SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Multicloud Infrastructure Broker for Edge Orchestration Framework

Michael Schlicker

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Multicloud Infrastructure Broker for Edge Orchestration Framework

Multicloud Infrastructure Broker für Edge Orchestration Framework

Author:	Michael Schlicker
Supervisor:	Prof. Dr.-Ing. Jörg Ott
Advisor:	Dr. Nitinder Mohan, Giovanni Bartolomeo
Submission Date:	15.12.2023

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 15.12.2023 Michael Schlicker

Abstract

Up-and-coming technologies like Augmented Reality, Internet of Things and Artificial Intelligence are trending towards increased use of edge computing philosophy, moving bandwidth heavy and low latency applications towards the user on the edge. However, there are still many use cases from offloading and usage peaks to specialty needs when an edge network alone reaches its limitations. In these cases using some cloud resources is very advantageous but integrating them in a way without risking a lock-in to one cloud provider is still difficult. Many frameworks that offer both cloud provisioning and orchestration are mostly designed for use in data centers under assumption that are too strict in a volatile edge environment. Here this thesis proposes a novel design for a multi-cloud broker which works together with an edge computing orchestration framework, enabling access to cloud resources from multiple providers directly from the edge. An example of this design is implemented for the Oakestra framework in a modular design which allows the dynamic extension for more cloud providers using a provider-agnostic interface. The implementation of the resource creation operations have been evaluated and are competitive with multi-cloud sky brokers designed for non-edge usage.

Kurzfassung

Aufstrebende Technologien wie erweiterte Realität, Internet der Dinge und künstliche Intelligenz tendieren zu vermehrter Nutzung der Edge Computing Philosophie, welche datenintensive und latenzempfindliche Anwendungen näher zum Nutzer. Allerdings, gibt es weiterhin viele Nutzungsszenarien bei denen Cloud Ressourcen sehr vorteilhaft sind allerdings ist das Integrieren dieser ohne dem Risiko des Lock-in Effektes zu einem Cloud-Anbieter noch immer schwierig. Viele Frameworkprogramme bieten zwar sowohl Cloud-Bereitstellung als auch Orchestrierungs Funktionalitäten allerdings unterstehen diese oft zu strikten Voraussetzungen, die in unbeständigen Edge Umgebungen nicht garantiert werden können. Diese Arbeit präsentiert ein neuartiges Design für einen Multi-Cloud Broker, welcher in Zusammenarbeit mit einem Edge Computing Orchestrierungsframework es ermöglicht direkt von der Edge Zugang zu Cloud Ressourcen zu schaffen. Eine Beispielimplementierung wurde umgesetzt auf Basis des Oakestra Frameworks in einem modularen Design, welche die dynamische Erweiterung um weitere Cloud Anbieter ermöglicht über eine anbieteragnostische Schnittstelle. Die Implementierung der Ressourcenerstellungsoperationen wurden ausgewertet und sind wettbewerbsfähig mit Multi-Cloud Sky Brokern, die für nicht Edge-Nutzung konzipiert wurden.

Contents

1 Introduction

1.1 Problem Statement

The increased popularity of edge-computing applications like the Internet of Things [26], Artificial Intelligence [20], and Augmented Reality [6] relying on low latency with high bandwidth usage that shouldn't waste capacity has sparked research on new edge frameworks. However, in many settings, available edge resources might not satisfy the requirements thoroughly, needing cloud computing to contribute additional resources to bridge the gap.

Provisioning them can be much faster, but in a fragmented cloud provider market [12], setting up the correct instances can involve many detailed and provider-specific steps leading together with data gravity to vendor lock-in [25].

Existing multi-cloud provisioning and orchestration applications too often rely on strong assumptions from a reliable world in the cloud, which is not given in mobile and edge environments. On the other hand, edge computing orchestrators often work in their trusted environment of contributed common resources, which come at a little extra cost.

1.2 Contribution

This thesis provides an approach for a solution by abstracting the necessary operations to set up and manage cloud infrastructure in a provider-agnostic form so that it can easily be extended to support many additional cloud providers. It introduces a novel multi-cloud broker that can perform and manage these complex operations of creating and provisioning clusters and workers automated and integrated into an existing edge orchestration framework. An implementation of this is evaluated against a somewhat comparable framework with the same goal but coming from a batch jobs side.

1.3 Thesis Structure

This thesis first introduces the basic concepts in Chapter 2, focusing on edge and multi-cloud computing and orchestrating services on them. Then, it will introduce in

Chapter 3 a design for such a multi-cloud broker for this edge environment with a detailed analysis of requirements. An implementation of this approach is presented in Chapter 4, including a detailed explanation of how modularity is achieved for multiple cloud providers. This solution is then tested in Chapter 5, and its speed performance is evaluated against the sky computing framework Sky Pilot. Finally, in Chapter 6, an outlook will show the future potential of this design when extended and deeply integrated into an orchestration framework.

2 Background

2.1 Edge Computing

2.1.1 Concept

One of the most promising distributed paradigms is edge computing. The core idea is to decentralize the computing needs from a few powerful units to many small ones, which are then placed closer to consumers in various geographic areas. [16]. This brings several benefits ranging from significantly shorter latencies and reduced bandwidth usage due to the shorter distance data has to travel to lower power consumption taking advantage of common energy efficiency in smaller units. [6] Also, it's possible to include nodes with very specialized hardware that is often not realizable in cloud infrastructure. [3]

On the downside, they introduce additional challenges to the generic distributed computing ones that break common underlying assumptions of cloud computing methodologies and techniques. [4] The first one is the wide heterogeneity of resources, so while one management application might work on one type, it might not work on very specific or very restricted hardware, so the Quality of Service requirements of such software might not be met, which creates these trade-off problems. Another issue is networking and, in particular, the bandwidth, as small devices often cannot handle as much incoming and outgoing data as specialized cloud infrastructure. Also, their networks tend to be more unreliable than at robust data centers, especially in mobile edge computing environments. [17] Here, the nodes need to be managed and operated in a way that is aware that nodes can be unavailable at any time and be very fault-tolerant.

2.1.2 Types of Edge Computing

There has been an emergence of various types that try to bridge the gap between edge and cloud computing, with one of the most notable being. **Fog Computing** It comes more from the cloud computing side and tries to preserve most of its benefits but layers its cloud resources hierarchically. This allows for some of them to be right at the edge of the cloud network, close to consumers. [11]

A similar concept is **Cloudlets**, which are small cloud-like data centers located close to the end-users. They still offer cloud-like services with more dense geographical coverage, reducing latencies, better reliability, and more computing power than pure edge computing. [1]

A third interesting approach is **Mobile Edge Computing (MEC)**. It integrates computing power and data storage inside a cellular provider's radio access network. It is increasingly common with 5G networks. This allows mobile devices to connect with these resources directly without ever needing to leave the networks, reducing latency and bandwidth use.[26]

2.1.3 Application Areas

While edge computing has still not that many real-life application, in certain scenarios its benefits are especially highlighted and useful. [23]

Since low latency is a one of the biggest advantages of edge computing it is a good approach to use edge processing power to offload time-critical but computing-intensive applications like Artificial or Virtual Reality from small wearable devices. [6]

Another good use case is artificial intelligence and especially Deep Neural Networks (DNN) as they have to process huge amount of data as an input and using edge computing they can span their network closer to the information source reducing the large amount of bandwith needed for the data transfer of raw input to some remote processing units. [20]

The MEC servers inside cellular networks are especially helpful for video streaming applications since these are among the largest bandwith applications. By caching some content in the networks a lot of backbone traffic to datacenters can be eliminated. [26]

Also with the increasing number of internet of things devices it makes sense for them to not always have to communicate back to a remote data center but instead take advantage especially since each IoT device is basically already a node in an edge network. Examples for these can be autonomous vehicles. [26]

2.2 Cloud Computing

2.2.1 Concept

Cloud Computing as a concept is best defined by the NIST [39] using five essential characteristics. They are computing capabilities in datacenters shared by multiple consumers via **resource pooling** that can be acquired via **on-demand self-service** without requiring a human interaction with the resource provider. These resources have **rapid elasticity** so that can be quickly scaled with changing demand while

providers optimize this resource use automatically by metering with a **measured service**. Consumers can interact with and use these capabilities via **broad network access** from heterogenous thin clients.

This allows customers a vast flexibility both in terms of scale and in time leading to fewer unused excess resources. Along with the effects of economies of scale from a cloud provider perspective it makes cloud computing at competitive price attractive to many companies. [14]

Besides the commonly associated **public cloud** providers which are available for use by any entity of the general public, the NIST also classifies additional deployment models. The opposite would be a **private cloud** which is exclusively used by (multiple) consumers of a single organization while still offering similar services just on organization-dedicated hardware managed either by themselves or a third party. When such a non-public cloud is available to a specific set of consumers from multiple organizations which share concerns like a common mission it is then often called **community cloud**. Additionally each of these distinct cloud infrastructures can also be combined to a common **hybrid cloud** approach, where these unique entities are tied together in a bigger architecture.

Service Models

These cloud computing offerings are then again categorized into four distinct service model types depending the level at which cloud resources are offered with different amount of abstracted setup.

The most low-level one of them is **Infrastructure-as-a-Service (IaaS)**. Here cloud providers offer individual resources like virtual machines, storage and networking to consumers which through virtualization act similar to dedicated on premise hardware while in the real underlying cloud infrastructure is abstracted away. This type gives the consumer total control over the operating systems, allows them to run arbitrary software and even control parts of the networking by e.g. hosting firewalls.

The next higher level is **Platform-as-a-Service (PaaS)**. Here the previously mentioned basic infrastructure is already provided and not controllable by the consumer. They can however deploy and run any applications which are supported by the underlying selected platform.

The third common service model is **Software-as-a-Service (SaaS)** where consumers directly consume provided applications running on cloud infrastructure without managing any of infrastructure and application deployment and the only customizations are in-app configurations.

Finally there has been recently the emergence of a fourth another service model **Function-as-a-Service (FaaS)** which is not yet defined like the previous ones by the

NIST [39].

Also known as Serverless Computing it completely abstracts the use of continuously running servers all together and instead offers the development of specific functions or tasks which are started by specific triggers or requests without any knowledge of the underlying programs, platforms or infrastructure environment [37].

One of the most intuitive analogies between the first three types with the process of getting and eating a Pizza has been provided by [2]. It compares in Figure X the traditional on premises computing with making a pizza at home to either baking a pizza at home with pre-purchased ingredients (IaaS), getting pizza delivered (PaaS) and dining out (SaaS). At each step the vendor manages more of the process and the consumer less of it.

2.2.2 Cloud Providers

Although there are hundreds of cloud providers around the world, just three biggest providers themselves (Amazon Web Services, Microsoft Azure and Google Cloud Platform) have reached a market share of 65% of all spending on cloud computing [12]. However many of the other cloud providers are relevant be it through special hardware, regulatory or commercial offerings.

2.2.3 Core Functionalities

While each cloud provider has their own set of features and APIs most of providers implement a similar set of core functionalities. The most important IaaS ones needed to implement are here explained by the example of the top three providers.

Virtual Machines

The core of Infrastructure-as-a-Service computing are virtual machine instances running on the virtualized abstracted datacenter hardware. These services are respectively called at the big three providers AWS Elastic Cloud 2 [36], GCP Compute Engine [10] and Azure Virtual Machines [21].

Instance Type Each Virtual Machine instance has a number of virtual CPUs, GPUs and memory available to it to perform computing tasks. These are usually grouped into different **instance type** configurations that consumers can choose from and vary in size and price. Many offer configurations grouped into families with different focuses like general purpose, compute- or memory-optimized and speciality accelerated hardware configurations.

Image Type The second major computing configuration is the selection of a template disk image from which instances are initially booted upon launching. These images contain primarily an installation of an operating system but can also have applications and services pre-installed. Some are offered by the provider themselves, third party image providers and these images can be created by consumers themselves as a snapshot from an existing root volume of a previously launched instance. However, since providers use different underlying virtual machines those images usually have different disk formats which can not interchangeably be used between them [14].

Networking

As these public clouds are not hosted at the consumers on premise they need to be reachable via the public internet. The instances can communicate both internally with other instances and externally with clients outside of a particular cloud. Providers offer therefore numerous network configuration and setup options.

Virtual Private Cloud One of the most commonly used ones are virtual private clouds (VPCs) which allows to logically isolate multiple instances into a virtual network separate from instances other consumers. These networks offer various services from subnets, routing to gateways.

Firewall A standard networking configuration in cloud computing are firewalls which restrict incoming and outgoing requests to certain ports and white- or blacklisting IP ranges. These firewall configuration can be reused via identifiers and attached in an additive way to instances.

Data Storage

As many applications for cloud computing need to persist data it's storage is another core functionality. There are usually two different types of storage first a low-latency disk / block storage used primarily as volumes by instances for both system and application data. For longer-term or larger data storage cloud providers offer dedicated storage services at reduced prices like AWS S3, GCP Storage and Azure Blob Storage. They are again classified into different storage classes from regular to long-term archive storage with varying retrieval time and minimum storage durations. Besides general purpose storage there are also specialized Platform-as-a-Service storage solutions like Key-Value, Relational Database and Secrets Management services which are optimized for these purposes.

Identity and Access Management

Since resources and services in the cloud incur costs access to them have to be regulated by robust identity and access management services (IAM). Additionally applications on these resources can contain sensitive data and should be protected from unauthorized interference. These IAM services allows for the creation of different user accounts assuming roles with fine-grained permission rules which can be grouped into policies. These users can then authenticate themselves to the clouds services via IAM using tokens, credentials or keys. Connections into virtual machines happens usually via SSH tunneling either through provider services or more universal via SSH keys.

Localization

Many of these services have data centers across the world and have regionally different offerings available. This can lead to a trade-off in certain applications of cloud computing between a geographically closer region or one further away but with some specialized resource types. Therefore most resources are not globally easily usable but instead are scoped to a specific region. One example would be images or keys where virtual machine instances can usually only use local ones or regionalized copies. Larger cloud providers offer also multiple availability zones within one region. These are distinct, separate locations within a wider region are that work completely independent from other zones within a region to ensure that a major outage in one zone doesn't affect the all locations within the same region.

Pricing

Just as diverse as cloud offerings are are various pricing models [19] although most providers bill each resource use very detailed. Typical pricing schemes are time-based (e.g. computing instances), volume based (e.g. data storage & transfer), access-based (e.g. secrets), free until a specific limit or a combination of multiple policies.

Price is also a tool to incentivize the use of specific resources. One popular one are spot instances which are temporarily unused cloud resources that can be used for these jobs at discounted prices.

2.3 Multi-Cloud Computing

2.3.1 Definition

Multi-Cloud computing is a very broad term which can range from simply using two clouds providers in an organization to a Federated Cloud. In the first most basic

case this could be just two completely separate workloads running each on manually provisioned different cloud providers [45]. On the other end at federated clouds, providers share their resources themselves among each other in a collaborative way without the user even having to be aware which provider's ones have been used [25].

Motivation

While most cloud providers have similar offerings there are still many arguments why organizations should consider consuming cloud services from multiple providers. First of all some do offer unique or specialized services which are not available by others or not available at specific locations, so by not restricting oneselve to a single provider user increase their number of options[25]. This also affects common services as the bigger choice allows for more price competition, increase availability and deal with peaks in services [25]. Diversifying with multiple clouds helps reducing the dependency on a single supplier of cloud resources who could leverage their position and create a vendor lock-in situation.

