Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelor Thesis No. 308

Provisioning of Docker Containers with TOSCA

Kevin Gödecke

Degree:	Bachelor of Science, Softwaretechnik
Examiner:	Prof. Dr. Dr. h. c. Frank Leymann
Supervisor:	Dipl.-Inf. Santiago Gómez Sáez
Commenced:	December 14, 2015
Completed:	June 14, 2016
CR-Classification:	C.2.4; D.2

Abstract

In order to master the administration and automation problem of distributed applications in the cloud age, topology & orchestration platforms have been established in the past few years. Application topologies and their entire lifecycle can easily be modeled and later on be deployed on various cloud environments. Standards like the Topology and Orchestration Specification for Cloud Applications (TOSCA) help to keep the description of applications platform independent and increase interoperability between components. Another recent paradigm in Cloud Computing is containerized virtualization. The particular and significant popularity of Docker containers was mainly driven be the needs of having less dependencies when moving from development to production environments. The technology around Docker container still evolves very fast and projects to provision and manage Docker container in a automated way have already been adopted by major Cloud providers (e.g. Amazon ECS[1], Azure Container Service[2], Google Container Engine[3]), but lack in topology & orchestration platforms like Cloudify[4] or OpenTOSCA[5]. The cloud provider offerings use container cluster technologies like Apache Mesos or kubernetes under the hood, as the lifecycle management of container is a complicated task. Container cluster technologies provide an easy way to automatically scale, deploy and manage multiple Docker container on various infrastructures.

This thesis aims to enable the support for the deployment of clustered Docker containers using a TOSCA compliant topology & orchestration language and execution environment. More specifically, the Cloudify environment is used as the basis to enable the modeling and deployment of container clusters hosted on kubernetes. By the usage of the Cloudify platform the interoperability with other non-containerized applications and general platform independence is assured, while still taking advantage the container cluster features. The resulting system is able to orchestrate, manage and scale application components individually, regardless of the underlying cloud technology.

[1] https://aws.amazon.com/ecs/
[2] https://azure.microsoft.com/en-us/services/container-service/
[3] https://cloud.google.com/container-engine/
[4] https://getcloudify.org
[5] http://opentosca.org

Zusammenfassung

In den letzten Jahren haben sich neue Herausforderungen durch den steigenden Verwaltungs- und Administrationsaufwand für verteilte Anwendungen in Cloud-Umgebungen ergeben. Um dieser Problematik entgegenzuwirken haben sich Topology & Orchestration Plattformen etabliert, welche es ermöglichen Topologien für Anwendungen und deren gesamte Lebenszyklen einfach zu modellieren und später auf verschieden Cloud Umgebunden zu deployen. Standards wie die Topology and Orchestration Specification for Cloud Applications (TOSCA) ermöglichen es Anwendungen plattformunabhängig zu beschreiben, um ein höheres Maß an Interoperabilität zwischen einzelnen Komponenten zu gewährleisten. Ein weiteres sehr verbreitetes Paradigma im Bereich Cloud Computing stellt die Container-Virtualisierung dar. Die steigende Beliebtheit von Containern, und speziell Docker Containern, ist zurückzuführen auf die Loslösung von sämtlichen Abhängigkeiten durch die Container-Virtualisierung, welche zum Beispiel zu einem barrierefreie Prozess beim Umzug von Entwicklungs- auf Produktionssysteme führt. Insbesondere Technologien rund um Docker Container entwickeln sich aktuell äußert schnell weiter und auch Provider wie Google, Amazon und Microsoft haben bereits Technologien zur automatisierten Provisionierung von Docker Containern eingebunden (z.B. Amazon ECS[6], Azure Container Service[7], Google Container Engine[8]). Da es sich bei der Verwaltung des gesamten Lebenszyklus, sowie der Skalierung von Container um eine komplexe Aufgabe handelt, bauen diese Services auf sogenannte Container Cluster Technologien auf. Container Cluster Technologien ermöglichen es containerisierte Anwendungen zu skalieren, deployen, ohne jegliche Abhängigkeiten zur unterliegenden Infrastruktur.

Diese Arbeit stellt Ansätze vor, um in einer TOSCA-basierten Topologie- & Orchestrierungsumgebung, Docker Container in Clustern zu deployen und auszuführen. Im Detail wird eine Cloudify-Umgebung als Basis für die Bereitstellung von geclusterten Containern durch Kubernetes verwendet. Die Cloudify Plattform ermöglich hierbei die Verbindungen zwischen nicht-containerisierten und containerisierten Anwendungskomponenten, während durch das Container Cluster nach wie vor Skalierung, Load Balancing und Service Discovery zur Verfügung gestellt wird. Das Gesamtsystem ist somit in der Lage Anwendungskomponenten individuell und unabhängig zu skalieren und zu verwalten. Als weitere Teil der Arbeit wird ein Ausblick gegeben, in welche Richtung ein solches System entwickelt werden kann, falls bereits angekündigte Technologien realisiert werden.

[6] https://aws.amazon.com/ecs/
[7] https://azure.microsoft.com/en-us/services/container-service/
[8] https://cloud.google.com/container-engine/

Contents

1 Introduction 13
1.1 Motivation & Problem Statement 14
1.2 Outline 14

2 Fundamentals 17
2.1 Cloud Computing 17
2.1.1 Deployment Models 17
2.1.2 Service Models 18
2.1.3 Providers 20
2.2 Topology & Orchestration 21
2.2.1 Application Topology 22
2.2.2 Orchestration 22
2.2.3 TOSCA 23
2.3 Container Based Virtualization 29
2.3.1 LXC 30
2.3.2 Docker 30
2.4 Container Compatible Cloud Services 32
2.4.1 Amazon Beanstalk 33
2.4.2 Amazon ECS 33
2.4.3 Google Container Engine 34
2.4.4 Microsoft Container Service 36
2.4.5 Comparison 36

3 Related Work 37

4 Specification 41
4.1 Requirements 41
4.1.1 Functional Requirements 41
4.1.2 Non-Functional Requirements 42
4.2 Use Case Description 43
4.3 System Overview 51

5 System Design 55
5.1 Architecture 55
5.1.1 Containers 55
5.1.2 Cloud Application Specification 55
5.1.3 Topology & Orchestration Platform 56
5.1.4 Container Cluster Technology 58

6 Implementation 63

7 Validation 79

8 Conclusion & Future Work 91

Bibliography 93

List of Figures

2.1 Cloud Deployment Models . . . 17
2.2 Cloud Service Models . . . 19
2.3 Example of a simple application topology . . . 22
2.4 Structure of TOSCA service templates [OAS13] . . . 24
2.5 OpenTOSCA ecosystem components [Uni16] . . . 25
2.6 OpenTOSCA architecture [BBH+13] . . . 26
2.7 Lego4TOSCA Node Types [HLNW14] . . . 27
2.8 Cloudify Architectural Overview [Gig16a] . . . 28
2.9 Comparison of Hypervisor-based Virtualization (right) and Container-based Virtualization (left) [Ber14] . . . 29
2.10 A typical Docker deployment workflow . . . 31
2.11 Cloud Provider Container Services Overview . . . 33
2.12 Google Container Engine Components - Kubernetes Overview . . . 35

3.1 Ubernetes Architecture [Kub16c] . . . 38

4.1 Distributed Containerized Application Deployment Use Cases . . . 44
4.2 Architectural System Overview . . . 51

5.1 Container Cluster Environment . . . 60
5.2 Cloudify Kubernetes Cluster Provisioning . . . 61

7.1 Validation Workflow . . . 79
7.2 Cloudify Dashboard on AWS . . . 81
7.3 Application Overview . . . 81
7.4 Cloudify Dashboard - Blueprint Topology Overview . . . 85
7.5 Cloudify Dashboard - Create Deployment Dialog . . . 86
7.6 Cloudify Dashboard - Blueprint Deployment Overview . . . 86
7.7 Cloudify Dashboard - kubectl scale Workflow . . . 88
7.8 Cloudify Dashboard - Regular Scale Workflow . . . 89

List of Tables

4.1 Description of Use Case: Model Distributed Application 45
4.2 Description of Use Case: Deploy Containerized Application 47
4.3 Description of Use Case: Scale Application 48
4.4 Description of Use Case: Monitor Application 49
4.5 Description of Use Case: Tear Down Application 50

5.1 Container Service Config Parameters by Cloud Providers 59

List of Listings

2.1 TOSCA Simple YAML Example [OAS16b] 25
2.2 Distributed web application modeled in YAML for Docker Compose . . . 32

5.1 Cloudify Webserver Example [Gig16b] 57

6.1 etcd Docker command 63
6.2 flannel Docker command 64
6.3 kubernetes Docker command 64
6.4 flannel Docker command for minion host system 65
6.5 kubectl command to start a service [Kub16a] 65
6.6 kubectl command to expose a service [Kub16a] 65
6.7 kubectl command to get service details [Kub16a] 66
6.8 kubectl command to delete a pod or service [Kub16a] 66
6.9 Docker installation using python and subprocess module 72
6.10 Disable iptables and ip-masq for Docker Daemon 72
6.11 Run ETCD in Docker Container 73
6.12 Docker installation using python and subprocess module 73
6.13 Run flannel using Docker image and save container ID 73
6.14 Extract the flannel subnet configuration from running flannel Docker container 74
6.15 Removal of the default docker0 bridge 74
6.16 Start the kubernetes master as Docker container 75
6.17 Install flannel on worker node 76
6.18 Install flannel on worker node 76
6.19 Run kubelet on worker node with master node IP 77
6.20 Run kubelet proxy service 77

7.1 Bootstrap Cloudify Manager VM on AWS 80
7.2 MongoDB Blueprint Modeling 82
7.3 Kubernetes Cluster Modeling 83
7.4 Application Service Modeling 84
7.5 kubernetes API Pods endpoint 87

1 Introduction

One of the reasons Cloud Computing has emerged so quickly in the last couple of years is the immense benefit of being able to run applications in a distributed manner. Major Cloud providers like Amazon[1], Microsoft[2] and Google[3] have extended their offerings in the Platform-as-a-Service sector (PaaS) to provide customers with services to make it even easier and faster to deploy distributed containerized applications without taking care of the underlying infrastructure. The provisioning support for container-based virtualization approaches like Docker has been adapted by most Cloud providers (e.g. Amazon ECS[4]) to further simplify the deployment process and enable users to automate processes like scaling and load balancing. Additionally recently evolved container cluster technologies like Apache Mesos or kubernetes, which are partly used by major cloud providers in their PaaS offerings, also provide interesting approaches on the way to keeping affords of deployment and maintenance to a minimum level.

In order to manage and describe such highly sophisticated applications and services it requires standards and frameworks to build a common ground between cloud providers. Standards like the Topology and Orchestration Specification for Cloud Applications (TOSCA)[OAS16a] have been evolved to describe and manage topologies and orchestration of applications in a cloud platform portable way and thus enhance the portability of cloud applications through describing their lifecycle, requirements and relationships.

The recent trend of container-based virtualization techniques has helped developers to further close the gap between development and production environments. Nevertheless the ability to enable TOSCA specified applications to be provisioned with container-based virtualization techniques like Docker[5] would help significantly to improve the interoperability of distributed applications in the cloud and will be the subject of this thesis.

[1] https://aws.amazon.com/
[2] https://azure.microsoft.com/en-us/services/cloud-services/
[3] https://cloud.google.com/
[4] https://aws.amazon.com/ecs
[5] https://docs.docker.com/

1.1 Motivation & Problem Statement

In todays IT and business world, applications become more and more complex. Different application components run on different platforms, require individual hardware and might even have special security or privacy constraints. Cloud computing in general provides a solution to parts of the stated problem, such as providing the underlying resources, but raises new problems like the organization and management of the application components itself. The containing components might have dependencies between one another or have to fulfill other platform specific requirements before being able to start. In order to fully automate entire lifecycles a topology and orchestration specification as well an execution engine is required. The Topology and Orchestration Specification for Cloud Applications (TOSCA) provides such a standard and is already well established in the industry. While topology describes how the application and its components are modeled in a platform neutral way, the orchestration describes the automation of deployment and other management tasks. By specifying the topology and orchestration details it is possible to automate the entire lifecycle of cloud applications.

The recent establishment of containerized virtualization approaches also opens new challenges for the orchestration of distributed cloud applications. While part of the topology & orchestration description is the creation of new node types, which also define the custom behavior during management operations, the design and implementation of such types for container and especially Docker container has only recently been tried to approach. Furthermore the technology around Docker containers is still evolving very quickly and new technologies like container cluster technologies appear very frequently. In order to fully take advantage of such new technologies in topology & orchestration platforms, it requires to extend and define the behavior and structure of these newly adopted technologies in a platform independent way, preferably using standards like TOSCA. The utilization of such practices to provision containerized, distributed applications with respect to a TOSCA based orchestration platform will help to automate processes for cloud applications and will be the subject of this thesis.

1.2 Outline

This thesis is structured in the following way:

Chapter 1 – Introduction: Brief introduction to the topic and disclosure of its motivation and problem statement.

Chapter 2 – Fundamentals: This chapter covers the fundamentals that are required during the course of this thesis. The basics of modern Cloud Computing will

be discussed, as well as the Topology and Orchestration of applications in cloud environments. This further leads to Container-based virtualization techniques including Container Cluster Technologies.

Chapter 3 – Related Work: There are numerous approaches that have similar intentions or provide approaches on other platform or ecosystems. This chapter discusses related work that interferes with the overall topic of this thesis.

Chapter 4 – Specification: This chapter specifies the requirements for the system, followed by the description of several use-cases as well as a high-level architectural system overview.

Chapter 5 – System Design: The overall system design details are covered and explained in this chapter. This covers the technology-specific architecture as well the connection between them.

Chapter 6 – Implementation: This chapter outlines the implementation of the proposed system and its components.

Chapter 7 – Validation This chapter describes a use-case which validates the introduced system and its requirements. The execution of the use-case will be covered step-by-step.

Chapter 8 – Conclusion & Future Work Draws a conclusion of this thesis and provides an outlook on the future use of this work and related topics.

2 Fundamentals

2.1 Cloud Computing

Cloud Computing and Virtualization are the key principles that made it possible to provide an even easier network access to a large amount of resources. Resources like networks, servers, storage, applications or even services are maintained in a big pool and can be allocated and assigned dynamically [BGPV12]. Through the dynamic allocation the underlying hardware and infrastructure is utilized in the best possible way [VRCL08]. Through virtualization physical servers can be divided into multiple virtual machines by using a so called Hypervisor. Each virtual machine has resources assigned to it and runs a separate Operating System. The Cloud Computing paradigm has been widely adopted in both research and industry domains due to the efficiency of scaling and the possibilities to deploy and host applications in a distributed manner.

2.1.1 Deployment Models

A Deployment Model describes how the physical infrastructure of a Cloud is hosted and deployed.

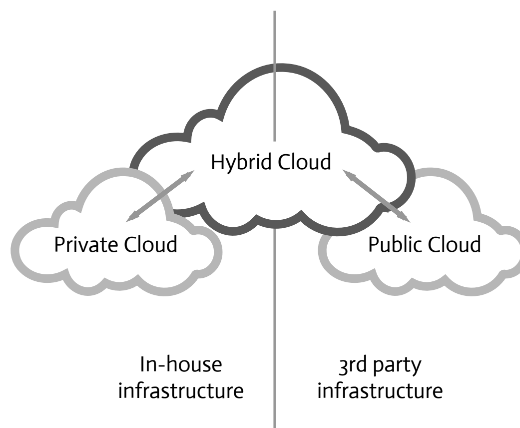

Figure 2.1: Cloud Deployment Models

1. Public Cloud Model
   The Public Cloud Model provides an infrastructure which is accessible by the public and lets multiple users share the same infrastructure to reach maximum efficiency and reduce overhead [BGPV12]. The usage is mainly billed by a monthly pay-per-usage model. The infrastructure is maintained by the provider, who also offers management tools for users to control their environment and provision new resources within very few clicks. The resources and services vary between providers and reach from Virtual Machines to Object Storage solutions to Mobile Services like Push Notifications. Popular providers including Amazon AWS, Google Cloud Platform and Microsoft Azure cover most of the market share and offer a huge variety of services.
2. Private Cloud Model
   The Private Cloud Model restricts public access and is mainly a result of security and privacy constraints within companies or organizations. The infrastructure is operated solely for single organization that can be divided into internal customers [BGPV12]. Interfaces similar to public cloud offerings are provided to employees and other eligible parties. A significant downside of a Private Cloud is the time and money intense maintenance which makes it only suitable for large scale enterprises and companies.
3. Hybrid Cloud Model
   Multiple cloud infrastructures can be combined into what is often referred to as the Hybrid Cloud Model [BGPV12]. This approach is often chosen when it is important to keep only a certain amount of data within a private data center while non-sensitive data can be stored off-shore. An other use case is to extend the resources of a private cloud infrastructure in times of traffic peaks or unpredictable bursts. Computing capacity and resources can be increased through a connection to a public cloud provider. Bandwidth limitations play an important role when extending a Private Cloud and need to be considered closely.

2.1.2 Service Models

Regardless of the selected deployment model the services offered within a cloud environment can be subdivided into different abstraction layers that have emerged over the last couple of years.

- Infrastructure-as-a-Service (IaaS) Model
  Clients get direct access to infrastructural resources like Virtual Machines (VMs),

databases or networks. The underlying physical hardware is fully virtualized to maximize elasticity and scaleability [DWC10]. This also means that new resources can be provisioned within a very short turnaround time in order to face temporary and unexpected workloads. One of the most popular services is Amazon Elastic Compute Cloud (EC2)[1].

