuilder.html
[11] https://ariatosca.incubator.apache.org/about/
[12] https://wiki.openstack.org/wiki/TOSCA-Parser
[13] https://github.com/tliron/puccini

tool to save the current state of the infrastructure, updating the already present infrastructure without unnecessary creation and deletion. For example, if you write a script requesting 5 virtual machines and there are already 3 present, you will only create 2 virtual machines when running the script.

Imperative orchestration means that a developer defines the steps the orchestrator has to take to build the correct infrastructure. The developer specifies a certain infrastructure and when the script is run the entire infrastructure will be created. Because the specified actions will sequentially executed the result and side effect are predictable. The downside is that it could create unnecessary infrastructure that was already present. For example, if you define a script requesting 5 virtual machines, every time the script is run your will create 5 new virtual machines.

Both have their advantages and disadvantages. We like the declarative model more for orchestration since it will reuse existing infrastructure to create an end state. If we relate both paradigms to the TOSCA model we see that the topology is declarative but the build plans and configuration are imperative.

Description to Orchestration

To be able to translate an application description to a working application, the infrastructure that is running the application has to be created first. The infrastructure can be derived from the topology in the TOSCA description that is provided.

The most basic infrastructure would be an application that is hosted in a virtual machine with specified properties. The TOSCA language contains a relation called *HostedOn*, we just follow this chain down to the virtual machine. If an application is running in the python environment that is running on a Linux machine in the cloud with 2 cores and 8GB RAM, we know exactly what to request from the service provider. If the TOSCA description includes all nodes with the machines that the application is hosted on, and they are easy to distinct from other nodes, the algorithm (listing 5.1) for orchestrating the vm's in this case is relatively simple, recursively going down from application to machine. It does require extra configuration for connecting them and defining security protocols.

```
def main(nodes):
    for node in nodes:
        host = getHost(node)
        if not host.isCreated:
            SLA.requestVM(host)
            host.isCreated = True

def getHost(node):
    # parent is defined by the HostedOn relation
    if node.parent is None:
        return node
    else:
        return getHost(node.parent)
```

Listing 5.1: Orchestratrion Example 1

Next to creating the virtual machines, the infrastructure also has to be created. For example, cloud providers provide security groups, dedicated data pipelines and custom port configurations. This configuration is normally done in hardware, but service provider's software defined networking allowing for programmable hardware. In TOSCA these are mostly described by relations. In some cases the relations just require extra configuration on a node and in other cases it requires the orchestrator to create extra entities.

New technologies provide new levels of abstraction. Containers can be deployed directly in the cloud without creating virtual machines. And simple functions can be ran with serverless functions, circumventing any infrastructure. This makes it hard to determine what actually has to be created during the orchestration phase. It would require nodes types to be accompanied with by the instructions on how to create them. Extensions for these new paradigms are proposed. E.g. Kritikos *et al.* [47] have proposed an extension to the CAMEL language for serverless computing.

Some nodes in the described application could be already provided and will not require orchestration. IoT devices are mostly already present and may only require some configuration. When translating the description to a set of entities that have to be created, the distinction has to be made between these nodes.

We could separate our set of nodes into 3 different categories: virtual hardware, hardware and software. Of these three categories only the virtual hardware has to be orchestrated. Meaning only this category needs orchestration instructions requesting resources at service providers. New technologies could introduce new categories.

To execute the orchestration, a set of instructions is required. We could create a pre-defined set of node types which will be recognized by the framework. A lookup table with orchestration instructions could be attached to these node types. If the parser recognizes a node in the set, the framework is able to execute the instructions to create the individual node. TOSCA supports properties attached to nodes, providing support for customization of the orchestration instructions, like port configuration and credentials. The other method is including the orchestration code in the nodes themselves. This requires the developer who is describing his application to have knowledge of IaC and write a lot of extra code, which is not desirable. However it does offer a lot of flexibility and makes the orchestrator more extensible for new technologies.

The best solution would be to have a pre-defined set of node types which already contain the artifacts for orchestration. Allowing a developer to create new node types and attach the correct orchestration artifacts when a new technology enters the market. Allowing developers to easily add support for paradigms like serverless and containers as a service next to virtual machines.

```
def main(topology):
    nodes, relations = topology
    for node in nodes:
        if nodeType(node) is "virtualHardware":
            # Orchestrate the infrastructure attached to this node.
            execute(node.IaC)
            # Once the node is created, provision the children
            if node.child is not None:
                provisionNode(node.child)

    # Update infrastructure with correct relations
    for relation in relations:
        update(relation.endNode)
        update(relation.startNode)

    # Configure the nodes with the right software etc.
    for node in nodes:
        configure(node)

# Recursively provision the node and its children
def provisionNode(node):
    provision(node)
    if node.child is not None:
        provisionNode(node.child)
```

Listing 5.2: Orchestratrion Example 2

This solution could also be extended to the relations, making adjustments in orchestration from attached nodes or creating entirely new entities and connecting them to other entities. The problem with configuring nodes correctly is that the relation is not always attached to the entity that needs to be updated. For example, let's assume the application opens an ssh connection to another entity. This would require the node that is running the application to allow outbound traffic on port 22. When combining the nodes and their relations we get an orchestration algorithm like listing 5.2.

A topology in TOSCA can be ambiguous and lack the information for specific orchestration. For instance, a user can specify that his application is hosted on a Linux system. If the Linux system has no specified *HostedOn* property, the specifications of the machine hosting the application are unknown, it could be a Raspberry Pi or a high grade server. If we relate this to service level orchestration, we are unable to know what type of machine we have to request with the service provider, requiring the framework to make this decision.

If the run-time specification of the application is known, an estimation of the machine that has to be created could be requested. The problem with this approach that estimating the run-time requirements of a single unknown application is still an open problem. Various researchers were able to roughly estimate workloads but this was by analyzing large pools of known tasks beforehand [53]. The framework that implements the orchestration is free to choose what estimation method, if it uses one at all, it uses.

After orchestration the framework should test if the topology that was provided was also created. Individual nodes in the infrastructure can easily be tested if they are present and their properties correct. It is however harder to test if the entire infrastructure is correct. This would also probably require the installation of software on the created infrastructure. For example, if we want to test if one instance is able to connect to another.

The simplest method of testing the infrastructure is by testing the application. The developer could provide the framework with a set of test which can be executed if everything is orchestrated and con-

figured. It does mean that the framework itself has no method of ensuring that the infrastructure is correct. If we see the orchestration as a black box, we are not testing the black box but the application that uses the output of the black box as input.

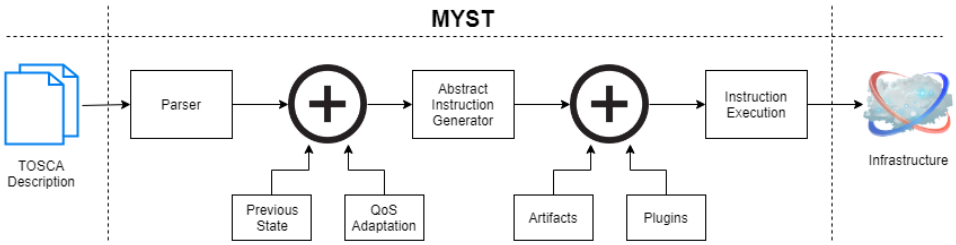

Figure 5.2: Orchestration Design

The solution we propose for orchestration in figure 5.2 is similar to the TOSCA build plan for static orchestration. However, we have added the ability to use the current state of the infrastructure and make adaptations based on quality of service violations.

Heterogeneous Devices

Automated orchestration is largely enabled by service level providers, like cloud providers. Heterogeneous devices that are not provided by service providers require other methods of orchestration. These devices have to be orchestrated manually, but still require to be added to the list of available infrastructure of the framework.

Devices should advertise themselves to the framework that they are available. We propose a sort of subscription service, where a device can subscribe with the framework. The framework in turn keeps a list of all the existing devices containing their properties. After the machine subscribed to the framework, the framework should do same basic configuration of the system.

The list of existing devices allows the framework to choose from the list and find a matching machine, instead of requesting the specific instance from a service provider.

Devices outside the cloud paradigm are mostly not provided by a service provider. These devices have to be added manually. we created a subscription service where devices are able to subscribe to be part of the existing infrastructure. this requires running a script on an existing machine. the script is currently Linux only and has some dependencies. we still have to describe the method of adding other types of devices to the network.

5.5 Configuration

After the orchestration we have several execution steps that create the infrastructure. We can add our configuration steps to the same list. After the resource is orchestrated we can trigger the configuration of it. In figure 5.3 we created a simple example of how this should look. On the left a simple description of a web server and database, and on the right the execution steps. On the top we see the original steps for orchestrating the machines. After a machine is created the configuration is triggered, and after the configuration the configuration of the next node is triggered. We also notices that the web server and the database have a requirement to capability relation, requiring the database to be configured before the web server can.

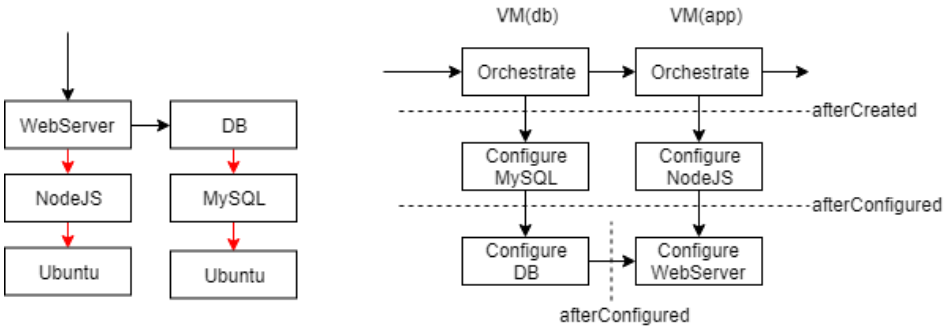

Figure 5.3: Configuration example

The approach essentially injects the configuration steps directly behind the orchestration steps. This means that we need to find a new way to orchestrate the infrastructure in parallel. Executing the orchestration, configuration and deployment serially could introduce latency. The framework could still orchestrate in parallel, but now we need extra checks during configuration to check whether dependencies have already been created. We can see this back in the extra checks in figure 5.3.

During the configuration step of the framework, it essentially executes code on the target infrastructure to prepare it for the application. Most commonly this code is executed by executing scripts. TOSCA enables the developers to attach artifacts to the nodes. The artifacts can be called on by the lifecycle event that are pre-defined in the language.

Since existing devices can be added during run-time, these devices have to be configured for the application during run-time too. After the device subscribed to the network, the framework will execute some logic on the device to make it ready.

TOSCA supports several lifecycle events (table 5.1). Interfaces support several properties like, create, configure, start, stop and delete. In table 5.1 we can see when the artifacts is executed and what happens next. The code that is supposed to configure the infrastructure is in artifacts that are attached to nodes in the TOSCA description. The execution of the artifacts can than be bound to the lifecyle events using the interfaces.

Node State	Transitional	Description
initial	no	Node is not yet created. Node only exists as a template definition.
creating	yes	Node is transitioning from initial state to created state.
created	no	Node software has been installed.
configuring	yes	Node is transitioning from created state to configured state.
configured	no	Node has been configured prior to being started.
starting	yes	Node is transitioning from configured state to started state.
started	no	Node is started.
stopping	yes	Node is transitioning from its current state to a configured state.
deleting	yes	Node is transitioning from its current state to one where it is deleted and its state is no longer tracked by the instance model.
error	no	Node is in an error state.

Table 5.1: TOSCA Lifecycle Node State [44]

We already discussed some tools that are able to configure machines. Preferably we would use a tool that does not need to be installed on a target machine to configure it. This allows us to easily add new machines without requiring manual set-up. For general purpose devices this would mean Ansible would be the best fit for configuring devices. But we do not want to be locked to a single tool, so we would like to be able to add multiple tools. This would also increase the usability across different types of devices.

Description to Configuration

It is important that nodes are configured in the right order. If not, it could happen that dependencies are not installed when a script is ran. For example, a web-server trying to connect to a database that

does not exist yet. TOSCA has two properties that help order the execution steps, the host relation, and capabilities and requirements relations. The *HostedOn* relation is simple, as the node is literally hosted on the other node, requiring it to be present. Capabilities and requirements can be used to describe endpoints on TOSCA nodes. The capability describes something the node provides. Another node can connect to it. A requirement describes that the endpoint it connects to has to be present before the node can work. If a requirement connects to a capability endpoint, the capability has to be created before the node with the requirement can be created.