Challenges

However, applying a multi cloud solutions to a concrete scenario is significantly more difficult than just using one cloud. First and foremost is the lack of standardization in this field. There have been attempts by the IEEE to establish Inter cloud standards including an service catalog and federation layer but they haven't been successful. The two biggest reasons are the lack of incentive by dominant cloud providers to adopt them when this would only increase competition while gaining few benefits and the sheer complexity of such a standardization effort. The scope needed for a complete standard would not only have to abstract and cater to countless specialized, proprietary infrastructure services but also operate together with higher-level PaaS and lower-lever orchestration services [45].

Another way cloud providers discourage using multiple clouds at once is pricing the outgoing data at a higher rate than incoming data. Users are then tempted to store huge amount of data which they can't inexpensively retrieve and therefore more likely also process at the storing provider leading to a data gravity[40].

2.3.2 Sky Computing

One alternative approach to standardization is sky computing where incompatible cloud resources are uniformly provisioned, aggregated and used through intermediaries as if they were all one big sky rather than separate clouds[45].

Parts

This architecture of the Sky computing concept builds on three main logical components [40]:

The first one is a **compatibility layer** which maps the needed proprietary cloud resources and their APIs into a common interface so that the other components can use them uniformly. There are many hosted service providers that offer multi-cloud services who built their implementations with open source offered compatibility layers as base.

The core builds an **inter-cloud broker layer** on top of the compatibility one. It handles the management of both the infrastructure and the applications concerns from provisioning cloud resource to the placement of jobs to specific cloud instances. This usually includes a catalog that stores all resources both infrastructure and application using a common naming scheme as well as an accounting service to keep track and optimize costs.

And thirdly **data peering** between different cloud providers. Ideally this should come at no extra cost however fees can often be offset by optimized pricing at target clouds. Such an agreement is especially beneficial when an inter-cloud common trusted subnet is created.

SkyPilot

One such sky computing implementation is SkyPilot from UC Berkeley which is primarily designed for batch jobs and machine learning pipelines with a big focus on spot instances [45].

The framework implements an inter cloud broker consisting of a **provisioner** for provisioning and managing the specified infrastructure needs, an **executor** that manages applications and deploys them to the concrete resources and an **optimizer** which calculates the best placements for jobs across clouds according to the application's DAG and requirements provided via yaml files or the API. It gets the cloud providers infrastructure offers from a **service catalog** which stores them along with long-term average prices as well as a **tracker** that observes spot prices. These prices are provided by **service publisher** for each cloud. Another cloud-specific component are **compatibility set** components that map the abstracted APIs to the cloud proprietary APIs.

SkyPilot builds when possible on top of existing solutions like Ray & Terraform for mapping infrastructure management and deploying to it.

Terraform

One of the most widely used foundations for implementing multi cloud setups is is the **Infrastructure-as-a-Code** (IaaC) tool Terraform. [42] It allows automated provisioning of cloud and on-premise resources through human-readable configuration files. Those configuration files define which resources there should be, they get then transformed together with the existing infrastructure into a directed acyclic graph of execution steps needed to attain the desired infrastructure, which can then be applied.

Each cloud provider has its own Terraform provider as intermediaries between Terraform and Cloud APIs they are available in a terraform registry and can be installed as need be. As a result the configurations itself are still very much tied to actual cloud resources and barely abstracted and Terraforms main focus is to automize these custom setups.

2.4 Orchestration

As a result of the emergence of microservice architecture consisting of many small services there has been an increased need to compose and orchestrate a large amount of them automatized. Each individual service has to be setup, specified and configured to not just run on it's own but to correctly run in cooperation with other services to a form a common bigger functionality. The most important functionalities for an orchestration framework are the facilitate communication among the services and their environment, syntactic as well as semantic alignment of services as configurations of them, and persistency of data from non-persistent service units to data storages [5]. Additionally they also allow monitoring of these services and offer life-cycle functionalities like restart policies.

2.4.1 Docker Compose

One of the most common ones for configuring and orchestrating services on a single computing instance is Docker Compose as it's the officially supported tool by Docker themselves and even has been integrated as a plug in to Docker itself. [7] It offers the functionality to start and stop services, build and pull docker images, create and attach data volumes to containers and also set up networking like bridges and bound ports from containers to the host machine.

This is configured via additive YAML configuration files that are fed as an input to the compose orchestrator command line interface which can then either orchestrate all services or selective ones.

2.4.2 Oakestra

Oakestra [22] is a lighweight edge-computing orchestration framework that tailors exactly to the specific needs of an edge environment. It acknowledges in its design the limitations on the edge like unreliable availability of nodes, nodes with small processing power and general high heterogeneity between nodes ranging from the smallest IoT devices to edge data centers. The principal philosophy of the framework is the pooling together of computing resources of various types into a shared federated infrastructure where users can provide their own resources to others.

Concept

Oakestra therefore proposes in their white paper the following concepts:

Three-tiered resource management Oakestra is hierarchically structured into three separate tiers consisting of worker nodes, which are organized into clusters, which are then again managed by a single root instance. This design allows fine-grained control of each cluster over their own worker nodes and keeps the cluster independent when the connection to the root is unreliable. A local cluster in proximity to their nodes also decreases the latency of intra-cluster management communication.

These separate clusters also remove overhead from the root component by only sharing aggregated capabilities and status of their workers which in an edge environment can easily go into the thousands or millions of nodes.

Delegated Scheduling Another key concept is that both root and cluster level have their own schedulers. On root it simply searches for the best cluster using its aggregated resource information and then delegates the concrete scheduling in a next step to the cluster. The cluster scheduler then finally schedules the to a concrete worker node. Services are scheduled to a fitting node which can fulfill the minimum requirements at the edge set by their job SLA's for processing power performance, networking requirements, location restriction and more. Oakestra allows the use of custom schedulers so that they can be individually tuned for different applications like a basic resource-only match, more complex latency & distance aware placement or other schedulers. This separation also lets the cluster completely handle node failures on its own as long as it has enough resources without needing management support from the root. Redeployments can happen without data transfers as long as the are services are stateless.

Semantic Network Overlay Additionally Oakestra also supports intra-service communication using an optional **NetManager** component which keeps address tables

to services on the same or other instances which can be accessed via proxying and tunneling.

Components

The implementation is organized into the same three logical tiers.

Root Orchestrator On the root level the **system manager** is the main component which controls the global infrastructure. It is the single entry-point for all incoming user requests like adding application SLA's to the root or deploying services. Therefore it also serves the frontend components with all necessary data. This data is stored in the root database which is accessed by both the system manager and **root scheduler**. Additionally there is a separate **root service manager** which coordinates the networking part in regards to subnets.

Cluster Orchestrator On cluster level the general structure is logically mirrored from the root tier except that it only manages all worker nodes within the cluster with the **cluster manager**. It uses the **cluster scheduler** to determine on which node of this cluster a job instance gets deployed to. It also stores them in it's own **cluster database** and the networking is once again coordinated by a **cluster service manager**.

Node Finally on the last tier there are two components. A **node engine** which executes the services on the runtime on deployments. It monitors these jobs and reports their status back to the cluster manager along with the nodes performance metrics like CPU, GPU and memory usage. The networking overlay part on the edge is handled by the **net manager** which doesn't allows Oakestra nodes to not require IP addresses.

3 System Design

This chapter offers an implementation-agnostic approach for a multi-cloud infrastructure broker. While its design was inspired to work with the Oakestra framework. the principles, general design, and operations are kept generic so that they can be applied to other three-tier orchestration frameworks.

The first section 3.1 specifies the requirements for this solution by first defining and motivating the four overall goals as system objectives, then breaking them down into concrete functional requirements, and finally stating the most important non-functional quality requirements that the proposed design should strive for.

These requirements are then used in section 3.2 to identify the key logical components, examine their design choices, and arranged them into an overall system architecture. Lastly, in section 3.3 the system's behavior is defined through a set of operations in which the introduced components realize the requirements through interactions with each other.

3.1 Requirements

3.1.1 System Objectives

While there exists a lot of models and methodologies to find and structure overall goals of a system many of them like GORE focus primarily on aiming for completeness by integrating as many stakeholders through a complex, structured process. These processes seem to be a bit too overarching for this narrow, technical system which is why this section only presents the systems major goals and focuses on their motivation.

Cloud Infrastructure Management

The core objective of the system is bringing cloud computing capabilities to an edge computing environment via an orchestration framework in accessible way. It extends the functionality of only application management by offering the possibility to provision, manage, and connect these cloud resources to existing edge infrastructure.

As a result an edge-computing framework can gain some of the benefits like flexibility of cloud computing without impacting the rest of the existing system. One potential

application scenario is the use of cloud instances to fulfill a need for more resources than the pure edge infrastructure can provide. Such cases can be both short time due to usage peaks or availability issues with unstable edge networks and more medium term like the need for resources at different locations and prototyping before committing to the establishment of own infrastructure. The proposed interface should automize the repetitive, manual steps of provisioning and configuring this cloud infrastructure to integrate it with an edge computing framework.

Multi-Cloud Sky Computing Broker

The second system objective is broadening the cloud capabilities by not just interfacing a singular cloud provider to the edge but instead offering resources from many different providers in a sky computing broker design which allows to seamlessly have clusters on multiple clouds.

This approach prevents the common issue of vendor lock-in as the proprietary APIs and interfaces which usually make moving applications between clouds hard and expensive can be easily instantiated and deployed via the orchestration framework. As each provider offers different types of resources like specialty hardware in different locations having the option to use any of them results in a more flexible and diverse offerings to choose from. This increased supply and comparability between options incentivizes market competition by making it easier for users to choose the most cost effective offers for each cluster need.

Simplified Scaling

Another path to use cloud resources more economically is providing a simple solution to scaling resources down if they are underutilized and scaling them back up in a short time when there's more demand.

While an automated scaling based on resource utilization monitoring and policies on when and how to scale is out of the scope of this thesis, the proposed system should be designed with this option in mind so that in the future these decision could be automated. This scaling option can not just save costs by keeping billed resources to a minimum, it also allows to better cover higher peaks with more bullish upscaling policies when scaling operations need less effort.

Automatic Security Configuration

The final system objective is addressing the security configuration needs arising from cloud providers operating over the public internet when interacting with edge-computing networks outside the cloud.

As the cloud resources need to be available from the open internet it should be prevented through appropriate security measures that unauthorized entities can interact with the systems components especially since the system can perform operations that incur unapproved costs at the cloud providers. Besides the system components also the applications running on the system need to be secured. Since each application can have different public interfaces these application-specific settings have to be taken into account when applying the principle of least privilege to the components security configuration. And as deployments happen automated also this configuration setting has to happen automated.

3.1.2 Functional Requirements

The functional requirements which define the systems main features are structured into eight separate use cases which can also be seen in Figure 4.1.

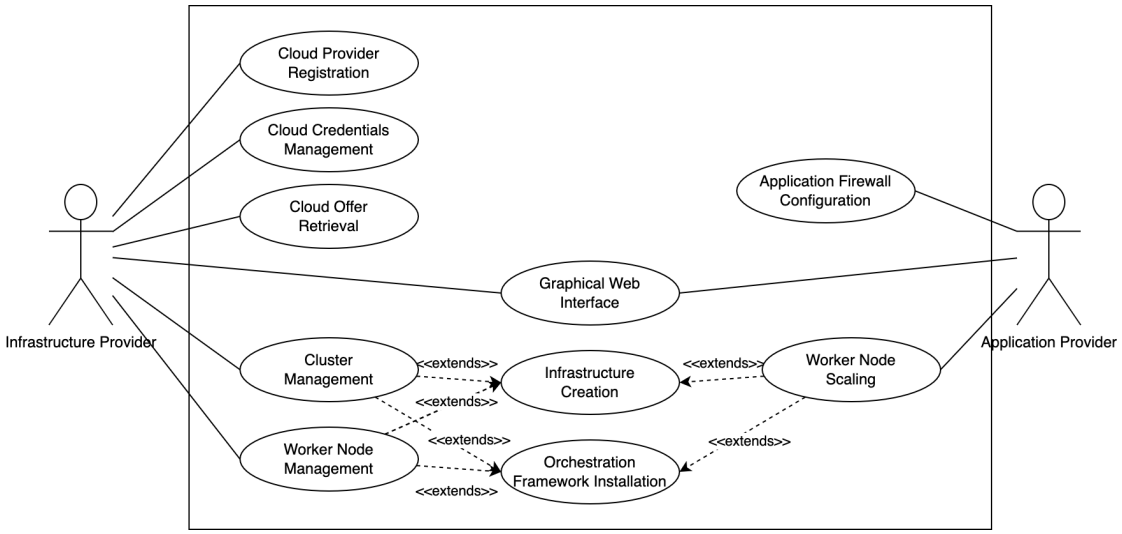

Figure 3.1: Use Case Diagram

These features are initiated by two types of actors. An Infrastructure Provider user whose role is to provide and manage the infrastructure that is available to the orchestration system and an Application Provider whose role is to run their registered applications on the system by deploying it to the available resources.

Cloud Provider Registration

Infrastructure Provider users can attach and detach cloud provider adapters to and from the system to either increase the support for more cloud providers or remove the support for unused ones. As a multi cloud system, adapters for many cloud provider can be attached at the same time. A list of currently available cloud provider adapters can be retrieved.

Cloud Credentials Management

Infrastructure Provider users can store their cloud credentials in the system so that the system can use them to create and manage the cloud resources. However the system may only use them for operations that their cost has been explicitly approved by the infrastructure providers. A list of available credentials can be retrieved to choose from but only the system can have access to the plain text credential data after they've been added, never the end user themselves. When a credential is no longer in use by any currently billed resource it can be removed from the system.

Cloud Offer Retrieval

Infrastructure Provider users can get a price estimate when creating new cloud resources in order to consent explicitly to the incurred costs. To receive an accurate offer they can select generalized provider-agnostic attributes for new cloud server instances like location, instance type and image type from a list of available ones for each option.

Cluster Management

Infrastructure Provider users can create new cloud clusters and attach them to the current system. Therefore this multi cloud broker should provision a cloud server instance as specified by the offer using chosen credentials, install and set up the cluster part of the system on this instance, and then connect the new blank cluster to the existing system. A cluster can then be deleted again if it has no actively billed nodes.

Worker Node Management

Infrastructure Provider uses can create multiple new worker node instances for an existing cloud cluster. Like the cluster creation the specific instance configuration can be selected from an offer. Using the credentials of the cluster new cloud instances are provisioned, the worker node orchestration software installed, and then those new nodes are connected to their cluster. Also worker nodes can be deleted if no applications are currently running on them.

Worker Node Scaling

Application Providers can scale a cluster by turning off or on individual worker nodes of the cluster. When nodes which are currently running application are being turned off they need to redeploy first those running service instances. Besides just stopping and restarting cloud instance, a user can also completely terminate and recreate worker nodes from scratch to further reduce costs. This recreation requires the same infrastructure creation and worker node software installation functionality as the Worker Node Management requirement instead of just reconnecting. Turning worker nodes back on or recreating them doesn't need the approval of an infrastructure provider user as the costs for running of all created nodes have been consented to upon initial creation. Scaling those nodes up and down only saves costs compared to them running constantly.

Application Firewall Configuration

Application Providers can trigger the reconfiguration of the worker node's firewall by deploying or undeploying applications to this worker. The system should automatically adjust the firewall upon deployment to allow traffic to the exposed applications port and upon undeployment the firewall should block traffic once again to the previously exposed application port.