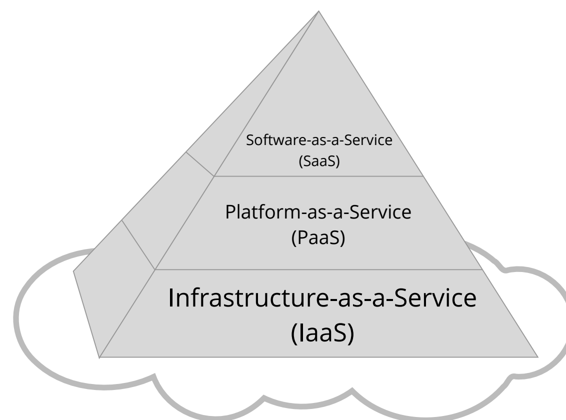

Figure 2.2: Cloud Service Models

- Platform-as-a-Service (PaaS) Model
  PaaS defines services which allow users to create, manage and run applications without maintaining the underlying infrastructure throughout their entire lifecycle. Clients are supplied with an application hosting environment that can be configured and be used automatically scale [BGPV12]. Examples include Amazon Beanstalk[2] and Microsoft Azure Container Service[3].
- Software-as-a-Service (SaaS) Model
  SaaS provides the highest abstraction layer in the service model section. Software hosted on the Cloud infrastructure is offered to the user. Clients can access the software through an API or a interfaces like a WebGUI [FLMS11]. The provider takes care of hosting the entire application. The infrastructure as well as the platform is fully managed by the provider [BGPV12]. Popular examples include Google Docs[4] or SalesForce[5].

[1] https://aws.amazon.com/ec2/
[2] https://aws.amazon.com/en/elasticbeanstalk/
[3] https://azure.microsoft.com/en-us/services/container-service/
[4] https://docs.google.com/
[5] https://salesforce.com/

2.1.3 Providers

Cloud providers differ not only in deployment models but also in their services and service models. This thesis focuses on three major Cloud providers (Amazon AWS, Microsoft Azure, Google Cloud Platform) as they cover over 42% of the entire IaaS market share in 2014 [Sta16].

Amazon AWS

Amazon started its Cloud offerings in 2006 as a subsidiary of Amazon.com with just infrastructure services and has ever since constantly added services in the IaaS and PaaS sector. The first services offered were Amazon Elastic Compute Cloud (EC2), Simple Storage Service (S3) and Simple Queue Service (SQS)[Gar07]. Amazon EC2 provides scalable Virtual Machines based on the Xen virtualization, whereas S3 is an affordable object storage. Nowadays Amazon has spread its data centers across the globe with more than 12 geographical regions and 33 availability zones [Ama16].

A few services have claimed more and more popularity especially with regards the deployment of distributed applications. With Amazon Beanstalk a PaaS service is provided that lets users deploy scalable web applications and services with a few clicks. Beanstalk uses EC2 resources under the hood and as the service per se is free of charge, customers get billed for the usage of EC2 resources. Load-Balancing, scaling and monitoring is fully handled by Amazon, without any needs of maintaining the underlying infrastructure.

One of the very recently added services is Amazon ECS (EC2 Container Service) which lets you easily deploy containerized Docker applications. Users can manage and run these applications either from the web dashboard or through multiple SDKs or CLIs. The containerized Docker containers are being automatically deployed to Amazon EC2 instances which run a custom ECS agent that is being handled by the service. As of most of the Amazon services it integrates really well with any other offerings like for example Elastic Load Balancing.

Microsoft Azure

Microsoft Azure is a cloud platform by Microsoft started in 2010 to help developers build applications that are highly scalable and flexible. The cloud infrastructure is based on the Microsoft Azure Hypervisor (WAH) technology [QLDG09] and is often referred to as an classic example of a PaaS platform [Sav15]. IaaS is an other part of their offering

and doesn't differ much from the competitors. The wide range of PaaS solutions that integrate especially really well with the .NET platform really set Azure apart.

Microsoft Azure Cloud Services lets you deploy your application in a fully automated way without taking care of the infrastructure, supporting Node.JS, PHP, Java, .NET Python and Ruby [Tul13].

Google Cloud Platform

The Google Cloud Platform provides multiple options in the PaaS and IaaS sector, giving the user different control over the environment. The Compute offerings can be divided into the following three services.

Google App Engine is a PaaS that provides support including Python, Java, PHP and Go to directly deploy and run applications, while letting Google take care of hosting, maintaining and scaling the infrastructure.

Google Container Engine was built due to the increasing impact of container-based development within the last couple of years. Developers can simply deploy their containerized applications to a cluster that is running on Google Compute Engine instances with Kubernetes[6].

Google Compute Engine is a IaaS that provides among others unmanaged Virtual Machines, while giving you control of the operating system and the environment.

2.2 Topology & Orchestration

Even though more and more cloud providers enter the market with all of them introducing different offerings and services, the description of distributed application remains the same and thus forms a key principle in the process of providing interoperability between different cloud providers.

[6] http://kubernetes.io/docs/whatisk8s/

2.2.1 Application Topology

The term network topology is used to describe how resources in a computer network are organized and connected [Dav06]. The topology of distributed applications in Cloud environments describes how different parts of the application are structured and linked to one another. Distributed applications and in particular cloud services and their topologies need to be described in a proper way in order to take full advantage of cloud environments and later on provide interoperability between providers.

For example the Topology of a simple Flask web applications might consist of a MongoDB database and the web application itself. The application needs to run on a web server like Apache or Nginx, which additionally requires to run Pythn in order to execute the application per se. The connection towards the MongoDB database describes a necessary relationship to provide the application with data and the ability to persist data.

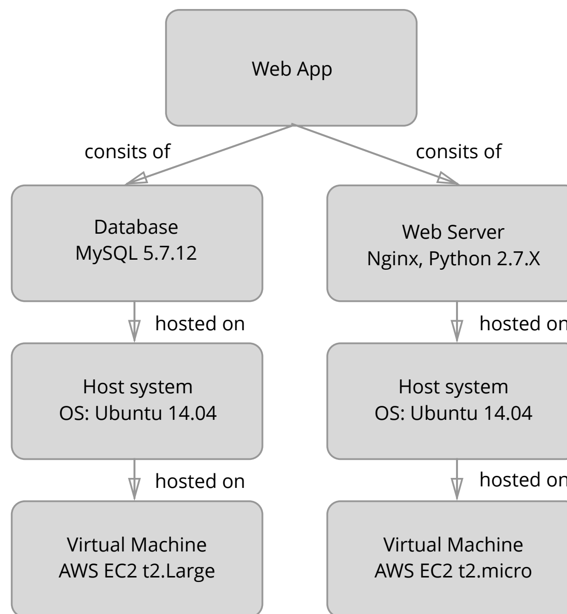

Figure 2.3: Example of a simple application topology

2.2.2 Orchestration

Oxford dictionary defines orchestrating as "to plan or coordinate the elements of (a situation) to produce a desired effect, especially surreptitiously)" [Ste10]. In terms

of Cloud Computing orchestration describes the automation of creation, management and manipulation of cloud resources like compute, storage and network in order to realize a user requests [LMVF11]. These requests can differ significantly and are mostly modeled by complex workflows that handle the provisioning of underlying resources. Orchestration is not only limited to delivering a service utilizing a particular provider, it rather strives to utilize all possible providers to achieve the desired goal in the best possible way. As part of this the orchestration also includes the execution of workflows on different providers, which due to their internal service logic and vendor-specific gateways and APIs describes an other important challenge.

2.2.3 TOSCA

The Topology and Orchestration Specification for Cloud Applications (TOSCA) is an open standard for distributed cloud based applications. TOSCA was established in 2014 by the OASIS committee and is being used widely to describe application topologies in a platform independent way. This solves the common vendor-lock issue and additionally provides a way to connect components even across different cloud providers [OAS13].

Besides the definition and description of relations between components of cloud applications, TOSCA also defines their life cycle. Life cycle definitions might later on be used by an orchestration engine to execute processes like deployment, scaling, termination or restart of the distributed application [BBLS12].

TOSCA is structured in a very simple and straightforward way. Service Templates describe topology as well as orchestration aspects through a specific TOSCA language. Figure 2.4 shows that each Service Template consists of a Topology Template, different Node and Relationship Types and Plans.

Topology Templates are further subdivided into Node and Relationship Templates. Node Templates have to be defined for different topology layers. A web application might have a Node Type (an instance of a Node Template) for a PHP application, which then is based on a Node Type that represents a Web Server, hosted on an other Node Type that models a cloud-hosted VM. Relationship Templates model connections between and to other Node Templates. Both Node Types as well as Relationship Types can have different properties and interfaces [BBH+13].

To describe entire workflows, TOSCA specifies so called Plans for management operations of components. Plans describe typically processes like termination and start and are based on existing workflow languages like BPEL and BPMN.

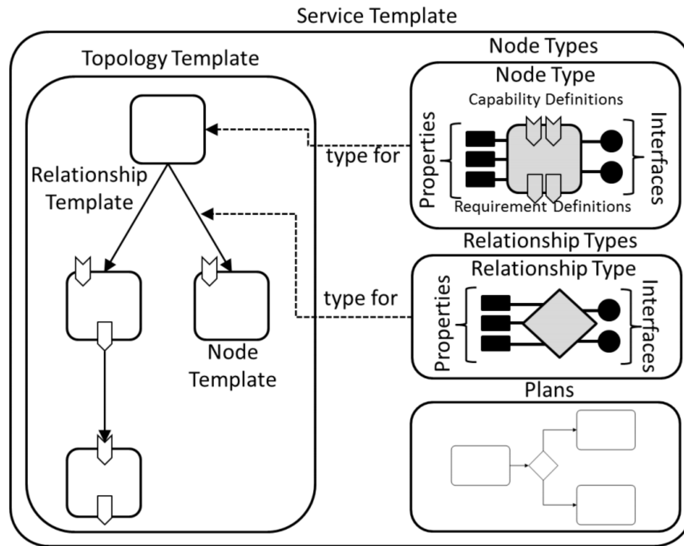

Figure 2.4: Structure of TOSCA service templates [OAS13]

TOSCA Simple YAML Profiles

OASIS specified a Simple YAML Profile format to further simplify the way Service Templates and thus Cloud Applications are defined and specified. The ultimate goal is that over time through community contributions a repository of existing node and relationship types will grow to help other users speed up the specification process. This will also enable users to provide software specific scripts that start for example a service. The format expects to build on top of a few base types, which later one can be used in a hierarchical way [OAS16b].

Example of base types could include a Compute Node or a DBMS Node. Figure 2.1 shows an exemplary definition which builds on top of a base node type called `tosca.nodes.Compute` defined in the `tosca_simple_yaml_1_0`. `my_server` then has specific details defined like CPU, RAM or even disk space.

```
tosca_definitions_version: tosca_simple_yaml_1_0
description: Template for deploying a single server with predefined properties.
topology_template:
  node_templates:
    my_server:
      type: tosca.nodes.Compute
      capabilities:
        # Host container properties
       host:
         properties:
           num_cpus: 1
           disk_size: 10 GB
           mem_size: 4096 MB
```

Listing 2.1: TOSCA Simple YAML Example [OAS16b]

OpenTOSCA Ecosystem

OpenTOSCA is a open source TOSCA-based ecosystem developed by the University of Stuttgart [Uni16]. Applications are provisioned in an imperative manner, meaning that applications are modeled through plans, which are similar to blueprints in Cloudify.

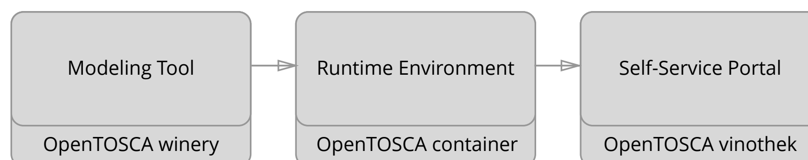

Figure 2.5: OpenTOSCA ecosystem components [Uni16]

As shown in Figure 2.5 the OpenTOSCA ecosystem is separated in three key components. OpenTOSCA Container provides a runtime engine to execute management and deployment operations by using the CSAR (Cloud Service Archive) format. CSAR files include information to fully deploy and instantiate distributed cloud applications [Tre13]. Among other things a CSAR archive can include Service Templates, Node Types and Relationships. On top the winery adds a modeling tool for topologies and management plans and the so-called Vinothek enables users to provision new applications across cloud providers through a web interface.

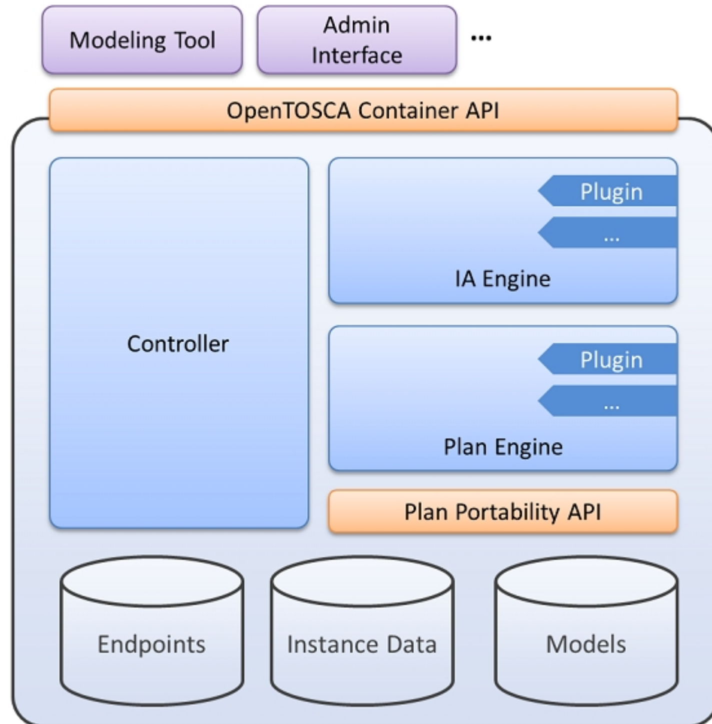

Figure 2.6: OpenTOSCA architecture [BBH+13]

Figure 2.6 gives a more detailed architectural overview of the OpenTOSCA container runtime component. In particular the runtime engine takes care of the execution of management operations, plan execution and application state management [BBH+13]. If needed the CSAR format can be extended with Implementation Artifacts in order to provide custom management operations. The Implementation Artifact Engine is generally responsible for the custom operations and for making them available in management plans. All plans are processed and validated by the Plan Engine and serve as a descriptor for management operations. A CSAR file can contain multiple plans which serve different needs like scaling, instantiating or similar. The controller component provides a general API access to add and remove CSAR files, whereas the remaining components and functions as an overlaying controller of the other components.

Lego4TOSCA

Lego4TOSCA builds on top of TOSCA and provides a generic format to describe reusable TOSCA node types. Figure 2.7 shows the basic nodes types of Lego4TOSCA called Building blocks. They can provided parameters on different hierarchical levels from infrastructural, which might include the configuration of an EC2 instance, to an operation

system level, which defines the operating system per se and certain runtime parameters [HLNW14].

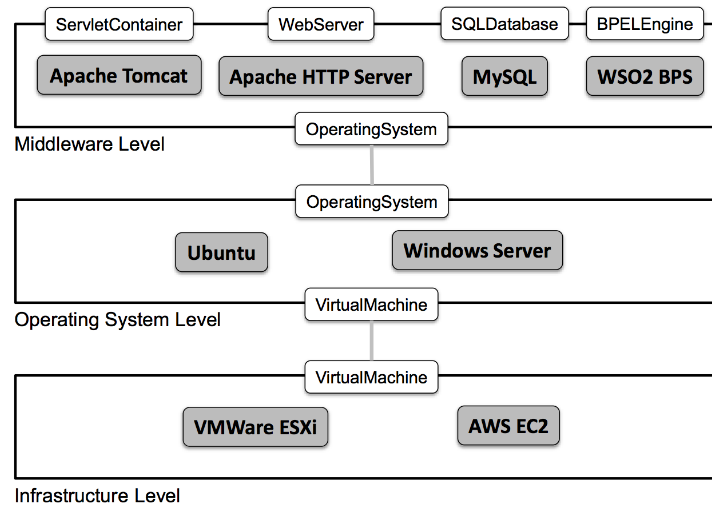

Figure 2.7: Lego4TOSCA Node Types [HLNW14]

Cloudify

Cloudify is an open-source platform to orchestrate and manage distributed cloud applications using the TOSCA specification. Similar to OpenTOSCA Cloudify provides tools to model applications and services, as well as a provisioning and orchestration engine. Through the usage of TOSCA the entire platform can provision multi-tier applications in a platform independent fashion. This gives users the possibility to even span application components across multiple providers [Gig16e].

Cloudify describes and provides tools to model topologies and orchestration configurations in blueprints which are defined using the YAML format and a specific Cloudify DSL (Domain Specific Language), which is derived from the TOSCA Simple YAML format specified by the OASIS committee. Blueprints include the application topology, workflows and policies and describe the entire application lifecycle. Additionally blueprints include runtime related parameters like URLs, usernames or passwords.