While translating the internal representation to the execution steps, the framework will go over each node in the topology. Attached to these nodes are the artifacts that execute the configuration. The artifacts are attached to lifecycle events defined by the TOSCA language, like configure or create. Depending on the trigger the command to execute the artifact should be added to the set of execution steps.

Heterogeneous Devices

Currently it is unclear how files attached to lifecycle events get invoked. If an artifact is attached to a lifecycle event, like *onCreate*, we do not know the execution environment. To support the execution of configuration scripts for other types of devices, we have to set the execution environment. TOSCA discusses the ability to add an artifacts processor to the description. This allows the developer to specify the execution environment where the artifacts should be executed in. This should allow us to add support for heterogeneous devices with their own execution environments. The grammar for this feature is however currently not implemented in the language. This means we would have to propose the grammar extension for it ourselves. With the grammar extension we could handle the required logic in the rest of the framework.

Embedded devices and sensors could be harder to configure, communication wise. One of the solution to configure these devices could be to let the developer do that himself. If he is able to deploy software to gateways where the sensors connect, he can configure the sensors from there. This would go against the idea of the framework, since it should handle the configuration and the deployment. We could also manage configuration from gateways using the framework, but we would need to update our design for this. Kenan Xu *et al.* [54] proposed a method that uses gateways to deploy software to distributed sensor networks. A method like theirs could be used as the basis for the configuration tactic. Often embedded devices come pre-configured from the manufacturer. Configuration could mean sending a few simple command, or flipping a few bits. Configuration would not resemble executing a script on the device itself.

5.6 Deployment

After the infrastructure is correctly configured, it is ready to run the applications. Our solution is responsible for deploying these applications to the individual machines.

Packaging

Virtualization is often used for packaging applications and has many benefits. Virtualised applications often come packaged with all their dependencies and run-time requirements. Because they use a hypervisor they are portable to different operating systems and will run the same on every machine. But because the application is bundled with the system that will run the application, it often has overhead both for storage and run-time performance. Also these platforms are often entirely replaced when the application is updated, requiring decoupled storage.

Virtual machines were used to package entire applications and make them portable. The virtual machine would include the application with all it's dependencies and the required configuration. A whole virtual machine could be copied to a physical machine and started without additional configuration.

Currently containerization is popular in the industry. This is essentially still a virtual machine that runs on a hypervisor but the amount of included overhead is less. It does not include an entire operating system like a virtual machine does. This makes them relatively small while still maintaining the advantages of the virtual machine. Calls from the containers are directly translated to calls in the host operation system.

Many platforms include package managers or dependency pulling mechanisms. For instance, node includes npm and python includes pip. These allow a developer to only specify what dependencies

are required by name. When the application is loaded and initialized, the framework will pull the dependencies from a package repository. These do however require the machines running the application to be connected to the internet.

Developers could also manually package an application and copy it over to the machine that will be running the application. This method is very error prone with developer mistakes and normally not platform independent. It often requires the developer to configure the host system manually before the application is able to be ran on the system. While this method allows automation it often requires more configuration of the automation scripts during development. It does however offer small packages and is really simple to implement for smaller applications.

Distribution

As discussed, virtual machines could also be seen as application packages. The application includes the entire machine it will run in. The application will be packaged in an ISO like file, and started on the target machine. They ISO file can be an artifacts of the TOSCA node, including a script that installs the machine. If the deployment of the virtual machine is not done during orchestration, double virtualization occurs, increasing the run-time overhead.

Containerized application can easily be deployed using tools like Kubernetes[55] or Docker Swarm[56]. These tools allow for the deployment, management and monitoring of many containers deployed across many machines. A single instance, like our solution, creates a containerized workload, and deploys it automatically to the right machine.

In its most basic way our solution is responsible for copying the artifacts to the target machines. The TOSCA description already contains the entry point of the application and how it should be executed. Many configuration tools already support copying the artifacts from a master instance to target nodes. This means the deployment step could mostly be combined with the configuration step. But starting the application should be delayed untill the entire infrastructure is realized.

Deployment Strategies

To speed up deployment in larger networks, we could use relay servers. In figure 5.4 we see an example with 12 machine to be deployed to. We inject 4 relay server into the infrastructure which each connect to three nodes. If we deploy the application from a single machine to all 12 machines, it will take 12 time units to finish. Deploying to the relay server will take 4 time units, and the relay units deploying to the machines will take 3 time units, adding up to a total of 7 time units. Drastically improving the time it takes to deploy an application. It should be mentioned this would increase the overhead of the infrastructure, since we have to add extra machines for relaying. Xu *et al.* [54] also show significant improvements when using this method and also test multi-hop approaches in comparison to this single hop approach.

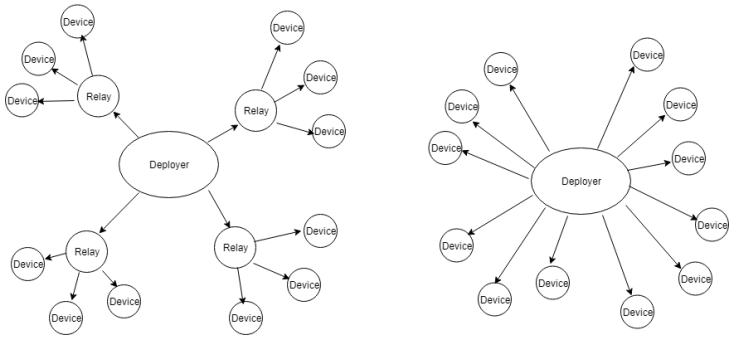

Figure 5.4: Relay Server Example

Heterogeneous Devices

Luk *et al.* [30] proposes a high level compilation tool chain that enables developers to write software for multiple types of devices concurrently. This allows us to abstract the code for devices in the edge and IoT and deploy to code to multiple types of devices. The output of their abstract language is currently focused on three types of processing units, micro processors, FPGA's and DSP's.

StreamPipes [57] deploys data processing pipelines across heterogeneous infrastructures. They utilize several communication standards such as, HTTP, Kafka, MQTT and OPC-UA. This allows them to deploy software on most types of embedded devices. If we could support most of these communication and deployments protocols, we are also able to support a wider range of devices.

5.7 Monitoring

To monitor the infrastructure we have to gather data from all the devices in the infrastructure. The data has to be saved in a central location, where it can be analyzed to check if the infrastructure complies with the QoS constraints. If the infrastructure does not comply, the system can take appropriate actions to correct it.

Data Gathering

We propose two methods of gathering data such that the monitoring block is able to check if the infrastructure upholds the quality constraints set by the developer.

When creating an API endpoint on the monitoring node, the fog nodes could send updated context information to the master, which then can update the information in it's lookup table. This could be used for updating utilization value or changing environment values. This method does require software running on the fog node that is actively gathering context information and actively send updates to the monitoring node. This would require the framework to set up the software on the fog node, which also increases the run-time overhead on the fog node.

While the previous method is focused on the nodes pushing the data, it could also be reversed and let the monitoring nodes actively pull the data from the nodes to save it in the lookup table. Security of this method would be questionable. Since it requires all the nodes to open certain ports to be accessible from the outside. The overhead of this method on the master node would be large. Constantly requesting information from the nodes, waiting for the response and updating it in the database would require a lot of resources. And in comparison to the previous method this puts the load more on a single instance instead of spreading it across all the nodes. Extra scheduling would be required because a round-robin method would probably not be suitable for all network configurations.

The first method, based on installing monitoring software on the nodes, is most suited for our application. The run-time overhead could pose problems with resource limited devices, but the security advantage overcomes this.

Insert Agents

As discussed, we will be installing monitoring agents on the devices in the infrastructure so that they are able to broadcast their context information. Installing the agents could be done by the same tooling that configures the infrastructure. Just as it installs software provided by the developer, we could inject our own software there.

We propose to inject the monitoring into the description of the infrastructure. In figure 5.5 we draw a situation with a single machine. Next to the software defined by the developer we can inject our own software that gathers the data and connects to the central monitoring system.

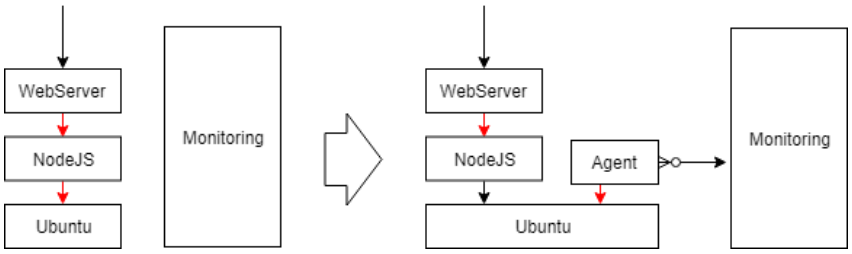

Figure 5.5: Insert Monitoring Agent

Heterogeneous Devices

By choosing the method that installs agents for monitoring the infrastructure, we also have to install these agents on edge and fog devices. Some of these devices require customized agents to be installed, because they do not run on a general purpose OS. The developer would have to write their own agents that sends the data to a pre-defined endpoint. Installation of these agents would change for different devices, would have to be defined by the developer.

Communication could pose a problem while sharing data. Even flexible solutions like API endpoints still require TCP/IP communication. Even if these nodes are able to send their data via customized communication protocols, the monitoring parts would also have to be extended to support it. This would require the developer to write software that runs on the monitoring part to cummunicate with the devices.

A standardized solution could be to deploy the monitoring agent to gateways that the devices talk to. These gateways are able to communicate through TCP/IP. The monitoring agent on the gateway should be able to communicate with the devices.

Howard *et al.* [58] proposed a monitoring service for embedded devices. Their method is based on embedded devices from a service provider. This means that most of the hardware is already abstracted. Their methods is comparable with saving all the data on a single location. They gather data from multiple service providers and aggregrate it.

5.8 Summary

In this chapter we discussed how we can automate the process of creating composite applications across heterogeneous infrastructures and what our requirements for it are. We talked about what tools are currently able to solve the problems we are facing and what they lack for our implementation. We discussed how we are able to automate the process on devices other than general purpose. And we proposed several solutions to the problems at hand.

Chapter 6

Implementation of MYST

Based on the initial design (chapter 3) and the problems explained in the previous chapters, we propose an implementation of the framework that is able to tackle these problems. We discuss the design of the framework and verify it with the proposed use case. We also talk about some of the experiments we have done to substantiate some of the choices we made.

6.1 Approach

To define the requirements of our framework we started by creating some user stories for what a developer want to achieve using our framework. From the user stories we can define actions of the framework. Next we create a simple design overview which we use to make a simple architectural description of the framework. Using the architectural description and the actions we can define how parts of the system interact with each-other and define the actions between them. Even if the framework is not finished we want to be able to describe the framework so that in future work all the parts can be filled in. We will decompose the system into smaller sub-systems, allowing us to describe inputs, outputs and actions more specifically. With the correct definitions of the blocks they can be implemented using correct tooling for them.

6.2 Top Level Design

In this section we will discuss the top level design of the framework. It is mostly an addition to chapter 3.

6.2.1 Challenges

The main challenges that we try to solve with this framework are.

- Providing a way to describe composite applications over heterogeneous infrastructures in a unified way.
- Automated orchestration, configuration and deployment of the composite application and its infrastructure.
- Ensure run-time quality of service.
- The ability to reconfigure the application.
- Abstract choosing the right machines with the right resources.
- Keeping the framework extensible.

6.2.2 Block Design

We have created a top level block design to illustrate the design of our framework (figure 6.1).