Graphical Web Interface

Both Infrastructure Provider and Application Provider users should have a graphical web interface where they can initiate all of their operations related to the multi-cloud broker. Additionally they should be able to see the status of their managed cloud infrastructure so that they can directly control it.

3.1.3 Non-Functional Requirements

Non-functional requirements are usually either defined as requirements describing properties, characteristics, constraints of a system, or as quality attributes that the product must have [18]. This section defines some of the important qualities that the proposed system should strive for in its design when making decisions. These qualities are than evaluated in section 5 to validate that they have been observed.

Extensibility

As there are so many cloud providers on the market that not everyone can be covered, one of the most important qualities of the system should be its extensibility to allow the easy development and integration of new cloud provider adapters. It should be possible to integrate such a new adapter without the need for changes to the rest of the systems code.

Modularity

Another result of the quantity of cloud provider most users only need the support for a few providers. Therefore it should be possible to only use and run the adapters for the used cloud providers. Adding or removing them should work without disrupting the running system.

Portability

While the number of cloud providers is already large, each of them has additionally many configuration options like instance types and images leading to an exponential amount of possible configuration combinations. It is clear that not every single one can be supported but in order to support as many out of the box the cluster and worker software should be able to run on at least basic Ubuntu or Debian distribution images with few preinstalled software as many build on top of those.

Performance (Speed)

Since a major benefit of the proposed system is the ability to provision cloud resource within a short time, the speed of the systems operations especially the key contributor to the systems quality especially when compared to similar broker orchestrators. Therefore the design should take steps to keep the duration of these operations low.

Responsiveness

Even if the provisioning operations would be implemented as high-speed as possible they would still take minutes to complete. Therefore this system should respond before that by having for a response time of never more than two seconds. This is especially relevant for the frontend to make the system not appear being stuck. Instead the current state of each operation should be communicated to the user.

Low Resource Overhead

As an edge-computing orchestration framework which is designed to have be lightweight because its just a helper for the actual running applications, this introduced multi-cloud functionality should only add minimum overhead in terms of CPU usage, memory usage and network activity during normal operations.

Operational Cost Effectiveness

Since operating on cloud instances incurs already significant costs. The system of creating and managing these resources should be not be wasteful and only consume as many cloud resources as it really needs for operating.

Security

By adding cloud computing to an edge computing network it becomes necessary to communicate between those components via the public internet. Therefore security measures have be taken as the system is not operating within a secure environment. While security considerations should be taken along the whole design process in particular these following security aspects need to be realized. First of all every interaction which is not on the same machine has to be authenticated so that only valid components of the system can interact with each other and the identity of users is ensured. This identity has then also be checked whether a user is authorized to perform certain operations especially with regards to operations which are billed by cloud operators. And thirdly, as the system handles a lot of secret credentials they have to be kept confidential across all components that interact with them.

Usability

Finally, the graphical web frontend which is the main way users interact with the proposed multi cloud broker has to be designed in such a way that the user can easily access with a few click all operations they could initiate.

3.2 Architecture

The proposed system is to be built on top of an existing three-tiered, hierarchical orchestration framework, consisting of a root orchestrator, cluster orchestrator and worker node component. Every feature that the multi-cloud system explicitly interacts with is a separate logical component from the existing components even if part of the functionality might already be part of the existing system. This separates the

concerns and highlights using this approach better where the edge orchestration and cloud infrastructure provisioning use common resources and functionalities. Besides extending these tier components, a new separate cloud adapter component is introduces which is responsible for translating the provider-agnostic operations to the provider-specific API. This section gives an overview over all the logical subcomponents which can also be found in Figure 3.2. The pre-existing abstract components of a generic three-tier orchestration framework are highlighted in the Figure 3.2.

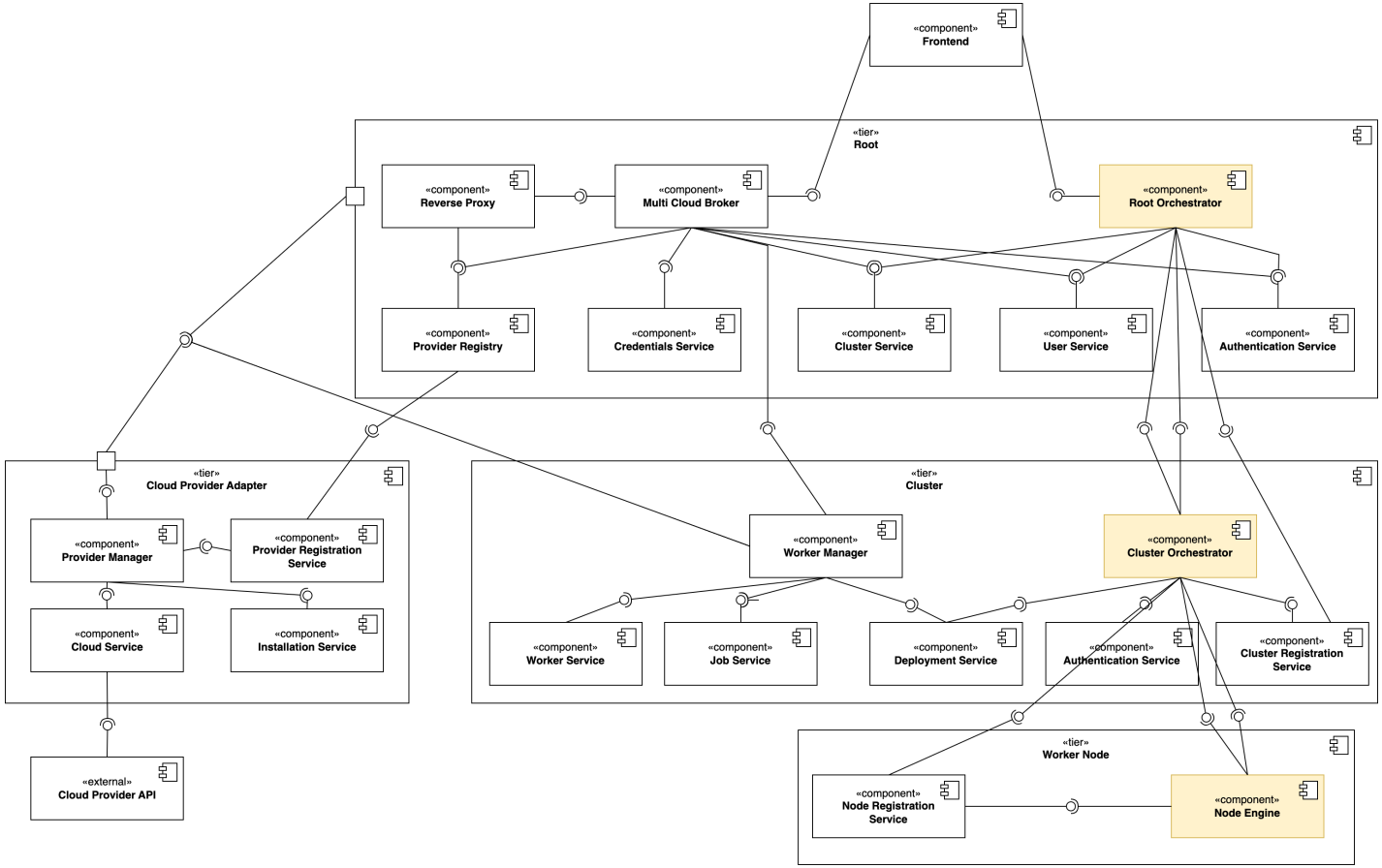

Figure 3.2: System Architecture

3.2.1 Root Components

Root Orchestrator

The root orchestrator abstracts the root component of an existing orchestration system. It contains all the root functionality that is needed to manage, schedule, and deploy micro-services and applications to specific clusters. The main functionalities that it shares with the multi cloud components are extracted into logical services. These services are used to interact with users, their authentication and the management of (edge-computing) clusters.

Multi Cloud Broker

The multi cloud broker component is the singular entry point for all newly introduced cloud infrastructure related operations on the root tier. Its main purpose is to validate incoming external requests and then direct the calls to other components, both inside and outside the root tier, acting as a driver managing the control flow of an operation. This approach was chosen over a design with multiple entry-points where external actors could directly address the direct components via an explicit api gateway and save one hop. However this would result in either the external actor having to do drive the many sub-calls or the driving functionality being located at the first call component which not necessarily. With a graphical web interface intended as the main interaction mode this complexity would fall onto the frontend which is abstracted by having a single target for all cloud-related requests. An additional benefit is that this single entry point is the only component which is required to check requests for authentication using a middleware. Therefore such a middleware can be omitted at the other components that are not exposed to external actors.

Authentication Service

As the multi cloud broker component introduces a second externally exposed component besides the existing root orchestrator both need to be able to handle authentication with a common secret to avoid duplicate authentication procedures. The best practice therefore is to have a separate authentication component which hold this common secret and can create tokens and validate them. Another benefit is that this component can be reused as a second instance on the cluster tier for cluster level authentication.

Cluster Service

The first components which interacts with a data repository is the cluster service. It's purpose is to persist the cluster states in the root component. It can register new

clusters, delete existing ones and retrieve a list of all clusters with their states as this information is needed to manage and interact with them. The persisted cluster objects should at least support the following attributes for this multi cloud functionality as depicted in Table 3.1

Field	Type	Description
Id	Identifier	Unique primary key for the cluster
UserId	Identifier	Identifier of the user who registered the cluster
ClusterName	Text	Unique name of the cluster
IP	Text	IP address of the cluster
Port	Text	Port of the cluster orchestrator
ProviderType	Text	Cloud Provider Name if created via MCB
InstanceId	Text	Cluster Instance Identifier on Cloud Provider
RegionId	Identifier	Identifier of the associated Region Info
PairingComplete	Boolean	Encrypted password for the user's account

Table 3.1: Cluster Model

Credentials Service

The credential service is for persisting and accessing the cloud credentials and cloud account dependent data. They are only passed temporarily to the provider adapters for creating, managing, and deleting cloud resources. Since each cloud provider has a different credential format, the repository has store them in a generic format like a binary-encoded file, which can then be decoded again by the respective provider adapter using their provider-specific implementation. This leads to the following schema with at least these attributes for the credentials as depicted in Table 3.2

Field	Type	Description
Id	Identifier	Unique primary key for the credential
UserId	Identifier	Identifier of the user who added the credential
Name	Text	Descriptive display name for the credential
ProviderType	Text	Name of Cloud Provider for this Credential
Credentials	Binary	Actual Credentials binary encoded
LastUsedRegionId	Identifier	Identifier of the region last used with this cred

Table 3.2: Credential Model

As these credentials can be used to incur significant costs for using resources they have to be kept secure and their use has to be restricted strictly with consent. This design assumes that all infrastructure provider of a root instance are allowed to use all the registered credentials, as this is exactly what the role is meant for. However when using one root instance in a larger organization then it can make sense to adapt the credential authorization from a simple role check to either a user check, so that only the user who added the credentials can use them, or that infrastructure providers can only used credentials provided by the same organization. In terms of storage security there are many different hardware and software based solutions like encryption. In pursuit of this design being portable it doesn't specify concrete storage security strategies. Instead it just assumes a trusted root environment for the database which is risky and has to be addressed by the operator according to their specific setup requirements. Besides the credentials themselves the design also caches two non-instance resources which can be reused across multiple clusters and thereby save time being spent on repeated setup before the actual computing instance creation and a reduction of cost on billed resources. The first reusable resources are firewall groups as the exposed ports by the system components are the same across all across the same component types. The other one are SSH keys for connecting to the cloud instances when they are created by the same credentials. Generating SSH key pairs is computationally expensive and should be stored securely by the clouds secrets management systems as they allow unlimited access to the worker nodes which can hold potentially sensitive application data as well as sensitive cluster and worker information. This storage of secrets is relatively expensive and since these instances are available anyways with the credentials it makes sense to reuse those keys. These kind of non-instance resources are usually not global but are only available within a cloud providers region. Therefore they are cached and reused on a credential and region pair base. This leads to the following schema for a repository of region information.

Field	**Type**	**Description**
Id	Identifier	Unique primary key for the region defaults
Region	Text	Provider Name for the region
ClusterFirewallId	Text	Provider Id for Default Firewall Rules of a Cluster
WorkerFirewallId	Text	Provider Id for Default Firewall Rules of a Worker
DefaultKeyName	Text	Provider Name for Stored Reusable Instance Access Key
ProviderType	Text	Cloud Provider Name if created via MCB
CredentialId	Identifier	Identifier of the associated Credential

Table 3.3: Region Info Model

User Service

The main purpose of the User Service from the Multi Cloud Broker perspective is the authorization checking. It is assumed that most of the other user service functionalities like registration and login are provided by the main orchestration system. A secondary purpose is to store with the region information containing cloud provider, region, key name and firewall groups used on the last created cluster by the user. This allows the pre-filling of the cluster creation fields in the web frontend.

Field	Type	Description
Id	Identifier	Unique primary key for the user
LastUsedRegionId	Identifier	Identifier of the associated Credential

Table 3.4: User Model

Provider Registry

The provider registry is a service registry in the spirit of the micro-service architecture specifically for registering dynamically multiple cloud provider adapters both the same or different providers. However to make sure that not just anyone can register as a cloud provider adapter and get access to the cloud credentials transmitted each prospective provider adapter has to register with a registration token through the multi cloud broker. The multi-cloud broker can retrieve the list of available providers and get the address of an available adapter for a specific provider type.

Reverse Proxy

Besides fetching a provider adapter of a specific type via an explicit call to the provider registry, calls can also be redirected to the correct by integrating the registry with a reverse proxy.

Web Frontend

The web frontend provides a graphical user interface for users to manage the system at the root tier. It connects to both the root orchestrator for most regular orchestration needs and to the multi cloud broker for the cloud infrastructure management as this is the singular entry-point for these operations.

Two web pages can be categorized in two main frontend subcomponents for credentials and cluster management.

The credentials feature has a list of existing credentials which also have a delete option if a deletion is possible and an add form where infrastructure providers can select a connected cloud provider and insert the credentials file. Registration tokens for additional providers can be generated via a registration button.

The clusters management page has a list of all clusters both manually provisioned clusters and managed cloud clusters, although only the latter ones can be deleted. A new cloud cluster creation request can be initiated by completing a cluster creation form where infrastructure providers have to select from the available credentials, regions, instance type and image type options. Display For cloud managed clusters there exists a worker list on the cluster detail page. Each managed worker instance can be here stopped, started, terminated, recreated or deleted. Similarly to the cluster there is also a form to create new worker instances however since the region and credentials are already defined by the cluster the user only needs to choose the instance type and image types for the new workers.

3.2.2 Cluster Components

The cluster level tier is similarly structured to the root tier except that each cluster is directly associated with only one provider adapter because of the constraint that managed cloud clusters consist of only workers in the same region within one cloud provider. Therefore it doesn't explicitly require a service registry as part of its system design however an implementation may include it. The authentication service component is directly duplicated from the root tier and is once again used for token issuance.

Cluster Orchestrator

The cluster orchestrator is the counterpart of the root orchestrator on the cluster tier. It abstracts all the functionality of an existing edge orchestration framework on a cluster level which is not shared with the following multi cloud support components. It's purpose is to manage, schedule, and deploy micro-services to specific worker nodes, as well as checking and aggregating the worker nodes state into a complete cluster wide state.