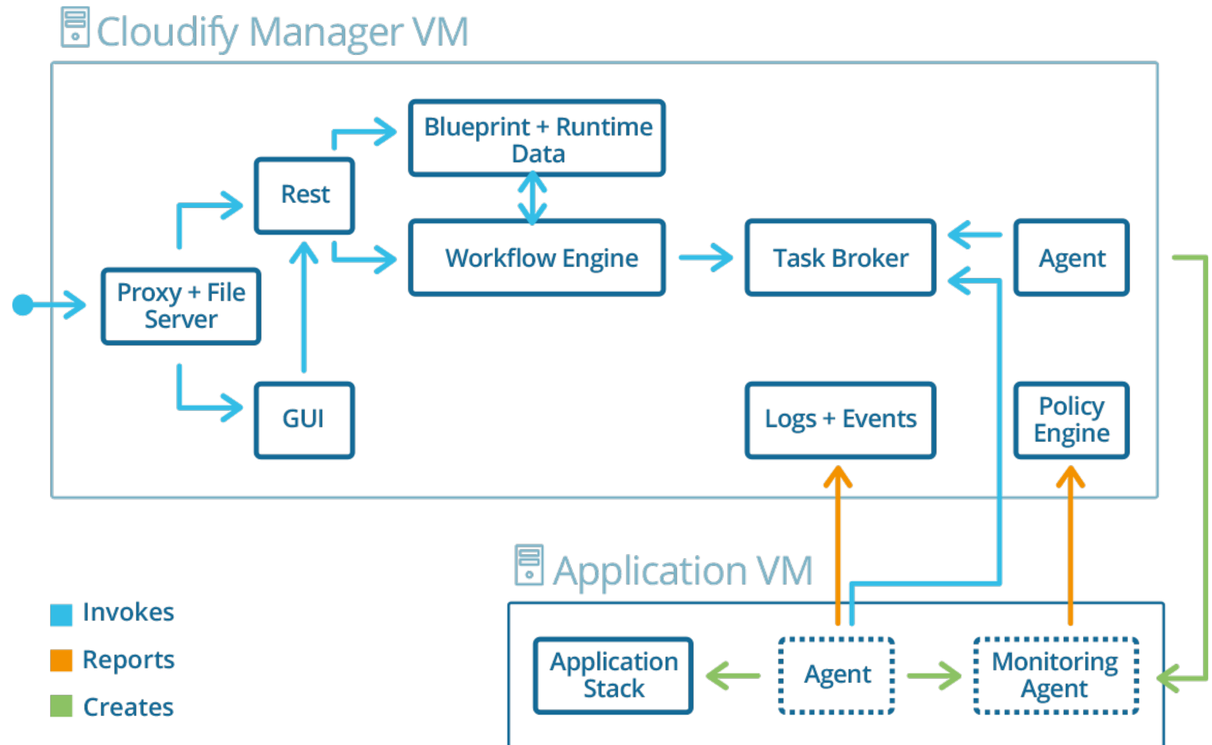

Figure 2.8: Cloudify Architectural Overview [Gig16a]

In figure 2.8 a general overview of the Cloudify Architecture is given.

Every Cloudify environment consists of a Cloudify Manager VM and multiple Application VMs. The Cloudify Manager VM provides access to a GUI as well as a Command Line Interface, which enables the user to manage application blueprints and provision and orchestrate them. The GUI further provides access to system and application monitoring and logging features as well as management tools to administrate deployed applications and blueprints.

Each provisioning task is described in Cloudify workflows, which are part of the modeling and can consist of custom Python and bash scripts. The Cloudify workflow engine, located on the Manager VM, parses the provided YAML application blueprint files and manages the orchestration tasks and their timing through the Task Broker. The Task Broker is built on top the Celery tasks broker, which is a asynchronous task queue based on distributed message passing system [Cel16] responsible for the distributed orchestration of tasks. The provisioning of common infrastructural resources like VMs or network interfaces is then passed onto a manager agent, which is also located within the Manager VM itself. After a successful deployment a general and a separate monitoring agent on the application VM report back to the central Manager VM in order for it to propagate the information to the user interface.

2.3 Container Based Virtualization

During the development process of an application developers usually work in their own development environment, which typically differs significantly from the environment that is later on used for production deployment. This directly leads to the problem of either adjusting the application to match the production environment or adjusting the deployment environment to match the application. Container based solutions promise a solution for this lack in the develop to deploy workflow, which is sometimes also referred to as "Dependency Hell" [Boe15]. On top this also provides a light weight alternative to full virtualization approaches [Mer14].

The main difference between virtualization and container-based virtualization is that it doesn't fully emulate a hardware layer and thus uses a different architectural virtualization approach by sharing the underlying operating system [DRK14] (also see Figure 2.9). This means generally less isolation but significantly lower overhead in resources like storage and CPU.

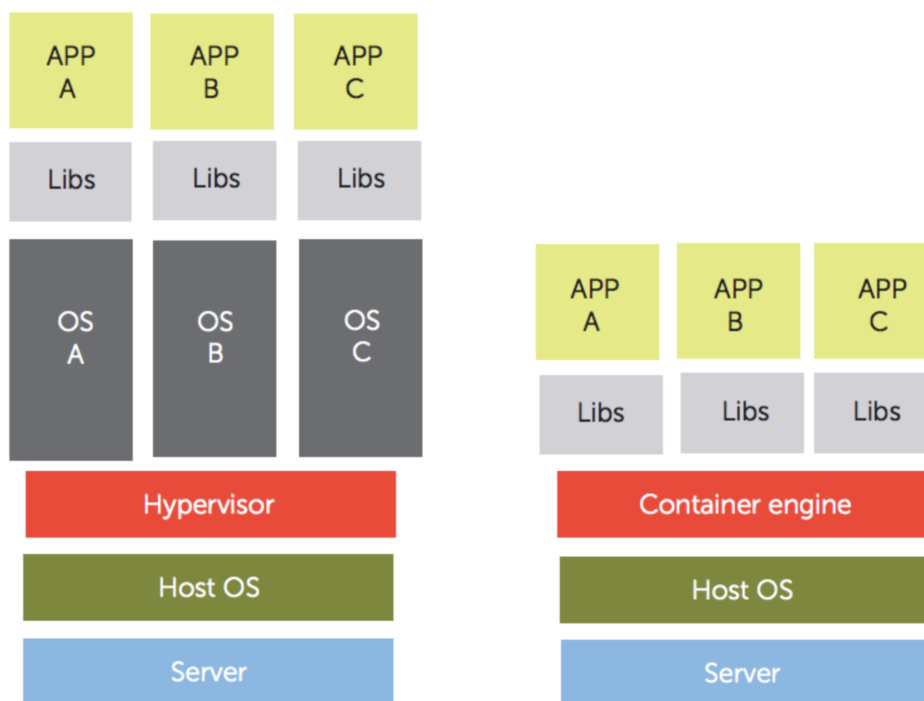

Figure 2.9: Comparison of Hypervisor-based Virtualization (right) and Container-based Virtualization (left) [Ber14]

2.3.1 LXC

LXC (i.e. LinuX Containers) are a container based virtualization method that uses core features of the Linux kernel like cgroups and kernel namespaces in order to isolate containers from one another. By accomplishing a detachment of users, processes and network management, LXC achieves a separation within a single host operating system.

The Linux kernel namespace provides a separation of the user management for every container, so that root privileges and user rights don't interfere and the host operating system can operate independently. A virtual network provides an abstraction layer for network interfaces of the container and an additional process namespace is responsible for isolating and managing processes on a container basis [XNR+13].

2.3.2 Docker

Docker containers are isolated packages of software that ship with their dependencies and config files already included. This makes it possible to run them in different environments out of the box.

The open source project uses different Linux-kernels and is build on the foundation of LXC features to achieve virtualization without setting up an entire operating system for every container [Boe15]. Each Docker container gets created by a Docker base image. New Containers can share base images (e.g. Ubuntu) and simply store new versions of files that get modified. This so called copy-on-write process is part of the AuFS (Advanced Multi-Layered Unification Filesystem) that Docker uses and is very efficient and light. When creating a new Docker image all steps taken are stored in the image and can then later on be used to re-create containers [Ber14].

Additional Dockerfiles can include instructions on what needs to be installed on top of a base image and can be used as a script to initially bring up the container and include environmental instructions like persistent storage, port mappings and further.

Docker Hub

Docker Hub is a central repository for Docker Images hosted by Docker, Inc. It enables users to collaborate on application and service container configuration, while providing features like automated builds, webhooks and more.

Typically Dockerfiles include a reference to Docker Hub in order to pull the required image and optionally resources. The image can be specified either directly through the repository/image name or via a Dockerfile. A Dockerfiles can additionally include further

instruction on the deployment process. After pulling the image successfully Docker can deploy the application. Figure 2.10 shows a workflow using Docker Hub as the Docker image registry.

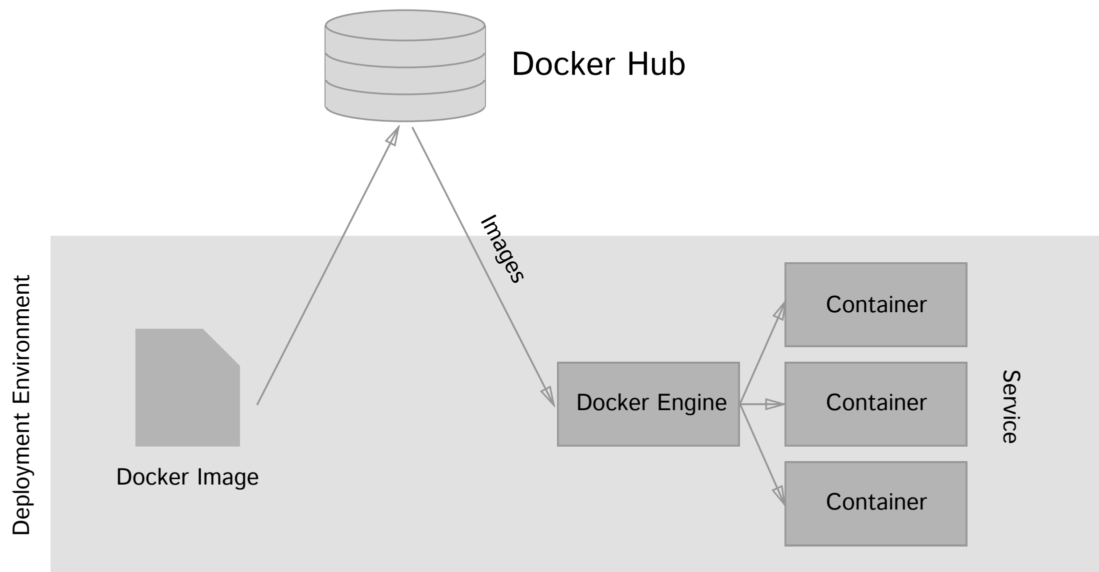

Figure 2.10: A typical Docker deployment workflow

Docker Swarm & Docker Compose

Orchestration of multiple Docker containers can be a challenging task. Containers in a distributed application may depend on one another and sometimes even require a specific starting order. The configuration and linking of multiple containers during the deployment process are key aspects that Docker Compose tries to solve. Through a single YAML file dependencies, links, volumes and other parameters can defined and later be used to boot up entire applications with just a single command. Docker Compose defines containerized parts of a distributed application as services [Tur14]. Listing 2.2 shows an example Python Flask web application consisting of a web service and a database service.

```yaml
web:
  build: .
  command: python -u webapp.py
  ports:
    - "5000:80"
  volumes:
    - .:/mountVolume
  links:
    - database
database:
  image: mongo:latest
```

Listing 2.2: Distributed web application modeled in YAML for Docker Compose

While Docker Compose provides an easy way to define relations in distributed containerized applications through a single YAML file, Docker Swarm moreover adds native clustering support to the Docker ecosystem. This means multiple Docker hosts can virtually appear as a single host, while providing additional cluster scaling capabilities. Docker Swarm serves the standard Docker API, so other tools like Docker Compose can be pointed to a Swarm cluster just like to a single Docker host.

2.4 Container Compatible Cloud Services

Due to the increasing popularity among developers most cloud providers have adopted solutions to support the deployment of container-based applications. The most important provider specific solutions are described in the following.

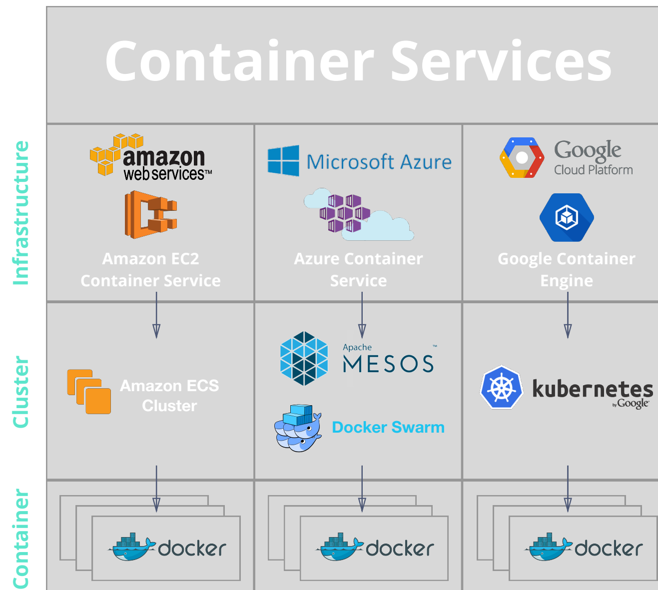

Figure 2.11: Cloud Provider Container Services Overview

2.4.1 Amazon Beanstalk

Amazon Elastic Beanstalk (EBS) provides an easy way to deploy containerized applications by providing a fully automated deployment, provisioning and monitoring platform on top of the AWS services. The underlying resources are provisioned by EBS and scale in and out to match user requirements and current workloads. All it requires is to upload and deploy an application to the environment.

2.4.2 Amazon ECS

Amazon ECS is a solution that lets you manage, scale and deploy containerized applications to a cluster of Amazon EC2 instances. The cluster infrastructure is managed by Amazon and provides functionality for load balancing, auto scaling and other AWS services.

In order to deploy a containerized application to Amazon ECS the user has to assign EC2 instances or an Auto Scaling group to the cluster. All Nodes within a cluster run ECS container agents to connect to the Cluster and get thus get managed.

Applications are defined by Task Definitions[7] which can contain different, even linked Docker containers as well as their resources and limitations. A Task Definition that is executed in a cluster is called a Task. In order to maintain a required number of Task instances, Services have to be created. Services make sure failed Tasks are restarted or replaced by new ones to assure a desired count of running Tasks.

2.4.3 Google Container Engine

Google Container Engine provides a comparable solution to Amazons ECS service. Docker containers get deployed into a cluster of assigned VMs (provided under the hood through Google Compute Engine). The cluster size can be defined by the user and if necessary can be modified and scaled at a later point.

The cluster is operated and managed through Googles Kubernetes[8], which is an open-source cluster management software which automates the deployment and management of containerized applications. Kubernetes supports different Container formats including the Docker container format.

Each Kubernetes cluster consists of a Kubernetes Master, which provides a publicly available API endpoint to communicate with the cluster. The Kubernetes Master is fully managed and thus doesn't need to be maintained by the user. Besides the Kubernetes Master the cluster consists of nodes which run Docker Hosts and execute containers. Additionally each node runs a Kublet Agent in order to connect and get managed by the Kubernetes Master.

[7] http://docs.aws.amazon.com/AmazonECS/latest/developerguide/task_defintions.html/
[8] http://kubernetes.io/

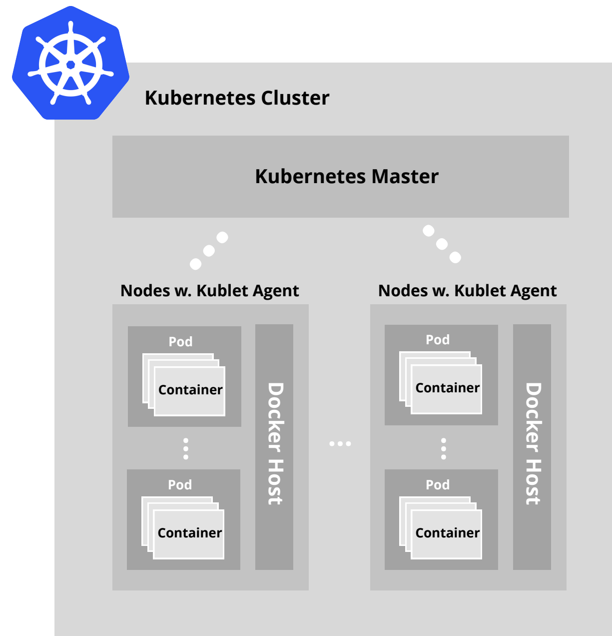

Figure 2.12: Google Container Engine Components - Kubernetes Overview

To get an actual containerized application running in the cluster *Pods* need to be defined. A *Pod* is defined as "a group of one or more containers, the shared storage for those containers, and options about how to run the containers"[9] and is defined by parameters like the Docker image, the number of replications, environment variables and further details (see comparison table). *Pods* are very similar to "Task Definitions" in Amazons Container Service (ECS) and can be defined through the provided CLI tools or the Kubernetes Dashboard which runs on the Kubernetes Master. The Dashboard is preconfigured on the Kubernetes Master and is generally available through `https://kubernetes-master-ip/ui`.

In order to maintain a desired count of *Pod* replications, Kubernetes automatically sets up a *Replication Controller* which internally deploys new *Pods* in case of crashes or failures. Kubernetes *Services* provide an abstraction layer for deployed *Pods* to guarantee consistent access (a proxy IP is assigned) to the application, while underlying containers might be switched out.

[9] `http://kubernetes.io/docs/user-guide/pods/`

2.4.4 Microsoft Container Service

By the time of this writing Microsoft Azure Container Service (ACS) is still in preview and is Microsofts attempt to provide a similar solution to Google Container Engine and Amazon ECS. The overall goal is to provide a platform to deploy containerized applications in a managed environment utilizing orchestration technologies like Apache Mesos[10] with the Marathon Framework and Docker Swarm[11].