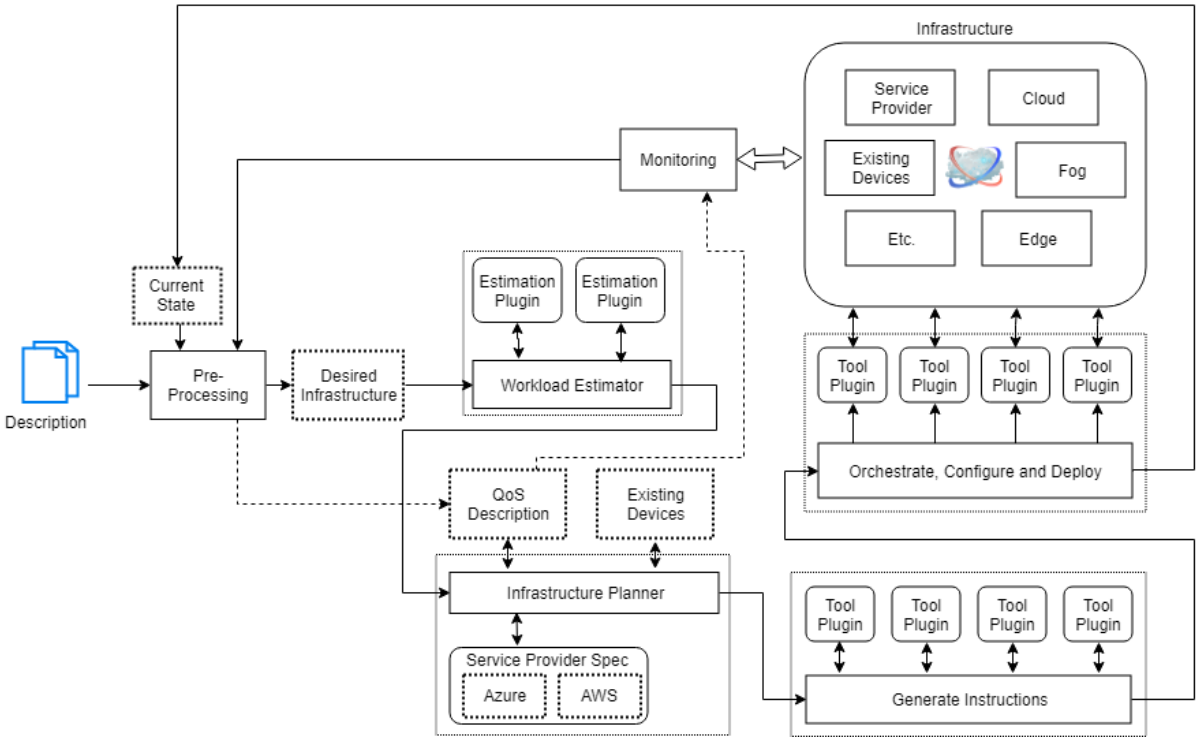

Figure 6.1: Top Level Design

Pre-Processing Once a developer inputs his description of the desired system into MYST, this will be the first block that processes it. This stage is responsible for translating the description into an abstract infrastructure that our framework is able to work with. And is also responsible for separating the QoS constraints from the description, so this can be used in other blocks. It combines the provided description with the current state/infrastructure of the system to only create the required infrastructure. The monitoring agent is able to request reconfiguration of the infrastructure if the QoS constraints are not met.

Workload Estimator The workload estimator is responsible for estimating the required resources for an application. This allows the framework to choose what resources should be requested from service providers or which ones to select from existing devices. Different types of applications and platforms require different estimators. By using a plugin based estimator, we hope to provide support for these open issues.

Infrastructure Planner Using the estimated workload and required resources the application will utilize, the application is able plan the actual infrastructure. It can now bind abstract computing resources from the description to the actual devices that need to be requested. It will ensure the planned infrastructure complies with the QoS constraints. Combining the existing infrastructure with service level resources allows for flexibility. The plugin system for service providers allows for easily adding new providers and easily updating their specifications once they add new functionalities.

Instruction Generator The instruction generator will translate the planned infrastructure to a set of abstract instructions that are required to set it up. After the abstract instructions are generated, it will add the actual implementation to the instructions. The tool plugins provide the actual instructions that are required for the tools to set up specific resources. It should be easy for implementing new types of resources or devices.

Orchestration, Configuration and Deployment Engine The engine is responsible for executing the instructions generated by the instruction generator. This will communicate with the service providers, existing infrastructure etc. The infrastructure that is generated by this block will contain virtual machines, virtual networks, databases, routers, and all the resources required to run the application. This block will create the infrastructure, configure it, and deploy the software to it. It is not only responsible for creating infrastructure, but also removing infrastructure. After creating the infrastructure, the state of the infrastructure should be saved. This allows for the declarative system to compare a newly provided description with the state of the system.

Monitoring The monitoring system will constantly check the realized infrastructure if it still complies

with the quality of service constraints. It will also be responsible for checking when resources break and have to be fixed. If actions are required to monitoring system will feed the required changes back into the framework, triggering a reconfiguration of the infrastructure.

6.2.3 Actions

Externally there are several actions the developer can take to influence the framework. These actions are required for triggering the framework to do something and also provide necessary information for it to function.

1. Initial provisioning of the application and its infrastructure.
2. Reconfigure infrastructure by adding new TOSCA description.
3. Attach plugins like, Executions, Instructions, Service Provider and Estimation.
4. Add existing devices to internal database of existing devices.

Internally the only action the framework takes is to trigger a reconfiguration. This happens when the QoS constraints are not met, and part of the infrastructure has to be adjusted so that it does.

We do not provide tooling to create the actual description. The description is textual but can be visualized by the right tools. The textual description can be created in any text editor. Tools like Winery [59] are able to visualize the description, but these will not support our extensions yet.

6.3 Block Specification

In this section we will go deeper into the implementation of the framework and the individual parts of it.

6.3.1 Pre-Processing

Specification

Pre-processing is responsible for translating the TOSCA description provided by the developer into an internal representation (IR) that is usable by the rest of the framework. The block will also combine the created IR with the current state of the infrastructure, and adjustments from the monitoring block, creating the newly desired end state.

Input This is the first block the description provided by the developer will go through. The developer has to provide it with the description of his application. This description should contain the topology, application artifacts and the QoS constraints of the application. The block also requires two internal inputs, the current state and implementation of the infrastructure, and the QoS violation events from the monitoring block.

Output The output of the block is an enhanced internal representation of the description. This description is annotated with information about what does and does not needs to be created. It still contains the topology and application information. It holds the desired end state of the infrastructure, which is combined with the requests from monitoring and the current state.

Implementation

In figure 6.2 we have highlighted how the data from the description flows through the pre-processing block. We start by converting the TOSCA description to an IR. This gets combined with inputs from the events from the monitoring block and the current state of the infrastructure. This should result in a desired infrastructure that is able to be created by the framework, contained in the annotated internal representation.

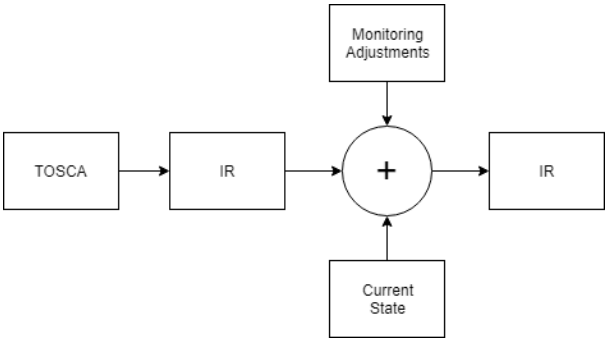

Figure 6.2: Pre-Processing design

To parse the TOSCA into the internal representation we will use the tosca parser provided by OpenStack [1]. We have to extend the parser so that it supports our extra extensions. Since the parser is open-source, and built in python we expect this to be relatively easy.

To compare the IR and state, and inject the adjustments from monitoring, we need to write our own software. In figure 6.2 we see how the IR is aggregrated with the data from monitoring and the current state. We have looked into software that is able to compare abstract syntax trees (AST), e.g. meta-programming languages like RASCAL [60]. But our IR consists of more relations than just AST's. We choose to write our own implementation in python that can be directly linked to the parser.

The software that is able to detect changes in the nodes can not be dependent on linking identifiers. Just like static code analysis, it has to detect how much a node compares to another and how much the relations do. Based on this it has to determine if the node is already in the infrastructure and has to be updated, or if it is an entirely new one.

We are using python which is an object based language. This allows the IR to be extended relatively easily. It is easy to add new properties to existing objects and update relationships between them. We do no have to deal with memory management and custom types, lowering change to make errors during development.

The IR we propose is semantically comparable to the TOSCA topology, but allows us to add our own properties. Just like TOSCA, single parts of the application should be built as a doubly linked tree structure, which is based on the *HostedOn* relation. Objects in this structure are connected to nodes in other tree structures just by attaching relation pointers. The three structure should enable us to easily inject or remove nodes in/from the tree structure or update relationships.

During the initial orchestration the current state of the infrastructure is empty. This means we do not have to compare the new description, and can just directly pass it on. During run-time events from the monitoring block will arive and tweaks will be made.

How does the block solve problems mentioned in chapter 4 and chapter 5

The block solves the reconfiguration problem by comparing the new TOSCA description provided by the developer with the current state of the framework. As a result the framework is able to know what should change and what not.

The block solves the QoS maintenance by taking adjustments from the monitoring block where QoS is not met and flagging the parts which needs to be adjusted. The parts that should be updated to maintain QoS are scaled accordingly and the rest of the framework will implement it.

Open problems

- We still have to write the extension to the OpenStack TOSCA parser.
- We still have to write the software that flags the differences between the new description and the current state.
- We still have to write the software that updates the IR according to the events from the monitoring block
- We still have to create the exact data model of the internal representation.

[1] https://github.com/openstack/tosca-parser

6.3.2 Workload Estimator

Specification

The workload estimator is responsible for determining the workload that will run on a certain piece of (virtualized) hardware. This allows the framework to make smart decisions on what resources to request from a service provider, or where to place software on existing hardware.

Input The workload of the application could be specified by the developer in the description, which can be used by the workload estimator. If the description does not contain this information, the estimator is able to use the artifacts and topology to estimate the workload. The estimation plugins should be provided as a separate library by either the developer or the supplier of our framework.

Output The workload estimator will output an updated application description where software components are annotated with the resources they require to be ran.

Implementation

In listing 6.1 we have written an overly simplified algorithm to illustrate the actions the workload estimator will take. The algorithm will iterate over the compute resources (hardware or virtualized hardware) and calculate the amount of processing power it should posses. It uses the *HostedOn* relation to find the nodes that are hosted on this machine. It will calculate the resources that node requires and iterate further. Summing up all the required resources of the software running on that machine, will give the indicator of the required hardware. It should be noted that this algorithm is really oversimplified, resources will not be specified by an integer, and calling the correct plugin and processing that information requires more effort.

Maria Calzarossa *et al.* [61] did a survey on workload characterization. They define 4 different methods of workload modeling techniques, 1) Formulation, 2) Run-time analysis, 3) statistical analysis and 4) Representativeness. They state that the problems in characterizing centralized systems have already been solved. While they also mention that these problems for more distributed systems are not yet solved. The larger the number of hardware and software components in the application make it harder to characterize the system. In 2016 they wrote a new paper [62] revisiting workload characterization. Chapter 7 discusses cloud workloads across the heterogeneous infrastructure. They mention how the studies are not really complete because they lack large amounts of real world data.

Goldin [63] describes run-time profiling of a software application. It starts up an application and attaches a profiling application which is able to read properties like CPU utilization and RAM usage. Our estimator could set up a test environment where it would run the application for a static amount of time while monitoring the properties. After the amount of time the actual application with the right resources could be deployed to the real environment. The problem with this is that it requires a lot of time to actually run the application, causing significant delay in the orchestration process. The accuracy of this method is also questionable, since the applications could have spikes in performance. This method would also require the emulation of real-world traffic for somewhat accurate results.

There are tools available that are able to profile application. Take for example the Java language. Mytkowicz *et al.* [64] wrote a paper comparing different profilers for the Java language. They mention multiple tools, hprof, xprof, jprofile and yourkit. Tools like these could be utilized as plugins for our workload estimator. The problem is that it is only able to analyze a single type of application, for other applications you require more plugins. This would require the estimator to recognize the type of application and call the right plugin for it.

To be able to link plugins to application workload, the workload has to be classified in some way. The reason for this is so that the framework knows what plugin to call when.

In the simplest form we could link file extensions to plugins. The problem with linking the plugins to file extensions is that a file is not necessary an application, which often consists of multiple files. This would require the framework to look for the entry point of the application. Also some workloads are not really specified by file artifacts, like containerized workloads. But since we are using TOSCA our software components are mostly defined using node types. We could link node types to certain plugins, which are then responsible for estimating the workload on that node. The problem is that node types can still contain different types of software, so that that the plugin needs to handle all of them. By linking plugins to multiple node types, we hope not to have to write too many plugins. This method would require the framework to keep a lookup table of the plugins and where they relate to.