Worker Manager

The worker manager is the main cloud infrastructure component on the cluster tier. Similar to the multi cloud broker component on root it is the singular entry-point for all cloud management requests and drives the control flow by calling other components. It is connected to a single cloud adapter due to the constraint that all workers have to

be in the same region of the same cloud provider. When interacting with this cloud adapter also it has to provide cloud credentials to it. Since infrastructure requests like creating or stopping worker nodes originate from the root there's no need to copy them permanently to the cluster tier and risk increased exposure. Instead they can be passed with each infrastructure call from the root to the cluster tier through the trusted and encrypted connection. The firewall configuration however is triggered by either an explicit deployment request from a root application provider or internally via a redeployment and both options don't have access can to the credentials on root but still require some credentials to use the cloud provider adapter. Therefore each cluster gets own credentials which are solely for creating and assigning firewall configurations to worker nodes of its cluster.

Worker Service

The worker service is analogue to the cluster service on the root tier. It handles the management of the workers in particular their creation, stopping, starting, termination, recreation and deletion. These states are persisted in a data repository this service interfaces. They worker node objects should at least have the following attributes:

Field	Type	Description
Id	Identifier	Unique primary key for the worker node
InstanceId	Text	Worker Instance Identifier on Cloud Provider
IP	Text	IP address of the worker node
NodeState	Text	Enumeration storing the nodes state
InstanceType	Text	Instance Type Identifier of Node on Cloud Provider
ImageId	Text	Image Identifier of Node on Cloud Provider
Storage	Number	Selected Storage Capacity for Node

Table 3.5: Node Model

Each cloud node in the database here primarily represents the consent to the costs that this node can incur by the user who initially created. Therefore these stored node entries are being kept even if they get temporarily shutdown or permanently terminated to lower costs. In order to recreate terminated nodes with the same type this system also stores the configuration attributes for each node. Because each logical node entry in the database can be realized by multiple actual cloud instances over its lifetime, the system needs to keep track of each node's state itself. This state attribute acts as part of a state machine which only allows the recreation of nodes when they are currently shutdown to prevent multiple cloud instances for one node and resulting

in conflicts that could lead to the inability to manage lost instances. The states and transitions of between them are visualized by Figure 3.3. Where yellow states represent the ones which incur costs and red the ones that don't.

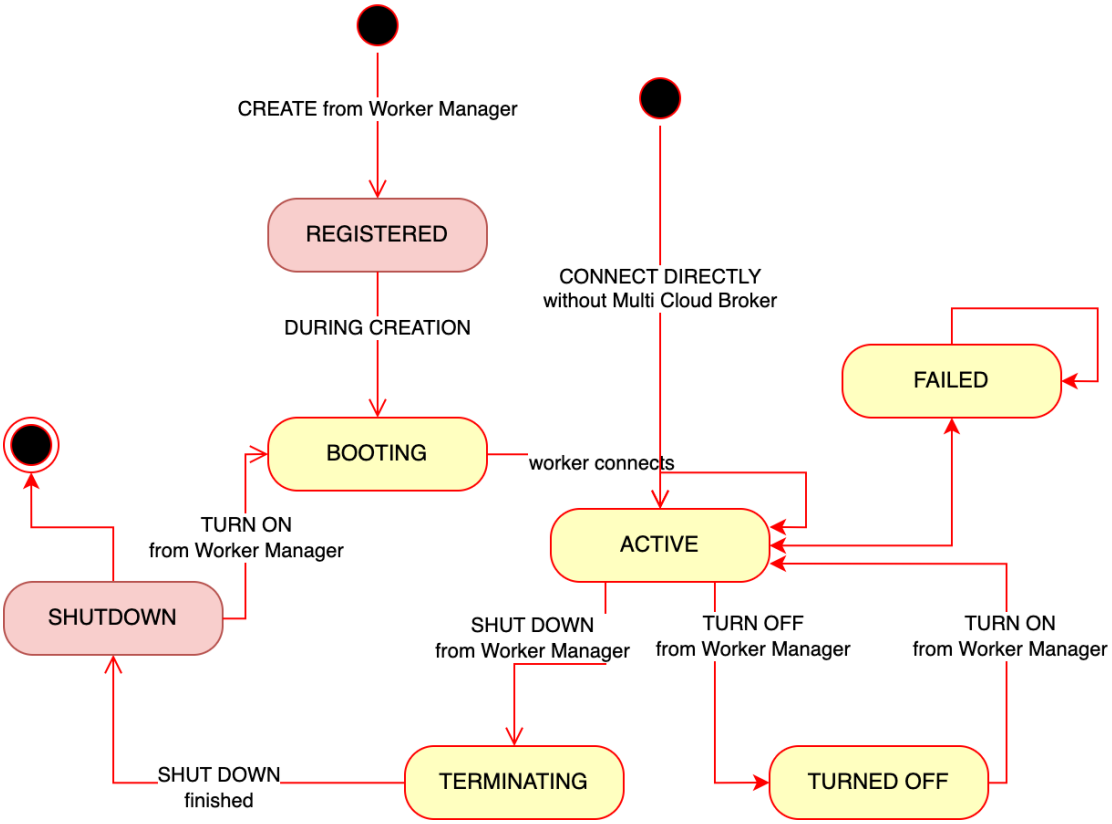

Figure 3.3: Worker Node State Diagram

Job Service

The job service handles the persistence of the states of deployed services as jobs. While it's mostly used by the cluster orchestrator for scheduling and state reporting, the worker manager still needs access to service information for correctly managing the firewall configurations of the worker nodes. Additionally before shutting down or turning off worker nodes, the worker manager needs to track the state of their job redeployments. Therefore the job objects in the repository must include at least these attributes:

Field	Type	Description
Id	Identifier	Unique primary key of the job
Port	Text	Port number used to receive traffic by the job
FirewallId	Text	Firewall Identifier on Cloud Provider for this job
Status	Enumeration	Running state of a job

Table 3.6: Job Model

Cluster Registration Service

This is a service which always runs upon startup. It connects the cluster tier to the root tier at its provided location and establishes the connection between the root and cluster orchestrator. To authenticate with the root it uses a token issued by the root, so that only the real registered cluster can connect to it.

Deployment Service

The deployment service handles the deployment on a cluster level to specific worker nodes. It is assumed that this is part of the core functionality of the existing orchestration framework. However this specific functionality is here modeled as a separate service since it has an explicit interaction with the worker manager. It has to notify the worker manager about deployments in progress so that it can adjust the firewall settings to either allow traffic to a newly exposed port of a deployed service or restrict traffic to a port no longer exposed as the service has been undeployed. Because the worker manager in particular and the multi cloud broker functionality in general should be an optional addition to an existing edge orchestration framework this interference has to be kept to a minimum and therefore this call about a deployment is a fire and forget request, that doesn't expect a response and is an optional trigger only. Also it can deploy and undeploy jobs which is needed before turning off or shutting down worker nodes.

3.2.3 Worker Node

The last pre-existing tier is the worker node. As it doesn't manage any other components it only has components that expose functionality to other cloud components or that are pre-existing components.

Node Engine

The main component of a worker node is abstracted into a node engine which receives deployment requests, runs the jobs locally and reports back their state.

Node Registration Service

The main registration service is the only service of this tier that is explicitly triggered by a multi cloud component. This service starts on startup after the installation and attaches a new node with its cluster using a token similar to how the cluster registers at the root through the Cluster Registration Service.

3.2.4 Cloud Adapter

The cloud adapter is a completely newly introduced parent-component logically entirely separate from the existing three tier system meaning that while it usually runs beside the root, it can also run on separate infrastructure or even as a microservice on a pre-existing node. The cloud adapter purpose is to provide a common interface for the required operations of a multi cloud broker, breaks them down into and coordinates them as smaller suboperations which are then forwarded to a concrete provider-specific suboperation implementation. For security reasons it doesn't store credentials which could come from various instead it just uses and deletes.

Provider Manager

Similar to the other multi-cloud components also the adapter has a single entry-point main component, called Provider Manager, which directs the provider operation by coordinating the sub operations to other logical and provider-specific components.

Provider Registration Service

Similar to clusters also provider have to register with the root component so that only trusted ones are called by the root. However these registrations are only unidirectional since the provider adapters don't initiate requests themselves to other tiers of the orchestration system and therefore don't need to keep track of which systems they are connected to. This makes it also easier to reuse the same adapter instance for multiple root system instances by reducing this state keeping.

Cloud Service

The cloud service contains all sub operations that interact with specific cloud providers API and SDKs to create, retrieve, modify or delete cloud resources. It implements these sub operations using a common provider-agnostic call signature and maps these calls to concrete provider-specific implementation using the concrete resources. In order to maximize reuse and minimize the share of the provider-specific implementation this is the only logical component which is different for each cloud provider. Since these API calls to the cloud providers have to be authenticated, this component requires the cloud credentials as an input and then casts them from their generic credential format to the cloud-specific one.

Installation Service

The installation service handles the second, cloud-provider-independent part of provisioning which is installing and setting up the software components of the system on new (virtualized) hardware. This design chooses to install all necessary software from scratch assuming a nearly blank cloud instance. Compared to the alternative of using images with the software pre-installed storage it offers more portability because it doesn't require to provide these custom images for each operating system on each cloud provider in each region. These images would not only cost effort to provide for the amount of combinations but each instance also incurs hosting fees at the cloud provider with no clear responsibility for these costs. The disadvantage of installing everything from scratch, however, is the time and computing effort spent on installing the same software over and over again. This can still be avoided with this design by specifying such an image with the pre-installed software. The installation service checks before installing each software component if it is already present and if doesn't reinstall it. After a successful installation on the new target resources this service launches the components with the required configuration so that they connect to the rest of the system.

3.3 Operations

Built on top of these logical components there are these distinct operation that realize the seven functional requirements. In this section the most important ones are presented in sequence diagrams with a description on how the logical components interact with each other. These operations are grouped into six phases of the program use to highlight similarities.

3.3.1 Infrastructure Creation

The most complex part operations are the actual creation of cluster and workers.

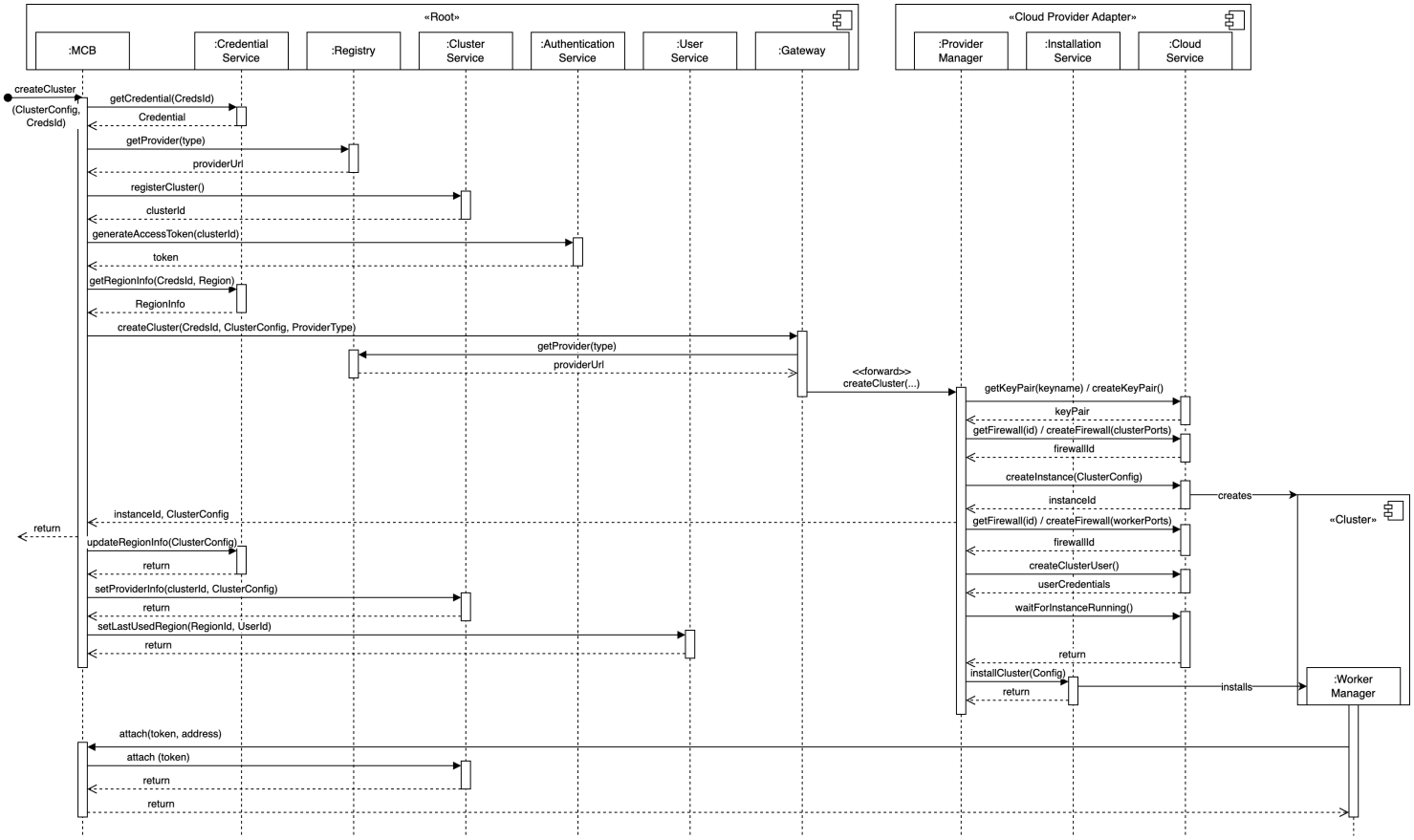

Figure 3.4: Cluster Creation Workflow

Cluster Creation

The creation of a cluster see Figure 3.4 starts in the Multi-Cloud Broker where first the initiating user is authenticated, the chosen credentials are retrieved if the user has access to them and checked in the registry whether an adapter for the chosen provider is available. Only then is a new cluster entry in the repository created and a registration token for this new cluster generated. Just before the provider adapter gets requested to create the actual infrastructure using the selected cluster configuration like instance type, the broker gets if available the cached ids for the cluster firewall and key that have been used previously with the same account in the same region. Depending on whether these cached reusable resources exist on the provider, the adapter then either creates those resources from scratch or uses the existing ones as the parameters to create the actual virtual machine instance for the cluster along with the passed cluster configuration. As soon as the actual provider API responds with the identifier of the newly created instance this request completes and returns the identifier to the initiating broker. The multi-cloud broker first completes the request as now the most likely failure scenarios regarding instance creation itself succeeded and finally updates the cached region defaults, stores the provider specific information to the cluster entity and sets the region as the users's last used one. While the adapter waits for the provisioning of the virtual machine to complete, it already creates checks or creates another firewall for the future workers of the new cluster. Additionally it creates a separate cloud provider user account which is solely used for creating job-specific firewalls and attaching them to worker nodes. Therefore these credentials can safely be stored on the cluster so that it retains the autonomy of (re-)deploying jobs within the cluster without interaction with the root. Once the created virtual machine is ready, the installation service connects to the virtual machine using the previously created key and installs all the cluster components with their necessary software and starts them up using parameters from the multi-cloud broker. Upon start-up of the cluster the cluster registration service attaches itself with its address to the root using the registration token, which completes the creation of a cluster.