During the initial cluster deployment the user can pick an Apache Mesos or Docker Swarm cluster deployment template to get started. The cluster management software will handle the underlying hardware and provide a way to easily scale and deploy containers. While Docker Swarm is directly able to handle Docker containers, Apache Mesos additionally requires Marathon[12] to orchestrate containers. It comes preinstalled with the Apache Mesos template provided by Azure and provides on top a web interface very similar to the one of Kubernetes.

2.4.5 Comparison

In order to scale containerized applications Amazon ECS, Google Container Engine and Azure Container Service utilize different cluster technologies as described in the previous section. By using such the user is still in control of the underlying infrastructure and assigns nodes to the cluster. Amazon Beanstalk provides an additional layer of abstraction by fully managing the underlying infrastructure. The user simply has to upload an application and it gets automatically deployed on AWS. Provisioning, scaling, load balancing and health monitoring are entirely managed by the Beanstalk service.

[10] http://mesos.apache.org/
[11] https://docs.docker.com/swarm/overview
[12] https://mesosphere.github.io/marathon/

3 Related Work

This chapter provides a general overview about related technologies with respect to the provisioning of Docker containers on multi-cloud platforms using an orchestration engine. Special attention was drawn towards the integration and orchestration with existing non-containerized application components and the distribution across multiple cloud providers. Various technologies were analyzed that attempt to solve either the multi-cloud provisioning problem or the problem that solves mixing of non-containerized and containerized components.

Ubernetes is a project by Google and derives from the main kubernetes project[1], which already provides a orchestration and provisioning platform for Docker containers. Ubernetes is still an early stage project, with various architectural and technological proposals but hasn't reached a beta stage yet. A drawback kubernetes brings along is its incapability of deploying worker nodes or containers on multiple cloud providers simultaneously. Ubernetes aims to open up the kubernetes platform by providing higher availability, application portability to avoid vendor lock-ins, on- and off-premise hosting because of privacy sensitive data and capacity overflowing on public cloud offerings. All those use-cases have been generated by the latest feedback of developer and large companies and formed the needs of a solution like Ubernetes.

Figure 3.1 gives a high-level overview of current status of the proposed Ubernetes architecture. A overlaying Ubernetes API is suggested which controls the deployment of multiple independent kubernetes clusters into different Cloud environments. The Ubernetes API further handles kubernetes cluster as first class objects, meaning that each cluster registered, listed, described and deregistered via the API. Each kubernetes cluster is unaware of other clusters. The Policy Engine decides which applications are deployed into which cluster but coordinates closely with the Migration Controller and the Desired Federation State storage. The Migration Controller assures that the replications are running on the cluster as specified. The Desired Federation State storage is proposed to be similar to a distributed ETCD storage, which is already used in kubernetes as introduced in Chapter 2: Container Cluster Technologies. The Ubernetes project looks

[1] https://kubernetes.io

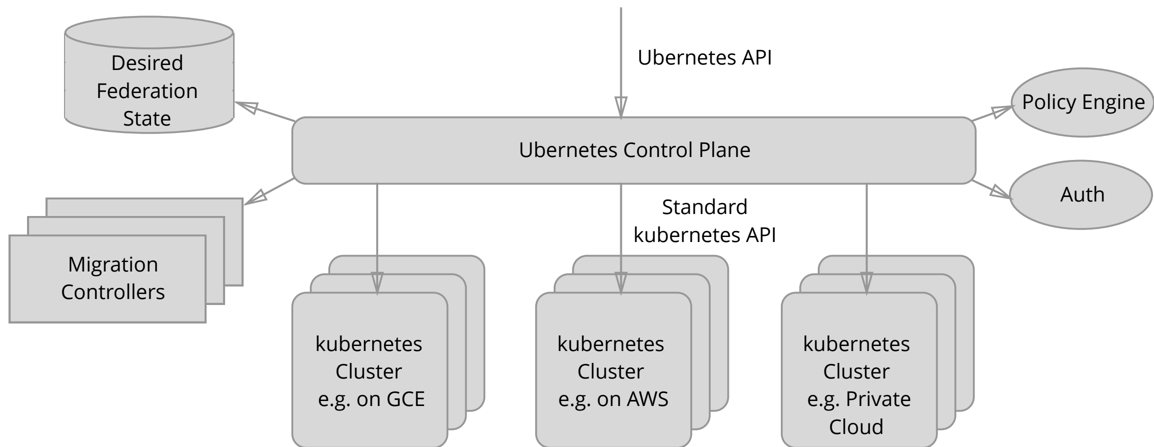

Figure 3.1: Ubernetes Architecture [Kub16c]

very promising, nevertheless by the time of this writing the project is not in a stage which is suitable for production use.

Part of the OpenStack system is the subproject OpenStack Heat[2] which provides an orchestration and provisioning engine for the OpenStack cloud platform. Applications can be launched based on templates which are specified in simple text configuration files. A templates describes the overall structure of an application including server, floating IPs, volumes, security groups and further [Ope16]. Currently OpenStack Heat uses custom DSL. Currently a 2nd version of the OpenStack Heat DSL (Heat DSL2[3] is evolving which is planned to provide compliance to the TOSCA specification. As of right now OpenStack provides a toolkit (project Heat-Translator[4] to translate TOSCA specified application templates into Heat Orchestration Templates (HOT). In OpenStack provides a very interesting approach, which has a wide range of supported types and an interesting architecture. Custom plugins even allow it to run Docker containers as part of Heat templates. A major drawback of the OpenStack Heat system is that it is limited to the OpenStack eco-system, which never the less could combine on- and off-premise data centers.

The open-source project Kansible[5] uses kubernetes and RedHats Ansible[6] in order to combine the usage of container clustering with non-containerized applications or compo-

[2] https://wiki.openstack.org/wiki/Heat
[3] https://wiki.openstack.org/wiki/Heat/DSL2
[4] https://github.com/openstack/heat-translator
[5] https://github.com/fabric8io/kansible
[6] https://www.ansible.com/

nents. Kansable uses Ansible playbooks in order to configure and run non-containerized applications. A playbook is a file which specifies an applications configuration, deployment and orchestration process. Playbook are also written in YAML and are very similar to the TOSCA Simple YAML Profile introduced in the Chapter 2. Kansible then allows to add the Ansible specified component to the kubernetes cluster and fully manage it. All kubernetes features are compatible including scaling, monitoring, service discovery and load balancing. A specifically assigned replication controller takes care of these tasks. Even though Kansible provides a way to host non-containerized applications within an existing kubernetes cluster, it is still tied down to the usage of one Cloud provider of the underlying kubernetes cluster. Further no compliance to a standard specification like TOSCA is provided, which further limits the portability of applications.

4 Specification

4.1 Requirements

This thesis aims to provide a orchestration & provisioning solution for distributed containerized cloud applications while providing a high level of platform interoperability. This section will outline the non-functional and functional requirements that the system must fulfill.

4.1.1 Functional Requirements

Application Modeling Tool: A tool is required in order to model distributed applications, including their topologies and orchestration tasks. Modeling plans can later on also be used to directly provision and deploy application using the provisioning & orchestration engine.

Virtualization Configuration: The modeling tool needs to provide support for different virtualization approaches, meaning that application components can be configured to run on bare Virtual Machine or within Docker Containers. The user will also have the possibility to connect components with different virtualization.

Configure Application Resources: Infrastructural resources (e.g. CPU, Memory, Network, ...) need to be configurable and allocated to specific application components.

Deployment of Distributed Applications: The platform needs to provide interfaces to provision and deploy distributed applications that have been modeled with the modeling tool. The deployment procedure should be accessible through a user interface.

Application Management Interface: An interface must be provided that gives easy access and control to the currently deployed applications. Application control operations include deploying, starting, stopping, scaling and deleting.

Scale Applications: The user should also be able to scale running applications and their components up and down by increasing the replication count or the resources available.

Monitoring & Logging: A specific interface for logging and monitoring capabilities is required in order to provide the user with error reporting and continuous updates on the system status.

Runtime Environment: The user needs to be able to select on which platform the entire runtime environment as well as the applications is deployed on.

4.1.2 Non-Functional Requirements

Portability: To increase the portability of distributed applications compliance to the TOSCA specification must be given. This will enable the usage of multi-cloud applications, while avoiding a vendor-specific adaption. Additional support for Docker will also provide portability in terms of moving between production and development environments.

Interoperability: Different application components need to be able to operate with each other, no matter what virtualization approach is used or which underlying cloud provider is used.

Clustered Container Virtualization: The system needs to be able to fully support container virtualization and proper service discovery by the use of a Container Cluster Technology.

Scalability: Deployed applications and separate components must be able to scale horizontally as well as vertically. Even containerized components need to be able to scale.

Failure Recoverability: In case of failure of an application component the system needs to be able to recover in an automated manner. Monitoring features need to continuously check if a component is properly running and trigger a restart or the deployment of a new replication.

Failure Isolation: The failure of a single application should have no direct impact on other applications.

Low Resource Overhead: The resource usage should have as little overhead as possible, while taking advantage of hyper-visor and container virtualization and container cluster technologies.

Easy Handling: The system itself should despite its complexity be easy to handle and use.

Extensibility: It should be easy to extend the system in terms of new applications node types, cloud providers or other custom operations.

4.2 Use Case Description

In todays business world applications become bigger and more complex. Distributed applications evolve in complexity and contain different components. With an increasing complexity, components are distributed among infrastructure and might even be hosted using different environment technologies like for example Docker. On top of the increasing complexity of applications, the usage of different cloud providers for a single distributed application is becoming more and more common. Mixed on- and off-premise setups out of privacy constraints are just one of the reasons for the latest trends in this direction.

This thesis suggests different use cases in order to run a containerized distributed application while using an existing application running on bare infrastructure. This shows that existing components can be continued to use in system using containerized parts. The use case further requires for the containerized components to automatically scale in case of increased load, while still providing unified access to the service (proper Service Discovery). Besides the topological description, the application needs to be orchestrated in an automated fashion, which includes the continues monitoring of system status in order to guarantee a maximum level of reliability. Figure 4.1 gives an overview about all use cases.

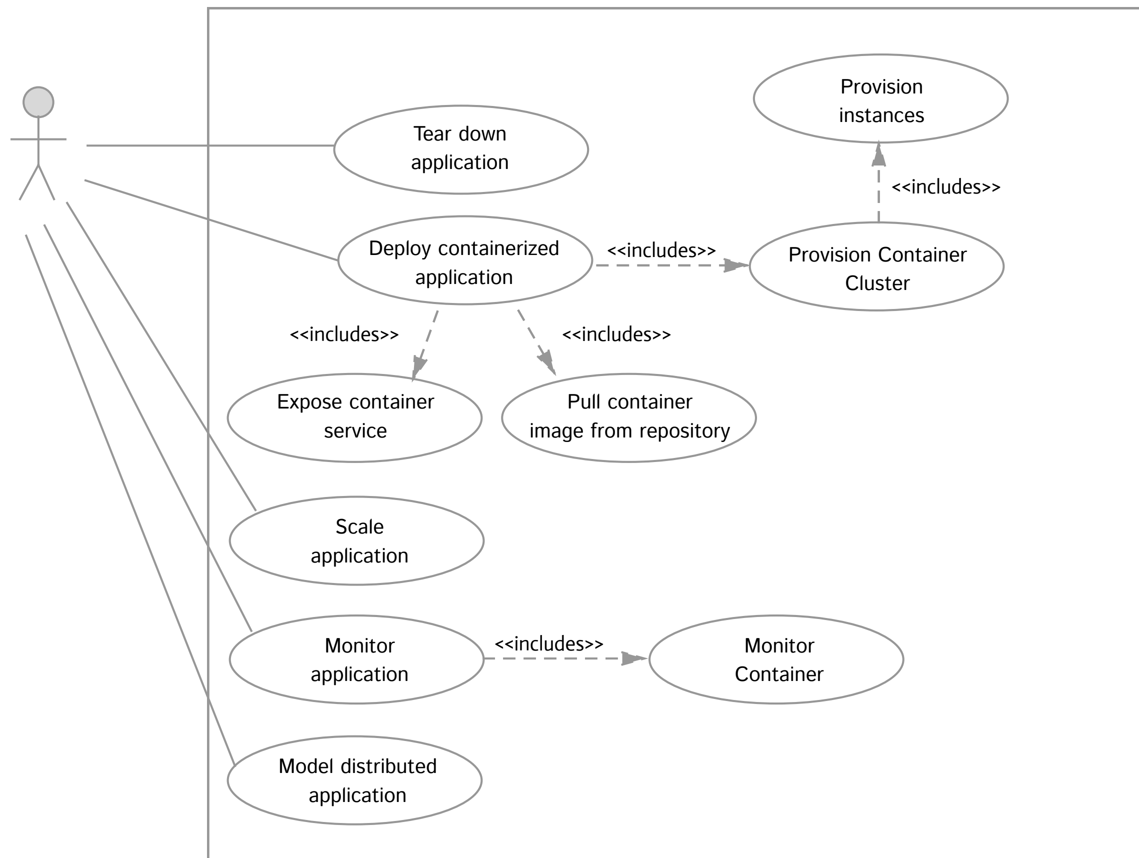

Figure 4.1: Distributed Containerized Application Deployment Use Cases

The following tables provide additional information about the use cases.

Name	**Model Distributed Application**

Goal	The user will be able to model complex distributed applications with a modeling tool. The applications will especially consist of native and containerized application components, which need to be connected. Also support for different Cloud provider needs to be given.
Actor	General User / Developer
Pre-Condition	The modeling tool provides a given set of node types
Post-Condition	The application is modeled in a way that can later on be used by a Topology & Orchestration engine to deploy the application.
Post-Condition in Special Case	The user needs to be provided with accurate error messages, if the modeling is incorrect.
Normal Case	1. The user builds a modeling plan out of existing components and node types 2. The modeling plan is valid
Special Cases	1a. The modeling tool provides not enough components a) The user needs to be provided with options to the extend given modeling components 2a. The modeling plan is invalid a) Provide accurate error messages that show inconstancies or other problems 2b. The modeling plan contains unknown types a) The user needs to be provided with options to the extend given modeling components

Table 4.1: Description of Use Case Model Distributed Application.

Name	**Deploy Containerized Application**
Goal	The user wants to deploy a distributed application containing components running on bare infrastructure and some being containerized
Actor	General User / Developer
Pre-Condition	The user has already modeled the application components and its dependencies
Post-Condition	The application with all its components was successfully orchestrated and deployed.
Post-Condition in Special Case	If the deployment fails then all resources should be teared down and the user needs to be informed about possible error sources
Normal Case	1. The user provides the modeling plan for the application 2. The Topology & Orchestration platform provisions the resources, including the container cluster 3. The container images get pulled from a central repository 4. The containerized components exposed their services 5. The components get orchestrated 6. The application is running and the orchestrated components are able to connect to each other

Special Cases	1a. The model plan is incorrect or missing a) The system points out sources of error in the modeling b) The system aborts the deployment 2a. The provisioning of resources fails a) Rollback all actions and inform user about possible error sources 2b. The deployment of the container cluster fails a) Rollback all actions and inform user about possible error sources 3a. The specified container image can't be pulled a) Inform user to provide a proper container image 4a. The service can't be exposed a) Check if an other service is exposing to the same port and report error sources 5a. Orchestration fails a) Rollback all actions and inform user about possible error sources

Table 4.2: Description of Use Case Deploy Containerized Application.

Name	**Scale Application**
Goal	The user needs to be able to scale the deployed application vertically and horizontally.
Actor	General User / Developer
Pre-Condition	The application is already deployed and running with a containerized part on a container cluster
Post-Condition	The application has scaled up or down according to the user input
Post-Condition in Special Case	If the scaling fails the application needs to recover its previous state
Normal Case	1. A distributed application with containerized components is already running 2. The containerized part that needs to be scaled will increase or decrease the replication count 3. New resources get provisioned and assigned to the cluster in order to scale the application up 4. The internal cluster load balancer distributes the traffic among the nodes
Special Cases	1a. No application is running 2a. A change of the replication count will not trigger the scaling a) Check if changes have been applied, if not provide the user with an error message 3a. The resource provisioning or resource assignment to the cluster fails a) Provide an error message, but keep the service running 4a. The load is not balanced between the nodes a) Monitoring capabilities need to watch if the load is balanced and in case of failure report with error messages

Table 4.3: Description of Use Case Scale Application.

Name	**Monitor Application**
Goal	The user needs to be able to monitor resources that have been provisioned.
Actor	General User / Developer
Pre-Condition	The application is already deployed and running with a containerized part on a container cluster, a workload can be monitored.
Post-Condition	The user is informed about the current status of the resources and the application components
Post-Condition in Special Case	If no information is provided, provide access possibilities to control the resources manually
Normal Case	1. The user select the application that he wants to monitor, preferably in a dashboard 2. The user receives information like logging, status and load
Special Cases	1a. No application is running or selected 2a. No data is provided, it seems like the components are not being monitored a) Check if there is no connection to the components and provided accurate error messages as well as possibilities to manually connect and monitor the components

Table 4.4: Description of Use Case Monitor Application.

Name	**Tear Down Application**
Goal	Applications and its components need to be able to be stopped and teared down
Actor	General User / Developer
Pre-Condition	The application is already deployed and running with a containerized part on a container cluster, a workload can be monitored.
Post-Condition	The application is stopped and the resources are deallocated
Post-Condition in Special Case	Parts of the application are still running
Normal Case	1. The user select the application that he wants to terminate, preferably in a dashboard 2. The platform stops the service and tears down the resources, including the cluster if no other container components are running
Special Cases	1a. No application is running or selected 2a. The application is not able to be stopped a) Provide possibilities to force an application to stop 2b. Parts of the application get stuck and won't shut down a) Provide possibilities manually tear down the resources

Table 4.5: Description of Use Case Tear Down Application.