We have to define a standard unit for the workload. However different software has different hardware intensive properties. It would be very hard to estimate everything with just operation per second. To

get a basic image of overall system requirement we propose to measure the workload based on 3 essential parts, CPU, RAM and storage. Defining a standard unit for each of the three allows us to simply communicate the requirements between the hardware and the framework. For CPU we could define the operations per second. For RAM we could define the maximum amount required. And for storage we could also define the maximum amount required. We do neglect a lot of important factors, like the speed of the storage, the multi-core processing, network bandwidth required and many more. But his should give the framework a rough idea of the hardware required to run the workload. The monitoring block could still request extra resources if required.

```
# Add workload requirements to ever
function main(topology):
    for node in topology.nodes:
        if type(node) is hardware:
            try:
                node.workload = estimateR(node)
            catch:
                node.workload = unknown

# Iterate up in the topology starting at the hardware
# Sum up all the workloads of the nodes running on one machine
function estimateR(node):
    int e = 0
    if type(node) is software:
        e = estimate(node) + sum [estimateR(h) | for h in node.hosts]
    if type(node) is hardware:
        e = sum [estimateR(h) | for h in node.hosts]
    return e

# Get the workload of a single node in the topology
function estimate(node):
    if node.workload is None:
        # throw error if no plugin is available for this type of application.
        return callPlugin(node.application)
    else:
        return node.workload
```

Listing 6.1: Recursive estimation pseudocode

Considering estimation for devices in the scope of IoT, Fei Li *et al.* [13] propose an extension on the TOSCA language to support IoT devices. Their extension includes binary artifacts and compiler specifications in the TOSCA description. Using these extra artifacts it would be easier to estimate the workload for heterogeneous devices. Their extension also proposes extra relation types for supporting IoT devices in TOSCA.

Challenges We have to define a standard model in which the plugins should be built. This means that we also have to define what the inputs and outputs of the plugin should be, and how these are able to be tied into the framework. Currently we have not proposed this model yet, requiring extra work in the future.

The plugins also have to load in the software before they are able to analyze the workload. Depending on the node type this could greatly differ. The software could be in the artifacts and copied by the plugin, or it could be a container which has to be pulled from a repository, or it could be pulled by an artifact. Plugins would have to pull the application or analyze the code to be able to analyze the workload.

Artifacts are not necessarily the actual applications that will run on the machines. They could easily be some scripts that pull and install an application from a repository. This would pose a problem with workload estimation since the provided artifact will not be the actual workload that is running on the machine and will only run once instead of continuously. Solving this problem is probably up to the plugin that is measuring the workload.

If you combine this with the ambiguous (virtualized) hardware, you need to estimate the workload of the software for different types of machines it is able to run on. This requires to run the estimator multiple times and attach all of this information to the node. Meaning the planner has to now which annotated information belongs to which hardware. And having to decide which one is the smartes to plan around.

6.3.3 Infrastructure Planner

Specification

The compute nodes are annotated with the physical nodes that will run the software, or annotated with the service provider specific information to realize the physical machine. If the workloads on the software components do not match the hosts, this block is also responsible for updating their descriptions.

Input The annotated internal representation which is accompanied with the resources required to run the application.

Output The description that is annotated with the information linking the compute node to a machine or the service level information.

Implementation

The block combines the infrastructure provided by a service provider, with the existing infrastructure and its characteristics. Using the QoS constraints defined in the TOSCA description, it is able to plan an infrastructure that adheres to the requirements.

Cloud providers often provide documentation of their platform and what software defined infrastructure they are able to provide. As we want our platform to be open, we do not want developers vendor locked to specific providers. This is why we chose that different provider specifications should be easy to plug in to the framework. Using the different specifications and characteristics the infrastructure planner should be able to plan across different providers and different types of devices. As new providers enter the market, with maybe new functionalities, they should be easy to add to the framework.

The pseudocode in listing 6.2 describes a simple algorithm that explains the basic workings of the infrastructure planner. The implementation of this algorithm and checking of the constraints would require a more complex implementation. This implementation is more geared towards general purpose and cloud hardware in comparison to more low level devices. Although workloads matching the device is still upheld.

```
def main(topology):
    nodes, relations = topology
    # match TOSCA nodes with physical
    for node in nodes:
        if nodeType(node) is "virtualHardware":
            node.device = findDevice(node, None)

    # check QoS on nodes
    for node in nodes:
        if not testQoS(node):
            # node has to be reevaluated exclude chosen device
            node.device = findDevice(node, node.device)

    # check QoS on relations
    for node in nodes:
        for relation in node.relations:
            # only test nodes that have a QoS specification
            if relation.qos is not None:
                # test if the the relation upholds qos constraints
                if not testQoS(relation.startNode, relation.endNode):
                    # both nodes have to be reevaluted
                    node.device = findDevice(node, node.device)

def findDevice(node, excludeDevice):
    for device in existingDevices:
        # if the device can handle the workload, select it.
        if device.characteristics > node.workload
                and (device is not excludedDevice):
            node.device = device
            existingDevices.remove(device)
            break
    # if no device is found in the existing devices, use SLA
    if node.device is None:
        node.device = serviceProvider.getMachine(node.workload, node.qos)
    return node
```

Listing 6.2: Simple Infrastructure Planner

How does the block solve problems mentioned in chapter 4 and chapter 5 The block solves the problem of choosing the right machines to run the right parts of the system. This block is able to optimize the infrastructure, and maybe in later stages it could be combined with soft quality constraints, like lowest cost as possible.

We mentioned how current tools in industry have no room to do their own interpretation. This should provide them with a way of selecting the right resources for the right applications.

Using a QoS specification and characteristics of the infrastructure, the tool should be able to plan the application over the infrastructure, in such a way it adheres to the standards set by the developer.

Challenges To be able to uphold QoS constraints during initial deployment, the framework would have to model characteristics between blocks. This requires modeling for every type of constraint we enable to be defined. If developers would add custom constraints, they would also have the code that models the constraints. This code would be required for initial deployment, but also for the monitoring block. How the information is gathered to make these decsions would also be custom for every type of constraint.

A problem with this system is that different providers, support different functionalities and provide different options to configure them. Describing the different provider specifications into an uniform method could pose a challenge.

In contrary to existing infrastructure, determining the characteristics of unexisting infrastructure, from service providers, is more guesswork. Depending on the requested resource, making it easy to estimate or not. For instance, testing the latency of a connection between two unexisting machine is uncertain.

6.3.4 Instruction Generator

Specification

The instruction generator is responsible for translating the plan generated by the previous blocks into executable commands. It should cycle trough every abstract command created and fill in the executable command for this. It uses plugins to generate the code or commands. It should be able to generate the instructions for orchestration, configuration and deployment and call the correct plugins for each command.

Input An internal representation annotated with the extra information as described in the previous steps.

Output The output of this block is the abstract plan annotated with the code for the execution steps. These execution steps are able to be used by the orchestration, configuration and deployment engine to create the infrastructure and get the application running on it.

Implementation

Our instruction generator is pretty similar and based on the plan builder proposed by the OpenTOSCA group [65]. *First* we transform the internal representation, that is still similar to the TOSCA topology, into set of steps that is required to realise the infrastructure. E.g. The machines have to be created before an application can be installed on it. *Second* we add the TOSCA life-cycle events in between the generated steps. If we have a artifact that should be ran after a node is created we add the artifacts execution step behind that. *Third* we add steps to install monitoring agents on the devices. This allows our framework to interact with the devices and maintain QoS standards. *Fourth* we add the platform specific instructions. The instructions differ per type of device and the tool that executes the instructions. For example, for cloud providers we can use Terraform to create infrastructure and Ansible to configure it. But for embedded device there is no need to create it but it could be configured through special software.

When creating the execution steps we start at the lowest or "hardware" nodes. We now have a starting step for creating the hardware. Now we can iterate up the *hostedOn* chain, ensuring dependencies for the following nodes are present. The execution steps for the following nodes are added to the list one by one. After iterating through all the nodes, it is also import to iterate through all connections. They could require the system to open up ports or realize data pipelines.

Challenges The instruction generator has to know the type of the node to generate platform specific code for it. Also relations to the node have to be checked since nodes being hosted on that node could also require that platform specific code. By creating a transitive relation based on the *hostedOn* property we could mark all nodes with the specified system.

Every abstract step generated, requires specific code to execute that step. That means that when writing a plugin, a finite amount of possibilities are present, and the developer has to write code for all the possible steps. But how do we decide what step belongs to what plugin, so that the plugin is able to generate an instruction for the step.

It is important that right order of execution steps is maintained. For example, let assume we have a database and a webserver. If the webserver is configured first and it tries to connect to the database it will error because it does not exist. Requiring the instruction generator to know that the database has to created first. TOSCA solves this by defining requirement and capabilities, where the node with the capability has to exist before the node with the requirement. For optimal performance we would like to be able to parallelize the process, however we do not have the algorithms to do this yet.

6.3.5 Orchestra, Configure and deploy

Specification

This block is responsible for executing the set of instructions, to realize the actual infrastructure and configure the application on top of it. It should also keep track of the current state of the infrastructure. After this block is executed there should be a working application running on top of the requested infrastructure.

Input The block will take a set of instructions generated by the previous block as an input. The instructions should already be in the right order. The instructions should be able to be executed by the installed plugins.

Output After this block is executed the infrastructure should be running with the application on top of it. It should be linked to the monitoring block and is running the composite application. To allow reconfiguration the block should output a file that describes the current state of the infrastructure. This allows the framework to compare a new description with the currently running infrastructure, and decide what changes should be made to realize the new description.

Implementation

To support different methods of communication and types of execution, we allow the use of plugins. If new types of devices get added or new methods of communication, a developer could simply write a plugin to support these. This makes the framework extensible for future technologies.

For reconfiguration purposes the block should output a file which describes the state of the infrastructure. It should tell our framework what resources are present, with what characteristics, and what is installed on them. The state file ensures that our framework is declarative. Allowing the framework to check the differences between the current end state and the desired end state. An example of this can be found in Terraform. It creates a *.tfstate* file in the *json* format. This file contains what infrastructure it has created, and the variables generated during creation. We would like to extend this with the configuration and deployment instead of only the infrastructure.

The framework itself will not realize the infrastructure, it will give the right commands to the plugins, which in turn are responsible for creating and configuring the infrastructure. These plugins are shared with the plugins in the instruction generator.

Current state of implementation We started development of the framework by automating the cloud infrastructure part. Using Terraform we started automating the initial setup of the framework. We are able to generate the initial cluster manager and are able to generate some test machines in the cloud. It should be noted that the orchestration is only tested on the Microsoft Azure platform. Currently we do not have the functionality of orchestrating, configuring and deploying the user's composite application description.

After the infrastructure was created, we used Ansible to configure the infrastructure and deploy the application to it. Ansible has two advantages that are critical to us. 1) Ansible is agentless, which means it does not require anything to run on the machine before we can configure it. 2) Ansible is pretty much platform agnostic on general purpose operating systems. Allowing it to configure Linux, Windows and Mac devices, nearing heterogeneity.

In future development both Ansible and Terraform would be expected to be integrated with the plugin system of the framework. But for the setup of the framework itself we can easily just use them without implementing them in the framework itself.

Initially we have set up a Kubernetes cluster to manage all the different machines. Although this solution would not work for some devices in the heterogeneous infrastructure, it was a good starting

point. Kubernetes itself also deploys monitoring agents to the machines. However, this is mainly for health checking by Kubernetes itself. The monitoring agent is not really extensible and maintaining custom QoS constraints only using Kubernetes will be hard to implement.

We have started testing a solution that smartly places containers on top of the infrastructure. We used Kubernetes for this and they call it *Custom Scheduling* and *Node Affinity*. It allows us to take an existing infrastructure and disperse containers across it. By injecting logic that takes QoS constraints into consideration, we where able to tell Kubernetes where the containers should be placed. Containers can not be ran on all of the devices in the heterogeneous infrastructure, not making it the ideal solution. But for testing purposes it allowed us to customly place workload where we wanted in the infrastructure.

To support existing devices to be added to the network, we created a script that is executable from an existing machine which will register it with the cluster manager. This script will also trigger to cluster manager to configure the existing machine correctly and deploy the right software to it. The cluster manager keeps a register of all the existing devices and their status. When resources are requested it is able to combine this list with the offerings from the service level providers, and able to chose the best instance to deploy to.

We have created a python scripts which automates the whole initial setup and creation of test instances. The python script currently calls Terraform and Ansible, which in turn execute scripts written by us to set up the initial framework. We had to create some custom logic to link everything together. For instance, after Terraform created the infrastructure, we had to save some properties like IP address and login information, allowing Ansible to connect to the newly created instance and configure it.

Challenges To speed up the process of orchestration, configuration and deployment it would be nice to be able parallelize the process. Parallel implementation could pose dependency problems. If the commands are executed in the wrong order, it could mean that requirements are not ready.

Currently we have not pitched how these plugins get added to the framework. We have not described the exact method of implementing the plugins in the framework yet.

A developer or someone else could practically make changes in the infrastructure or the software running on top of it with another tool than our framework. If this happens, the state file our framework generated is not correct anymore. This would mean a reconfiguration could not correctly compare the running and the desired infrastructure. And would probably mean the whole infrastructure has to be deleted and a new one created.

6.3.6 Monitoring

Specification

The monitoring block is responsible for maintaining the quality standard, set by the developer, during run-time of the application. It will be linked to the infrastructure generated by the framework. If the quality standards are not met, the monitoring block is responsible for tweaking the application and its infrastructure in such a way that it does meet the quality standards again.

Input The block has two roughly defined inputs, the first one being the QoS constraints set by the developer that should be upheld, the second one the communication and events from the infrastructure itself.

Output The monitoring block flags parts of the system that are not meeting the QoS constraints. This allows the framework to update the description, and in turn update the infrastructure so that it does meet the QoS constraints.

Implementation

The block should set up an endpoint to which the agents are able to connect to. By connecting to this endpoint, they can send data and events. Allowing the monitoring block to aggregate all the data from all the devices in the infrastructure. The advantage of this is that it is more secure than letting the monitoring block request information from the devices in the infrastructure. It would require to let public entry points open at all the devices, which is not secure.

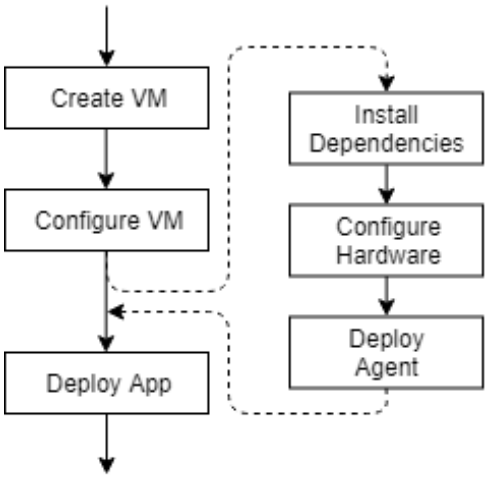

Figure 6.3: Insert Agent Build Plan

The framework is responsible for configuring the monitoring agents on top of the infrastructure. We propose to insert the instructions for configuring this into the same description as the application. On the left in figure 6.3 we see a set of steps that could be generated by earlier steps. To insert the monitoring agent we just insert the new steps where the dotted line is drawn. The agent has some dependencies that have to be installed, hardware has to be configured (like opening ports), and the agent has to be deployed and started. The following blocks in the framework will take care of the implementation and execution of the newly added steps.

The monitoring block checks if the information sent from the monitoring agents exceeds the QoS constraints. If it does, it will trigger the system to update. The QoS constraints are not shared with the agents themselves, they will only send their observations, no events. This allows the constraints to be easily update-able and not having to redistribute them.

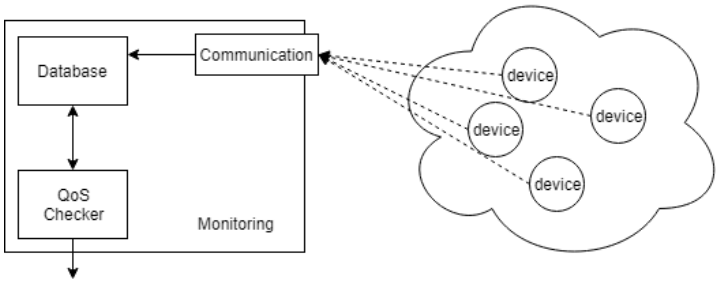

Figure 6.4: Monitoring block

In figure 6.4 we see a how the monitoring part of the system works. It receives the information from the infrastructure via a communication endpoint. After the information is aggregated in the database, the software is able to detect if QoS constraints are broken. If one of the constraints is broken, the software is able to send an event back into the system with the correct information to recover the infrastructure.

How does the block solve problems mentioned in chapter 4 and chapter 5 The block solves the problem of maintaining the Quality of Service. By keeping steps that are inserted fairly ambiguous, the framework is able to deploy the monitoring agent to multiple types of devices, depending on the device it will be hosted on.

Challenges Our current proposal uses agents that are installed by the framework on the devices in the infrastructure. With the heterogeneous infrastructure it could be hard to create these agents for all different types of devices. It would be relatively easy creating an agents that gathers information about the device on a general purpose machine, but less trivial with an embedded device. Also the type of relevant information that is able to be gathered is much different. Depending on the method the devices are configured and connected to the framework other methods could be used to gather their data. For instance, if a gateway is used, the monitoring agent could be deployed to that device and will gather

information on multiple embedded devices.

If the infrastructure would be adjusted manually, and the state file is not correct anymore, the monitoring block could flag false positives or the other way around. Making the monitoring block not fully reliable anymore, and not to be trusted to update the infrastructure to maintain QoS.

After the block determined that a part of the infrastructure does not meet the QoS constraints, we still have to determine how to scale that part of the infrastructure. For instance, if an extra instance is required, logic is required that inserts a load balancer and links it to the multiple instances. Or if an instance is created with more compute power, the framework has to know how to vertically scale it up and reroute traffic from the old machine to the new one.

Chapter 7

Validation & Performance Characteristics

In this chapter we will validate our research using the pre-defined use case, and experiments substantiating some of the design decisions we have made.

7.1 Verification

To validate our framework we use the proposed use case from chapter 1. We test how easy it is to create the application and its infrastructure using the framework. We will start by creating a (simplified) description of the application. The description adheres to the TOSCA form with the proposed extensions. After this we put the description through the framework and check step by step what the framework will do with the description, and what is required like plugins. After verifying the framework during initialisation, we are verifying the framework during run-time.

We target a game where users are grouped into time limited games. For instance, a shooting game where the first person to reach a certain amount of points will result in the game to end. In this use case we want optimize the latency between the server and the users while optimizing the resource pool of machines. The idea is that we have a single server running a matchmaking algorithm. The matchmaking algorithm will create pools of users that are geographically close to each other. Once enough users are in the pool, a game should be created for these users. The game will partly run on the user machine (edge) while the back-end, communication with other users, will run on a server. When dealing with a distributed network of servers that are geographically scattered, we are able to dynamically choose where to run this back-end.

We aim to create a back-end that is replicable and able to be created and destroyed for a single game. The matchmaking server should be able to distribute a back-end instance to a node that is geographically closest to the users. Once the back-end is correctly started we can tell the user application to which node they should connect. After the game is finished the back-end can be cleaned up, freeing the resources for future processing.

We created a simplified visual TOSCA description. This allows the reader to easily get an overview of the architecture of such an application. In figure 7.1 you can see that we added some of our extensions to the description, such as the ambiguous hosts and the QoS descriptions. We will talk about how the framework implements this use case, going through three stages, initial deployment, monitored scaling and reconfiguration.

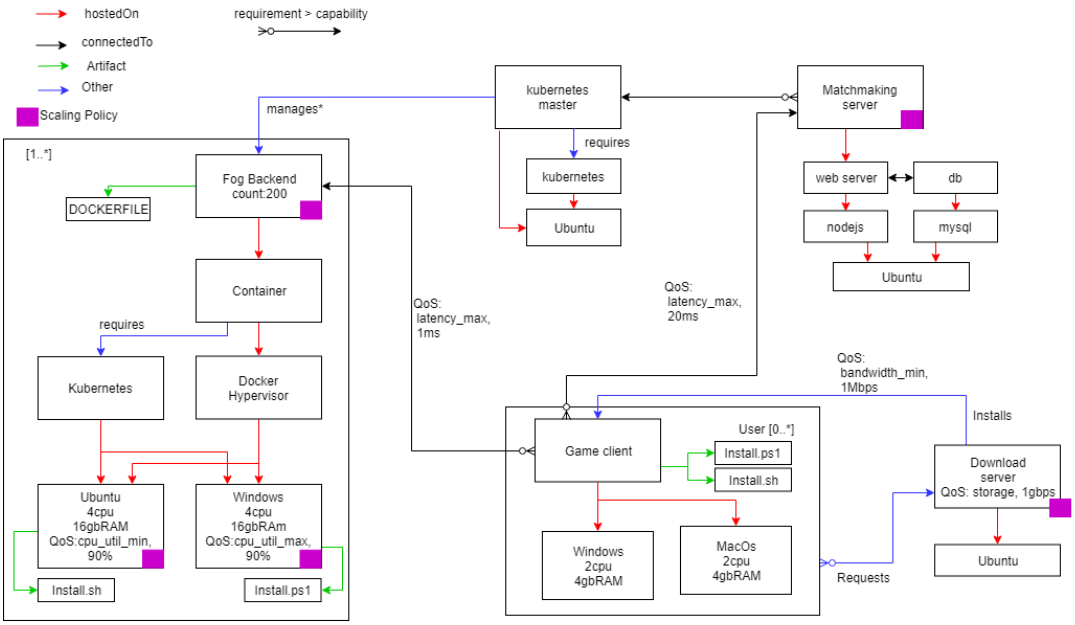

Figure 7.1: TOSCA topology, Gaming example

Initial Deployment

We start by discussion the actions the framework will take during initial deployment. During initial deployment we assume the framework starts with an empty state and has no run-time knowledge about the application.

Pre-Processing The framework starts by processing the description and translates this to an internal representation (IR). The object oriented way of the IR should make it easy to annotate it and inject blocks.

It will then pull the QoS constraints from the description and save this in such a way the infrastructure planner and the monitoring block are able to access it and use the information.

The framework will compare the newly deliver description with the current state of the infrastructure that was generated by the orchestration, configuration and deployment block. But since this state is empty, the entire infrastructure in the description has to be created.

Some block in the description are flagged as not to be created. As we see in figure 7.1 the game client is to be installed by a user, but QoS constraints have to be maintained by the framework. This means the IR should represent that the infrastructure planner can ignore this block, but the monitoring block should still know about it.

Workload Estimation Several workload estimation plugins should be added to framework. We see that multiple blocks are not fully described as in the hardware that is required. Based on the nodes in the TOSCA description, we see that we at least require estimation tools for NodeJS, MySQL, Kubernetes, Docker, Ubuntu and Windows.

We notice that three blocks are only specified that the software is ran on a Linux system. Not only does the framework has to estimate the workload of all the software in the *HostedOn* chain, it also has to inject the compute block at the end of it. After injecting the block, the estimator bubbles up all the *HostedOn* relations and determines and adds up all the workloads that are running on that instance.

Depending on the description the estimation differs per block. For instance, we notice that the *Kubernetes Master* has no scaling policy while the *Download Server* does have one. This means that the *Kubernetes Master* requires the entire application to be ran on a single node, which means the QoS constraints have to be met, or the instance has to be replaced. The *Download* server has a scaling policy, so if the QoS constraints are broken, a new instance is added. This allows the framework to pick the best cost-to-performance machine.

Infrastructure Planner After the workload estimator determined the hardware required to run the software, the planner has to get this hardware. The infrastructure will fetch specifications from the existing resources and get the specifications of the service level resources. Preferably the framework should use the existing resources, but if they do not comply with the QoS constraints, the resource should be requested from the service provider. After finding the best resource to run that part of the composite

application, it should link the instructions to fetch that resource to the plan.

In the fog back-end we see that it can either be hosted on a Windows system or an Ubuntu system. When the infrastructure planner is able to chose between existing nodes running both operating systems, or service level machines, it will select one with the required properties. After the planner has chosen, the instruction generator, is responsible for inserting the right artifacts, belonging to that operating system, into the build plan.

We expect the developer to deliver at least one service provider specification. If he does not do this, the composite application is dependent on the existing infrastructure. This will probably limit the scaling and maintaining capabilities of the framework.

Instruction Generator The infrastructure in this description is general purpose, not requiring specialised tools for orchestration, configuration and deployment. For orchestration Terraform could be used to orchestrate service provider infrastructure. Orchestration existing devices is done by the framework. Configuration is the same for both types of hardware. Ansible can be used as a plugin for configuration but also for deployment. To be able to deploy containers to the *fog back-end* a plugin is required for Kubernetes.

Based on the *HostedOn* property the instruction generator is able to determine that it first needs to create the (virtual) hardware and then install the other nodes on top of that. The order after that is based on other relations.