Worker Node Creation

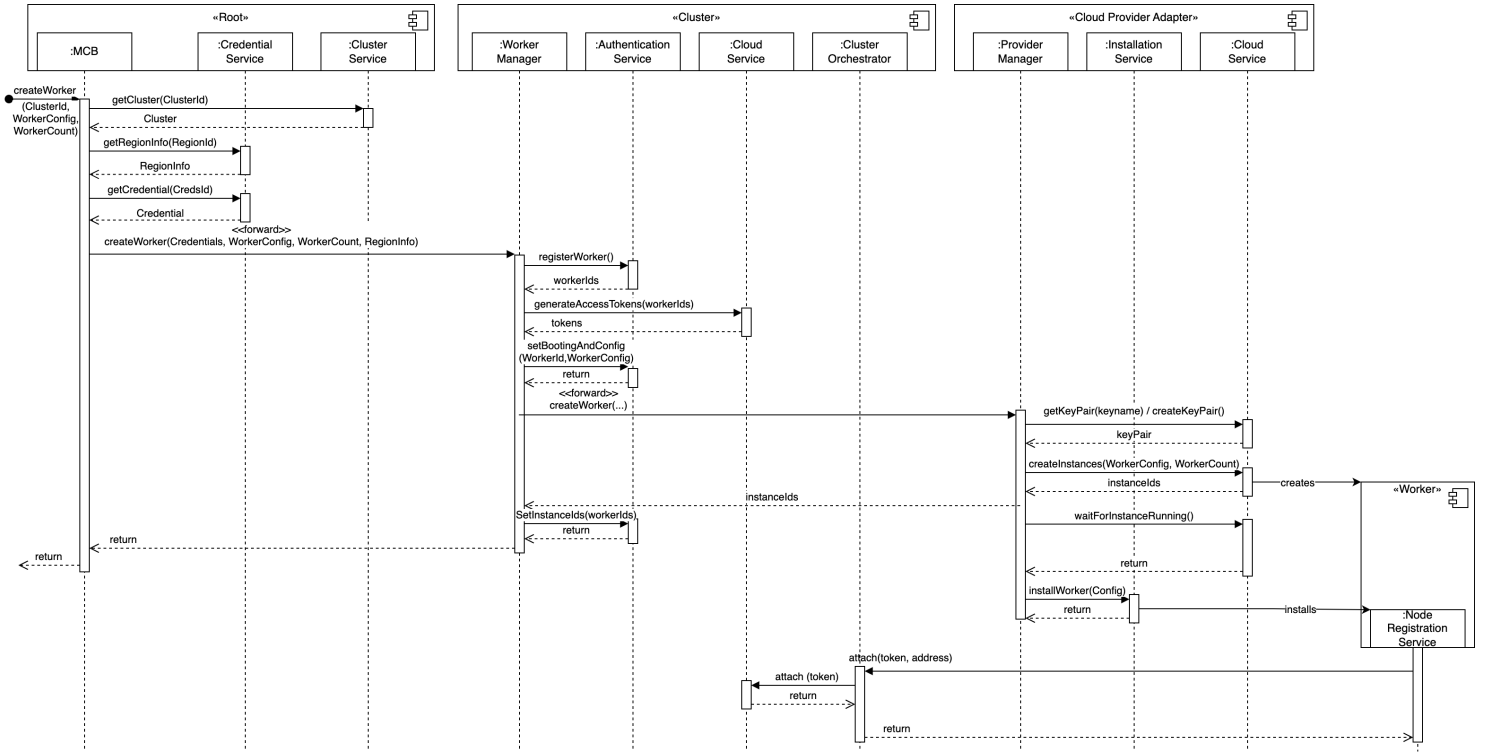

Figure 3.5: Worker Creation Workflow

The worker node creation starts as shown in Figure 3.5 similarly in the multi cloud broker as the user won't interact directly with the cluster's worker manager. As the cluster already exists containing the region info which in turn links to the credentials they can simply be retrieved from the credential service using the information without needing to select the credentials explicitly. Thereby the same credentials are used for both the cluster and all its provisioned worker instances reducing the risk of inaccessibility when credentials are retracted. After all this necessary cached information has been gathered on the MCB it is then forwarded in a worker creation request to the worker manager on the cluster. There the worker manager mirrors the behavior of the cluster manager on root by registering new worker nodes using the Worker Service and creating registration tokens with the Authentication Service. Each newly created worker database entry has then added the worker configuration settings and its state set to booting so that it's in the correct state for later attaching. Then the create worker request gets forwarded a second time now to the Provider Manager. It fetches like on root level the key pair although it can't create a new one. It also doesn't have to prepare firewalls and user accounts as they have all been already created upon cluster creation. If the creation from the cloud service succeeds then the provider sends a response back to the worker manager with a list of the just created instanceIds. These are then added to the node database entries. In the meantime provider waits for the instance to become up and running. When the instances are finally ready they get the worker code installed and then upon start of the node the node registration service attaches via the cloud orchestrator using the cloud service.

3.3.2 Deployment

While deployments are separate from resource management they still require some adjustments for security reasons to adapt the firewalls of a worker node. So that only the necessary ports are exposed to public internet. These adaptions are triggered unlike all other operations not explicitly via the multi cloud broker but instead by the deployment service itself. In order to keep the multi cloud components an optional part these triggers are sent without waiting for responses and the changes inside the deployment service are kept to a minimum.

Assigning Firewall

Upon deploying the deployment service sends the assigning trigger to the worker manager with the chosen job and node ids. The worker manager receives this request and then first retrieves the needed ports for the job from the job service. Using the cluster user credentials, that have been set during cluster creation, the worker manager

requests the provider adapter to create a new firewall with those ports if the job doesn't already have an existing firewall. The provider adapter calls the create firewall operation from the cloud service and returns the firewall id. This firewall id is then saved for the job so that it can be reused on future deployments. After the optional creation the firewall then needs to be assigned to the node, where the job is deployed to. Therefore the nodes cloud instance id is retrieved and then sent to the provider who executes the assign firewall to instance operation with the cloud service. Once this has succeeded then the firewall configuration is adapted for the node to the new job.

Unassigning Firewall

During the deletion of a job as part of undeploying it is checked for which nodes that previously had the job deployed there is no longer a single instance of this job. For these nodes is then an unassigning request to the worker manager triggered containing the node and job id again. It simply retrieves the node from the worker service for the node's cloud instance id and the job for the job's firewall id. Using these two ids and the cluster user credentials again the worker manager calls the provider who performs the unassign operation with the cloud service.

3.3.3 Infrastructure Scaling

The scaling down and up of previously created and therefore consented workers is in certain parts pretty similar so this section will first show how the worker manager gets the call and then go into the different operations individually.

Set Worker Status

The scaling is from a root perspective a simple setting of a worker nodes status to either STOP, RESTART, TERMINATE or RECREATE. Traditionally the root level of orchestra had no detailed information about individual worker nodes to keep the independence between the tiers. However when managing and provisioning the infrastructure of a cluster itself then we need to show these individual nodes on the root level. On root level the cluster is received from the cluster service along with the credentials and region info. From there the set status request is forwarded to the worker manager where they are then handled differently depending on the selected status. The result of the operation is then returned again as a status back to the root to show whether it succeeded.

Stopping, Terminating & Deleting Worker

Deleting a worker requires the node also to be terminated first before it gets deleted. Therefore all three operations can be described together. All of these turning off worker operations start by setting the node to a `TERMINATING` state so that it is no longer counted as an active node and can no longer accept new deployment instances. Next all currently running jobs of the node are being un- and redeployed by the deployment service to the now remaining nodes of the cluster or in case of not enough resources back on a global root level. After the undeployment has been triggered the node can then be stopped or deleted on the provider via the provider manager using the stop or delete provider operation. The delete operation is used for both just terminating the worker instance and permanently deleting it. Once the provider has confirmed the successful stopping or termination the node status is set accordingly to the STOPPED or TERMINATED state. In case the node should be completely deleted and not just stopped or terminated, then the node gets deleted in the repository by the worker manager as well using the worker service. This means that application provider can no longer recreate a completely deleted instance. A completely new one has to be approved again by the infrastructure provider. Therefore only those can delete workers permanently.

Restarting Worker

Restarting a currently stopped worker begins with the reissuing of a new registration token by the authentication service. With that the restart request is directly forwarded to the Provider Adapter. There cloud service first triggers the restart instance cloud operation and then already responds back to the worker manager. Next, asynchronously, the restart of the node components is prepared by retrieving the key to access the worker instance and then waiting for the cloud instance to become ready with an external IP address. As soon as it is ready the installation service is called to connect to the instance and there restart the node component using the new IP address. The node then starts with the node registration service and just like on creating a worker it connects to the cluster using the reissued registration token. Upon this registration the node becomes `ACTIVE` and ready to accept jobs again.

Recreating Worker

After terminating an instance it only exists anymore in the cluster's node repository but no longer in any form on the cloud. Therefore the recreating is a mix of the worker restart and creation operation. First it reissues a token again via the Authentication Service, then it retrieves the node configuration information with the Worker Service

and then it sets the node to booting again. The stored configuration information from the initial worker creation is then reused for the input to trigger the same create worker operation of the provider which is explained in more detail in its section. The returned new instance id of the worker is then set as the current one to the node in the repository. Then the attaching process is the same as soon as the node is running with the node components as in restarting and creation of a worker.

4 Implementation

This chapter presents an MVP implementation for the previously introduced design of a multi-cloud infrastructure broker. It is applied to the Oakestra because its low-overhead, heterogenous approach is uniquely fitting to this design. This chapter first 4.1 introduces and justifies the chosen technologies for the implementation of the design. The following section 4.2 maps the operations and logical components to the actual endpoints and components in the implementation with a particular focus on the integration into the existing Oakestra implementation. Next in section 4.3 the general implementation design of the server structures are presented. And finally there'll be a more in-depth look into the design of the Provider Adapter so that they can easily be implemented to support more cloud providers.

4.1 Technologies

4.1.1 Server

Golang Gin Server Almost all of the newly introduced components are implemented using the Golang [9] programming language. It was chosen due to it's high performance, low-overhead design and robust lightweight multi-threading using Goroutines. Therefore it is widely used in networking and cloud computing. As for the HTTP server and networking frame framework the choice went to Gin [41] as one of the two most used frameworks since Gorilla Mux was at the time of the start of the project in an archived state and only has been recently revived [44].

Python Flask The only server written using a different language is the JWT Generator with Python Flask as it's used in most of the Oakestra framework including some token issuance functionality which was extracted from the main components and therefore could be reused in large parts without breaking changes.

4.1.2 Technologies

Consul For the service registration and discovery functionality Consul was chosen as the solution. A great benefit of it, is the option to combine it in the future with Consul

Template which allows it to work with the lightweight API gateway NGINX.

MongoDB Apart from being already used as the existing database, mongoDB [13] is the optimal choice for an optional component like this multi cloud broker. It allows with its document-oriented design to create collections and entries without the need for database migrations. Therefore the main oakstra components don't even have to be aware of these multi cloud components.

Docker In order to allow the system to run in a multitude of environments each component on root and cluster level is virtualized as Docker containers. Along with docker compose they are easy to run either by building or pulling an image.

JWT As a result of so many different components interacting in various different locations over the public internet they need to be properly secured. For this JSON Web Tokens (JWT) were chosen as they are the industry standard and widely used inside Oakestra [43]. There even existed already a registration mechanism on cluster level to the root which has been extended to work on the node and provider registrations. Also it is used to authenticate individual users interacting with the root system.

4.2 Implementation Mapping

This section maps the in the System Design to actual components and endpoints in the implementation. It especially details the separation of functionality within a tier.

4.2.1 Root

Root Orchestrator

The root orchestrator is the collection of all pre-existing Oakestra root components and therefore consists of several components like the **service manager** and **scheduler**. Although only the **root database** in connection to the **system manager** are the only directly interacted with components as part of the multi-cloud functionality. The database is shared among both the multi-cloud broker component and system manager with each of them implementing their respective database interfaces for the needed functions. This database contains all the user, cluster and credentials data. The root orchestrator uses them directly for multi-cloud purposes to register clusters and indirectly by handling the login of users and deployment of services which can trigger firewall changes in the cluster tier. Apart from bugfixing and minor changes the only big change needed was the extraction of the JWT issuance to a dedicated service,

which is a good idea independent from the multi-cloud functionality but a necessity to authenticate incoming requests with the broker as a second user entrypoint for root.

Multi-Cloud Broker

The main component for multi-cloud functionality is the broker itself which is the only entrypoint for infrastructure-related requests. It handles the functionality of the same named logical component in handling all incoming requests, orchestrating and delegating all steps on the root level. Besides that it also implements the whole credentials service with the credentials and region caching data as well as most cluster objects related interaction, aside from registering them, using the root database. It also interacts with the users in the database by setting the last used region. While most authentication handling is part of the system manager the broker also uses the JWT generator to generate and validate registration tokens for provider adapters. [33]

Consul Registry

The consul component which is so far exclusively used for provider registration, but should be extended to also register internal root services, is kept stock from the Docker image. Only the Multi-Cloud Broker interacts with it to attach providers after handling the registration process ahead on its own and to retrieve provider adapter server instances.

JWT Generator

The JWT generator implements the Authentication Service functionality by generating a private signing key and then issuing JWT tokens to other root services which can be validated via the public key it provides. It is a complete custom implementation which offers token issuance and public key retrieval as http endpoints and maps them to the `jwt_flask_extended` library functions. The introduction of this generator replaces the previously used symmetric token issuance using the same library in the root orchestrator's system manager. The change to asymmetric JWT signing is needed to allow both system manager as well as the multi cloud broker to validate the tokens in their respective middlewares without having to dangerously share a common shared or private key. [35]

4.2.2 Cluster

Worker Manager

The main multi-cloud component on the cluster is again combined into a single-entry component called worker manager which manages all the worker related infrastructure concerns on a cluster level. It is not just called from the system manager who authenticates, interprets and relays worker requests on the root level to the cluster tier but it also receives requests directly from the **cluster manager** when firewall changes are needed. The worker manager itself contains the logical functionality of the worker and job services which act as a repository for those objects in the **cluster database**. [34]

Cluster Orchestrator

Similarly to the root orchestrator it is also made up of four main components however with an adapted functionality. The multi-cloud components again only interact with the **cluster database** and **cluster manager** directly. However upon notification of the schedulers result the cluster manager also now sends a firewall adaption request to the worker manager so that it is configured for the newly deployed service. Also in reverse it triggers the removal of a firewall if the last job instance of a type is being undeployed. Since the worker manager can also stop or terminate workers and to ensure continual uptime of services the cluster manager also offers a new endpoint to redeploy all jobs of a given node. Besides the mainly deployment related changes to the cluster manager the introduction of creating workers also required the adaption of the existing cluster registration to the root a new node registration to this cluster manager. For this task exclusively as of now the cluster tier gets the copy of the roots JWT generator to generate and validate the node registration tokens it issues.

4.2.3 Node

The node tier stayed almost the same with the incorporation of the logical node registration service into the **Node Engine**. It basically just adds the node registration token as a parameter and sends it as part of the handshake with the cluster.