4.3 System Overview

To accomplish the previously named features a system based on the following components is introduced. The components are also outlined in Figure 4.2. The system covers the entire workflow from modeling of an application over orchestration its components to scaling and managing the application throughout its lifecycle. This section gives an high level overview of the system whereas chapter 5 gives insights into the concrete implementation and the technology providers used.

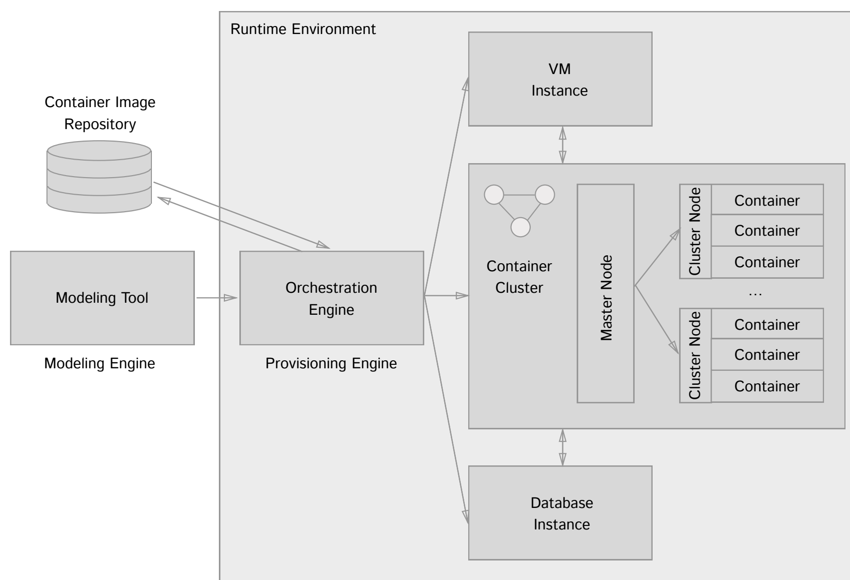

Figure 4.2: Architectural System Overview

1. Modeling Tool
   A modeling tool will be provided to model distributed applications as well as relationships between them. The modeling will later be used by the orchestration & provisioning engine in order to deploy the distributed components. It's noteworthy that the modeling of applications not only covers topological specifications, like infrastructural requirements, but also states dependencies and other orchestrational details. The modeling mostly starts off with more generic types, which later on will derive into more specific node types like for example Cloud provider specific node types. An important part of a modeling tool is that it should be able to easily extend it in case new types are needed.

2. Topology & Orchestration Engine
   In the introduced system an Topology & Orchestration Engine will be used. By

applying compliance to a standard like TOSCA a very high level of platform interoperability for applications will be assured. The Topology & Orchestration Engine will take advantage of the TOSCA specification and provide functionality to execute deployments and other orchestrational tasks in various environments. Further the engine needs to be aligned with the modeling tool in order to application modeling structures and provision them.

3. Runtime Environment
   In order to provide scaling capabilities monitoring functionalities of the runtime environment will assure accurate up and down-scaling of the underlying resources. A proper monitoring is also required to determine failure of running instances and maintain proper execution.

4. Container Cluster Technology
   Recent Container cluster technologies like Docker Swarm, kubernetes or Apache Mesos have been evolved to satisfy common needs like auto-scaling, service discovery and load balancing, while providing a container-centric infrastructure [Kub16d]. The management and maintenance of containers is a complicated task, which a container cluster handles in a fully automated way, while providing very little overhead. Further the following benefits can be achieved:

   a) Scaling of Containers
      Scaling a Docker host system vertically can increase the resource in times of a traffic peak. Nevertheless it's sometimes inevitable to scale a Docker container horizontally. While scaling the host vertically can be easily done by simply adding resources to the underlying virtual machine, scaling horizontally requires an additional load balancer to distribute requests between deployed instances. This becomes a difficult tasks if multiple Docker containers run separately on multiple hosts and still need to scale automatically and discover services provided among themselves. Docker cluster technologies solve this by providing an entire infrastructure to host Docker containers, which provides auto-scaling and load balancing.

   b) Service Discovery
      The service provided by a container needs to be accessible in a unified way regardless of the current number of replications and current scaling. To achieve this each Cluster provides different approaches for proper Service Discovery, which is handled in a fully automated fashion. This way containers can connect to each other or can even be accessed from components outside of the cluster.

5. Docker Container
   The Container Cluster technologies discussed in this thesis cover solely the orches-

tration of containers and provide no capability to manage applications running on an other infrastructure. The high flexibility of containers make it a common tool for developer and thus are common input and runtime format for every cloud orchestration engine.

6. Native Services
   While newly develop application components are mainly build on the basis of Docker containers many old components are still running on bare infrastructure. Nevertheless it is required to incorporate those components in topologies with containerized applications and provide interoperability between them. In the suggested system the orchestration and topology engine is able to provision and orchestrate both types.

5 System Design

The previously introduced system will consist of different components, which will be integrated in a topology and orchestration eco-system. Some fundamental architectural design choices have already been introduced in chapter 4. This chapter will not only argue the selected design decisions but also introduce the specific technologies.

5.1 Architecture

5.1.1 Containers

The general benefits of container virtualization have already been covered in Chapter 2. Through the increased popularity of Docker within the latest few years special attention was drawn towards the deployment of Docker containers. A huge community has been established, that provides not only great support but also a huge variety of Docker images that are hosted on a public Docker image repository called Docker Hub. Images from Docker Hub are publicly accessible and can be pulled from the orchestration engine during deployment.

5.1.2 Cloud Application Specification

Cloud specifications allow an easy platform independent description of distributed applications. The previously introduced TOSCA standard shows a high market acceptance and adoption even by market leading IT companies. Further the TOSCA standard provides the Simple YAML format (see chapter 2: TOSCA Simple YAML Profiles) which can easily be extended to cover custom or provider specific nodes types. This way a common ground for the modeling of distributed applications is already given.

5.1.3 Topology & Orchestration Platform

The foundation of the introduced system is a topology and orchestration platform. OpenTOSCA and Cloudify as TOSCA based orchestration platforms have been taken into consideration, while specifying the architectural design. Cloudify provides a wide range of already supported node types, which can be used to model and manage complex and distributed applications. Deriving and extending new node types is very intuitive and easy as Cloudify uses the official TOSCA Simple YAML format (see Chapter 2). Basic node types like compute instances for AWS or OpenStack have already been published by the Cloudify Team. While Lego4TOSCA offers similar types for the OpenTOSCA eco system based on XML templates, the overall number of available node types for Cloudify outreaches the number available types for OpenTSCOA through Lego4TOSCA. Additionally benefits of Cloudify include a big community, a well structured documentation and free support.

The modeling of distributed applications for Cloudify including their lifecycle, dependencies and components is also done through specifying YAML files following the Simple YAML format. A custom DSL deriving from the official TOSCA Simple YAML specification (`tosca_simple_yaml_1_0`), forms the foundation. This DSL provides types for compute nodes and other basic types in the Cloudify eco system. Listing 5.1 shows a simple web server application specified using the Cloudify DSL 1.3. The web server derives from the `cloudify.nodes.WebServer` node type, and specifies the port as well as the VM it's running on. The underlying VM derives from the `cloudify.nodes.Compute` and specifies the IP address. The `http_web_server` further specifies custom life cycle operations, for the configure, start and stop tasks.

```yaml
tosca_definitions_version: cloudify_dsl_1_3

description: >
  This blueprint installs a simple web server on the manager VM.

imports:
  - http://www.getcloudify.org/spec/cloudify/3.4m5/types.yaml

inputs:
  server_ip:
    description: >
      The ip of the server the application will be deployed on.
  webserver_port:
    description: >
      The HTTP web server port.
    default: 80

node_templates:
  vm:
    type: cloudify.nodes.Compute
    properties:
      ip: { get_input: server_ip }
  http_web_server:
    type: cloudify.nodes.WebServer
    properties:
      port: { get_input: webserver_port }
    relationships:
      - type: cloudify.relationships.contained_in
        target: vm
    interfaces:
      cloudify.interfaces.lifecycle:
        configure: scripts/configure.sh
        start: scripts/start.sh
        stop: scripts/stop.sh
```

Listing 5.1: Cloudify Webserver Example [Gig16b]

Custom Cloudify plugins that are loaded into the Cloudify management VM, provide an easy way to define provider specific life-cycles and provisioning tasks by using a Python interface. Plugins can also be loaded onto the Application VM to provide custom runtime tasks. An example of a very common plugin is the Cloudify AWS plugin, which provides functionality to deploy EC2 instances and offers access to other AWS services. An other example for an Application VM plugin would be the official Cloudify script plugin, which enables the execution of scripts on a running Cloudify Application VM during the runtime. The Cloudify plugin interface will be used in the proposed architecture

of this thesis to extend the Cloudify eco system with a scalable solution to host Docker containers through a Cluster technology.

5.1.4 Container Cluster Technology

Even though the Cloudify eco-system already provides a Docker plugin to run containerized components, the capabilities are very limited. In general all aspects of cluster technologies mentioned in chapter 4 apply here. Different Cloud provider like Amazon AWS, Microsoft Azure and Google already started including cluster technologies with services like Amazon ECS, Azure Container Service and Google Container Engine. Table 5.1 shows a comparison of the configuration parameters by cluster technology. The parameters were divided into three sections, which give an abstraction of different stages during the deployment procedure. Each cluster consisted of more general parameters like the cluster name, the region of deployment and network details. Applications running on the cluster have separate parameters and only different in naming conventions, which Table 5.1 tries to clear by giving them more generic names. The biggest difference between the providers was seen in the network or linking section. Each cloud technology had specific parameters, which influences the options but can also lead to a more complex user experience. During the initial testing it became clear that kubernetes had the most user friendly and hassle free approach, which drew special attention to the straightforwardness of service discovery.

Table 5.1: Container Service Config Parameters by Cloud Providers

m,	Amazon ECS	Azure Container Service	Google Container Engine
General	Credentials	Credentials	Credentials
Cluster	Cluster Name Region* Network Details* Agent VM Size* # Agents	Cluster Name Region* Network Details* Agent VM Size* # Agents # Master Nodes* Orchestrator Type*	Cluster Name Region* Network Details* Agent VM Size* # Agents
Tasks/ Pods/ Applications	Name # Instances Image CPU/Memory Available Env Variables Port Mapping RUN Command Volume Mounting* Labels	Name # Instances Image CPU/Memory Available Env Variables Port Mapping RUN Command Volume Mounting* Labels	Name # Instances Image CPU/Memory Available Env Variables Port Mapping RUN Command Volume Mounting* Labels
Network/ Linking/ Service Discovery	Container Linking* Hostname, DNS, ...* Docker Security Options* Working Directory, Entry Point* ...	Network Mode* Hostname, Ports, User, ...* Parameters* Executor* Ports* Constraints* User*	Namespace* ...

* Provider specific parameter

The extension of the Cloudify ecosystem by a kubernetes plugin in order to run, manage and scale Docker micro-services, is a viable solution to achieve the requirements stated in Chapter 4. The comparison of the different cluster technologies showed that kubernetes was very straightforward in terms of service discovery and lifecycle management of Docker containers in general. An other benefit of cluster technologies is that they provide an other abstraction layer, so the underlying infrastructure can easily be exchanged. Figure 5.1 shows that the underlying IaaS provider can vary. The container cluster provisions the containers independent of the infrastructure it runs on.

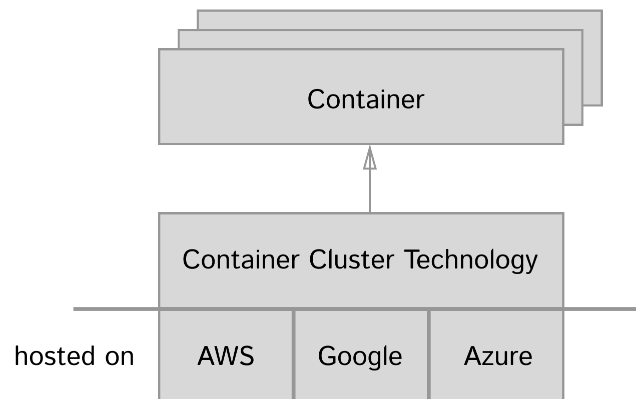

Figure 5.1: Container Cluster Environment

A typical kubernetes Cluster contains a kubernetes Master as well as multiple minion nodes, which represent the worker nodes in the cluster. Workloads are spread between the minions, on which a kublet agent and a Docker host is running in order to execute packaged applications called pods. A pod is considered a package of components of an application which are closely related and can contain one or more containers. Kubernetes handles the scaling, lifecycle management as well as service discovery in an automated fashion. The capacity available to the cluster is dependent on the minions assigned to it. The challenge of using a kubernetes cluster as a node type in Cloudify lies in the provisioning of the kubernetes master and its minion nodes.

In general two options are considerable, while deciding how to provision the kubernetes master inside of Cloudify. The direct deployment on the Cloudify management VM and a separate deployment on a VM orchestrated by Cloudify. A provisioning on a separate host provides a higher level of scaling flexibility with slightly more orchestration affords. In order to implement this approach Cloudify suggest the definition of a separate kubernetes master node type as well as node types for the minion hosts and the pods (microservice).

Figure 5.2 shows the deployment process of an containerized application on Cloudify using a kubernetes cluster. First Cloudify provisions a new VM hosting the kubernetes Master node. The minion nodes are not provisioned by the kubernetes Master node as expected, but rather by the Cloudify management VM itself during the initial deployment of the cluster. Afterwards the successful deployment of the master and minion nodes, the kubernetes cluster is ready to deploy applications. Cloudify afterwards reads the microservice specification from the YAML blueprint and lets the Master node deploy

it into the cluster. After the service is available Cloudify can continue connecting it to other parts of an application.

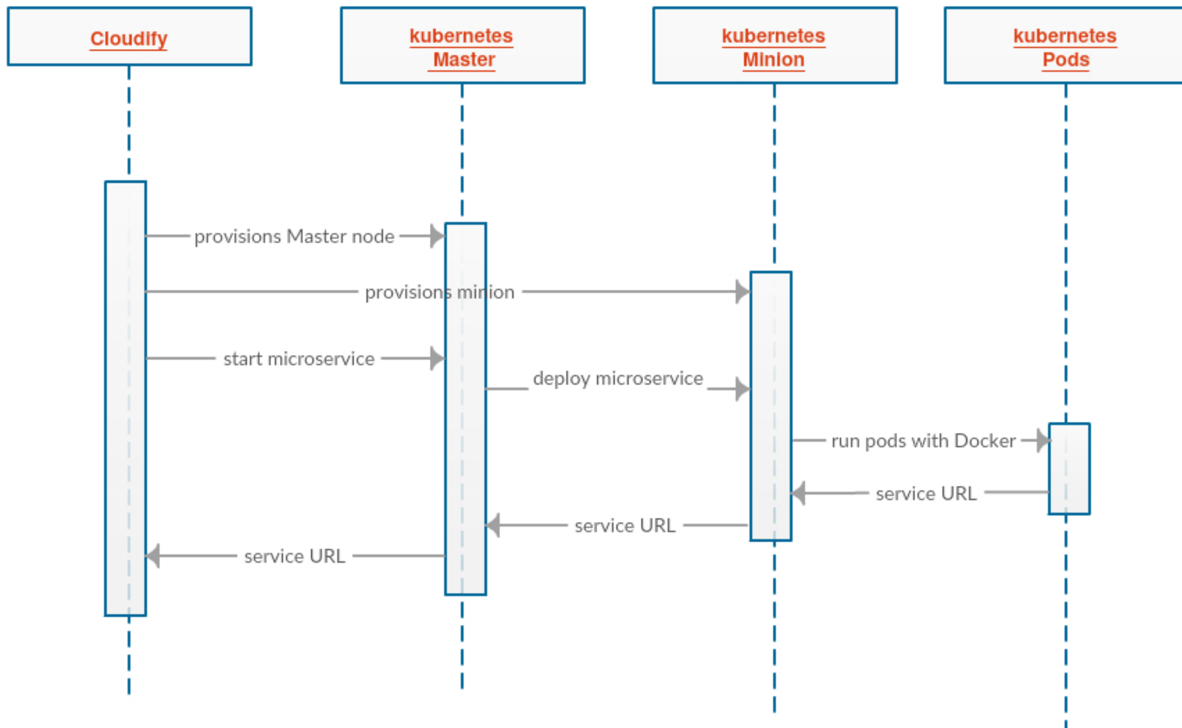

Figure 5.2: Cloudify Kubernetes Cluster Provisioning

6 Implementation

This chapter will discuss the implementation of the previously introduced architectural design and its challenges. In particular this section will cover the creation of a plugin and blueprints to use kubernetes as a cluster on the AWS or OpenStack cloud platforms. While Cloudify already provides a proprietary and outdated solution for the OpenStack cloud [Gig16c], the challenge mainly was to provide support for the latest version of Cloudify and add additional support for other cloud providers.

Kubernetes Multi-Node Setup

Kubernetes provides very detailed information on their website on how to setup a multi-node cluster [Kub16b]. The guidelines use Docker itself for provisioning the kubernetes master and other components on any infrastructure. In order to connect the different components of the multi-node setup flannel [Cor16b] is used. Flannel is a virtual subnet, reliant on etcd, a distributed key-value store specifically designed for service discover and shared configuration management [Cor16a]. A separate Docker container will host flannel and take care of connecting the different pods and services host later on.