The instruction generator also has to determine the order in which the commands have to be executed. Let's take a look at the matchmaking server in figure 7.1. The *db* block hosts an endpoint, which is a requirement of the *web server* block. This means the instruction generator has to create the *db* before the *web server*. Roughly speaking, if we look at the capability to requirement relation in *connectedTo*, the block containing the capability should be created first. The order in this design would be as follows, Kubernetes master, Download server, matchmaking server and then the fog back-ends.

Orchestrate, Configure, Deploy The plugins used in this block are tied in with the plugins from the instruction generator. These are mostly just the tools, so that they can be called by the instructions.

Parallel execution is not implemented yet, but in the future this block should be able to execute certain instructions in parallel to speed up the process.

After the block created the infrastructure, configured it and deployed the composite application to it, it will save the state of infrastructure. This state encompasses the infrastructure it created, so not the *client*. The file contains what instances are linked to what, what their properties are and how they are linked together.

Monitoring We see multiple QoS constraints defined in the topology, 2 based on latency, 1 based on bandwidth, and 2 based on CPU utilization. All the machines in the system have monitoring agent installed on it which talks to the monitoring block of the framework.

The latency based QoS constraints require the developer to create a custom policy that includes code to check the latency between instances. The code from this policy will be executed by the monitoring agent that runs on the devices. No plugins are required for this since the additional code should be implemented as an extension on the TOSCA policy type.

For the CPU utilization the agent installed on the device will continuously check the utilization and send this to the monitoring block. If they monitoring block notices that the utilization exceeds the limits set in the QoS constraints, it will trigger an update to the application. The block will update the description and this will go through the rest of the framework again.

Run-time

We will not discuss every block again, only the things that change in them during run-time.

The pre-processing block gets a trigger from the monitoring block with where the error is and what to adjust. The internal representation is updated so that it meets the QoS constraints again. The internal representation should be annotated with what should be updated. This only requires the framework to adjusted what is needed and leave what is fine.

Let's take a look at the download server in figure 7.1. It holds a QoS constraints that says the download speed of the user should be minimally 1 Mbps. It also contains a scaling policy. If the QoS policy is not upheld, the monitoring block will trigger the framework to scale this part of the framework up. The internal representation has to be adjust to support the scaled block. When the extra block is inserted, we also have to insert a load balancer (figure 7.2) which is able to reroute the traffic from a single endpoint to both instances. Using the QoS constraints of the node, a new machine is requested with the provider that contains storage with a speed of 1 Gbps. If the load balancer is present it has

to be updated, and if not a load balancer has to be fetched from the service provider. The framework is responsible for adjusting the internal representation such that the load balancer is inserted correctly. The rest of the framework is responsible for actually realizing it.

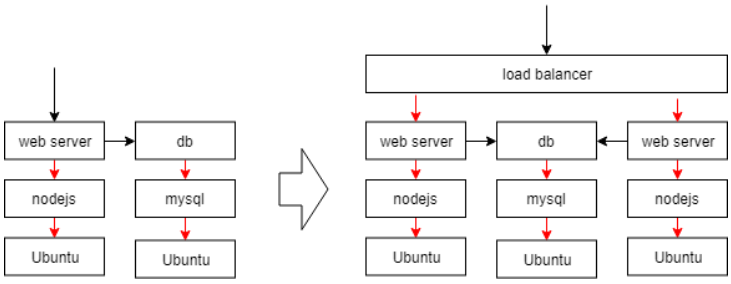

Figure 7.2: Insert Load Balancer

The framework is also responsible for cleaning up the old infrastructure and configurations. This requires it to create some extra instructions. These instructions get executed by the execution block before the new infrastructure is realized.

In figure 7.1 we added purple boxes to blocks that require scaling policies. These policies will determine the number instances that will spread the load of the block. For example we could attach the scaling policy in listing 7.1 to the download server block. During initial deployment the framework would deploy 5 instances and inject a load balancer. During run-time the framework will match the load. If the load is lower than expected, it will scale down the number of instances. If the load is higher, it will scale up the instances.

```
my_scaling_policy_1:
  type: tosca.policy.scaling
  description: Simple node autoscaling
  properties:
    min_instances: 1
    max_instances: 20
    default_instances: 5
    increment: 1
```

Listing 7.1: Scaling policy

Currently we have set no scale down policies in the description. Scale down policies could require the framework to remove entities, but could also require the framework to transform the infrastructure, if for instance a load has to be removed. For example, by adding a "minimum cpu utilization" property we can instruct the framework to scale down the number of instances.

Reconfiguration

Once the developer wants to reconfigure the composite application, he can input a new application description into the framework. The new description is compared with the current state of the framework. The internal representation is updated with the new desired infrastructure and the changes that have to be made. From hereon the process is comparable to the run-time update from the monitoring block.

The new description can contain parts that require new plugins in the framework. The developer is responsible for adding these.

7.2 Experiments

We have created some simple experiments to substantiate certain decisions in the framework.

To substantiate some of the claims and choices we have made we executed some experiments. These experiments are about the execution order of the commands, scaling parts of the composite application, and the placement of the application over the distributed network.

7.2.1 Parallel vs. Serial Execution

The orchestration, configuration and deployment can be done either parallel or serial. We perform an experiment to prove which method would be best for our framework.

During orchestration there are no dependencies on each other. This means that if parallel execution is used, everything can be orchestrated at once. The configuration and deployment does require dependencies to be installed first. In figure 7.3 we see a visual representation of how serial and parallel scheduling would compare. Blue is the time for orchestration, green for configuration and red for deployment.

We will take the gaming use case to prove that parallel execution is faster than serial. The use case requires the execution of 4 blocks, that are dependent on each other. The scheduling order is dependent on the requirements and capabilities of the blocks.

Hypothesis

We expect the parallel execution to be faster than serial execution.

Definitions

```
Block references
KM = Kubernetes Master
DS = Download Server
MM = Matchmaking Server
FB = A single Fog Backend instance
n = number of fog node back-ends

Functions
O(x) = Time it takes to execute orchestration
C(x) = Time it takes to execute configuration
D(x) = Time it takes to execute deployment
OCD(x), OC(x), CD(x) = Sum of the two or three execution times

max(x,y,...) = Return maximum value of the arguments
```

Listing 7.2: Definitions Serial-Parallel

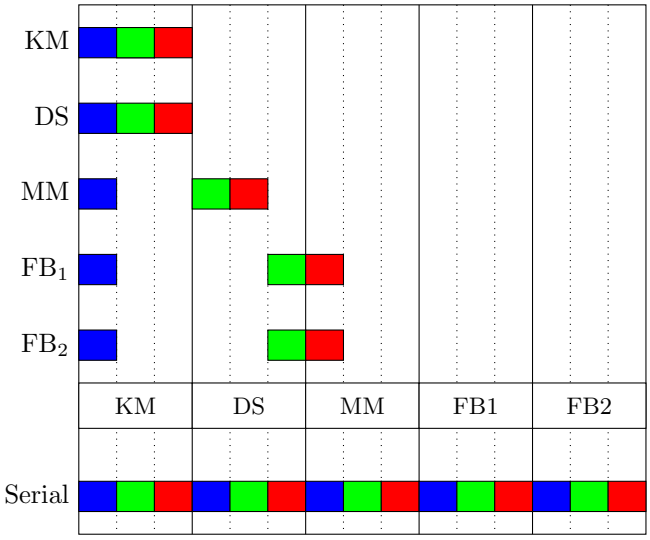

Figure 7.3: Serial and Parallel execution

To prove that parallel execution would be faster for the example case, we formulated the solution using the previously defined definitions.

Proof

```
# parallel time
par = max(OCD(KM), OCD(DS), O(MM), O(FB)) + CD(MM) + CD(FB)

# serial time
ser = OCD(KM) + OCD(DS) + OCD(MM) + (OCD(FB) * n)

# subtract CD(MM) from both
par = max(OCD(KM), OCD(DS), O(MM), O(FB)) + CD(MM) + CD(FB) - CD(MM)
par = max(OCD(KM), OCD(DS), O(MM), O(FB)) + CD(FB)

ser = OCD(KM) + OCD(DS) + OCD(MM) + (OCD(FB) * n) - CD(MM)
ser = OCD(KM) + OCD(DS) + O(MM) + (OCD(FB) * n)
```

```
# subtract CD(FB) from both
par = max(OCD(KM), OCD(DS), O(MM), O(FB)) + CD(FB) - CD(FB)
par = max(OCD(KM), OCD(DS), O(MM), O(FB))

ser = OCD(KM) + OCD(DS) + O(MM) + (OCD(FB) * n) - CD(FB)
ser = OCD(KM) + OCD(DS) + O(MM) + (OCD(FB) * n-1) + O(FB)
ser = (OCD(KM) + OCD(DS) + O(MM) + O(FB)) + (OCD(FB) * n-1)

# max has the same arguments as the sum
max(OCD(KM), OCD(DS), O(MM), O(FB)) < (OCD(KM) + OCD(DS) + O(MM) + O(FB))

# par is smaller than the first part of ser
par < (OCD(KM) + OCD(DS) + O(MM) + O(FB))

# which means
par < ser
```

Listing 7.3: Parallel and serial execution comparison

Since the value of *(OCD(FB) * n-1)* can not be lower than zero, the time it takes to execute all steps will always be greater using the serial method. Depending on the amount of fog back-ends that have to be orchestrated and configured, the serial time will drastically increase.

7.2.2 Dynamic Scaling

We have created a simple experiment to test the influence of dynamic scaling. In our experiment we have a machine that is able to handle a static amount of requests per second. A source sends a static amount of requests to the machine. The source purposely sends more request than the program is able to process. This requires the framework to scale the program up. In figure 7.4 the framework adds a second machine for processing the request on time unit 8.

Hypothesis We that expect by dynamically adding compute instances we are able to adjust live user load.

Method An analytical method is applied where we simulate incoming requests against the capabilities of a system of processing these requests. The method is purely mathematical, using algebra to represent the properties of the system.

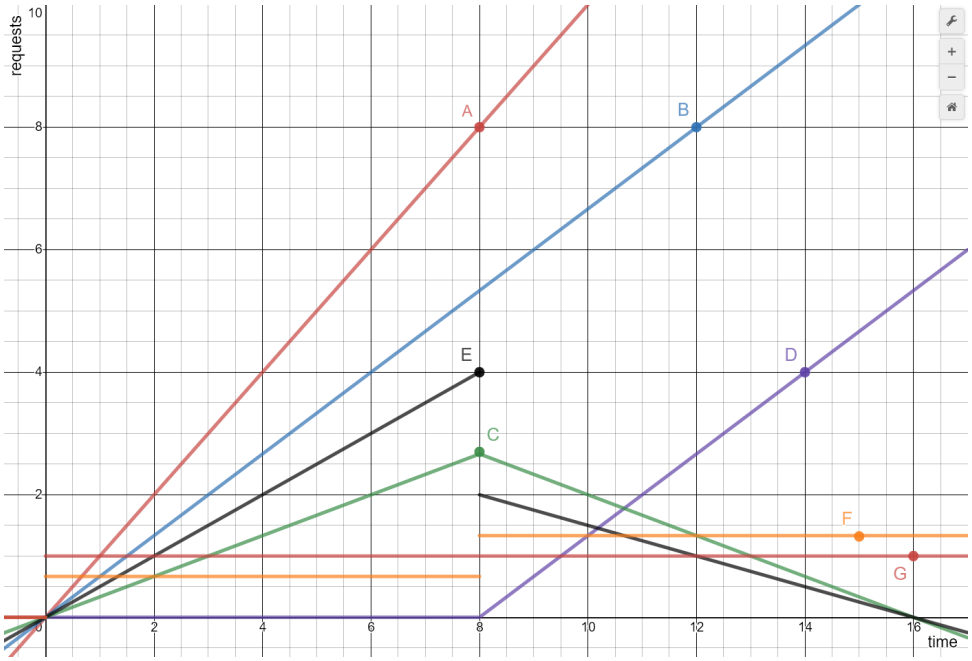

Figure 7.4: Scaling experiment

Definitions

```
G = Incoming requests per second
F = Requests per second the system is able to process
```

```
A = The Incoming requests
B = The amount of requests virtual machine 1 processes
D = The amount of requests virtual machine 2 processes

C = The amount of requests in the queue
E = The amount of time a request is in the queue before it is processed
```

Listing 7.4: Definitions Scaling

In lines F and G we see that the amount of requests coming in per second is greater than the requests that are being processed per second. But after 8 time units the second virtual machine kicks in and doubles to processing capacity.

If we subtract line B from line A we have the difference in processing capacity. This difference is the amount of requests that are not processed, waiting in the queue to be processed. We see this back in line C, the queue increasing in length with a single machine, but decreasing once the second machine joins.

If all the requests are cached, and we assume a queue is used and not a stack, the time in the queue will increase. Once the second machine joins, the wait time will drastically decrease, since requests are handled twice as fast.

In conclusion, we see that dynamically adding instances can mitigate an overabundance of requests. By adding instances during run-time, the developer does not need to include full capacity during initial setup of the application.

7.2.3 Context Aware Deployment

We want to test the influence of context aware deployment of the application. We are using the gaming use case, where every edge node is a user and every fog node a server back-end. We are testing if it matters how we connect the users to the servers. The quality of service constraints we are using for this is the distance between the user and the server. We are testing two methods, 1) connect a user to random server or 2) connect a user to the closest fog node. We do not test the quality of the connection are the actual latency. For simplicity we will keep the experiment to distance.

Hypothesis We expect the selection algorithm that links nodes based on distance should significantly lower the overall distances between edge and fog nodes.

Method We wrote a Python script that allows us to simulate a grid environment. We defined a number of edge nodes and a number fog nodes (back-end services). The Python script places these nodes randomly in a 2d grid. Calculations are performed on the nodes, and the results returned to us. We automated the script in such a way that we are able to run multiple of randomly generated tests, to test the effects on multiple situations.

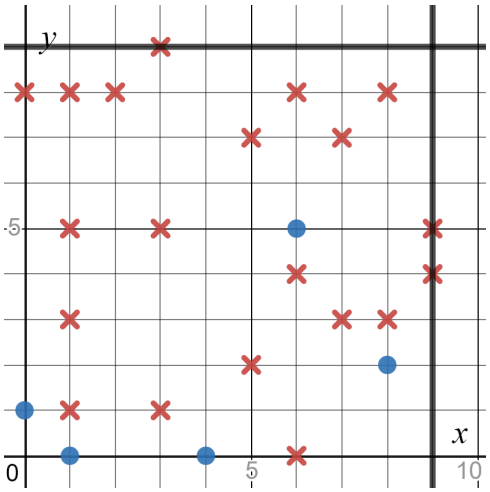

Figure 7.5: Placement Example

The raster used for the experiment is of the size 10 by 10. In the raster there are 20 edge nodes placed randomly and 5 fog nodes placed randomly. Where edge and fog nodes cannot have the same location.

To determine the distance we used the following algorithm. We chose to not use the Pythagorean equation because our is more comparable with real world infrastructure.

$$distance = |x_{edge} - x_{fog}| + |y_{edge} - y_{fog}|$$

We ran 100 tests and plotted the average distance from every node to their connected fog node. This essentially created the a random plot like in figure 7.5 a 100 times. In figure 7.6 we see the result of 100 runs. The Red dots plot the results for random fog node selection, and the blue dots for closest node selection. Every point resembles the average of the distances between the edge nodes and the linked fog node.

```
d = 0
for e in edgeNodes:                # Iterate through all edge nodes in situation
  f = closestFogNode(e)            # Find the closest back-end server
  d = d + distance(e,f)            # Calculate total distance for all the nodes
average = d / length(edgeNodes)    # Calculate average distance between edge and fog nodes
```

Listing 7.5: Situation Distance Algorithm

In order to calculate the average of the distances between all edge and their linked fog node in a single situation, we applied the algorithm in listing 7.5. A single point in figure 7.6 represents the outcome of this algorithm. *Note. In this case for the closest fog node. For random allocation the closestFogNode function can be replaced by a function that finds a random fog node.*

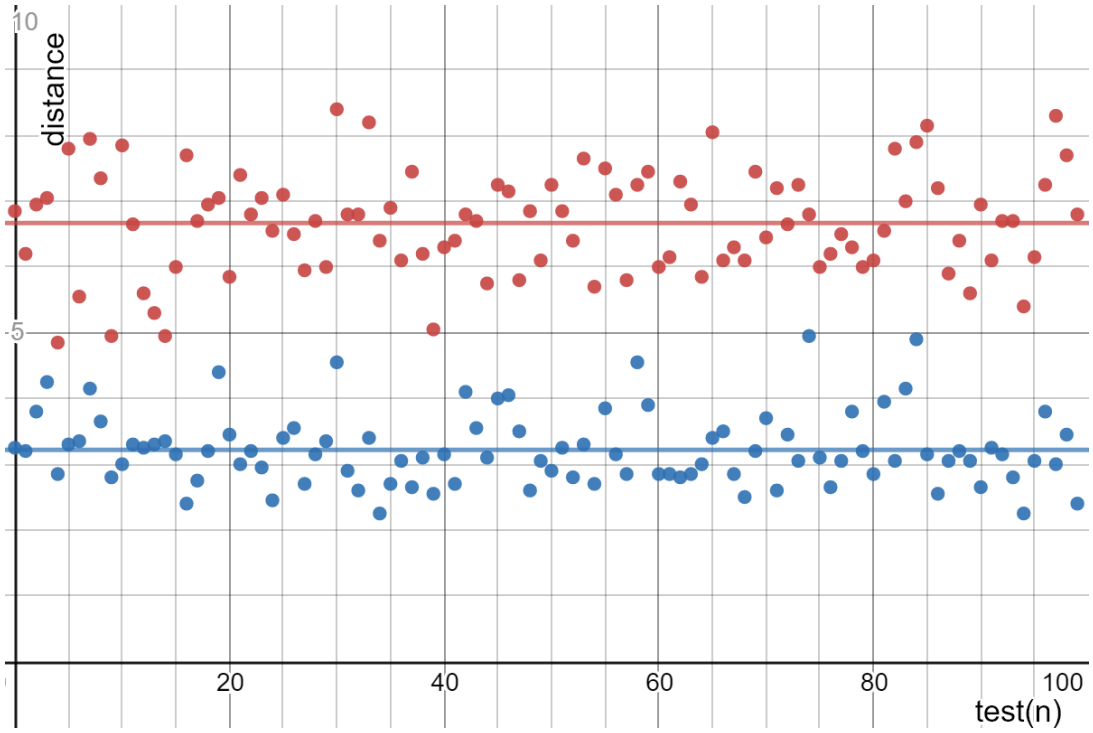

Figure 7.6: Placement Distance

We see that an algorithm that connects edge nodes to the closest fog node, significantly reduces the distance data has to travel. The average distance over the 100 tests for random is *6.6675* with a median of *6.7*. While the average distance over the 100 tests for closest node is *3.21749* with a median of *3.15*.

Chapter 8

Conclusion

In this chapter, we will conclude the work that we have done in this thesis. We will link the work we have done to the research question we proposed in chapter 1. Finally, we will discuss the work that is still open for future research.

8.1 Discussion

Our approach was really practical by combining academic literature with literature from industry and practical experimentation. This allowed us to get a good grasp of the latest developing technologies and relevant literature. The issue with this approach is that we use a lot of material that is not academically reviewed, hurting the validity of the research.

In comparison with the current tools in the DevOps environment, which are mostly single purpose, we proposed a framework that is able to automate across multiple parts of the pipeline. Instead of coding individuals steps of creating a composite application, a developer can use the design of his application to orchestrate, configure and deploy it.

By implementing ambiguity in the design process and workload estimation in the framework, our design allows the framework to make its own decisions for resource selection. DRIP[3] enables cross-cloud provisioning and self-planning of infrastructure but does not give as much flexibility to the framework and the developer, reducing flexibility during run-time. MYST adds support for infrastructure not managed by service providers.

In comparison with the solution proposed by Binz *et al.* [2], our solution adds the run-time reconfiguration and ensuring quality of service. Solutions they proposed could be combined with MYST to prevent implementing existing functionality.

The results of our experiments showed that MYST could increase the quality of experience significantly for users. While making it easier for developers to create and maintain their applications in a changing environment.

8.2 Threats

We have identified some threats that could hurt the validity of the research we have done.

- As the framework is conceptual, the verification is also conceptual. If the framework is implemented, results could differ.
- The results of this study have not been compared to a baseline. A lack of comparison could hurt the relevancy of our solution.
- A use case is susceptible to researcher bias. The use case could be specifically picked to fit the solution. The researches were also fairly biased towards "automating as much as possible" is good. Automation could have negative effects which we did not consider.
- The researchers were relatively inexperienced in the field of DevOps and cloud computing before they started with the research. A lack of experience could greatly influence the assumptions made during the research.
- The research was not shared in the DevOps community. This could end up in the research being less relevant as expected since the community never had the chance to actually confirm they wanted a solution for the problem.

8.3 Conclusion

The purpose of the research was to add a new tool to the DevOps environment. MYST makes it easier for developers to create composite applications across the Cloud, Fog, and Edge. Although most DevOps tools are purposed towards a single task, our tool encompasses a wide range of automation tasks.

We were able to set the first steps towards a framework that enables cloud developers to extend their applications to the fog and edge. The framework abstracts the tasks of orchestration, configuration and deployment while ensuring the quality of service of the applications and their infrastructure. Automation of these tasks will reduce the time and effort spent by developers moving their applications to the fog and edge. By reducing complexity more applications can move their processing closer to where the data is, reducing distance data travels, offload networks and reduce latency.

We flagged several problems that are currently present in the development of composite applications in heterogeneous infrastructures. To solve these problems, we started by finding how composite applications could be described across the development and operational pipeline. We automated the orchestration, configuration, deployment and operational process using a combination of industry tooling and custom software. The framework is verified by proposing a use case that could benefit from distributed computing and implementing them in the framework.

Research Question 1: How to effectively model a distributed application and its underlying heterogeneous virtual infrastructure during the application lifecycle? The developer can describe composite applications using the TOSCA language. The language includes the ability to describe the infrastructure that is required for composite applications. We proposed to extend the language to support heterogeneous devices.

Research Question 2: How to automate the orchestration, configuration and deployment of distributed applications over heterogeneous infrastructures? We proposed a framework which enables the developer to describe their composite applications and automate the orchestration, configuration, and deployment. The framework reduces design effort by making its own decisions on infrastructure planning, resource estimation, and placement. It will convert the description to a set of steps that can be executed by industry tooling.

Research Question 3: How to autonomously maintain the quality of service of a distributed application over heterogeneous virtual infrastructure during run-time? The framework implements a monitoring agent combined with a feedback loop. This allows the framework to continuously check if the composite application complies with the quality of service constraints. And if it does not, update it where needed, so that it does comply.

8.4 Future Work

As the framework is still conceptual, the first step is to implement the framework. This allows us to verify the framework, and run tests on it. Implementing the framework will also result in a better understanding of the technologies used. We propose that the framework should be decomposed, and the individual blocks should be built and tested one by one before they are integrated.

The extensions to the TOSCA language that we proposed still have to be formalized, implemented and verified. Once the language has been defined correctly, visual design tools like winery should be extended to support the extensions. This should improve the usability of our tool during the design phase. The parser and the internal representation have to be extended to support the extensions and the ability to annotate the topology. Once everything is correctly formalized the extensions should be proposed to the entity that maintains the language, to implement it into the TOSCA standard.

All the interfaces between the framework and the plugins have to be formally defined. This contains estimation, service provider and tool plugins.

The data model of the internal representation is roughly defined in the content of the thesis. However the exact specification has to proposed yet.

Although we have discussed creating and updating the infrastructure, we have not discussed how to destroy it. Creating the right instructions, in the right order, to delete the infrastructure should be researched and formalized in future research.

During development we did not always take security concerns into consideration. We have left gaps open like access management. In the future we would like to analyze the security flaws in the framework and the applications deployed with it, and optimize the security.

Currently we have conceptually verified the framework using a use-case, often called scenario testing. In the future we want to test the framework using a scalability analysis, to test how the framework reacts

to larger infrastructures. We would like to verify if the framework is compatible with multiple application architectures. To do this we would like to apply adaptability testing, using peer-to-peer, micro-service architecture, distributed computing and federated analytics.

Acknowledgements

I would like to thank dr. Zhiming Zhao for his guidance. He helped me greatly during the weekly meetings we had and with the feedback he provided.

Thank you to Roger Salhani, my supervisor from KPMG, for giving me advice and helping me in the company.

I would like to thank my colleagues and other interns in the Digital Advisory for making me feel at home and ensuring I enjoyed my time at KPMG.

Thanks to the University of Amsterdam for providing me with all the resources to finish my master. In addition to this, thank you Ana-Maria Oprescu for all the advice during the master, and ensuring the master program is a success.

And I would also like to thank my fellow students from the master, especially Stephan, Andrea, Aynel, Rens and all the other people I worked with.