4.2.4 Cloud Provider Adapter

The four logical components inside the provider tier are combined into a single cloud provider server which contains all those functionalities. The provider registration and installation services are simply realized as service as described in the server structure 4.3. The cloud service has a unique architecture and integration to allow for maximum

reuse and more in-depth explained in the provider implementation section 4.4. [30] [29] [32]

4.3 Server Structure

In total the multi-cloud functionality introduces three completely brand new go gin servers into the system with the **multi-cloud-broker**, **worker manager** and **cloud provider**. While they each serve very functionality the general structure of the server is the similar, especially the first two, and therefore will be examined together. Most custom structures of the cloud provider server are examined in the next section 4.4.

4.3.1 Main & Server

The `main.go` file is only for solely for initializing the server, which has its initialization function in `server.go`. This server initializes there all the special low-level clients which are used across the whole application which is in this case is a generic http client and the `mongoClient` connected to either the root or cluster database. Finally the server creates a router and runs them on the chosen port.

4.3.2 Router

The router itself is configured in `router.go` along with controllers, services, repositories and clients. The main functionality of the router of routing the requests to concrete functions is implemented using the routers `GET()`, `POST()` and `DELETE()` functions which each take in a path with parameters `/path/:parameter/subpath` and a function from a controller which handles the request using the `gin.Context`.

JWT Middleware Lastly on the multi cloud broker server the router also initializes a JWT middleware and sets the JWT public key it received in a setup key retrieval request inside the router. This middleware simply checks when a JWT authorization bearer token is provided if it's valid using the public key and parses it to retrieve the tokens data which is primarily the signed identity as the `sub` claim and can be extended to also check roles. The token and identity are then attached to the context via `Set(key string, value any)` before the request are forwarded to the controllers where those two attributes are now available through the context.

4.3.3 Controllers & DTOs

The controllers's purpose is to receive the incoming requests, process the inputs, calling the respective service functions and then respond with the results or http errors. The requests are typically grouped into controllers by their first path component in the url. In the requests implemented there are three sources of inputs that are getting processed. First values from the middleware like identity that have been processed there and stored in the context. They can be mandatory or optional and have to be casted from any `context.MustGet("identity").(string)`. The second source of input are path variables that are passed via the URL and caught via the router to the context where they can be read (e.g. `context.Param("workerId")`). And lastly input from the requests body formatted in json which can be bound to an empty pointer of the expected object `err := context.ShouldBindJSON(&requestBody)`. Typically these object types are defined as Data Transfer Objects (DTO) structs which contain a json key that can be different to the attribute name (e.g. `ClusterName string ` json:"cluster_name"` `). When all inputs are retrieved they are then passed as parameters to specific service functions which do the most work. These services are held within the controller objects. Once these service functions conclude they provide a result either in form of a response object (mostly DTOs) and potentially raised errors. A response with either the DTO or error message are then sent along with a http status code via the context as a JSON `context.JSON(http.StatusOK, responseObject)` or string `context.String(httpError.StatusCode, httpError.Error())`.

The multi-cloud broker has the following controllers: `ClusterController`, `CredentialController`, `ProviderController` and `WorkerController`.

The worker manager has only these two controllers: `firewall` and `worker`.

The cloud provider has these controllers: `AttachController`, `FirewallController`, `OfferController` and `WorkerController`.

4.3.4 Services

The core of the orchestration is achieved by services. They can exclusively focus on that without need to bother about http request and response parsing (controller) or actions on others (clients, repositories). It handles this control-flow and operates on input, intermediate and response values. However for simple requests like creating credentials they have been omitted as there are no complex flows but instead controllers call directly the repository. Services themselves usually don't contain other services but they store clients and repositories. The workflows inside these services are for the most part just the operations from System Design 3.3 from the Multi Cloud Broker, Worker Manager and Provider Manager components.

4.3.5 Clients

Clients are primarily interfaces for request to other components or certain library clients (consul). They contain the code to perform these requests with url creation, request encoding, response decoding and error handling so that the functionality of these requests appear as easy black boxes from the services point of view like local function calls.

The clients include on the multi-cloud broker `Consul`, `JWTGenerator`, `Provider`, `SystemManager`, and `WorkerManager`.

The worker manager has only two clients: `cluster-manager` and cloud `provider`.

4.3.6 Repositories & Models

Repositories interface the MongoDB collections. They are each initialized with a mongoClient on which the collection is retrieved and stored as the repository attribute like this:

```
func NewClusterRepo(mongoClient *mongo.Client) ClusterRepo {
  return ClusterRepo{
    clustersCollection: mongoClient.Database("clusters").Collection("clusters"),
  }
}
```

Each database operation is then interfaced here with input validation mapping to filters and updates. The results from the database are then either directly returned as model objects, gathered from cursors on list operations, or only returned as an optional error to indicate whether or not the operation worked.

4.4 Provider

4.4.1 Architecture

The overall structure of the provider server is pretty similar to the multi-cloud broker and the worker manager with its controllers and services. A key difference however is that the provider-specific clients (like AWS EC2Client) can't be created at start-up as they for the most part require authentication credentials to initialize which are only provided by the requests. This would mean in a traditional set-up that upon one request from a worker manager or the multi-cloud broker which triggers multiple API calls to the cloud provider that on each one of them they would need to initialize a separate client which is quite inefficient.

Instead, at the beginning of each request, the handling services instantiate the provider-specific, credentials-dependent provider service from a factory type which takes the credentials as an argument and creates the provider service with the provider-specific clients. Since the request handling services are part of the provider-independent provider manager logical component, they can't contain concrete provider factories. Therefore inheritance is used via an abstract interface which defines all needed cloud operations in provider-agnostic functions representing the cloud service from the design 3.2.

Together this forms an **abstract factory pattern** [**designPatterns**] as depicted in Figure since the handling services expect any provider factory which then creates the provider-specific service.

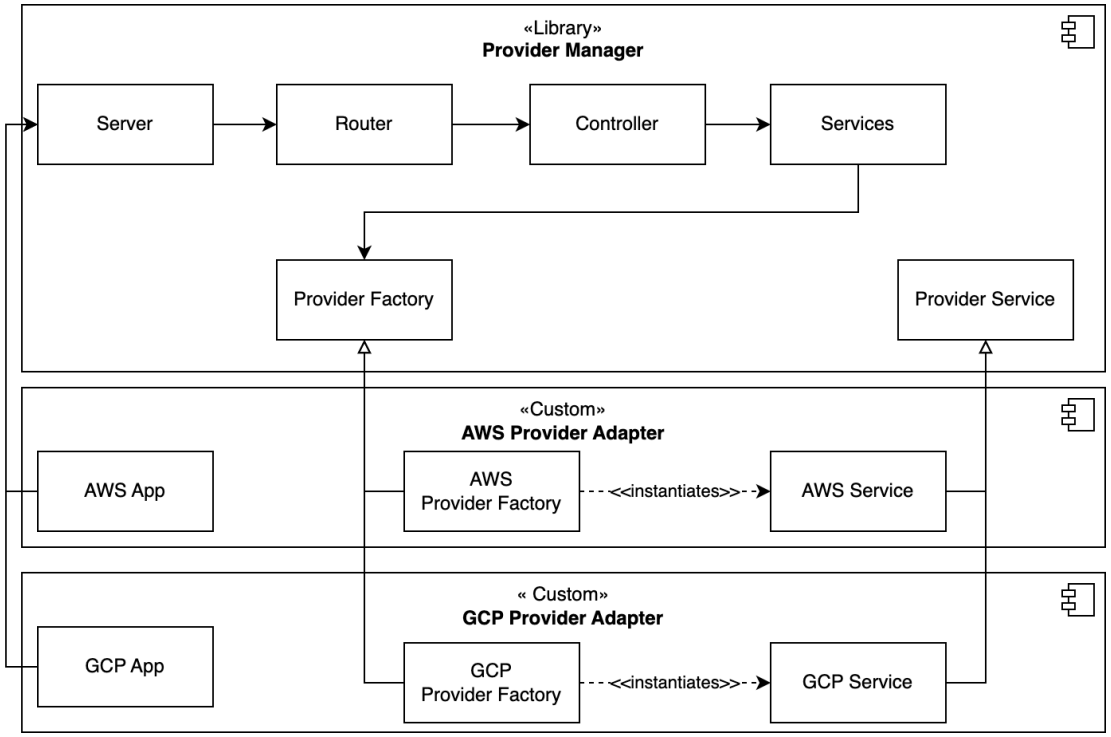

Figure 4.1: Abstract Factory Pattern

The is factory is passed through the whole structure starting from the server via the controllers to the handling services. This allows the whole server to be decoupled from the provider-specific code and only take the factory as a single parameter. Using the decoupling the common server can be a separate library in form of a go module that can be imported by provider implementations. These then only need to implement

a provider specific service implementing the specified functions, a factory which can instantiate this service and a base application that calls the creation of the server from the library. As a result there's very little implementation overhead when creating a new adapter for another cloud provider. With only small adaptions in the router and server input it would be even possible to host multiple provider adapters in one application using the same abstract factory pattern.

4.4.2 Interface

Since all providers need to implement the uniform interface for the factory and service these have to be both generic enough to support any provider and specific enough to provide all the information needed by the concrete implementation. Therefore this interface is very generic and on a high level leaving all these details to the provider service implementations. In this subsection the interface's function signatures are provided and the mapping described for the AWS and GCP providers showing how they differ and how these differences are handled using this architecture.

Instance Management

```
CreateInstances(
        instanceCount int,
        instanceType string,
        imageId string,
        diskSize int,
        keyName string,
        firewallId string,
        clusterName string,
        workerInstances bool,
) ([]string, error)
StopInstance(instanceId string) error
RestartInstance(instanceId string) error
DeleteInstance(instanceId string) error
WaitForInstanceRunning(instanceId string) (string, error)
```

The instance functions represent the four actions on instances of creating and deleting them as well as stopping and restarting instances. All very common features for cloud computing instances and needed for the multi-cloud broker operations. On AWS they are realized with the `ec2.Client` and on GCP with the `compute.InstancesClient`.

CreateInstances first of all takes in all the instance configuration attributes from the operation itself like `instanceType`, `imageId` and `diskSize`. Then it is assumed that firewall and keys to connect to instance later on can be configured separately and reused so therefore just ids or names are needed. Then a cluster name from the root orchestrator is passed as many resources are based on this name and it helps identifying them along with a flag whether not the created instances represent workers or a cluster. Finally there is also an instant count which makes it possible to support batch creations of this complex operation. This operation returns then an array of strings with instanceIds of the created instances which are provider specific ids and used for identification with the other instance provider functions.

On AWS this is a very straight forward using `ec2.Client.RunInstances` although in order to connect the created `Amazon Elastic Block Store` start disk it is needed to bind it correctly to the instance according to the image's specification as different operating systems have different boot disks. This information is retrieved by a second `ec2.Client.DescribeImages` request.

Using GCP on the other hand the boot disk is directly specified via a boolean attribute so such a check and specification is not needed here. But in comparison to AWS the SSH keys are not explicitly stored so therefore it needs to first retrieve the public key from the SSH secret using the service's `GetKeyPairData` function and then add it as a metadata to the created instance. Two additional differences are that GCP Compute instances have the firewalls automatically attached via instance tags and they don't have by default a public network interface. To be accessible by the internet and have a public IP address it needs to be configured which can happen as part of the instance creation request which is `compute.InstancesClient.Insert`.

StopInstance, RestartInstance and DeleteInstance work pretty similar for both cloud providers as a simple `ec2.Client.StopInstances`, `ec2.Client.StartInstances` & `ec2.Client.TerminateInstances` as well as `compute.InstancesClient.Stop`, `compute.InstancesClient.Start` & `compute.InstancesClient.Delete` requests.

WaitForInstanceRunning is a provider operation which is really different between both implementations. While AWS provides with `ec2.InstanceRunningWaiter.Wait()` a blocking function out of the box which waits until the instance is running and ready for interaction, GCP doesn't offer anything like this. Therefore for Google Compute Instances there's a retry loop with exponential backoff to get the instance `compute.InstancesClient.Get` and check whether the network interface has already a public IP address assigned. This address is then returned by this operation while on AWS after the wait has completed the IP address has to be fetched in a separate

`ec2.Client.DescribeInstances` request.

Key Management

```
GetKeyPairData(keyName string) (dto.KeyPairData, error)
CreateAndStoreKeyPair(keyName string) (dto.KeyPairData, error)
```

On both AWS and GCP secret managers are used to store the SSH keys that are used to connect.

CreateAndStoreKeyPair however is very different between them. AWS offers via the `ec2.Client` to generate key pairs itself `ec2.Client.DescribeKeyPairs` which can then be directly attached to instances via the key name. This key's content is retrieved via `ec2.Client.CreateKeyPair` and then stored `secretsmanager.Client.CreateSecret`.

GCP's SDK itself doesn't offer the functionality to create new keys or even store them directly. Therefore the private key is generated with the `rsa.GenerateKey` function, marshaled with `x509.MarshalPKCS1PrivateKey` and stored as pem bytes `pem.EncodeToMemory`. The public key is calculated using `ssh.NewPublicKey` and also marshaled `ssh.MarshalAuthorizedKey`. The storing itself has also one key difference as in GCP the secrets are created blank `secretmanager.Client.CreateSecret` and only have the content stored explicitly in secret versions `secretmanager.Client.AddSecretVersion`.

GetKeyPairData is very similar to the storing of the secret itself as on AWS the secret value can be retrieved directly with `secretsmanager.Client.GetSecretValue` while GCP again operates over the version `secretmanager.Client.AccessSecretVersion`.

Firewall Management

```
CreateFirewall(ports []int, name string, clusterName string) (string, error)
AttachFirewall(firewallId string, instanceId string) error
DetachFirewall(firewallId string, instanceId string) error
CheckIfFirewallExists(firewallId string) (bool, error)
```

Firewalls are realized again in different way between cloud providers. AWS EC2 has them as `SecurityGroup` resources that can be directly attached to instances while Google Compute attaches their firewalls via a tagging system where target tags are specified on the firewall itself and it is then attached to those instances who have one of these tags. Both EC2 SecurityGroups and Compute Firewalls are pretty similar with

a configuration of traffic directions, IP protocols, IP & port permissions. However in AWS the creation `ec2.Client.CreateSecurityGroup` of a security group and setting up the ingress rules `ec2.Client.AuthorizeSecurityGroupIngress` are two separate requests while on Google Cloud it's a single one `compute.FirewallsClient.Insert`. In the Oakestra case only tcp ports are supported as a parameter.

Attaching and Detaching of firewalls is works on both clouds by adjusting the instance attributes. On AWS this achieved by fetching the instance `ec2.Client.DescribeInstances`, changing the `SecurityGroups` array of string ids and then updating again the instance attributes `ec2.Client.ModifyInstanceAttribute`. On GCP it's pretty much the same with fetching the instance `compute.InstancesClient.Get`, adapting the target tags array and then setting the tags `compute.InstancesClient.SetTags`. Since the firewalls use a specific naming scheme it's necessary to check whether or not the firewalls exist which is achieved by the `CheckIfFirewallExists` function using `ec2.Client.DescribeSecurityGroups` and `compute.FirewallsClient.Get`.