The following command which will later on be used in a custom kubernetes plugin will launch a Docker daemon running etcd:

```
sudo docker -H unix:///var/run/docker-bootstrap.sock run -d \
   --net=host \
   gcr.io/google_containers/etcd-amd64:2.2.1 \
   /usr/local/bin/etcd \
      --listen-client-urls=http://127.0.0.1:4001 \
      --advertise-client-urls=http://127.0.0.1:4001 \
      --data-dir=/var/etcd/data
```

Listing 6.1: etcd Docker command

A custom CIDR address range needs to be set, which will later on will be used to provision the pods in the cluster. Afterwards flannel can be launched in a container on the master node.

```
sudo docker -H unix:///var/run/docker-bootstrap.sock run -d \
   --net=host \
   --privileged \
   -v /dev/net:/dev/net \
   quay.io/coreos/flannel:flannel:0.5.5’
```

Listing 6.2: flannel Docker command

After getting the subnet details from the previously started flannel Docker container, the Docker configuration of the kubernetes master host system needs to be changed accordingly. Afterwards the Docker service on the master node needs to be restarted in order to apply the latest network changes.

The following command will execute the final launch of the kubernetes master node on the host system:

```
sudo docker run \
   --volume=/:/rootfs:ro \
   --volume=/sys:/sys:ro \
   --volume=/var/lib/docker/:/var/lib/docker:rw \
   --volume=/var/lib/kubelet/:/var/lib/kubelet:rw \
   --volume=/var/run:/var/run:rw \
   --net=host \
   --privileged=true \
   --pid=host \
   -d \
   gcr.io/google_containers/hyperkube-amd64:v1.2.3 \
   /hyperkube kubelet \
      --allow-privileged=true \
      --api-servers=http://localhost:{} \
      --v=2 \
      --address=0.0.0.0 \
      --enable-server \
      --hostname-override=127.0.0.1 \
      --config=/etc/kubernetes/manifests-multi \
      --containerized \
      --cluster-dns=10.0.0.10 \
      --cluster-domain=cluster.local".{master_port}
```

Listing 6.3: kubernetes Docker command

The setup of the minion nodes also requires a configured Docker host in order to deploy and connected the kubelet agents to the master node. After setting up the Docker service

on the node itself and adjusting the network settings with regards to the IP of the master node, the installation of flannel can be started:

```
sudo docker -H unix:///var/run/docker-bootstrap.sock run \
   -d --net=host --privileged \
   -v /dev/net:/dev/net \
   quay.io/coreos/flannel:0.5.3 \
   /opt/bin/flanneld \
   --etcd-endpoints=http://{master_ip}:4001
```

Listing 6.4: flannel Docker command for minion host system

After the execution an other command is required to retrieve the flannel subnet configuration from the Docker container. The procedure is identical to the the master node. The settings are necessary to change the general Docker host settings of the minion node. After getting the subnet the Docker configuration needs to be changed. This way the Docker host system is aware of the the flannel network and uses the IP ranges and subnet instead of its default settings.

After those actions have been completed and the Docker service was restarted, the kubelet agent can be installed. The steps taken connect the deployed kubelet agents to the master node.

The final step is to run a containerized service as a Docker image on the deployed multi-node cluster. In order to control the cluster and execute commands, the kubectl command line tool is used [Kub16a]. The execution is fully run by the kubernetes master node.

```
kubectl run NAME --image=image [--env="key=value"] [--port=port] [--replicas=replicas]
    [--dry-run=bool] [--overrides=inline-json] [flags]
```

Listing 6.5: kubectl command to start a service [Kub16a]

In order to make the service publicly accessible, the service needs to be exposed to a specific port.

```
kubectl expose (-f FILENAME | TYPE NAME | TYPE/NAME) [--port=port] [--protocol=TCP|UDP]
    [--target-port=number-or-name] [--name=name] [----external-ip=external-ip-of-service]
    [--type=type] [flags]
```

Listing 6.6: kubectl command to expose a service [Kub16a]

After the service has been exposed you can easily retrieve the public and private IP address of the service. The kubectl get command will output all details for a service including the IPs.

```
kubectl get (-f FILENAME | TYPE [NAME | /NAME | -l label]) [--watch] [--sort-by=FIELD]
    [[-o | --output]=OUTPUT_FORMAT] [flags]
```

Listing 6.7: kubectl command to get service details [Kub16a]

If service are not longer required the kubectl delete command lets you delete resources such as pods and services.

```
kubectl delete ([-f FILENAME] | TYPE [(NAME | -l label | --all)])
```

Listing 6.8: kubectl command to delete a pod or service [Kub16a]

Kubernetes Cloudify Plugin

In order to integrate the kubernetes cluster and service deployment in Cloudify a custom plugin is required. The plugin will be loaded onto the Cloudify master node. The master node will then be instructed to load the plugin onto the node that is going to host the kubernetes master.

Cloudify Plugin File Structure   Cloudify plugins help to define custom operations that get executed during certain life cycle operations in the deployment process of a node type. This thesis introduces an approach containing a Cloudify plugin to provision a multi-node kubernetes cluster and run and scale applications on it. In general each Cloudify plugin follows an architectural design, which consists of a few basic components:

- `__init__.py`
  The `__init__.py` file is required to treat the directory and files as a python package. It can contain function definitions that are being used by multiple script files in the package directory.
- `setup.py` - The initial setup file
  Contains information like the name, author, version, license as well as dependencies on other Cloudify plugins. Any Cloudify plugin is a python module with a few simple restraint. Each plugin requires the install dependency of cloudify-plugins-common and a unique name. Further the `setup.py` file includes a packages

parameter that needs to correspond to a folder in the plugin directory, which will include python files with task definitions and similar.

- `plugin.yaml`
  The `plugin.yaml` file contains custom node type definitions, as well as declarations of custom relationships and workflows required for the cluster deployment and management. The detailed implementation of lifecycle operations will be discussed later on and are implemented in specific python scripts in the Package directory.

  The following custom node types have been defined:

  - `cloudify.kubernetes.Base`
    The kubernetes master as well as the kubernetes nodes share a few base properties which the `cloudify.kubernetes.Base` node type defines. It derives from the `cloudify.nodes.Root` node type.

    `cloudify.kubernetes.Base` node type properties:

    * `ssh_keyfilename`: SSH key file for passwordless access
    * `ssh_username`: The username corresponding to the SSH key
    * `ssh_password`: The password corresponding to the SSH key and username
    * `ssh_port`: The port needed to connect via SSH
    * `install_docker`: Determines if Docker needs to be installed on the host
    * `install`: Determines if kubernetes needs to installed on the host

  - `cloudify.kubernetes.Master`
    Derives from the `cloudify.kubernetes.Base` and defines a node type for the kubernetes master node. The master node also coordinates the deployment of services within the cluster.
    `cloudify.kubernetes.Master` node type properties:

    * `master_port`: The kubernetes master nodes default port, which is also used when kubernetes nodes connect to it

    `cloudify.kubernetes.Master` lifecycle operations:

    * start: The `cloudify.kubernetes.Master` node type requires a custom start operation that brings up the Master node of the cluster.

  - `cloudify.kubernetes.Node`
    Derives from the `cloudify.kubernetes.Base` and defines a node type for the kubernetes nodes (minions) within a cluster.
    `cloudify.kubernetes.Node` lifecycle operations:

  * start: The `cloudify.kubernetes.Node` node type also requires a custom start operation that will deploy the minion node and connect it to the kubernetes cluster master node.

- `cloudify.kubernetes.Microservice`
  The `cloudify.kubernetes.Microservice` doesn't derive from the `cloudify.kubernetes.Base` node type but rather from the `cloudify.nodes.Root` node type. The `cloudify.kubernetes.Microservice` represents a service that gets deployed in the kubernetes cluster. `cloudify.kubernetes.Microservice` node type properties:

  * `name`: A service name
  * `image`: The Docker image that the service runs. Will be pulled from Docker Hub.
  * `port`: The port of the service that the Docker container will publish to.
  * `target_port`: The target port on which the service will be available.
  * `protocol`: The protocol that will be used to publish the serivce. Defaults to TCP.
  * `replicas`: The number of pod replications that will be deployed within the cluster
  * `run_overrides`: If the default kubectl run command needs to be overwritten, this property can be used
  * `expose_overrides`: If the default kubectl expose command needs to be overwritten, this property can be used
  * `config`: kubernetes configuration in key/value format
  * `config_path`: If an external kubernetes configuration needs to be loaded this path will be applied
  * `config_overrides`: If the default config command needs to be overwritten, this property can be used

  `cloudify.kubernetes.Microservice` lifecycle operations:

  * start: The `cloudify.kubernetes.Microservice` node type uses a custom start operation to deploy the service into the cluster. Internally the `kubectl create -f FILE` command is used to deploy the service within the kubernetes cluster.

Additional the following custom relationships have been introduced. The detailed implementation of the defined relationships are located in the `tasks.py` file in the Packages directory.

- `cloudify.kubernetes.relationships.connected_to_master`
  Derives from `cloudify.relationships.connected_to` and defines a relationship specifically for the connection from minion nodes to the kubernetes master node. The operation hooks into the postconfigure workflow [Gig16d] and retrieves details like IP, port and SSH details and makes them available in the Cloudify context object.

- `cloudify.kubernetes.relationships.contained_in_host`
  Directly derives from the `cloudify.relationships.contained_in` relationship and extends the default behavior by making the IP address of the containing host available to the included host. This relationship will later on be used to share network details between the host of the master node and the kubernetes master itself.

On top of custom node types and relationships, workflows have been defined. Workflows can be executed on deployments and can contained multiple tasks. Each workflow is deployment-sensitive, meaning that certain workflows are only available in certain stages of the deployment. Applied to the kubernetes cluster this means that workflows to run, expose, scale and delete containers and services are only available when the kubernetes cluster has been successfully deployed.

As already mentioned the following workflows will be available after the successful deployment of a kubernetes cluster. The implementation of the workflows is located in the `workflows.py` file.

- `kube_run`
  This workflow executes the `kubectl run` command to run a pod and maintain a give number of replications throughout the cluster.

  Parameters:

  * `master`: Name of the master node
  * `name`: Name of the pod to run
  * `image`: The id of the image that will be run
  * `port`: The port that will internally be opened
  * `replicas`: The number of replications maintained by the replication controller

    * `dry_run`: If the operation should be performed as a dry run or not
    * `overrides`: If specified the parameters will be overwritten by the provided string in JSON format

- `kube_expose`
  Exposes the the provided resource on a given port by using the `kubectl expose` command.

  Parameters:

    * `master`: Name of the master node
    * `name`: Name of the resource that needs to be exposed
    * `resource`: The type of the resource that needs to be exposed
    * `protocol`: The protocol used to expose the resource (TCP or UDP)
    * `port`: The port that will be used to expose the resource
    * `target_port`: The port of the resource that will be used to map it to the external port
    * `service_name`: Name of the exposed service that will be created
    * `overrides`: If specified the parameters will be overwritten by the provided string in JSON format

- `kube_scale`
  Scales the given resource to a given number of replications using the replication controller and the `kubectl scale (-f FILENAME | TYPE NAME | TYPE/NAME)--replicas=COUNT [--resource-version=version] [--current-replicas=count] [flags]` command.

  Parameters:

    * `master`: Name of the master node
    * `name`: Name of the resource that needs to be deleted or stopped
    * `replicas`: Number of replicas to adjust scaling to

- `kube_delete`
  Deletes the resource using the `kubectl delete (-f FILENAME | TYPE [NAME | /NAME | -l label | --all])[flags]` command.

  Parameters:

    * `master`: Name of the master node

        * `name`: Name of the resource that needs to be deleted or stopped
        * `resource`: The type of the resource that needs to be deleted or stopped
        * `all`: Needs to be set to True if all resources need to be deleted or stopped

- Package directory
  As already mentioned in the previous section the package directory includes files implementing workflows, operations and node type specific lifecycle methods. In order to execute the command line commands a few python modules have been used:

  - The subprocess [Pyt16b] and os [Pyt16a] python modules were used to execute the command line tasks on nodes.
  - The ctx module provides an easy access to context related data like the Cloudify logger or node properties that have been defined in a Cloudify blueprint during the modeling process.

  The following will give a more detailed overview about the implementation of the actual files:

  - `start_master_ubuntu14.py`
    Defines the start operation for the `cloudify.kubernetes.Master` node type mentioned previously. The entire bootstrapping of the kubernetes master node is implemented in this file. The function `start_master` includes the following steps:

1. Installation of Docker

```
1  subprocess.call("sudo apt-get install apt-transport-https
       ca-certificates",shell=True)
2  subprocess.call("sudo apt-key adv --keyserver
       hkp://p80.pool.sks-keyservers.net:80 --recv-keys
       58118E89F3A912897C070ADBF76221572C52609D",shell=True)
3
4  subprocess.call("sudo apt-get install apparmor",shell=True)
5
6  subprocess.call("sudo touch
       /etc/apt/sources.list.d/docker.list",shell=True)
7  subprocess.call("sudo bash -c 'echo \"deb
       https://apt.dockerproject.org/repo ubuntu-trusty main\" >
       /etc/apt/sources.list.d/docker.list'",shell=True)
8
9  subprocess.call("sudo apt-get update",shell=True)
10 subprocess.call("sudo apt-get --assume-yes install
       linux-image-extra-$(uname -r)",shell=True)
11
12 subprocess.call("sudo apt-get --assume-yes install
       docker-engine",shell=True)
13
14 subprocess.call("service docker start",shell=True)
```

Listing 6.9: Docker installation using python and subprocess module

2. Changing configuration of Docker daemon

```
1  subprocess.Popen(['sudo','nohup','docker','daemon',
       '-H','unix:///var/run/docker-bootstrap.sock',
       '-p','/var/run/docker-bootstrap.pid', '--iptables=false',
       '--ip-masq=false', '--bridge=none',
       '--graph=/var/lib/docker-bootstrap'], stdout=open('/dev/null'),
       stderr=open('/tmp/docker-bootstrap.log','w'), stdin=open('/dev/null'))
```

Listing 6.10: Disable iptables and ip-masq for Docker Daemon

3. Start ETCD

```
1  os.system("sudo docker -H unix:///var/run/docker-bootstrap.sock run -d
       --net=host gcr.io/google_containers/etcd-amd64:2.2.1
       /usr/local/bin/etcd --listen-client-urls=http://127.0.0.1:4001
       -advertise-client-urls=http://127.0.0.1:4001
       --data-dir=/var/etcd/data")
```

Listing 6.11: Run ETCD in Docker Container

4. Set CIDR range for flannel

```
1  os.system("sudo docker -H unix:///var/run/docker-bootstrap.sock run
       --net=host gcr.io/google_containers/etcd-amd64:2.2.1 etcdctl set
       /coreos.com/network/config '{ \"Network\": \"10.1.0.0/16\"}'")
```

Listing 6.12: Docker installation using python and subprocess module

5. Run flannel

```
1  pipe=subprocess.Popen(['sudo','docker', '-H',
       'unix:///var/run/docker-bootstrap.sock','run', '-d','--net=host',
       '--privileged','-v', '/dev/net:/dev/net',
       'quay.io/coreos/flannel:0.5.5'], stderr=open('/dev/null'),
       stdout=subprocess.PIPE)
2
3  # get container id
4  cid=pipe.stdout.read().strip()
5  pipe.wait()
```

Listing 6.13: Run flannel using Docker image and save container ID

6. Get flannel settings and adjust docker config

```
1  # get flannel subnet settings
2  pipe = subprocess.Popen(['sudo','docker',
       '-H','unix:///var/run/docker-bootstrap.sock', 'exec',format(cid),
       'cat','/run/flannel/subnet.env'], stdout=subprocess.PIPE,
       stderr=subprocess.PIPE)
3  out, err = pipe.communicate()
4
5  # Extract flannel subnet
6  result = out.decode()
7  flannel=";".join(result.split())
8
9  # Adjust local Docker configuration to use flannel subnet
10 with open("/tmp/docker","w") as fd:
11    with open("/etc/default/docker","r") as fdin:
12      for line in fdin:
13        fd.write(line)
14   with open("/tmp/docker","a") as fd:
15     fd.write(flannel+"\n")
16     fd.write('DOCKER_OPTS="--bip=${FLANNEL_SUBNET} --mtu=${FLANNEL_MTU}"\n')
17
18   subprocess.call("sudo mv /tmp/docker /etc/default/docker",shell=True)
```

Listing 6.14: Extract the flannel subnet configuration from running flannel Docker container

7. Remove Docker bridge

```
1  # remove existing docker bridge
2  os.system("sudo /sbin/ifconfig docker0 down")
3  os.system("sudo apt-get install -y bridge-utils")
4  os.system("sudo brctl delbr docker0")
5
6  # restart docker
7  os.system("sudo service docker start")
```

Listing 6.15: Removal of the default docker0 bridge

8. Get the master port from the blueprint and start kubernetes master

```
# get the port from the blueprint
master_port=ctx.node.properties['master_port']

# start the master
subprocess.call("sudo docker run \
  --volume=/:/rootfs:ro \
  --volume=/sys:/sys:ro \
  --volume=/var/lib/docker/:/var/lib/docker:rw \
  --volume=/var/lib/kubelet/:/var/lib/kubelet:rw \
  --volume=/var/run:/var/run:rw \
  --net=host \
  --privileged=true \
  --pid=host \
  -d \
  gcr.io/google_containers/hyperkube-amd64:v1.2.3 \
  /hyperkube kubelet \
      --allow-privileged=true \
      --api-servers=http://localhost:{} \
      --v=2 \
      --address=0.0.0.0 \
      --enable-server \
      --hostname-override=127.0.0.1 \
      --config=/etc/kubernetes/manifests-multi \
      --containerized \
      --cluster-dns=10.0.0.10 \
      --cluster-domain=cluster.local".format(master_port), shell=True)
```

Listing 6.16: Start the kubernetes master as Docker container

- `start_node_ubuntu14.py`
  Defines a `start_node` Cloudify operation which is used to deploy a kubernetes node. The kubernetes nodes connect to the master node using the kubelet agent command line tool. Additionally every kubernetes node runs the kubernetes proxy in order to provide unified access for services and support load balancing across multiple replicas.