Bibliography

[1] Ericsson, *Internet of things forecast*, https://www.ericsson.com/en/mobility-report/internet-of-things-forecast.

[2] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, and S. Wagner, "Opentosca–a runtime for tosca-based cloud applications", in *International Conference on Service-Oriented Computing*, Springer, 2013, pp. 692–695.

[3] S. Koulouzis, P. Martin, H. Zhou, Y. Hu, J. Wang, T. Carval, B. Grenier, J. Heikkinen, C. de Laat, and Z. Zhao, "Time-critical data management in clouds: Challenges and a dynamic real-time infrastructure planner (drip) solution", *Concurrency and Computation: Practice and Experience*, e5269, 2019.

[4] M. H. et al., *Terraform*, https://www.terraform.io, 2014.

[5] Openstack, "Heat", *Openstack wiki. Available online: https://wiki.openstack.org/wiki/Heat*,

[6] LogicMonitor, *Cloud vision 2020:the future of the cloud*, https://www.logicmonitor.com/resource/the-future-of-the-cloud-a-cloud-influencers-survey/.

[7] N. Peter, "Fog computing and its real time applications", *International Journal of Emerging Technology and Advanced Engineering*, vol. 5, no. 6, pp. 266–269, 2015.

[8] F. Bonomi, R. Milito, J. Zhu, and S. 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*, ACM, 2012, pp. 13–16.

[9] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges", *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[10] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano, "Devops", *Ieee Software*, vol. 33, no. 3, pp. 94–100, 2016.

[11] S. Neely and S. Stolt, "Continuous delivery? easy! just change everything (well, maybe it is not that easy)", in *2013 Agile Conference*, IEEE, 2013, pp. 121–128.

[12] R. Kohavi, R. M. Henne, and D. Sommerfield, "Practical guide to controlled experiments on the web: Listen to your customers not to the hippo", in *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, ACM, 2007, pp. 959–967.

[13] F. Li, M. Vögler, M. Claeßens, and S. Dustdar, "Towards automated iot application deployment by a cloud-based approach", in *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications*, IEEE, 2013, pp. 61–68.

[14] M.-G. Avram, "Advantages and challenges of adopting cloud computing from an enterprise perspective", *Procedia Technology*, vol. 12, pp. 529–534, 2014.

[15] M. Vögler, J. M. Schleicher, C. Inzinger, and S. Dustdar, "A scalable framework for provisioning large-scale iot deployments", *ACM Transactions on Internet Technology (TOIT)*, vol. 16, no. 2, p. 11, 2016.

[16] L. Bass, I. Weber, and L. Zhu, *DevOps: A software architect's perspective*. Addison-Wesley Professional, 2015.

[17] J. S. Van der Veen, E. Lazovik, M. X. Makkes, and R. J. Meijer, "Deployment strategies for distributed applications on cloud computing infrastructures", in *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, IEEE, vol. 2, 2013, pp. 228–233.

[18] D. Zeng, L. Gu, S. Guo, Z. Cheng, and S. Yu, "Joint optimization of task scheduling and image placement in fog computing supported software-defined embedded system", *IEEE Transactions on Computers*, vol. 65, no. 12, pp. 3702–3712, 2016.

[19] L. Liu, Z. Chang, X. Guo, S. Mao, and T. Ristaniemi, "Multiobjective optimization for computation offloading in fog computing", *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 283–294, 2017.

[20] H. Zhang, Y. Xiao, S. Bu, D. Niyato, F. R. Yu, and Z. Han, "Computing resource allocation in three-tier iot fog networks: A joint optimization approach combining stackelberg game and matching", *IEEE Internet of Things Journal*, vol. 4, no. 5, pp. 1204–1215, 2017.

[21] S. Nastic, S. Sehic, D.-H. Le, H.-L. Truong, and S. Dustdar, "Provisioning software-defined iot cloud systems", in *2014 international conference on future internet of things and cloud*, IEEE, 2014, pp. 288–295.

[22] A. Amazon, "Cloud formation", *Amazon Web Services. AWS CloudFormation. Available online: http://aws.amazon.com/en/cloudformation*,

[23] M. Azure, "Azure resource manager (arm)", *Azure cloud services. Available online: https://docs.microsoft.com/en-us/azure/azure-resource-manager/resource-group-overview*,

[24] A. Carzaniga, A. Fuggetta, R. S. Hall, D. Heimbigner, A. Van Der Hoek, and A. L. Wolf, "A characterization framework for software deployment technologies", COLORADO STATE UNIV FORT COLLINS DEPT OF COMPUTER SCIENCE, Tech. Rep., 1998.

[25] G. D. Abowd, A. K. Dey, P. J. Brown, N. Davies, M. Smith, and P. Steggles, "Towards a better understanding of context and context-awareness", in *International symposium on handheld and ubiquitous computing*, Springer, 1999, pp. 304–307.

[26] I. Satoh, "Context-aware deployment of services in public spaces", in *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*, Springer, 2008, pp. 221–232.

[27] C. Taconet, E. Putrycz, and G. Bernard, "Context aware deployment for mobile users", in *Proceedings 27th Annual International Computer Software and Applications Conference. COMPAC 2003*, IEEE, 2003, pp. 74–81.

[28] T. C. Chieu, A. Mohindra, A. A. Karve, and A. Segal, "Dynamic scaling of web applications in a virtualized cloud computing environment", in *2009 IEEE International Conference on e-Business Engineering*, IEEE, 2009, pp. 281–286.

[29] R. N. Calheiros, R. Ranjan, and R. Buyya, "Virtual machine provisioning based on analytical performance and qos in cloud computing environments", in *2011 International Conference on Parallel Processing*, IEEE, 2011, pp. 295–304.

[30] W. Luk, J. Coutinho, T. Todman, Y. M. Lam, W. Osborne, K. W. Susanto, Q. Liu, and W. Wong, "A high-level compilation toolchain for heterogeneous systems", in *2009 IEEE International SOC Conference (SOCC)*, IEEE, 2009, pp. 9–18.

[31] F. Fjellheim, "Over-the-air deployment of applications in multi-platform environments", in *Australian Software Engineering Conference (ASWEC'06)*, IEEE, 2006, 10–pp.

[32] J. Sendorek, T. Szydlo, M. Windak, and R. Brzoza-Woch, "Fogflow-computation organization for heterogeneous fog computing environments", in *International Conference on Computational Science*, Springer, 2019, pp. 634–647.

[33] C. Boettiger, "An introduction to docker for reproducible research", *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.

[34] N. Chen, Y. Yang, T. Zhang, M.-T. Zhou, X. Luo, and J. K. Zao, "Fog as a service technology", *IEEE Communications Magazine*, vol. 56, no. 11, pp. 95–101, 2018.

[35] A. Bergmayr, U. Breitenbücher, N. Ferry, A. Rossini, A. Solberg, M. Wimmer, G. Kappel, and F. Leymann, "A systematic review of cloud modeling languages", *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, p. 22, 2018.

[36] A. Rossini, "Cloud application modelling and execution language (camel) and the paasage workflow", in *Advances in Service-Oriented and Cloud Computing—Workshops of ESOCC*, vol. 567, 2015, pp. 437–439.

[37] B. Di Martino, G. Cretella, and A. Esposito, "Defining cloud services workflow: A comparison between tosca and openstack hot", in *2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems*, IEEE, 2015, pp. 541–546.

[38] V. Andrikopoulos, A. Reuter, S. G. Sáez, and F. Leymann, "A gentl approach for cloud application topologies", in *European Conference on Service-Oriented and Cloud Computing*, Springer, 2014, pp. 148–159.

[39] E. Brandtzæg, "Cloudml: A dsl for model-based realization of applications in the cloud", Master's thesis, 2012.

[40] D. K. Nguyen, F. Lelli, Y. Taher, M. Parkin, M. P. Papazoglou, and W.-J. van den Heuvel, "Blueprint template support for engineering cloud-based services", in *European Conference on a Service-Based Internet*, Springer, 2011, pp. 26–37.

[41] A. Bergmayr, "An architecture style for cloud application modeling", PhD thesis, Technische Universität Wien, 2016.

[42] T. Binz, G. Breiter, F. Leyman, and T. Spatzier, "Portable cloud services using tosca", *IEEE Internet Computing*, vol. 16, no. 3, pp. 80–85, 2012.

[43] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann, "Tosca: Portable automated deployment and management of cloud applications", in *Advanced Web Services*, Springer, 2014, pp. 527–549.

[44] M. Rutkowski, L. Boutier, and C. Lauwers, "Tosca simple profile in yaml version 1.2", *Tech. Rep.*, 2019.

[45] M. DeHaan, *Ansible*, `https://www.ansible.com`, 2013.

[46] Z. Zhao, P. Martin, J. Wang, A. Taal, A. Jones, I. Taylor, V. Stankovski, I. G. Vega, G. Suciu, A. Ulisses, *et al.*, "Developing and operating time critical applications in clouds: The state of the art and the switch approach", *Procedia Computer Science*, vol. 68, pp. 17–28, 2015.

[47] K. Kritikos, P. Skrzypek, A. Moga, and O. Matei, "Towards the modelling of hybrid cloud applications", *IEEE 12th International Conference on Cloud Computing (CLOUD)*, 2019.

[48] R. Han, M. M. Ghanem, and Y. Guo, "Elastic-tosca: Supporting elasticity of cloud application in tosca", in *CLOUD COMPUTING 2013, The Fourth International Conference on Cloud Computing, GRIDs, and Virtualization*, Citeseer, 2013, pp. 93–100.

[49] M. Fowler and U. Distilled, *A brief guide to the standard object modeling language*, 2003.

[50] G. Cloud, "Google deployment manager", *Google cloud docs. Available online: https://cloud.google.com/deployment-manager/docs/*,

[51] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees", in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, IEEE, 1998, pp. 368–377.

[52] C. Endres, U. Breitenbücher, M. Falkenthal, O. Kopp, F. Leymann, and J. Wettinger, "Declarative vs. imperative: Two modeling patterns for the automated deployment of applications", in *Proceedings of the 9th International Conference on Pervasive Patterns and Applications*, 2017, pp. 22–27.

[53] J. Patel, V. Jindal, I.-L. Yen, F. Bastani, J. Xu, and P. Garraghan, "Workload estimation for improving resource management decisions in the cloud", in *2015 IEEE Twelfth International Symposium on Autonomous Decentralized Systems*, IEEE, 2015, pp. 25–32.

[54] K. Xu, H. Hassanein, G. Takahara, and Q. Wang, "Relay node deployment strategies in heterogeneous wireless sensor networks", *IEEE Transactions on Mobile Computing*, vol. 9, no. 2, pp. 145–159, 2009.

[55] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes", *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[56] N. Naik, "Building a virtual system of systems using docker swarm in multiple clouds", in *2016 IEEE International Symposium on Systems Engineering (ISSE)*, IEEE, 2016, pp. 1–3.

[57] D. Riemer, F. Kaulfersch, R. Hutmacher, and L. Stojanovic, "Streampipes: Solving the challenge with semantic stream processing pipelines", in *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, ACM, 2015, pp. 330–331.

[58] M. L. Howard and W. R. Harper Jr, *Service provider for providing data, applications and services to embedded devices and for facilitating control and monitoring of embedded devices*, US Patent 6,601,086, Jul. 2003.

[59] O. Kopp, T. Binz, U. Breitenbücher, and F. Leymann, "Winery–a modeling tool for tosca-based cloud applications", in *International Conference on Service-Oriented Computing*, Springer, 2013, pp. 700–704.

[60] P. Klint, T. Van Der Storm, and J. Vinju, "Rascal: A domain specific language for source code analysis and manipulation", in *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, IEEE, 2009, pp. 168–177.

[61] M. Calzarossa and G. Serazzi, "Workload characterization: A survey", *Proceedings of the IEEE*, vol. 81, no. 8, pp. 1136–1150, 1993.

[62] M. C. Calzarossa, L. Massari, and D. Tessera, "Workload characterization: A survey revisited", *ACM Computing Surveys (CSUR)*, vol. 48, no. 3, p. 48, 2016.

[63] M. Goldin, *Methods and apparatus for software profiling*, US Patent 7,716,643, May 2010.

[64] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Evaluating the accuracy of java profilers", *ACM Sigplan Notices*, vol. 45, no. 6, pp. 187–197, 2010.

[65] K. Képes, "Konzept und implementierung einer java-komponente zur generierung von ws-bpel 2.0 buildplans für opentosca", B.S. thesis, 2013.