Cluster User Management

```
CreateClusterUser(clusterName string) (json.RawMessage, error)
DeleteClusterUser(clusterName string) error
```

Despite different user and identity management systems the creation of dedicated cluster user accounts that can adapt firewall walls independent from the root orchestrator is very similar. First a new user account gets created. On AWS they are regular users `iam.Client.CreateUser` and on GCP they are called ServiceAccount `iam.IamClient.CreateServiceAccount` when they are meant to be used by an API like this. Then policies are defined which for AWS are pretty complex JSON objects describing the various permissions with conditions `iam.Client.CreatePolicy`. On GCP AMI this is realized via roles describing the permissions `iam.IamClient.CreateRole`. Once the policy / role is created it needs to be attached to the user which was previously created. AWS has for that a single endpoint `iam.Client.AttachUserPolicy` while on the Google Cloud the roles mappings are stored in a project-wide policy object. So therefore this policy object has to be fetched `cloudresourcemanager.Service.Projects.GetIamPolicy`, adapted to include a new binding between the role and the service account and finally once again set `cloudresourcemanager.Service.Projects.SetIamPolicy`. Now that the user account is correctly configured account keys still need to be generated for the cluster to be able to use them as credentials in the provider adapter. This is achieved by `iam.Client.CreateAccessKey` on Amazon and `iam.IamClient.CreateServiceAccountKey` on Google.

Offer Retrieval

```
GetRegions() ([]string, error)
GetInstanceTypes() ([]dto.OfferInstanceTypeResponse, error)
GetImagesIds() ([]dto.OfferImageTypeResponse, error)
```

Besides the actual management of the multi-cloud infrastructure through the provider it is also used to retrieve information about offered resources so it can be displayed uniformly by a frontend.

GetRegions When it comes to regionalization Google Cloud resources are much more restricted to zones within a region. So in order to have the best consistency in all resources it was decided to use directly zones instead of regions as intermediaries. Therefore the GCP implementation simply retrieves all zones and returns their names `compute.ZonesClient.List`. On Amazon the regular regions are used `ec2.Client.DescribeRegions`.

GetInstanceTypes is used to get not just the name of available instance types but also attributes like number of vCPUs, vGPUs and memory which are the performance constraint attributes of Oakestra. Google provides these details already in the standard retrieval `compute.MachineTypesClient.List` of MachineTypes, as they call it. Amazon offers this detail only in a separate `ec2.Client.DescribeInstanceTypes` request for which the instance type names are fetched from `ec2.Client.DescribeInstanceTypeOfferings`.

GetImagesIds is basically the same on both cloud platforms with the `ec2.Client.DescribeImages` and `compute.ImagesClient.List` APIs.

Additional Information

```
GetDefaultUserName() string
```

Finally there is one more additional info function `GetDefaultUserName` that enables the rest of the provider adapter to get the user name that is used on the instances which is needed to login via SSH. Some providers have default names that can be provided here and some allow the specification of one in which case the choice is also available there.

4.4.3 Installation

The second major part of the Provider Adapter besides the mapping of operations to provider-specific code is the installation of the Oakestra and its multi cloud components to newly created instances so that they can connect.

SSH Connection

The implementation is executed using the Secure Shell Protocol (SSH) [15] and the Go implementation [8]. A connection is established using the username from `GetDefaultUserName`, ip address from `WaitForInstanceRunning` and the private key from either
`CreateAndStoreKeyPair` or `GetKeyPairData`. Most of the installation is done via bash scripts that is copied to the using the Secure Copy Protocol (SCP) [24].

Cluster Installation

The cluster installation first copies the `cluster_install.sh` and `installer_function.sh` helper script and then two environment files parsed to via ssh using the `cat` program. The cluster installation script first installs using the helper script git, curl and docker along with docker-compose. Then it clones worker manager repository and the oakestra one for the cluster orchestrator. Finally it runs both of these docker compose files with the respective env file.

Worker Installation

The worker installation runs in comparison to the cluster not within a docker container but as an executable. As it has a significantly lower overhead which is necessary for less powerful edge nodes. First of all also here it installs all necessary helper programs with wget, tar, iptables, runc and jq. The net manager installation starts with the download of the binary and an accompanying installation script which gets executed. The necessary parameters get set via its configuration JSON file. Then the NetManager itself gets started.

The same procedure happens then also with the NodeEngine except that here all parameters are passed as command line arguments which include the cluster's IP address and the nodes registration token. Ahead of the start of the NodeEngine there's a sleep function of ten seconds called. NetManager usually takes a bit of time to fully boot up and it is necessary to have it ready when starting the NodeEngine with the NetManager enabled.

Worker Restart

A worker restart after an instance has been stopped also requires the execution of some commands again as the stopping has terminated both NodeEngine and NetManager processes. Since the worker node gets usually a new IP address assigned after restart it has to be updated as well in the NetManager configuration json using the jq program. Then both NetManager and NodeManager can be started again just like after the installation also with a ten seconds pause between them.

5 Evaluation

Following the implementation of the multi cloud broker integration into Oakestra this chapter aims to aims to show how efficient and quick this implementation in comparison to a similar cloud broker. The focus is the provisioning of new cloud instances, set up and connection time until they are ready to receive computing jobs as this is the most significant measurable benchmark.

5.1 SkyPilot Comparison

As a comparison broker and orchestrator SkyPilot [45] was chosen as it's another novel approach of implementing the vision of Sky computing. While Oakestra approaches this topic from the edge computing point of view and extending it by cloud functionality, SkyPilot is primarily designed for running queued batch jobs and other non-urgent as cost-effective as possible on the cloud taking full advantage of cloud spot management. However, since both can provision cloud instance worker nodes organized into clusters and run services on them dependent on SLA requirements they make this aspect an interesting and fair comparison under the consideration of their different application areas and goals.

5.1.1 Key Differences

These differences in goals is also mirrored by key differences in both their offered functionality and implementation structure. First of all while Oakestra is mixed of Python and Golang components, with the latter almost exclusively used for the multi cloud integration, SkyPilot is completely written and running in Python.

Cloud Provider & Credentials Management Similar to Oakestra it allows the installation of various cloud provider plugins however they are not dynamically bound as separate server instances. Instead they are part of the SkyPilot installation itself upon which the to be installed provider plugins can be chosen and even later on extended. Each provider plugin however relies on the installation of their respective python command line tool like `boto3` for Amazon Web Services and `gcloud` for Google cloud Platform. The credentials setup runs via those installed command line clients. All these

steps are not abstracted in a uniform cloud provider agnostic fashion like the Oakestra Multi Cloud Broker does with both a service discovery based provider plugin system to dynamically attach providers without interacting with a provider-specific interface and storing and managing credentials from multiple users and organizations.

Cluster Architecture The biggest architectural differences of the provisioned clusters is that on Oakestra each cluster usually and especially using the broker runs on its own instance consuming hardware exclusively in a strict three tier system. SkyPilot on the other hand doesn't dedicate resources only for cluster management but instead delegates this responsibility to one worker node called the head node. So from a cloud resources point of view it looks more like a two-tier system than the logical three tier one. Therefore SkyPilot needs one less instance saving costs while taking computational power from one worker node.

Immutable Cluster Management A third major difference is the immutability of cluster configurations on SkyPilot. Once a cluster's configuration has been defined it can't be changed anymore This makes sense as it was designed to dynamically spawn up resources taking advantage of cloud spot management to execute a finite job after which those resources can be deleted again. Oakestra coming from an edge computing background had a number of fluctuating resources it can offer to the system and host longer-term offloading use cases where it's not expected for resources to be turned off or on. This results in various features being available only to the Oakestra broker. First the number of nodes in a cluster is already fixed upon creation of the cluster and can't be scaled up or down. Either the whole cluster is running or it is completely down. The Oakestra MCB on the other side can dynamically scale by adding workers to a cluster, stopping or terminating individual ones and restarting or recreating them.

This immutability is not only limited to cluster wide configurations but also to node configurations but also extends to the port exposure. In SkyPilot all about to exposed ports have to be defined by during start up and can't be adapted depending on which service is currently running. In contrast Oakestra checks as part of the deployment and undeployment operations whether the firewall needs to be adjusted for these jobs.

5.2 Setup

To ensure the best effort in terms of fairness between the two evaluated competitors the root tier of both run inside of docker containers. They evaluated operations are executed, monitored and controlled by a python script.

However, since SkyPilot is exclusively a Command Line Interface program and Oakestra is an HTTP server based solution the operations are sent to the broker respectively via executing commands inside the docker container and as HTTP requests to the containers port.

Tested have been the version 0.1.0 of the Oakestra Multi Cloud Broker enabled with extensive logging and the nightly build of SkyPilot from the 23rd September 2023 with the AWS and GCP plugins installed and a minor adaption to the logging increasing the accuracy to nanoseconds level. [38]

Additionally the root database of the Oakestra instance has been regularly queried during the evaluation. Except for that there has been no external influence on them. Both Oakestra and SkyPilot instances ran on a single, shared, neutral Digital Ocean server with 2 Intel vCPUs, 4 GB memory and 120 GB disk space running on Ubuntu 22.04 (LTS) in Frankfurt, Germany.

5.3 Testing Procedure

The evaluation python script loops alternating over two very similar procedures implemented for Oakestra and SkyPilot respectively so that they are as comparable as possible. Each of them logs the start, end, failure and some in between steps as formatted rows into a csv log file containing all necessary context information like parameters, operation and steps. [27]

Set Up On Oakestra the setup builds and starts both the regular Oakestra root orchestrator and the multi cloud broker components via their respective docker compose files. The AWS and GCP plugins are started from their pulled DockerHub image in their pre-librarization state [31]. Once all containers are running the credentials are added for both AWS and GCP to the broker via the `POST /credential` endpoint. Then both provider adapters get registered and attached to the broker. The already running SkyPilot container is retrieved from the docker demon.

Cluster Creation The cluster creation including the initial workers is executed twice per iteration. Once from a clean slate without any other clusters running where information is being cached. These two clusters are called `ColdCluster` and `WarmCluster`.

On Oakestra each cluster creation is split up into two parts first the creation of the cluster itself via `POST /cluster` and once the cluster is paired, which is checked by a direct database query every five seconds whether the `pairing_complete` flag is set. The second step is creating the worker via the `POST /cluster/<clusterId>/worker` request since the cluster is running independent from workers on a separate instance

and therefore has a separate endpoint. Just like with the cluster pairing the evaluation script also waits for the all requested workers to become active by querying the database for the number of `active_nodes` of the cluster.

For SkyPilot all cluster operations use the same empty yaml configuration file as all parameter are instead directly passed as command line arguments. New clusters including workers are created by running `sky launch` with the specified parameters. This command is executed via the Docker python library's `exec_run`. Once the command has successfully completed then a cluster creation finished log entry is created. Additionally, the internal logs of SkyPilot are parsed to create additional logs to store the time when the head node is created compared to the other nodes.

Worker Stop & Restart For stop and restart an according status request is sent to the Oakestra MCB. The restart is once again confirmed by waiting for the correct number of active nodes of the cluster.

The SkyPilot evaluation uses `sky stop` and `sky launch` as apparently the durations suggest that for a stopped cluster the `sky launch` command is equivalent to `sky start`. Unlike Oakestra which enables the workers fine-grained individually, SkyPilot always stops or starts the whole cluster.

Worker Termination & Recreation The termination and recreation of worker is very similar in the evaluation to the previous stop and restart behavior. Except that for Oakestra the statuses parameter is adapted and the SkyPilot evaluation uses `sky down` to completely destroy the instance.

Additional Workers Besides restarting and recreating workers or clusters, Oakestra offers a unique possibility compared to SkyPilot to create additional workers. The evaluation in total triggers this thrice as besides the explicit increase in additional workers in this step it was part of the cluster creation as the second step.

Clean Up Finally in the end both evaluation processes are cleaned up so that the next iteration is completed without interference. On Oakestra this includes a complete destruction of all containers while SkyPilot already gets rid of all data with the command `sky down -all`.

5.3.1 Parameters

For the the evaluation the following parameters where executed and evaluated.

Instance Type Initially the goal was to test both AWS and GCP with the most basic general purpose instance types `t2.micro` and `e2-micro`. However, the latter on GCP was so slow that most of the times timeouts were triggered. Therefore on GCP it was evaluated with the one size larger instance type `e2-small`.

Locations As the locations the region near Frankfurt in Germany where also the evaluating root server on Digital Ocean is hosted. On AWS it's `eu-central-1`, on GCP `europe-west3` and since Oakestra uses zones for GCP `europe-west3-c`.

Images The images have been chosen to be the default ones on all four configurations. For Oakestra this means on AWS a basic `Amazon Linux 2` image and on GCP a `Debian 11` image. SkyPilot uses its own default images `Google, Deep Learning VM, M113, Debian 11, Python 3.10` and `Deep Learning AMI GPU PyTorch 2.1.0 (Ubuntu 20.04)` which have a lot of Python Deep Learning kits like NumPy and SciPy already preinstalled while the ones Oakestra tested with were pretty blank.

Node Counts The evaluation was carried out with a single worker cluster, to see how the head node performs in SkyPilot. Along with cluster with a worker count of two, five and ten to get an indication how the provisioning times correlate with number of worker nodes.

Miscellaneous Besides these specific parameter settings the disk size was set to 60 GB in general as these Python installed SkyPilot default images are relatively big. And in general both frameworks Oakestra and SkyPilot were evaluated for both AWS and GCP provider.

5.4 Data Preprocessing

The raw data was delivered as csv files containing individual timestamps with context about operation, configuration and run identifiers. As cloud provisioning is very likely to result in failures from time to time which can be resolved by rerunning operations also a lot of initial errors occurred. The amount of retrying and error handling that the evaluation script is more limited than what an end user resolve. Therefore the evaluation run itself was stopped multiple times and resumed with continuing higher iteration numbers. From there it has been preprocessed into durations for each operation on each run. Failures resulted in either 0 values, negative values or impossible high values. There has been one small bug in the Oakestra evaluation where the no timestamp was created for the `AddWorker` start. This has been resolved by just using the previous

finished timestamp from the previous `RecreateWorker` timestamp which was logged directly before a 30 seconds delay and the start of the adding of workers. Therefore this timestamp was used with just 30 seconds subtracted. Another distinction worth noting is that for the workers there were always two types finished timestamps `Finished` and `WorkerReady` with the latter showing the time when the worker node was up compared to when the creation was completed also on root side. Especially on the SkyPilot evaluation this was important as finished was also logged if the operation failed so using this extra timestamp all failed tries were identified.

As a final step of the preprocessing all failed attempts had to be filtered out from averages as well as unusually high durations which occurred occasionally but were so big as to influence the averages significantly but also rare enough to have them as clear outliers and no regular variance.

5.5 Results

Analyzing the resulting durations two very interesting conclusions have been drawn first on the parameters influence of the parameters of cluster size and cloud provider. The second analysis is on how the operations compare in duration between them with the same configuration. The raw data and all processing steps have been made available on GitHub. [28]

5.5.1 Influence of Cluster Size & Cloud Provider

The first graph Figure 5.1 is combining the average cluster creation duration including worker creation for both Cold Cluster and Warm Cluster starts and comparing it primarily with start durations of the same cloud provider and platform set up but with different number of worker nodes. The goal is to find whether there is a correlation between creation duration and number of nodes.

Given the modest sample size and large variance of start up times with the same setup due to the large number of factors inside a datacenter that can influence them, there can only be one such correlation be drawn. The start up time for a single cluster using SkyPilot is about 42-45% faster than clusters with two or more worker nodes. A quick look into the raw data confirms the theory that this is due to the head node being setup first in SkyPilot just like the cluster instance in Oakestra. However since SkyPilot doesn't create a separate instance for the cluster this first created head node is the only one created in this operation and therefore there is no second step cutting the start up time almost in half.

From the same figure can also a comparison be drawn between SkyPilot and Oakestra with the same provider and how the same platform performs with different cloud

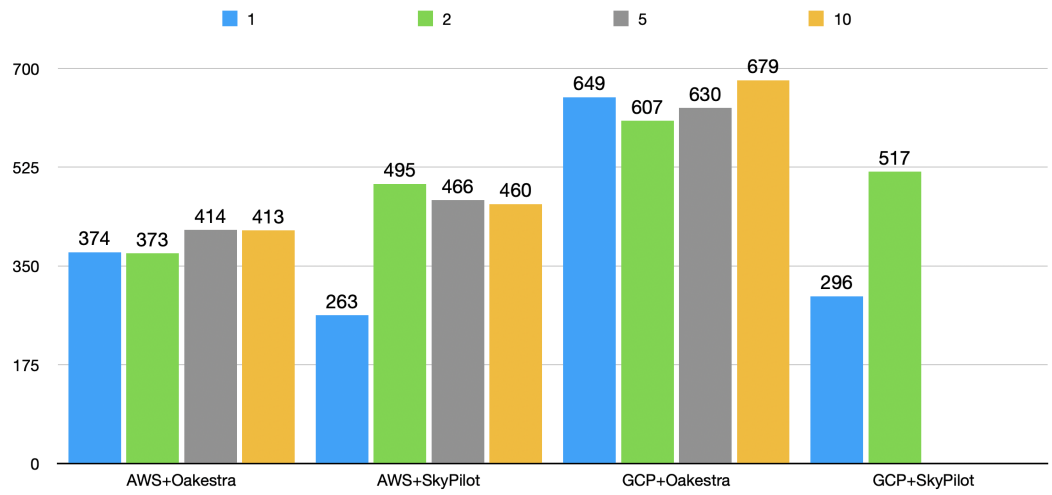

Figure 5.1: Provider Platform Duration Evaluation

providers. On AWS the two platforms are about the same speed and on GCP SkyPilot is a bit faster. While these comparisons between platforms always use the same instance type it was observed during the evaluation preparation that the type has a very strong impact on start up time. For example the `e2-micro` machine type on Google Cloud Platform was so slow setting up that SkyPilot always timed out the operation. In conclusion it can be said that the multi cloud broker implementation from a performance point of view is competitive with the proven SkyPilot. It seems that the overhead of each is to some point small even though both do heavy long lasting setups once the the instances are reachable. However the speed of the cloud provider seems to be the biggest determining factor.

5.5.2 Comparison of Operation Durations

A second analysis in Figure 5.2 compares the duration of various instance creations for all node counts now that we have determined that except for the single node cluster on SkyPilot they are about the same with the same provider and platform setup.

It shows that there is no significant speed difference between a cold and warm start despite there being a lot more setup work on Oakestra on the first start like creating key pairs, firewalls and user accounts.

Surprisingly the duration for restarting a stopped cluster or worker and recreating a terminated one is about the same in SkyPilot while in Oakestra it is about 45-55% slower. The reason for that being that worker nodes get installed from scratch on Oakestra, it

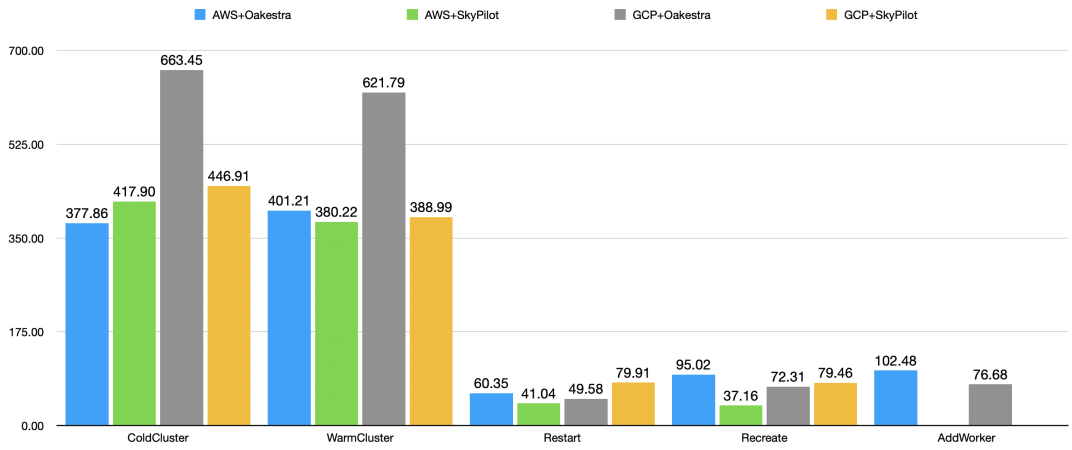

Figure 5.2: Operation Duration Evaluation

hasn't been determined why this difference was not observable. The Oakestra exclusive feature of adding new workers to existing clusters is only marginally slower with about 7% than recreating a terminated worker. It is however still important to have these two options since the recreating can be triggered by an application provider and not only by an infrastructure provider user.

In conclusion Oakestra is from a speed point of view competitive with SkyPilot while offering many unique features targeted to an edge computing application. The results between operations were about as expected with those operations with more heavy lifting taking longer than lightweight operation and in a range as expected.

6 Conclusion

6.1 Final Remarks

The in this thesis presented multi-cloud broker approach for an edge orchestration framework offers a new, easy and provider-agnostic approach to provision cloud resources and integrate them in existing systems. It helps bridging the gap between the edge and the cloud coming from an edge-first orchestration without the often strong underlying assumptions of cloud orchestration frameworks. The modular design of the provider plugins with a common generic interface requiring only the minimum set of features needed prevents vendor lock-in by making it easy and flexible to create and manage supporting cloud resources from multiple cloud providers. It doesn't require knowledge resulting in another step towards the vision of sky computing. The evaluation showed that it can rival established novel sky computing frameworks in terms of provisioning speed while providing unique features like scaling worker nodes with restarting or recreating nodes that have been previously consented to before stopping or terminating by infrastructure provider users. Additionally it automatically reconfigures firewall configurations for the currently deployed services.

6.2 Future Work

The implementation provides already the necessary functionality for it to work properly but certain improvements haven't been completed at the time of this thesis submission. This is includes especially security related topics like encrypting communication with TLS, access control using organizational accounts and the planned frontend interface even though except for the quote offer endpoint all necessary APIs are ready. Besides the immediate completion there are several ways this architecture was designed to incorporate future additions. First and foremost the number of supported cloud providers can be increased from the currently only two to more using the generic interface and the common library. The biggest future potential lies in the possibility to integrate the multi cloud broker with the scheduler who could then within a certain policy automatically create and terminate worker nodes for clusters depending on demand instead of like currently needing a manual trigger to scale from any user.

List of Figures

List of Tables

Bibliography

[1] A. Bahtovski and M. Gusev. "Cloudlet Challenges." In: *Procedia Engineering* 69 (2014). 24th DAAAM International Symposium on Intelligent Manufacturing and Automation, 2013, pp. 704–711. ISSN: 1877-7058. DOI: https://doi.org/10.1016/j.proeng.2014.03.045.

[2] A. Barron. *Pizza as a Service*. 2014. URL: https://www.linkedin.com/pulse/20140730172610-9679881-pizza-as-a-service.

[3] G. Bartolomeo. "Enabling Microservice Interactions within Heterogeneous Edge Infrastructures." PhD thesis. Master's Thesis. TUM, 15.09, 2021.

[4] G. Bartolomeo, M. Yosofie, S. Bäurle, O. Haluszczynski, N. Mohan, and J. Ott. "Oakestra: A Lightweight Hierarchical Orchestration Framework for Edge Computing." In: *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. Boston, MA: USENIX Association, July 2023, pp. 215–231. ISBN: 978-1-939133-35-9.

[5] S. Chollet and P. Lalanda. "An Extensible Abstract Service Orchestration Framework." In: *2009 IEEE International Conference on Web Services*. 2009, pp. 831–838. DOI: 10.1109/ICWS.2009.14.

[6] V. Cozzolino, L. Tonetto, N. Mohan, A. Y. Ding, and J. Ott. "Nimbus: Towards Latency-Energy Efficient Task Offloading for AR Services." In: *IEEE Transactions on Cloud Computing* 11.2 (2023), pp. 1530–1545. DOI: 10.1109/TCC.2022.3146615.

[7] Docker. *Docker Compose*. https://docs.docker.com/compose/. 2023.

[8] Google. *Go: Crypto SSH*. https://pkg.go.dev/golang.org/x/crypto/ssh. 2023.

[9] Google. *Golang*. https://go.dev. 2023.

[10] Google. *Google Cloud Compute Engine*. https://cloud.google.com/compute?hl=en. 2023.

[11] O. C. A. W. Group. *OpenFog reference architecture for fog computing*. https://www.iiconsortium.org/pdf/OpenFog_Reference_Architecture_2_09_17.pdf. 2017.

[12] S. R. Group. *Quarterly Cloud Market Once Again Grows by USD 10 Billion from 2022; Meanwhile, Little Change at the Top*. https://www.srgresearch.com/articles/quarterly-cloud-market-once-again-grows-by-10-billion-from-2022-meanwhile-little-change-at-the-top. 2023.

[13] M. Inc. *MongoDB*. https://www.mongodb.com. 2023.

[14] K. Keahey, M. Tsugawa, A. Matsunaga, and J. Fortes. "Sky Computing." In: *IEEE Internet Computing* 13.5 (2009), pp. 43–51. DOI: 10.1109/MIC.2009.94.

[15] C. M. Lonvick and T. Ylonen. *The Secure Shell (SSH) Connection Protocol*. RFC 4254. https://www.rfc-editor.org/info/rfc4254. Jan. 2006. DOI: 10.17487/RFC4254.

[16] P. López, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, and E. Riviere. "Edge-centric Computing." In: *ACM SIGCOMM Computer Communication Review* 45 (Sept. 2015), pp. 37–42. DOI: 10.1145/2831347.2831354.

[17] S. Maheshwari, D. Raychaudhuri, I. Seskar, and F. Bronzino. "Scalability and Performance Evaluation of Edge Cloud Systems for Latency Constrained Applications." In: Oct. 2018. DOI: 10.1109/SEC.2018.00028.

[18] D. Mairiza, D. Zowghi, and N. Nurmuliani. "An Investigation into the Notion of Non-Functional Requirements." In: *Proceedings of the 2010 ACM Symposium on Applied Computing*. SAC '10. Sierre, Switzerland: Association for Computing Machinery, 2010, pp. 311–317. ISBN: 9781605586397. DOI: 10.1145/1774088.1774153.

[19] A. Mazrekaj, I. Shabani, and B. Sejdiu. "Pricing Schemes in Cloud Computing: An Overview." In: *International Journal of Advanced Computer Science and Applications* 7.2 (2016). DOI: 10.14569/IJACSA.2016.070211.

[20] F. McNamee, S. Dustdar, P. Kilpatrick, W. Shi, I. Spence, and B. Varghese. "The Case for Adaptive Deep Neural Networks in Edge Computing." In: *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. 2021, pp. 43–52. DOI: 10.1109/CLOUD53861.2021.00017.

[21] Microsoft. *Microsoft Azure Virtual Machines*. https://azure.microsoft.com/en-us/products/virtual-machines. 2023.

[22] T. Munich. *Oakestra*. https://www.oakestra.io. 2023.

[23] S. A. Noghabi, L. Cox, S. Agarwal, and G. Ananthanarayanan. "The Emerging Landscape of Edge Computing." In: 23.4 (May 2020), pp. 11–20. ISSN: 2375-0529. DOI: 10.1145/3400713.3400717.

[24] OpenSSH. *Secure Copy Protocol*. https://man.openbsd.org/scp.1. 2023.

[25] D. Petcu. "Multi-Cloud: Expectations and Current Approaches." In: *Proceedings of the 2013 International Workshop on Multi-Cloud Applications and Federated Clouds*. MultiCloud '13. Prague, Czech Republic: Association for Computing Machinery, 2013, pp. 1–6. ISBN: 9781450320504. DOI: 10.1145/2462326.2462328.

[26] D. Sabella, A. Vaillant, P. Kuure, U. Rauschenbach, and F. Giust. "Mobile-edge computing architecture: The role of MEC in the Internet of Things." In: *IEEE Consumer Electronics Magazine* 5.4 (2016), pp. 84–91.

[27] M. Schlicker. *Multi-Cloud Broker Evaluation Code*. `https://github.com/michaelschlicker/oakestra-mcb-evaluation`. 2023.

[28] M. Schlicker. *Multi-Cloud Broker Evaluation Result*. `https://github.com/michaelschlicker/oakestra-mcb-evaluation-result`. 2023.

[29] M. Schlicker. *Oakestra AWS Provider*. `https://github.com/michaelschlicker/oakestra-aws-provider`. 2023.

[30] M. Schlicker. *Oakestra Cloud Provider*. `https://github.com/michaelschlicker/oakestra-cloud-provider`. 2023.

[31] M. Schlicker. *Oakestra Cloud Provider Image 0.1.0*. `https://hub.docker.com/r/michaelschlicker/oakestra-provider`. 2023.

[32] M. Schlicker. *Oakestra GCP Provider*. `https://github.com/michaelschlicker/oakestra-gcp-provider`. 2023.

[33] M. Schlicker. *Oakestra Multi Cloud Broker*. `https://github.com/michaelschlicker/oakestra-multi-cloud-broker`. 2023.

[34] M. Schlicker. *Oakestra Worker Manager*. `https://github.com/michaelschlicker/oakestra-worker-manager`. 2023.

[35] M. Schlicker. *Simple JWT Generator*. `https://github.com/michaelschlicker/simple-jwt-generator`. 2023.

[36] A. W. Services. *Amazon Web Services: Elastic Cloud 2*. `https://aws.amazon.com/ec2`. 2023.

[37] H. Shafiei, A. Khonsari, and P. Mousavi. "Serverless Computing: A Survey of Opportunities, Challenges, and Applications." In: *ACM Comput. Surv.* 54.11s (Nov. 2022). ISSN: 0360-0300. DOI: 10.1145/3510611.

[38] SkyPilot. *SkyPilot AWS GCP Build*. `https://hub.docker.com/r/michaelschlicker/oakestra-worker-manager`. 2023.

[39] N. I. of Standards and Technology. *The NIST Definition of Cloud Computing*. Tech. rep. Federal Information Processing Standards Publications (FIPS PUBS) 140-2, Change Notice 2 December 03, 2002. Washington, D.C.: U.S. Department of Commerce, 2011. DOI: 10.6028/NIST.SP.800-145.

[40] I. Stoica and S. Shenker. "From Cloud Computing to Sky Computing." In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. HotOS '21. Ann Arbor, Michigan: Association for Computing Machinery, 2021, pp. 26–32. ISBN: 9781450384384. DOI: 10.1145/3458336.3465301.

[41] G. Team. *Gin*. https://gin-gonic.com. 2023.

[42] Terraform. *Terraform*. https://www.terraform.io. 2023.

[43] M. N. Vi nals. "Designing Interaction Framework for Multi-Admin Edge Infrastructures." In: ().

[44] A. Vulaj. *Gorilla Mux archived*. https://www.oakestra.io. 2023.

[45] Z. Yang, Z. Wu, M. Luo, W.-L. Chiang, R. Bhardwaj, W. Kwon, S. Zhuang, F. S. Luan, G. Mittal, S. Shenker, and I. Stoica. "SkyPilot: An Intercloud Broker for Sky Computing." In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 437–455. ISBN: 978-1-939133-33-5.