  1. Docker Installation
     The Docker engine installation works identical to listing 6.12.

  2. Changing the Docker daemon configuration
     The Docker configuration needs to changed just like on the master node. The command in listing 6.10 is used to disables iptables, ip masquerade and network bridging for the hosts docker configuration.

3. Installing flannel

```
1  # Get the master ip address from the blueprint
2  master_ip=ctx.instance.runtime_properties['master_ip']
3
4  # Start flannel using as docker container
5  pipe=subprocess.Popen(['sudo','docker',
       '-H','unix:///var/run/docker-bootstrap.sock','run',
       '-d','--net=host','--privileged', '-v','/dev/net:/dev/net',
       'quay.io/coreos/flannel:0.5.3', '/opt/bin/flanneld',
       '--etcd-endpoints=http://{}:4001'.format(master_ip)],
       stderr=open('/dev/null'),stdout=subprocess.PIPE)
```

Listing 6.17: Install flannel on worker node

4. Get flannel settings and adjust docker config

```
1  # get container id from previously started flannel container
2  cid=pipe.stdout.read().strip()
3  pipe.wait()
4
5  # get flannel subnet settings
6  pipe = subprocess.Popen(['sudo','docker',
7      '-H','unix:///var/run/docker-bootstrap.sock', 'exec',format(cid),
8      'cat','/run/flannel/subnet.env'], stdout=subprocess.PIPE,
9      stderr=subprocess.PIPE)
10 out, err = pipe.communicate()
11
12 # extract flannel subnet
13 result = out.decode()
14 flannel=";".join(result.split())
15
16 # Adjust local Docker configuration to use flannel subnet
17 with open("/tmp/docker","w") as fd:
18    with open("/etc/default/docker","r") as fdin:
19     for line in fdin:
20       fd.write(line)
21  with open("/tmp/docker","a") as fd:
22    fd.write(flannel+"\n")
23    fd.write('DOCKER_OPTS="--bip=${FLANNEL_SUBNET} --mtu=${FLANNEL_MTU}"\n')
24  subprocess.call("sudo mv /tmp/docker /etc/default/docker",shell=True)
```

Listing 6.18: Install flannel on worker node

5. Remove Docker bridge
   This step is identical to the one shown in listing 6.15 for the master node.

6. Install kubelet with master node

```
# get master ip and port from blueprint
master_ip=ctx.instance.runtime_properties['master_ip']
master_port=ctx.instance.runtime_properties['master_port']

# run kubelet
subprocess.call("sudo docker run --net=host -d -v
    /var/run/docker.sock:/var/run/docker.sock
    gcr.io/google_containers/hyperkube:v1.0.1 /hyperkube kubelet
    --api-servers=http://{}:{} --v=2 --address=0.0.0.0 --enable-server
    --hostname-override={} --cluster-dns=10.0.0.10
    --cluster-domain=cluster.local".format(master_ip, master_port),
    shell=True)
```

Listing 6.19: Run kubelet on worker node with master node IP

7. Start kubernetes service proxy
The kubernetes service proxy functions as a load balancer in order to provide services from a consistent address, while having multiple replications throughout the cluster.

```
subprocess.call("sudo docker run -d --net=host --privileged
    gcr.io/google_containers/hyperkube:v1.0.1 /hyperkube proxy
    --master=http://{}:{} --v=2".format(master_ip,master_port),shell=True)
```

Listing 6.20: Run kubelet proxy service

– `tasks.py`
The `tasks.py` file defines functions annotated with the `@operation` mark, in order for Cloudify to recognize them as Cloudify operations. The following operations have been defined:

  * `cloudify.kubernetes.relationships.connected_to_master` implementation
  * `cloudify.kubernetes.relationships.contained_in_host` implementation
  * `cloudify.kubernetes.Microservice` start operation: Writes the service description into a YAML file as required by the kubernetes documentation. Afterwards the file can be passed into the `kubectl create` command to create a service. The implementation also includes the usage of `kubectl run` and `kubectl expose` to cover the full deployment of the microservice.

7 Validation

The validation of the proposed system will be conducted by deploying a distributed application using a mixture of containerized and non-containerized application components. In order to deploy such application a modeling is required. The modeling will be done by creating a Cloudify YAML blueprint file. The blueprint will not only define the components and their relationships but also the underlying infrastructure resources, including the Iaas Cloud provider.

Figure 7.1 gives an overview about the validation process that will be applied. After setting up the environment and the Cloudify platform, the application can be modeled and deployed. To validate the application specific functions, the general application service is tested, followed by scaling the application components and then tearing the application down.

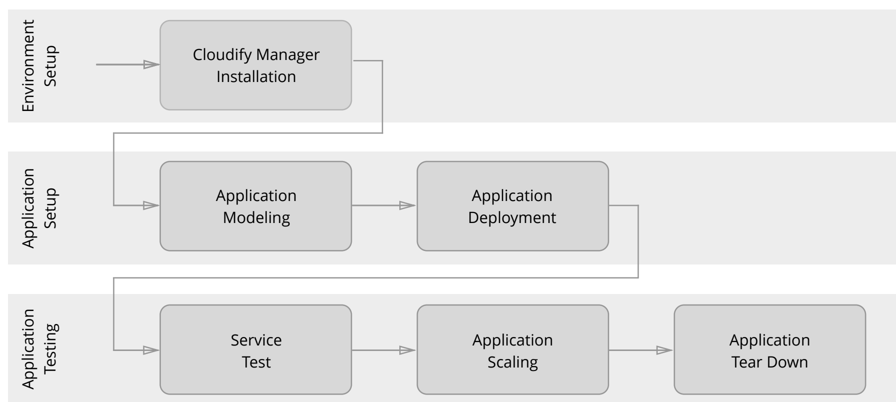

Figure 7.1: Validation Workflow

1. Cloudify Manager Installation
   The primary focus for this validation will be the deployment on Amazon AWS. Therefore a Cloudify Manager needs to be setup within AWS in order to provision the application nodes when required. Cloudify provides specific Manager Blueprints for different IaaS providers, which are available on GitHub. The AWS

Manager Blueprint[1] requires the AWS Access Key ID and Secret Access Key to make requests to AWS and provision resources. Besides the AWS credentials it's required to either specify a path to a local SSH key file to connect to the manager VM as well as the application VM's, which will be provisioned later on, or if non is specified a new one is created by Cloudify. The same applies for the AWS EC2 security groups, the user either needs to specify a group that will be applied to the manager and application VM's or a new one will be created during the bootstrapping process. The last required parameters are the image id as well as the instance type. For this validation the `CentOS Linux 7 x86_64 HVM EBS` image with a `m3.medium` instance type was used.

To execute the bootstrapping of the Manager VM the Cloudify CLI is required. Instructions on how to install the Cloudify CLI on all major platform can be found in the official documentation[2]. After the installation the following bootstrap command can be executed in order to provision the Cloudify Manager VM on AWS.

```
cfy bootstrap --install-plugins -p aws-ec2-manager-blueprint.yaml -i
    aws-ec2-manager-blueprint-inputs.yaml
```

Listing 7.1: Bootstrap Cloudify Manager VM on AWS

Afterwards the CLI should output the IP address of the Cloudify Manager VM and the user should be able to see the provisioned EC2 instance in the AWS dashboard.

The Cloudify Dashboard should now be accessible through a browser by the IP. Figure 7.2 shows the Cloudify Dashboard after the Manager Bootstrap process.

[1] https://github.com/cloudify-cosmo/cloudify-manager-blueprints/blob/master/aws-ec2-manager-blueprint.yaml

[2] http://docs.getcloudify.org/3.3.1/installation/from-packages/

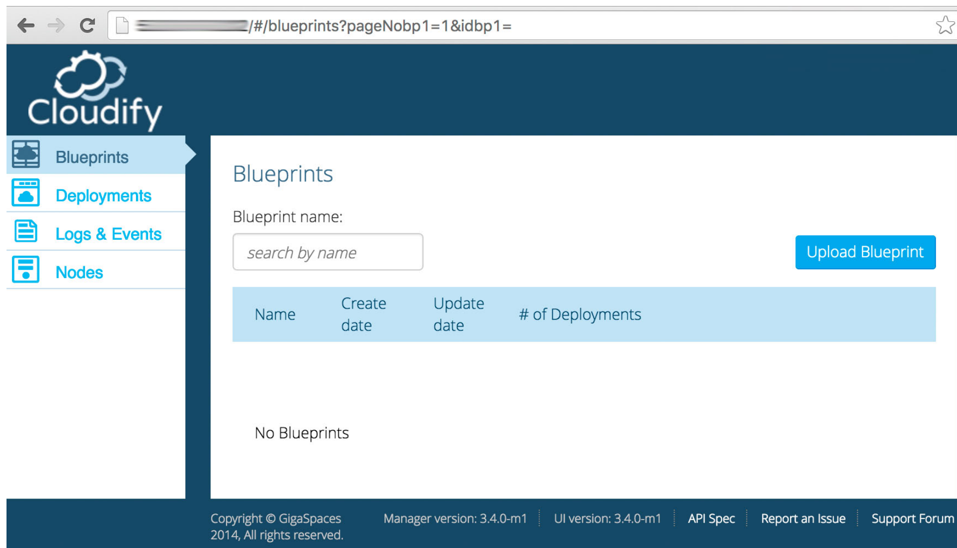

Figure 7.2: Cloudify Dashboard on AWS

2. Application Modeling
   After the successful deployment of the Manager VM on AWS, the application itself needs to be modeled using the Cloudify YAML blueprint format.

   The application that will be deployed consists of three different components and derives from the official Cloudify nodecellar example[3]. Figure 7.3 shows a containerized web application deployed on a kubernetes cluster using Docker, which retrieves data from a database that is located outside of the kubernetes cluster and runs on a separate EC2 instance. The application blueprint will use the kubernetes plugin introduced in Chapter 5 and also the Cloudify AWS plugin[4] to provision infrastructural resources on AWS.

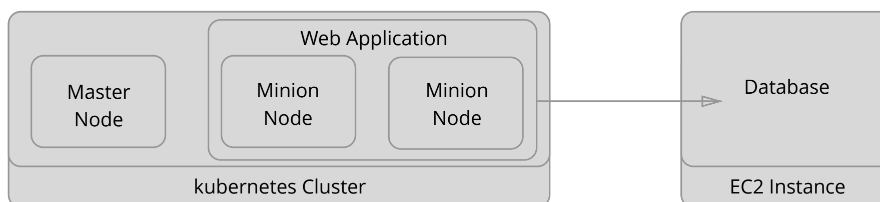

Figure 7.3: Application Overview

[3]https://github.com/cloudify-cosmo/cloudify-nodecellar-example
[4]https://github.com/cloudify-cosmo/cloudify-aws-plugin

Listing 7.2 shows the first part of the blueprint. A MongoDB database is specified to be run on an EC2 instance. Additionally a specifically assigned security group is defined, which will later on allow the web application to access the data by using the default MongoDB port 28017.

```
mongod_host:
  type: cloudify.aws.nodes.Instance
  properties:
    image_id: {get_input: image}
    instance_type: {get_input: size}
    agent_config:
      user: { get_input: agent_user }
  relationships:
    - type: cloudify.aws.relationships.instance_connected_to_security_group
      target: mongod_security_group

 mongod_security_group:
  type: cloudify.aws.nodes.SecurityGroup
  properties:
    resource_id: mongod_security_group
    description: mongod security group
    rules:
      - ip_protocol: tcp
        from_port: { get_property: [ mongod, port ] }
        to_port: { get_property: [ mongod, port ] }
        cidr_ip: 0.0.0.0/0
      - ip_protocol: tcp
        from_port: 28017
        to_port: 28017
        cidr_ip: 0.0.0.0/0

 mongod:
  type: nodecellar.nodes.Mongod
  instances:
    deploy: 1
  properties:
    port: 27400
  relationships:
    - type: cloudify.relationships.contained_in
      target: mongod_host
  interfaces:
    cloudify.interfaces.lifecycle:
      create: scripts/mongo/install-mongo.sh
      start: scripts/mongo/start-mongo.sh
      configure: scripts/mongo/install-pymongo.sh
      stop: scripts/mongo/stop-mongo.sh
```

Listing 7.2: MongoDB Blueprint Modeling

Further Listing 7.3 shows the modeling of the kubernetes cluster containing a master node and a minion node. The master node host gets a specific security group, a separate Elastic IP and a relationship that binds to the kubernetes cluster. The minion node host is part of the same security group and also gets bind to the same kubernetes cluster. The security group opens all ports required to access the kubernetes dashboard and the kubernetes API.

```
kubernetes:
  type: cloudify.nodes.Tier

master_ip:
  type: cloudify.aws.nodes.ElasticIP

master_host:
  type: cloudify.aws.nodes.Instance
  properties:
    image_id: { get_input: image }
    instance_type: { get_input: size }
    agent_config:
      user: { get_input: agent_user }
  relationships:
    - target: kubernetes
      type: cloudify.relationships.contained_in
    - target: master_security_group
      type: cloudify.aws.relationships.instance_connected_to_security_group
    - type: cloudify.aws.relationships.instance_connected_to_elastic_ip
      target: master_ip

minion_host:
  type: cloudify.aws.nodes.Instance
  properties:
    image_id: { get_input: image }
    instance_type: { get_input: size }
    agent_config:
      user: { get_input: agent_user }
  relationships:
    - target: master_security_group
      type: cloudify.aws.relationships.instance_connected_to_security_group
    - target: kubernetes
      type: cloudify.relationships.contained_in

master_security_group:
  type: cloudify.aws.nodes.SecurityGroup
  properties:
    resource_id: master_security_group
    description: kubernetes master security group
    rules:
```

```
    - ip_protocol: tcp
      from_port: { get_property: [ master, master_port ]}
      to_port: { get_property: [ master, master_port ]}
      cidr_ip: 0.0.0.0/0
    - ip_protocol: tcp
      from_port: 4001
      to_port: 4001
      cidr_ip: 0.0.0.0/0
    - ip_protocol: tcp
      from_port: 3000
      to_port: 3000
      cidr_ip: 0.0.0.0/0
```

Listing 7.3: Kubernetes Cluster Modeling

Lastly Listing 7.4 shows how the services get assigned to the hosts which have been model in the previous listings. The nodecellar web application is handled as a kubernetes microservice that gets deployed through the master node and thus requires additional parameters.

```
master:
  type: cloudify.kubernetes.Master
  properties:
    install: true
    install_docker: true
  relationships:
    - type: cloudify.kubernetes.relationships.contained_in_host
      target: master_host

minion:
  type: cloudify.kubernetes.Node
  properties:
    install_docker: true
  relationships:
    - type: cloudify.kubernetes.relationships.connected_to_master
      target: master
    - type: cloudify.relationships.contained_in
      target: minion_host

nodecellar:
  type: cloudify.kubernetes.Microservice
  properties:
    config_path: service.yaml
    config_overrides:
      - "['spec']['containers'][0]['env'][0]['value'] = '@{mongod_host,ip}'"
      - { concat: ["['spec']['containers'][0]['env'][1]['value']=","'",{
          get_property: [mongod,port]},"'"]}
```

```
relationships:
  - type: cloudify.relationships.contained_in
    target: master_host
  - type: cloudify.kubernetes.relationships.connected_to_master
    target: master
  - type: cloudify.relationships.connected_to
    target: mongod
```

Listing 7.4: Application Service Modeling

The `service.yaml` contains the specifics about what that needs to be deployed into the kubernetes cluster. This could either be a simple kubernetes pod or specification for a service, a replication controller or a kubernetes deployment.

After the blueprint has been specified, it needs to be packaged including the kubernetes plugins, which can be skipped if the plugin has been published online and specified in the blueprint accordingly.

3. Application Deployment
   The deployment of a blueprint on Cloudify requires two steps, the blueprint deployment and the deployment of the application itself. After the initial blueprint upload a validation check on the topology will be triggered. If the validation was successful the topology is displayed graphically on the Cloudify dashboard. Figure 7.4 shows the topology of the nodecellar example application using a kubernetes cluster. This gives users the possibility to double check the blueprint modeling and the entire topology.

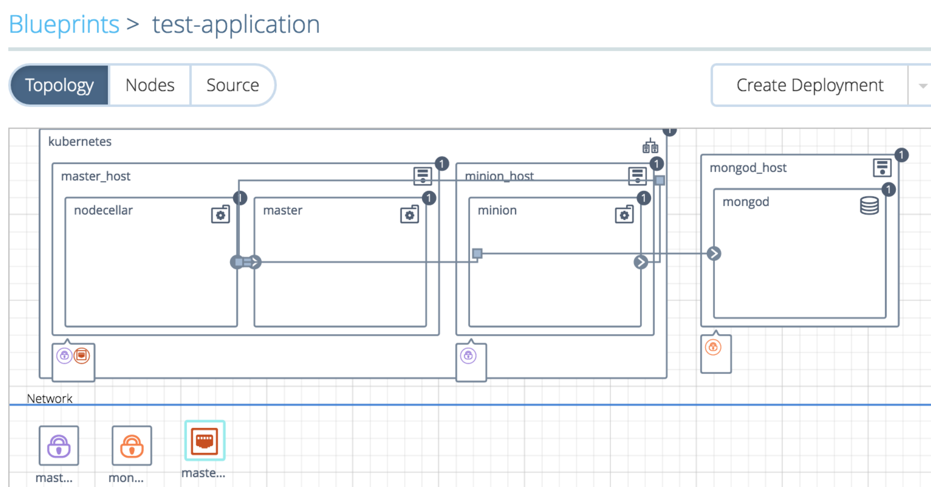

Figure 7.4: Cloudify Dashboard - Blueprint Topology Overview

The blueprint deployment itself, can be started by clicking "Create Deployment" and specifying a name, an image for the virtual machine (AMI ID) and the appropriate agent user name as well as the EC2 instance type. Figure 7.5 shows the deployment creation dialog with the test input. After the creation dialog was confirmed the blueprint deployment will be executed.

Deploy test-application ×

Please enter deployment name

test-application-deployment

Parameters Raw

image: ami-fce3c696

agent_user: ubuntu

size: t2.large

Create

Figure 7.5: Cloudify Dashboard - Create Deployment Dialog

The Deployments tab in the Dashboard gives an overview about all currently deployed blueprints (see Figure 7.6). By clicking "Execute Workflow" and then selecting the Install workflow, the deployment and provisioning of the entire application gets started.

Cloudify

Blueprints
Deployments
Logs & Events
Nodes

Deployments

Deployment name: search by name
Blueprint: Choose Blueprints

ID	Blueprint	Created	Updated	Action
test-application-deployment	test-application	2016-06-12 20:48:15	2016-06-12 20:48:15	Execute Workflow

Figure 7.6: Cloudify Dashboard - Blueprint Deployment Overview

After the process has been completed the user can double check the AWS dashboard if the instances have been provisioned.

4. Service Test
   The Logs & Events tab gives detailed information about the status of the deployment and if the install workflow has been executed successfully an event with the type "Workflow ended successfully" will be logged. Afterwards the user can access the kubernetes API through the IP of the kubernetes master node. The endpoint `/api/v1/pods` gives information about the currently active pods in the cluster. Listing 7.5 shows the determining section for the nodecellar example application that has been deployed.


```
...
"metadata": {
      "name": "nodecellar",
      "namespace": "default",
      "selfLink": "/api/v1/namespaces/default/pods/nodecellar",
      "uid": "42c1a2b5-30eb-11e6-a19a-0a87ce970989",
      "resourceVersion": "72",
      "creationTimestamp": "2016-06-12T22:16:07Z"
    },
    "spec": {
      "volumes": [
        {
          "name": "default-token-pfz9l",
          "secret": {
            "secretName": "default-token-pfz9l"
          }
        }
      ],
      "containers": [
        {
          "name": "nodecellar",
          "image": "dfilpi/nodecellar:v1",
          "command": [
            "../node/bin/node",
            "server.js"
          ],
          "workingDir": "/root/nodecellar-master",
          "ports": [
            {
              "hostPort": 3000,
              "containerPort": 3000,
              "protocol": "TCP",
              "hostIP": "0.0.0.0"
            }
            ...
```


Listing 7.5: kubernetes API Pods endpoint

The user can also access the web application now by accessing the master node on port 3000 using a browser.

5. Application Scaling
   The scaling of the application can also be done using the Cloudify Dashboard. It is possible to scale the replication count of a kubernetes deployment, replication controller or service (see Figure 7.7).

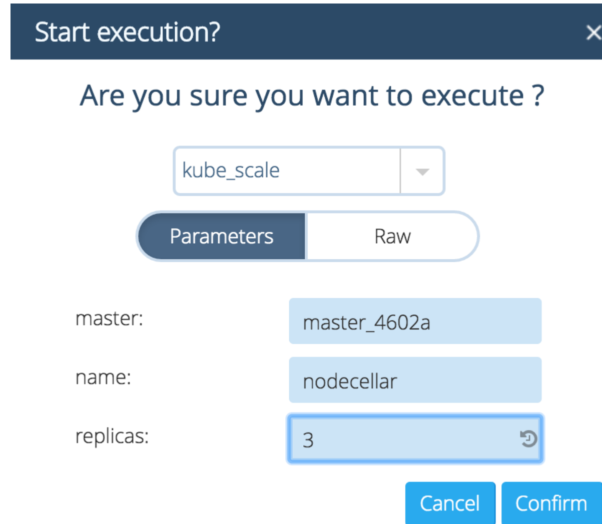

Figure 7.7: Cloudify Dashboard - kubectl scale Workflow

Additionally Cloudify provides support to scale provisioned nodes up and down. Figure 7.8 shows the dialog for the scaling of nodes outside of kubernetes. After the dialog has been confirmed Cloudify will provision a new node and load balance all requests among them.

Figure 7.8: Cloudify Dashboard - Regular Scale Workflow

6. Application Tear Down
   An application tear down is being handled in a fully automated fashion using the Cloudify dashboard. The uninstall workflow will shut down the service and terminate all provisioned virtual machines and other resources on AWS.

8 Conclusion & Future Work

The provisioning and management of distributed applications in the cloud is a complex process. Orchestration platforms like Cloudify help to provide a high level of platform interoperability and reduce the complexity of deploying and managing applications throughout their entire lifecycle. Because of the significant popularity of containerized applications a Cloudify kubernetes plugin for the Cloudify ecosystem was introduce, in order to enable the provisioning of containerized applications. The plugin adds support for the modeling of kubernetes clusters, containerized applications and their entire lifecycle. It became clear that features like scaling, load balancing and proper service discovery are very specific for containerized applications and couldn't easily be added to a orchestration engine without the use of a container cluster technology.

The fundamentals in Chapter 2 have covered basics about container virtualization and cloud computing, while also looking into the market and paying special attention towards the current offerings in the PaaS section of all major cloud providers for container management services. It was shown that even Amazon ECS, Google Container Service and Microsoft Azure Container Service utilize different container cluster technologies to manage Docker containers. In Chapter 3 similar approaches to enable the orchestration and management of containerized applications have been analyzed, while drawing special attention towards platform interoperability. The most interesting approach was Ubernetes, which enables the federation of multiple kubernetes clusters on different platforms by applying an other API layer on top of all deployed kubernetes clusters. By the time of this writing the project only reached proposal status and therefore couldn't be included in the proposed implementation. In Chapter 4 the non-functional and functional requirements of a system which deploys containerized applications are listed. It became clear that such requirements can only be fulfilled by using a container cluster technology in combination with an orchestration platform. The proposed system integrates a container cluster as a separate node type into the platform including the handling of all lifecycle operations for the cluster and its container services. The underlying runtime would then also be responsible for connecting the container services with other provisioned resources.

Chapter 5 then maps the requirements and the proposed system to specific technologies. The system is based on the Cloudify orchestration engine and platform, which provides

compliance to the TOSCA specification and supports modeling via the TOSCA Simple YAML Profile format. After comparing different container cluster technologies, kubernetes was selected due to an easier overall deployment and service discover process. Further the chapter outlines the inclusion into the Cloudify ecosystem by specifying the needs of a custom Cloudify plugin to inject kubernetes node types and functionality to manage the cluster and deploy containers on it.

The implementation of the system is stated in Chapter 6. A Cloudify plugin is introduced which bootstraps an entire multi-node kubernetes cluster using Docker itself on separate nodes within Cloudify. The implementation follows official instructions in the kubernetes documentation and wraps them into the Cloudify plugin written in Python. Management operations are executed using the `kubectl` command line tool through the kubernetes master node. Additionally custom node types are being defined to be available for application modeling and provisioning.

The Validation described in Chapter 7 deploys a distributed application including non-containerized and containerized components to verify the use with a practical application. The process includes the modeling of a web application using a database running on a VM, while running the web application itself on the kubernetes cluster. The result show that the deployment was successfully created and a following scaling procedure could also be successfully executed. It is noteworthy that the implementation as of right now is only suitable for the use within AWS, but should be easily transferable to other cloud providers. If the already mentioned Ubernetes project gets into development it would be easier to include a Ubernetes cluster into Cloudify, which would provide shift the cross-provider support out of Cloudify towards Ubernetes. Further the deployment of multiple container cluster could be considerer in the future.

Bibliography

[Ama16] Amazon Web Services, Inc. *AWS Global Infrastructure*. Mar. 2016. URL: https://aws.amazon.com/about-aws/global-infrastructure/ (cit. on p. 20).

[BBH+13] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, and S. Wagner. "Service-Oriented Computing: 11th International Conference, ICSOC 2013, Berlin, Germany, December 2-5, 2013, Proceedings." In: ed. by S. Basu, C. Pautasso, L. Zhang, and X. Fu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013. Chap. OpenTOSCA – A Runtime for TOSCA-Based Cloud Applications, pp. 692–695. URL: http://dx.doi.org/10.1007/978-3-642-45005-1_62 (cit. on pp. 23, 26).

[BBLS12] T. Binz, G. Breiter, F. Leyman, and T. Spatzier. "Portable Cloud Services Using TOSCA." In: *IEEE Internet Computing* 16.3 (2012), pp. 80–85 (cit. on p. 23).

[Ber14] D. Bernstein. "Containers and Cloud: From LXC to Docker to Kubernetes." In: *IEEE Cloud Computing* 1.3 (Sept. 2014), pp. 81–84 (cit. on pp. 29, 30).

[BGPV12] L. Badger, T. Grance, R. Patt-Corner, and J. Voas. *Cloud Computing Synopsis and Recommendations: Recommendations of the National Institute of Standards and Technology*. USA: CreateSpace Independent Publishing Platform, 2012 (cit. on pp. 17–19).

[Boe15] C. Boettiger. "An Introduction to Docker for Reproducible Research." In: *SIGOPS Oper. Syst. Rev.* 49.1 (Jan. 2015), pp. 71–79. URL: http://doi.acm.org/10.1145/2723872.2723882 (cit. on pp. 29, 30).

[Cel16] Celery Project. *Official Website*. May 2016. URL: http://www.celeryproject.org/ (cit. on p. 28).

[Cor16a] CoreOS, Inc. *ETCD Open Source Repository*. May 2016. URL: https://github.com/coreos/etcd (cit. on p. 63).

[Cor16b] CoreOS, Inc. *Flannel Open Source Repository*. May 2016. URL: https://github.com/coreos/flannel (cit. on p. 63).

[Dav06] T. S. David Groth. *Network+ Study Guide: Exam N10-003*. Wiley Publishing, Inc., Feb. 2006 (cit. on p. 22).

[DRK14] R. Dua, A. R. Raja, and D. Kakadia. "Virtualization vs Containerization to Support PaaS." In: *Cloud Engineering (IC2E), 2014 IEEE International Conference on*. Mar. 2014, pp. 610–614 (cit. on p. 29).

[DWC10] T. Dillon, C. Wu, and E. Chang. "Cloud Computing: Issues and Challenges." In: *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*. Apr. 2010, pp. 27–33 (cit. on p. 19).

[FLMS11] C. Fehling, F. Leymann, R. Mietzner, and W. Schupeck. "A collection of patterns for cloud types, cloud service models, and cloud-based application architectures." In: *University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, University of Stuttgart, Institute of Architecture of Application Systems, Technical Report Computer Science* 5 (2011) (cit. on p. 19).

[Gar07] S. Garfinkel. *An Evaluation of Amazon's Grid Computing Services: EC2, S3, and SQS*. Technical Report TR-08-07. Harvard Computer Science Group, Aug. 2007 (cit. on p. 20).

[Gig16a] GigaSpaces Technologies. *Cloudify Architecture Overview*. May 2016. URL: http://getcloudify.org/guide/3.0/overview-architecture.html (cit. on p. 28).

[Gig16b] GigaSpaces Technologies. *Cloudify Hello World Example*. May 2016. URL: https://github.com/cloudify-cosmo/cloudify-hello-world-example (cit. on pp. 11, 57).

[Gig16c] GigaSpaces Technologies. *Cloudify Kubernetes OpenStack Plugin*. May 2016. URL: https://github.com/cloudify-examples/cloudify-kubernetes-plugin-blueprint (cit. on p. 63).

[Gig16d] GigaSpaces Technologies. *Cloudify Reference Builtin Workflows*. May 2016. URL: http://getcloudify.org/guide/3.0/reference-builtin-workflows.html (cit. on p. 69).

[Gig16e] GigaSpaces Technologies. *What is Cloudify*. May 2016. URL: http://docs.getcloudify.org/3.3.1/intro/what-is-cloudify/ (cit. on p. 27).

[HLNW14] F. Haupt, F. Leymann, A. Nowak, and S. Wagner. "Lego4TOSCA: Composable Building Blocks for Cloud Applications." In: *2014 IEEE 7th International Conference on Cloud Computing*. June 2014, pp. 160–167 (cit. on p. 27).

[Kub16a] Kubernetes. *Documentation kubectl Overview*. May 2016. URL: http://kubernetes.io/docs/user-guide/kubectl-overview/ (cit. on pp. 11, 65, 66).

[Kub16b] Kubernetes. *Kubernetes Docker Multinode Setup - Master*. May 2016. URL: http://kubernetes.io/docs/getting-started-guides/docker-multinode/master/ (cit. on p. 63).

[Kub16c] Kubernetes. *Ubernetes Architectural Proposal*. May 2016. URL: https://github.com/kubernetes/kubernetes/blob/master/docs/proposals/federation.md (cit. on p. 38).

[Kub16d] Kubernetes. *What Is Kubernetes*. May 2016. URL: http://kubernetes.io/docs/whatisk8s/ (cit. on p. 52).

[LMVF11] C. Liu, Y. Mao, J. Van der Merwe, and M. Fernandez. "Cloud resource orchestration: A data-centric approach." In: *Proceedings of the biennial Conference on Innovative Data Systems Research (CIDR)*. Citeseer. 2011, pp. 1–8 (cit. on p. 23).

[Mer14] D. Merkel. "Docker: Lightweight Linux Containers for Consistent Development and Deployment." In: *Linux J.* 2014.239 (Mar. 2014). URL: http://dl.acm.org/citation.cfm?id=2600239.2600241 (cit. on p. 29).

[OAS13] OASIS. *Topology and Orchestration Specification for Cloud Applications Version 1.0*. Specification. Nov. 2013 (cit. on pp. 23, 24).

[OAS16a] OASIS. *Topology and Orchestration Specification for Cloud Applications Version 1.0*. May 2016. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.pdf (cit. on p. 13).

[OAS16b] OASIS-Open. *TOSCA Simple Profile in YAML Version 1.0*. Specification. Feb. 2016 (cit. on pp. 11, 24, 25).

[Ope16] OpenStack Foundation. *OpenStack Heat Documentation*. May 2016. URL: https://wiki.openstack.org/wiki/Heat (cit. on p. 38).

[Pyt16a] Python Software Foundation. *Python Documentation 2.7.11: os - Miscellaneous operating system interfaces*. May 2016. URL: https://docs.python.org/2/library/os.html (cit. on p. 71).

[Pyt16b] Python Software Foundation. *Python Documentation 2.7.11: Subprocess management*. May 2016. URL: https://docs.python.org/2/library/subprocess.html (cit. on p. 71).

[QLDG09] L. Qian, Z. Luo, Y. Du, and L. Guo. "Cloud Computing: First International Conference, CloudCom 2009, Beijing, China, December 1-4, 2009. Proceedings." In: ed. by M. G. Jaatun, G. Zhao, and C. Rong. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. Chap. Cloud Computing: An Overview, pp. 626–631. URL: http://dx.doi.org/10.1007/978-3-642-10665-1_63 (cit. on p. 20).

[Sav15] J. Savill. *Mastering Microsoft Azure Infrastructure Services*. Indianapolis, Ind. : Sybex, a Wiley brand, 2015 (cit. on p. 20).

[Sta16] Statista. *Global market share of cloud infrastructure services in 2014, by company*. May 2016. URL: http://www.statista.com/statistics/477277/cloud-infrastructure-services-market-share/ (cit. on p. 20).

[Ste10] A. Stevenson. *Oxford Dictionary of English*. Oxford University Press, USA; 3rd Revised ed. edition (August 1, 2010), 2010 (cit. on p. 22).

[Tre13] R. Trefft. "Design and development of a generic file service for OpenTOSCA." MA thesis. University of Stuttgart, 2013 (cit. on p. 25).

[Tul13] M. Tulloch. *Introducing Windows Azure for IT Professionals*. Introducing. Pearson Education, 2013. URL: https://books.google.de/books?id=tlxuAwAAQBAJ (cit. on p. 21).

[Tur14] J. Turnbull. *The Docker Book*. James Turnbull, 2014. URL: https://books.google.de/books?id=CtMEBwAAQBAJ (cit. on p. 31).

[Uni16] Universität Stuttgart. *OpenTOSCA - Open Source TOSCA Ecosystem*. May 2016. URL: http://www.iaas.uni-stuttgart.de/OpenTOSCA/ (cit. on p. 25).

[VRCL08] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. "A Break in the Clouds: Towards a Cloud Definition." In: *SIGCOMM Comput. Commun. Rev.* 39.1 (Dec. 2008), pp. 50–55 (cit. on p. 17).

[XNR+13] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. D. Rose. "Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments." In: *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. Feb. 2013, pp. 233–240 (cit. on p. 30).

All links were last followed on June 13, 2016.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature