FH JOANNEUM - University of Applied Sciences

Multi Cloud Application Deployment and the Development of the Cloud Infrastructure Landscape

Master thesis submitted at the Master Degree Programme IT-Architecture

for the degree „Diplomingenieur für technisch-wissenschaftliche Berufe“

„Diplomingenieurin für technisch-wissenschaftliche Berufe“

Author:

Tom Kleinhapl, Bsc

Supervisor:

Dipl.-Ing. Georg Mittenecker

Graz, 2022

Obligatory signed declaration

I hereby confirm and declare that the present Bachelor's thesis/Master's thesis was composed by myself without any help from others and that the work contained herein is my own and that I have only used the specified sources and aids. The uploaded version is identical to any printed version submitted.

I also confirm that I have prepared this thesis in compliance with the principles of the FH JOANNEUM Guideline for Good Scientific Practice and Prevention of Research Misconduct.

I declare in particular that I have marked all content taken verbatim or in substance from third party works or my own works according to the rules of good scientific practice and that I have included clear references to all sources.

The present original thesis has not been submitted to another university in Austria or abroad for the award of an academic degree in this form.

I understand that the provision of incorrect information in this signed declaration may have legal consequences.

Graz, September 14, 2022

Tom Kleinhapl

Contents

List of Figures

List of Abbreviations

ACU	Azure Compute Unit
AD	Active Directory
API	Application Programming Interface
ARM	Azure Resource Manager
AWS	Amazon Web Services
CAGR	Compound Annual Growth Rate
CAMP	Cloud Application Management for Platforms
CD	Continuous Deployment
CDK	Cloud Development Kit
CDMI	Cloud Data Management Interface
CLI	Command-Line Interface
CI	Continuous Integration
CRUD	Create, Read, Update, Delete
CSB	Cloud Service Broker
DB	Database
DRY	Do not repeat yourself
DSL	Domain Specific Language
ECS	Elastic Computer Service
GUI	Graphical User Interface
HCL	HashiCorp Configuration Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IaaS	Infrastructure as a Service
IaC	Infrastructure as Code
IDE	Integrated Development Environment
iPaaS	Integration Platform as a Service
JSON	JavaScript Object Notation
MSAL	Microsoft Authentication Library
OASIS	Organization for the Advancement of Structured Information Standards
OCI	Open Container Registry
PaaS	Platform as a Service
RDS	Relational Database Service
RDBMS	Relational Database Management System
REST	Representational State Transfer
SOA	Service Oriented Architecture
SKU	Stock Keeping Unit
SLA	Service Level Agreement
SQL	Structured Query Language
TDD	Test Driven Development
TOSCA	Topology and Orchestration Specification for Cloud Applications
VM	Virtual Machine
VPC	Virtual Private Cloud

Abstract

This thesis examines the current multi cloud market, specifically regarding multi cloud application deployment and portability. The standardisation of cloud technologies and the combination of technology advancements made with Infrastructure as Code (IaC), microservices and multi cloud should make it possible for end users to leverage advantages of multi cloud within a single application. The central research question this thesis aims to answer is: Can single applications be hosted on a multi cloud architecture using a high degree of automation (deployment and configuration) in an economical and feasible way in the current cloud infrastructure landscape?

To achieve this, a practical example is implemented and used to demonstrate the current effort, problems and difficulties encountered when developing a flexible deployment model using multiple cloud providers. The practical implementation revolves around a microservice application, that was conceived specifically for the Azure cloud, which is converted into a multi cloud capable architecture for Azure and Amazon Web Services (AWS). Using the IaC tool Terraform, multiple deployment plans are constructed, each using a different configuration. These deployment plans can be deployed interchangeably with minimal to no changes to the application code.

As such, the research question can be affirmed. However, the effort to design and implement multi cloud capable applications is still very high, in part due to a lack of standard adoption in the cloud. Not only the deployment of services to the cloud lacks standardisation, also storage interfaces are mostly proprietary in the cloud. This extends the challenge of creating multi cloud applications from the deployment plans to the application code, which needs to be able to interface with multiple standards. Still, using third party abstraction tools, building provider-agnostic applications is possible. Whether such a setup is needed and beneficial can only be evaluated on a case to case basis.

Portability between providers out of the box is still a long way away and unlikely to be available in the near future. This thesis conceives three main paths through which infrastructure portability in the cloud can be achieved. These are custom in-house solutions, portability through cloud brokers and finally the enforcement of standardised interfaces through policy. However, given the current cloud infrastructure market, which is dominated by a few big players, it is difficult to envisage an organic development towards a more open or broker driven market.

Kurzfassung

Diese Arbeit untersucht den aktuellen Multi-Cloud-Markt, insbesondere im Hinblick auf die Bereitstellung und Portabilität von Multi-Cloud-Anwendungen. Die Standardisierung von Cloud-Technologien und die Kombination von technologischen Fortschritten, die mit IaC, Microservices und Multi-Cloud erzielt wurden, sollten es Endbenutzern ermöglichen, die Vorteile von Multi-Cloud innerhalb einer einzigen Anwendung nutzen zu können. Die zentrale Forschungsfrage, die diese Arbeit beantworten soll, lautet: Können einzelne Anwendungen auf einer Multi-Cloud-Architektur, mit einem hohen Grad an Automatisierung (Deployment und Konfiguration), in der aktuellen Cloud-Infrastrukturlandschaft wirtschaftlich sinnvoll und praktikabel gehostet werden?

Zur Beantwortung der zentralen Forschungsfrage wird ein praktisches Beispiel implementiert und verwendet, um die aktuellen Bemühungen, Probleme und Schwierigkeiten zu demonstrieren, die bei der Entwicklung eines flexiblen Applikationsbereitstellungsmodells mit mehreren Cloud-Anbietern auftreten. Die praktische Umsetzung dreht sich um eine speziell für die Azure-Cloud konzipierte Microservice-Anwendung, die in eine Multi-Cloud-fähige Architektur für Azure und AWS umgewandelt wird. Mit dem IaC-Tool Terraform werden mehrere Bereitstellungspläne erstellt, die jeweils eine andere Konfiguration der Hosting-Architektur darstellen. Diese Bereitstellungspläne können mit minimalen bis gar keinen Änderungen am Anwendungscode austauschbar ausgeführt werden.

Insofern kann die Forschungsfrage bejaht werden. Der Aufwand für Design und Implementierung von Multi-Cloud-fähigen Anwendungen ist jedoch immer noch sehr hoch, was zum Teil auf die fehlende Standardakzeptanz in der Cloud zurückzuführen ist. Nicht nur die Bereitstellung von Diensten in der Cloud ist nicht standardisiert, auch Speicherschnittstellen sind in der Cloud meist proprietär. Dadurch erweitert sich die Herausforderung der Erstellung von Multi-Cloud-Anwendungen von den Bereitstellungsplänen auf den Anwendungscode, der in der Lage sein muss, mit mehreren Interfacestandards zu kommunizieren. Durch die Verwendung von Abstraktionstools von Drittanbietern ist es jedoch möglich, anbieterunabhängige Anwendungen zu erstellen. Ob eine solche Einrichtung notwendig und sinnvoll ist, kann nur im Einzelfall beurteilt werden.

Von einer Out-of-the-Box-Portierung zwischen Anbietern ist man heute noch weit entfernt und ist in naher Zukunft kaum denkbar. Diese Arbeit konzipiert drei Hauptpfade, durch welche Infrastrukturportabilität in der Cloud erreicht werden kann. Dies sind kundenspezifische Inhouse-Lösungen, Portabilität durch Cloud-Broker und schließlich die Durchsetzung standardisierter Schnittstellen durch Richtlinien. Angesichts des aktuellen Cloud-Infrastrukturmarktes, der von einigen wenigen großen Anbietern dominiert wird, ist es jedoch schwierig, sich eine organische Entwicklung hin zu einem offeneren oder von Cloud-Brokern getriebenen Markt vorzustellen.

1 Introduction

With the current cloud developments towards a progressively more service oriented business model, in theory it should not matter which company actually fulfils this service. The increasing standardisation of technologies should ideally make it possible to substitute one provider for another without much need to reconfigure affected applications or services. This would allow consumers to leverage not only availability options from multiple providers, to create a more resilient infrastructure, but also enables users to benefit from price differences in the market and build dynamic infrastructures that are more economical than ones bound to a single provider. For instance, a number of deployment plans could be devised for an architecture and dynamically deployed, during scheduled downtimes, based on a periodical evaluation of market prices of individual services.

The emergence of microservice oriented application design theoretically makes it possible to make use of these advantages within a single applications' infrastructure. The goal of a microservice infrastructure is to break down a single application into smaller services, that each fulfil a very particular goal. The idea is that each of these services should be easily exchangeable and the failure of one service should not bring the whole application down. Therefore, coupling between these services is very loose. This should make it possible to distribute these services to different providers and create deployment plans that dynamically deploy and remove parts of the infrastructure based on predefined conditions.

Solutions and standards exist that aim to make seamless multi cloud hosting possible and give enterprises the tools to build cloud infrastructures spanning multiple providers. While the use of multiple cloud providers by single enterprises is very common by now, they are often used for different use cases and practically interoperate very little, if at all. The aim of this thesis is to investigate, whether applications running on a multi cloud infrastructure are uncommon due to technical limitations, or whether such implementations are feasible and likely to shape the future of cloud hosting.

1.1 Problem Statement

When deploying an application to the cloud, currently the most common approach is to choose one cloud provider and then design the application around services available from this single provider. With technologies and trends like microservice architectures, the possibility to distribute even a single application amongst multiple cloud providers becomes a possibility, at least in theory. Most cloud providers still try to avoid the possibility to easily switch from their service to a competing one, and keep customers locked into their platforms by using proprietary technologies. The use of open standards across providers and the emergence of tools that can interface with multiple providers aim to change this and might usher in a new cloud era driven primarily by brokers and not by service providers themselves. [13]

1.2 Objectives

The goal of the thesis is to evaluate the current situation regarding multi cloud deployment of resources and explore possible future developments of the cloud infrastructure landscape. A practical example will aim to demonstrate the current effort required to achieve a flexible deployment plan using multiple cloud providers with a high degree of automation. The goal will be to deploy a microservice oriented application using two cloud providers interchangeably. Each service required for the application should be substitutable by an equivalent service from a different provider. The goal is not to develop an application from the ground up with a multi cloud architecture in mind, but rather to take an existing application originally developed to run on a single cloud provider. This will evaluate to what degree a separation of services across providers is feasible (e.g. just frontend and backend, or individual services) and to what degree the deployment and service configuration can be automated.

Furthermore, the thesis aims to assert whether an evolution into a broker driven market place for cloud resources is a realistic prospect for the near future. Insights on current limitations gained from the practical implementation of a cross provider infrastructure will be combined with theoretical research on trends and standardisation efforts to gauge whether this is a realistic presumption for the future of the cloud. Simultaneously, the current competition in the market will be analysed and a possible change to the competition in a broker market evaluated.

1.3 Research Question

The central research question for this thesis is:

- *Can a single application be hosted on a multicloud architecture using a high degree of automation (deployment and configuration) in an economical and feasible way in the current cloud infrastructure landscape?*

Additionally, the thesis aims to answer the following questions:

- *What current development efforts are there in the standardisation of cloud technologies?*
- *How will cloud provider markets and competition overall evolve in the future? Will the trend of service oriented cloud computing evolve into a broker oriented market?*

1.4 Relevance

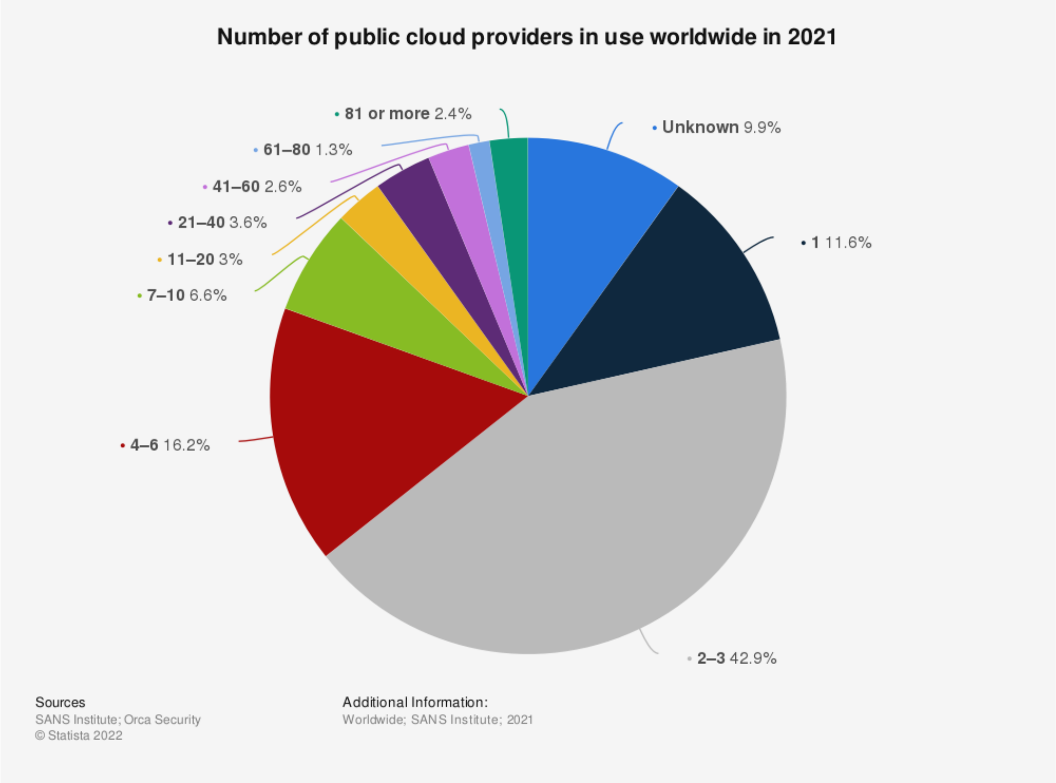
While there are papers that develop approaches to the deployment of multi cloud applications, the focus is often on developing a framework for designing such an application from the ground up. This thesis takes a different approach, in taking an existing application designed specifically for one provider and evaluating the effort and difficulties when changing this single cloud environment to incorporate more cloud providers. The scenario, where an existing application or service is moved from one cloud provider to another is not uncommon in businesses around the world, therefore the results and information generated from the paper will be of interest in the in the industry. Furthermore, the practical implementation is completely open source and can serve as guidance for other implementations in a practical environment.

2 Multi Cloud

The following chapter serves as an introduction to the subject of multi cloud. It provides some use cases for multi cloud and in the final subchapter details how the subject matter relates to the practical implementation in chapter 5. Insights into current solutions and the current state of multi cloud management are covered in chapter 6.2.

2.1 Definition and Use Cases

The term multi cloud generally refers to the use of multiple public cloud providers in a single, heterogeneous cloud architecture. This is in contrast to a hybrid cloud scenario, which describes the combination of a public cloud with an on-premise architecture, a private cloud or both. [14] It is not uncommon for a company to end up in some sort of multi cloud scenario. As can be seen in the statistic depicted in figure 2.1, only about 12 % of companies worldwide make use of just a single provider for their cloud needs. Usually, different providers are used for different use cases, meaning that there is very little interaction and data transfer between providers. For example, many companies will use the Azure cloud for their office needs, but use a different provider for hosting an application. The real difficulty in managing multiple clouds arises when data needs to be transferred between two providers.



Sources
SANS Institute; Orca Security
© Statista 2022

Additional Information:
Worldwide; SANS Institute; 2021

Figure 2.1: Number of public cloud providers in use worldwide in 2021 [1]

Use cases for hybrid cloud and multi cloud also often differ. A common reason for a hybrid cloud setup is cloud bursting, which can be solved by scaling in the public cloud. Common use cases and benefits of multi cloud environments are:

- **Optimise costs or improve quality of services**: One of the biggest benefits to using multiple cloud providers is competition amongst them. This allows the consumer to better optimise their infrastructure service quality or costs. Better quality can be achieved either by choosing the provider with the best redundancy options and best Service Level Agreement (SLA) or by combining services from multiple providers. Cost savings can be achieved by opting for the cheapest options from multiple providers.
- **React to changes of the offers by the providers**: If an architecture is drawn-up using a single cloud provider, that provider has a lot of room for manoeuvre in terms of service pricing. A multi cloud environment can offer end users the necessary flexibility to be able to react to changes made to services in use by instead switching to a similar offer from another provider. This might go beyond just price and be for instance the deprecation of a framework.
- **Follow the constraints, like new locations or laws**: Some regions have special requirements imposed on the data that is processed by applications in the cloud. One such example is the EU, which through the GDPR mandates that personal data must be stored within the European Economic Area. Therefore, using an additional cloud provider that offers hosting in that region might be necessary. [15]
- **Improve availability**: Not all cloud providers are able to provide hosting locations all over the world. Especially if using providers outside the big three (AWS, Azure and Google Cloud) a multi cloud solution might be necessary to make services available in new locations. Or a company might want to decrease downtime, without paying for a better SLA, by combining hosting options of multiple providers.
- **Backups and disaster recovery**: Once again, this will be especially true for smaller providers, which might not have enough data centres to offer satisfactory disaster recovery through redundancy zones. Furthermore, backups to another provider's storage solution might be a cheaper alternative to redundancy options from the same provider, albeit being more hands-on.
- **Make use of exclusive offers and technologies**: Cloud providers will try and differentiate themselves by offering new exclusive features occasionally. Often it is only a question of time till other providers offer the same or a very similar service, however, if one wants to use the newest technologies as they are released, a multi cloud setup is one way to do so, without the need to shift all services to a new provider.

[16]

2.2 Difficulties and Obstacles

Proprietary Technologies and a Lack of Standardisation

The use of proprietary technology makes creating and managing architectures spanning multiple clouds intricate and time-consuming. For example, looking at services as simple as block storage, both Azure and AWS have two different proprietary implementations, which out of the box offer no compatibility. While both Microsoft and Amazon promote tools, which allow for data transfer between the proprietary standards to occur, the procedure is more difficult and has more possible points of failure than transfers within the same provider. While there are standards for uniform access to most cloud technologies, these often lack support. This is also the case for the Cloud Data Management Interface (CDMI), an open standard that describes a RESTful Hypertext Transfer Protocol (HTTP) interface for accessing and managing cloud storage. Despite the publishing of the first version in April 2010, there are to this date only very few native implementations from cloud providers and only a few third party compatibility implementations for other providers. Cloud providers themselves clearly do not view adhering to open standards as a priority or even consider it a danger to the market advantaged gained from vendor lock-in. [17], [16]

To overcome these shortcomings from providers, there are third party libraries, that implement compatibility to certain standards and can make it possible to communicate with multiple providers in a unified way. These third party implementations usually abstract provider Application Programming Interface (API)s to create a new API. The major drawback of these abstracted implementations have, compared to native provider API, is that either they only offer the common denominator between supported providers or they do not allow true unified access. Terraform for instance is a case of the latter. It offers support for many providers and is able to offer access to platform specific features through its abstraction, but configurations for every provider differ slightly, making it not truly uniform. The result is that identical services from different providers have different names and properties. An example for a unified API would be Apache Libcloud, a Python library that abstracts services from over 50 providers and enables access to their services through their abstracted API. [18]

Latency

Having data travel within the same data centre will ensure that there is very low latency between two services. If the data needs to move between two data centres of the same provider, the provider will have made sure latency is kept as low as possible. However, if data travels between data centres from different cloud providers, there is no guarantee for low latency. For latency critical tasks, such as user interactive processes, this can be a notable downside. It is therefore important, especially when developing for environments involving multiple clouds, to design applications in a way to make latency a non issue for user interactive tasks.

Monitoring and Cost Management

Cost management also becomes significantly more difficult when multiple providers are involved. In a single provider set-up, one can simply use the cost analysis and forecast from that provider, which usually offers fine-grained cost monitoring. Having to monitor costs for multiple providers at a time, using different cost management interfaces, can become a tedious task prone to little mistakes that might cost a lot of money. There are solutions that allow cost management through a unified interface, but these often do not offer the same level of information and control that native provider solutions offer. Similarly, most cloud providers offer out of the box monitoring solutions for their services. These monitoring solutions have the main benefit of requiring little to no effort to set up. Although, unlike with billing services, there are plenty of monitoring solutions that offer support for multiple clouds. Still, they represent another tool a company will need to familiarise itself with.

Auto-Scaling

Auto-scaling features usually work using the integrated performance monitoring capabilities of cloud providers. An auto-scaling solution that takes into account instances scattered around multiple providers, will need to be custom build using third party monitoring and management solutions.

2.3 Relevance for the Practical Implementation

The practical implementation represents a multi cloud scenario involving the Azure Cloud and AWS. Services of a microservice-oriented application are deployed to the two cloud providers in different configurations. It represents a very specific use case involving the orchestration of resource deployment as well as data migration. However, practical implementation does not address other multi cloud management challenges, such as the increasing complexity of monitoring and cost control. The main multi cloud difficulty that is focused on in the practical implementation, is the lack of standardisation and the thereof resulting differences in interaction with providers. Also, the transfer of data between providers is a central problem covered in the practical implementation, documented in chapter 5.

3 Infrastructure as Code

This chapter introduces the subject IaC. It lays out the basic working of IaC, lays out benefits of working with IaC and looks at the opportunities IaC offers for multi cloud. Finally, at the end of the chapter, the relevance for the practical implementation of the thesis is discussed.

3.1 Deploying Services to the Cloud

Deployment of infrastructure in the cloud can be a tedious task, especially if the requirements are more than a couple of Virtual Machine (VM)s or containers. The most basic, but also most beginner-friendly, approach is to deploy resources through a graphical user interface, usually in the form of a web application. This is great for starting out and getting familiar with the assortment of services offered by a provider, but will quickly start to become tedious and show some obvious drawbacks. Services need to be manually configured one by one using a walkthrough wizard, which was useful for the first couple of times, but once a user knows what their configuration should look like, quickly become a productivity barrier.

The next approach which will allow for much more productive workflows, albeit having a higher learning curve, is to interact with a cloud provider using the command-line. Once the required services and their performance configuration are known to the user, the Command-Line Interface (CLI) to cloud providers becomes a very powerful and time saving tool. Services can be deployed in their desired performance configuration using a single command. If the service requires additional configuration, this can usually be done with subsequent commands. Automation becomes possible through scripting, allowing for the deployment of entire infrastructures from a single file. In listing 3.2 is a simple example from the Microsoft documentation for the deployment of a database and the creation of a firewall rule, to show what such a deployment script might look like.

Listing 3.1: Deployment script using the Azure CLI to deploy a database and configure a firewall rule. [19]

```
# Create a single database and configure a firewall rule
# Variable block
let "randomIdentifier=$RANDOM*$RANDOM"
location="East US"
resourceGroup="msdocs-azuresql-rg-$randomIdentifier"
tag="create-and-configure-database"
server="msdocs-azuresql-server-$randomIdentifier"
database="msdocsazuresqldb$randomIdentifier"
login="azureuser"
password="Pa$$w0rD-$randomIdentifier"
# Specify appropriate IP address values for your environment
# to limit access to the SQL Database server
startIp=0.0.0.0
endIp=0.0.0.0

```

```
echo "Using resource group $resourceGroup with login: $login, password: $password..."
echo "Creating $resourceGroup in $location..."
az group create --name $resourceGroup --location "$location" --tags $tag
echo "Creating $server in $location..."
az sql server create --name $server --resource-group $resourceGroup --location "$location" --admin-user $login --admin-password $password
echo "Configuring firewall..."
az sql server firewall-rule create --resource-group $resourceGroup --server $server -n AllowYourIp --start-ip-address $startIp --end-ip-address $endIp
echo "Creating $database on $server..."
az sql db create --resource-group $resourceGroup --server $server --name $database --sample-name AdventureWorksLT --edition GeneralPurpose --family Gen5 --capacity 2 --zone-redundant true # zone redundancy is only supported on premium and business critical service tiers
```

IaC is another, more recently developed approach to deploying cloud services. It manages to address some of the shortcomings of deploying through CLI scripts and offers additional benefits by using practices from software development. Unlike CLI scripts, which imperatively describe the actions the cloud environment should execute, IaC frameworks allow for declarative descriptions of the desired state of a cloud environment. IaC started gaining popularity with the DevOps movement in the 2010s. The goal of DevOps, which is a combination of the terms development and operations, was to reduce the time for changes made to a software design to reach an operational level by converging software development and IT operation workflows. [20] IaC makes it possible to create consistent, repeatable routines for provisioning and changing cloud services and their configurations. This was instrumental in accommodating certain DevOps practices such as Test Driven Development (TDD), Continuous Integration (CI), and Continuous Deployment (CD). [21] The example in listing 3.2 shows the deployment of the deployment of a database using the popular open-source software tool Terraform. Terraform was developed by HashiCorp and uses a domain specific infrastructure language known as HashiCorp Configuration Language (HCL). Through its modular structure, Terraform supports a multitude of cloud providers.

Listing 3.2: Database deployment using terraform. [22]

```
provider "azurerm" {
  features {}
}

resource "azurerm_resource_group" "example" {
  name     = "example-resources"
  location = "West Europe"
}

resource "azurerm_mssql_server" "example" {
  name                         = "example-sqlserver"
  resource_group_name          = azurerm_resource_group.example.name
  location                     = azurerm_resource_group.example.location
  version                      = "12.0"
  administrator_login          = "4dm1n157r470r"
  administrator_login_password = "4-v3ry-53cr37-p455w0rd"
}
```

```

resource "azurerm_mssql_database" "test" {
  name           = "acctest-db-d"
  server_id      = azurerm_mssql_server.example.id
  collation      = "SQL_Latin1_General_CP1_CI_AS"
  license_type = "LicenseIncluded"
  max_size_gb  = 4
  read_scale    = true
  sku_name       = "BC_Gen5_2"
  zone_redundant = true

  extended_auditing_policy {
    storage_endpoint                        =
        azurerm_storage_account.example.primary_blob_endpoint
    storage_account_access_key       =
        azurerm_storage_account.example.primary_access_key
    storage_account_access_key_is_secondary = true
    retention_in_days                       = 6
  }
}
```

With IaC, deployment files are more than just deployment scripts, they can also serve as Documentation. As can be seen in the example in listing 3.2, these files are much easier to read than a conventional script, as seen in listing 3.1 and while they might not completely replace conventional documentation, especially for people with a non-technical background, they offer a valuable extension to it. For instance, if new employees are onboarded to work on an existing infrastructure, having exact records of what they will be maintaining including exact service configurations will be extremely useful.

Most IaC Languages are so-called Domain Specific Language (DSL)s, meaning they are conceived for a specific application, in this case declaring infrastructure. This is likely due to the fact that most cloud engineers and infrastructure admins are better versed in scripting than coding. With the emergence of the DevOps movement, this is starting to change. Today, it is not uncommon for engineers working primarily in infrastructure to also be well versed in general purpose coding languages. There have therefore also been movements that allow for general purpose languages, such as Java, Python, JavaScript etc. to be used for infrastructure deployment and to a certain extent management. Again, there are proprietary implementations, such as the AWS Cloud Development Kit (CDK) and provider independent solutions, such as Pulumi. These two implementations also offer a substantially different structure. The AWS CDK translates the code to an AWS CloudFormation markup file, which is then used for the deployment, while Pulumi interacts directly with an AWS API. Furthermore, Pulumi offers support for multiple cloud providers. [23]

3.2 Benefits of Infrastructure as Code

Automation

The first benefit associated with IaC is automation. The possibility to deploy a whole infrastructure through a single button press is a comfort that offers considerable productivity benefits in an organisation. It also means anyone can deploy infrastructure as needed. For instance, if a development team needs a new VM or needs to upgrade to a more powerful instance, theoretically there is no need for an infrastructure admin to provision it for them, if there are preconfigured deployments available to the development team. To be fair, this benefit is not exclusive to IaC and can also be achieved through scripting, it is, however, greatly enhanced by IaC tools. Looking back at the examples in the previous chapter, one can tell that deployments using CLI scripts can quickly become unclear and difficult to retrace. The difficult readability of these procedural scripts also makes it difficult to determine exactly where a script has failed.

Idempotency

In IaC the desired state the environment should achieve is described, if this state is not reached, either all changes, or at least the affected resource is rolled back and an error message detailing the encountered problems is thrown. Furthermore, due to their declarative nature, IaC tools allow for the creation of idempotent deployments. This means that no matter how often the deployment is executed, it will always lead to the same result. If one uses a CLI script deploying 5 VMs using randomly generated IDs, and it is executed twice, 10 VMs will be provisioned. On the other hand, when using an IaC deployment plan, the second execution will do nothing, since the desired state of 5 VMs is already reached.

Avoid Configuration Drift

IaC deployment scripts are consistent. Executing a deployment plan in resource group A will lead to the exact same result as executing the same deployment plan in resource group B. This is a very common scenario for testing changes made to an application in a test environment that is exactly the same as the productive environment. Not only is there no possibility for human error in recreating the productive environment, but if changes are made directly to the deployment plan and applied to both environments, there is no possibility for the configurations of both environments to drift apart over time.

Speed

Pressing a single button to execute a deployment plan is obviously much faster than a person clicking through deployment wizards in web interfaces. Changes to existing infrastructure now also only require editing a file. The initial creation of the deployment plan, will likely require more effort and time than using a Graphical User Interface (GUI), but that extra effort will definitely pay off in the future when making changes to the infrastructure. Once familiar enough with the documentation of an IaC tool, it can also be faster to create a new infrastructure using code snippets from the web or previous configurations and adjusting them to one's needs.

Documentation

As mentioned in the previous chapter, deployment plans can be an excellent way of documenting infrastructure, being especially useful for onboarding or a change of Infrastructure administrator. In addition to the plan itself being a good documentation, it can also be used to generate visual graph representations of the infrastructure. Figure 3.1 shows a visualized Terraform execution plan.

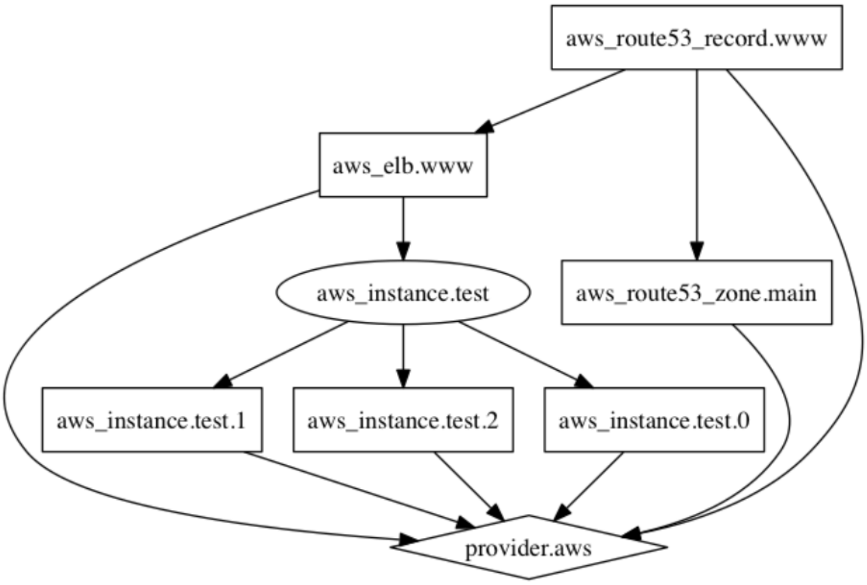


Figure 3.1: Graphical representation of a Terraform deployment plan.[2]

Version Control

Having deployment plans represented in code means they can be checked into version control tools, like Git. This not only facilitates collaboration, but also means the entire history of the file will be saved in the commit log. This is useful for debugging, by looking at the changes since the last working configuration. Additionally, if things go sideways, one can simply revert to a previous version.

Code Completion and Validation

This is one of the major benefits IaC has over conventional CLI scripting. When using a general purpose programming language, one can make use of the powerful code completion and suggestion functionalities from Integrated Development Environment (IDE)s. IDEs will also make sure that code is syntactically correct. For most popular DSLs there are packages that will add this functionality for most well known IDEs. In addition to any IDE capabilities, most IaC tools come with their own validation functionality, that will run before the plan is executed and will make sure the code is semantically correct. Many IaC tools also allow users to preview changes that will be made to the current state of the cloud environment, which can help avoid unwanted modifications due to an overlooked mistake or fallacy.

Reuse

Certain parts of an infrastructure will be the same for many people that have similar use cases. Therefore, it can be useful to check the web to find out if there are already configurations available online for a particular use case. Furthermore, infrastructure developed for a productive environment will often also be used to create a similar staging environment, with IaC this task becomes just another deployment plan execution. Certain IaC tools even allow you to define minor changes in environments in a Do not repeat yourself (DRY) way, by exposing changes as input variables.

Flexibility

With infrastructure provisioning becoming as simple as pressing a button, a new door to even greater use of the cloud's flexibility is opened. It becomes so simple and fast to deploy new infrastructure that infrastructure can not only be scaled back, but be completely removed if it is not needed for a few days. Since the time for automated deployments is predictable, it is possible to have entire infrastructures removed and redeployed according to a dynamic schedule.

Dependency Inference

Most IaC tools come with a dependency engine, that will ensure that resources are deployed in the right order. Usually, these dependencies can be inferred by the dependency engine as implicit dependencies. If dependency inference is not possible, for example with resources that are not declared in the same configuration, there is always the option to declare explicit dependencies. This allows more freedom in the structuring of the configuration files compared to a CLI scripts, which will need to declare resources in the exact order they need to be deployed.

[24]

3.3 Infrastructure as Code and Multicloud

Most cloud platforms offer a proprietary IaC implementation. The Azure cloud for instance implements the Azure Resource Manager (ARM) templating engine, which uses JavaScript Object Notation (JSON), as well as a more recent abstraction called Azure Bicep which aims to make ARM templates more readable. [25] Amazons AWS implements an IaC called AWS CloudFormation, which is also a proprietary solution offering no compatibility to other cloud providers. [26] Generally, the way these infrastructure coding languages work is by interfacing with the respective cloud platform through an API. If this API is accessible to third parties, they can also develop support for the cloud platform in a custom DSL, by interacting with these provider APIs. This is, for instance, how Terraform works. Unfortunately, these APIs are not standardised, APIs from different providers will be structured differently. This means that simply changing the destination of an API call will in most cases not work and not yield the same output. For tools like Terraform, that support multiple providers, this means that every provider needs to be maintained separately. [27] This is where Terraform benefits from having a large community. While HashiCorp does maintain and develop implementations for some providers themselves, there are many more that are maintained by their community, showing once again the benefits and power of open-source technologies.

While Terraform through the support of multiple providers betters the situation of vendor lock-in in the cloud platform landscape, it is still not a provider independent solution and lacks seamless portability of resources from one provider to another. It offers the possibility for infrastructures to be drawn up across providers using a single definition language and simplifies the move from one provider to another, since the fundamental semantics stay the same, but resource names and properties can differ substantially between providers. For instance, listing 3.3 depicts a Terraform snippet, that if executed will provision a MySQL database instance on AWS. In listing 3.4 one can see the declaration of a database server on the Azure cloud.

Listing 3.3: Deployment snippet for a MySQL database on AWS

```
resource "aws_db_instance" "example" {
  allocated_storage = 10
  engine            = "mysql"
  engine_version    = "5.7"
  instance_class    = "db.t3.micro"
  name              = "mydb"
  username          = "foo"
  password          = "foobarbaz"
  parameter_group_name = "default.mysql5.7"
  skip_final_snapshot = true
}
```

Listing 3.4: Deployment snippet for a MySQL database on Azure

```
resource "azurerm_mysql_server" "example" {
  name                = "mysql"
  location            = azurerm_resource_group.example.location
  resource_group_name = azurerm_resource_group.example.name

  administrator_login          = "foo"
  administrator_login_password = "foobarbaz"

  sku_name = "B_Gen5_2"
  storage_mb = 5120
  version   = "5.7"

  ssl_enforcement_enabled = true
}
```

While the examples shown in the listings seem very similar at first, due to the difference in implementing managed database services in AWS and Azure, the outcome of the two deployments is actually quite different. In listing 3.3, specifically deploys an instance of a MySQL database on AWS, this means that AWS will deploy a database server and instantiate a new database with the name mydb. In listing 3.4, as can be derived from the resource name, only the database server is deployed, there will be no instance called mydb on that server. To instantiate a database instance using Terraform, we need to declare another resource, which is depicted in listing 3.5.

Listing 3.5: Instantiating a MySQL database on Azure.

```
resource "azurerm_mssql_database" "test" {
  name         = "mydb"
  server_id    = azurerm_mssql_server.example.id
  collation    = "SQL_Latin1_General_CP1_CI_AS"
  license_type = "LicenseIncluded"
  max_size_gb  = 4
  read_scale   = true
  sku_name     = "S0"
  zone_redundant = true
}
```

It is differences like this that show that being familiar with Terraform does not mean that it is straight forward to port a configuration from one provider to another. There are, however, also some efforts to create standards that allow for a common method of interacting with cloud resources across providers, by abstracting the differences between them. These are analysed in chapter 6.

3.4 Relevance for the Practical Implementation

IaC is one of the central technologies that make the practical implementation of a microservice-oriented multi cloud application possible. Specifically, the IaC language Terraform was chosen for the implementation of the project. Terraform configurations will make up the central orchestration tool for the different configurations. The main reason for the choice of Terraform as a tool for the practical implementation, is that it supports a multitude of providers and that it offers a simple solution for transferring information between providers, such as endpoint DNS names and access keys. Terraform is also the most popular DSL on the market, as can be gathered from a Google Trends statistic comparing the search term Terraform to the terms CloudFormation (AWS native IaC DSL) and Azure Bicep (Azure native IaC DSL), depicted in figure 3.2.

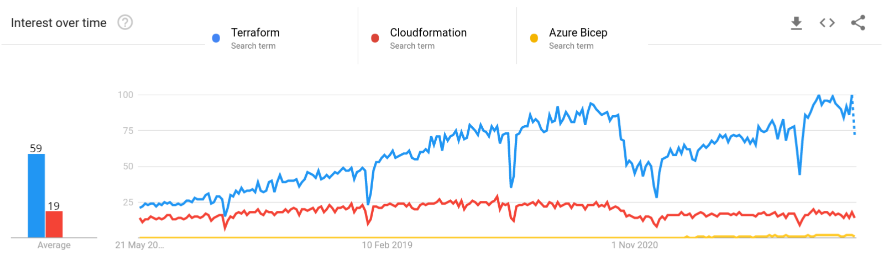


Figure 3.2: Popularity comparison between the search term Terraform, Cloudformation and Azure Bicep, from May 2017 to May 2022, worldwide. [3]

The graph in figure 3.2 shows the popularity scores of the respective search terms, which is calculated relative to the peak interest in one of the search terms. A value of 100 is the peak popularity for the term. A value of 50 would mean that the term is half as popular. A score of 0 means that there was not enough data for the term. The low popularity of Azure Bicep can be explained by the fact that it is relatively new, being initially released as an alpha version in August 2020. [28] The average score of 59 for the search term Terraform compared to the average score of 19 for CloudFormation shows that Terraform is much more present in Google searches. Based on this statistic, it is probable that Terraform is also more widely used.

4 Microservice Architectures

The following chapter covers the basics of microservice architectures, looks at some of the advantages they procure and elaborates their relevance for the practical implementation, which is covered in chapter 5. There is no universally accepted definition of microservices, here is a very general definition, that concisely summarises the essence of what makes up a microservice: Microservices are independently deployable modules. [29]

4.1 Microservices and Service Oriented Architectures

In general, microservice architectures can be seen as a modern derivative of Service Oriented Architecture (SOA) in software development. SOA was originally a response to the increasing complexity and size of enterprise applications. Conceptually, microservices and SOA are very similar and share many communalitie. The goal is to break open big, monolithic, siloed applications and distribute their functions into smaller entities that mesh and interact with one another. Interaction over a network also makes it possible to integrate services from different platforms in a heterogeneous environment. There is no universally accepted definition of what a microservice is or when a service becomes a microservice, but generally the difference comes down to scope and even looser coupling of services. SOA usually takes a whole enterprise architecture as its base scope and isolates services from that holistic perspective. Microservices, on the other hand, are scoped around a single application, as is the case in the practical part of this thesis. [4]

Furthermore, the SOA approach connects all of its services through a common enterprise service bus, and exerts centralised control over the individual services. With microservices, the coupling between services becomes much looser. Individual microservices are developed as independent isolated units, without a dedicated centralised communication hub. Instead, communication is often implemented using lightweight HTTP communication architectures such as Representational State Transfer (REST) APIs. This is why SOA is sometimes referred to as an orchestration of services and microservices as a choreography of services. An illustration showing the difference between these approaches is depicted in figure 4.1. Ultimately, this should give microservices the possibility to develop and evolve on an individual level without restrictions from the context of the application. [4]

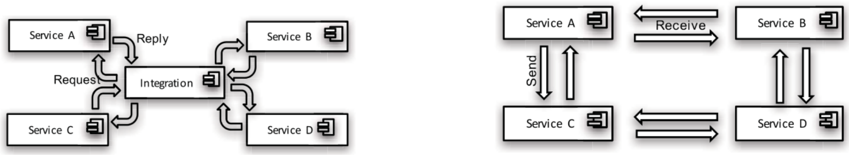


Figure 4.1: Service orchestration on the left and service choreography on the right.[4]

4.2 Advantages of Microservice Architectures Compared to Monolithic Applications

In contrast to a microservice architecture, all functions in a monolithic architecture are strongly interconnected and are executed and viewed as one service. The monolithic term does not necessarily refer to the software architecture of an application, but can just describe the architecture used to host that application. For instance, an application might be broken down into several services, but be hosted using a single VM. That VM can then be seen as a single monolithic service and has some notable drawbacks to a more distributed microservice architecture. That being said, there are still some reasons to build monolithic applications or run applications in a monolithic way, the primary advantage being cheaper development and in certain situations cheaper hosting.

Application Resilience

Services in a microservice architecture are built to be independent of one another. To be able to run in an isolated environment without relying on the existence of another services for operation. This has a big advantage for the application that is made up of these microservices, effectively eliminating single points of failure. If a single service fails, only that particular service will be unavailable which will inevitably have an impact on the application, where a certain functionality will be unavailable, but essentially the application itself should still be running and available and will only require restarting that particular service to restore full functionality.

Development

Due to the very loose coupling of services, which will be able to run in an isolated environment, it is very easy to divide responsibilities and development efforts amongst large development teams. A team or person working on a specific service can then also work mostly independently of other teams. Apart from decisions concerning the interfaces, that will enable the interaction with the services, all technology decisions within a service should ideally have no impact on other services. This also means that development teams will mostly have a very open choice of technologies for the application development and can choose to work with what they are familiar with.

Resource Scaling

Another advantage, due to the clean cut separation of services, is that scaling can be done at service level in contrast to scaling the whole application. For instance, services that serve logged-in users might require less resources than services that serve all traffic to the application. Microservices make it possible to simply scale these individual services based on the actual demand for them. This fine-grained resource control makes it possible to optimise the costs for applications and possibly gain an economic advantage. This also makes microservice applications optimal for cloud environments, where scaling of an individual service is mostly supported out of the box.

Replacement, Reuse and Revisions

The strong decoupling and independence of microservices, that should enable completely autonomous operation of every service, makes it very easy to move a service from one application to another. For instance, a user management service will likely be useful and very similar in many applications, thus existing services can greatly accelerate the development of new applications. Furthermore, the easy substitution of services means that if part of a microservice application is based on a framework or programming language one is not familiar with, it would be quite easy to replace that particular service, without having to rewrite the whole application, ideally even without having to alter other services. This also means that services can evolve and be updated without the need to release a whole new version of the application. The absence of dependencies between service modules, paves the way to rolling release models, where updates can be released much more frequently for individual services.

Economic Advantage

Whether a microservice architecture is ultimately more economical than a monolithic representation of the same application depends heavily on the application scenario. One of the biggest influencing factors are the size of the application and the characteristics of the workload. A microservice architecture allows the individual services of the application to be individually adapted to the requirements of the application, which can have a major impact, especially in the case of dynamic workloads that are difficult to predict. For example, an app service that receives accesses from users might scale higher than a database that is not needed every time the website is accessed. Achieving higher revenue by offering better SLAs plays another important role, but these could also be achieved, especially on a smaller scale, through cheaper backup and failover plans. Nonetheless, an application that has a microservice architecture could still be deployed in a monolithic fashion, taking away many of its advantages, but giving it an extra bit of flexibility, that might save costs in certain situations. [29]

4.3 Relevance for the Practical Implementation

The application that is deployed in the practical implementation, described in chapter 5, was originally developed with a microservice oriented approach in mind. Here the term microservice oriented is used because it does not strictly comply with some definitions of microservices. Also, since the scope of operation of the application is quite small, the services are not very granular. Still, services are separated without central control, and most services are capable of running autonomously, the one exception being the backend, which requires a connection to the database to start up. During the development of the application in a previous project, some benefits of the chosen architecture could already be felt during development. Especially the possibility to develop services independently was beneficial since it was developed in a team of four students.

The choice of architecture for the application was largely driven by the fact that it was developed as a cloud first application. The use of microservices meant that during development, it was possible to deploy individual services to the cloud and incorporate them with services running locally that were still being worked on. This made the transition into the cloud very easy and allowed us to pinpoint connectivity issues to particular services upon moving them to the cloud.

Finally, coming back to the initial definition of microservices, at the beginning of chapter 4, microservices are independently deployable and therefore make it possible to deploy different parts of a single application to different environments, as long as there is the possibility to transfer information between these environments. So, not only does this make it possible to deploy individual services to the cloud and have them interact with services still running on a local machine, it also makes it possible to deploy services belonging to the same application architecture to different cloud providers, making multi cloud application architectures possible. This is exactly what the practical implementation aims to accomplish, to try and make use of multi cloud benefits detailed in chapter 2 and asses the feasibility of such a setup.

5 Practical Implementation

This chapter covers the implementation of a multi cloud microservice oriented application, in the Microsoft Azure and AWS. The application was not designed from the ground up with a multi cloud architecture in mind but was designed as a cloud first application, specifically for the Azure cloud. It was initially developed as a proof of concept for a microservice oriented application in the cloud in a related project at the FH JOANNEUM. The application in question is an inventory platform, that uses object detection in computer vision to automatically classify objects imported to the platform. The functional development was not a primary goal of the previous project, nor is it the focal point of this thesis.

This practical implementation aims to demonstrate the current effort required to achieve a flexible deployment scenario using multiple cloud providers with a high degree of automation. Accordingly, multiple deployment plans were drawn up, which are described in chapter 5.4. The practical implementation was realised using the IaC tool Terraform, as well as additional tools needed for the transfer of data between providers. The results of this implementation consist of the documentation and implementation files of the practical example itself, as well as a discussion on the current experience of managing a multi cloud architecture, specifically using Terraform. The use of standardised orchestration engines to manage multi cloud architectures is evaluated in chapter 6.

5.1 State of Departure

The application was designed specifically around a few proprietary technologies, namely Azure functions and the Azure cosmos database. It was subdivided into the following services:

- Backend: REST API written in Python using the FastAPI framework.
- Frontend: Written using the JavaScript framework Vue.js, an Azure cosmos database.
- Database: Cosmos DB instance, for storing information about objects.
- Storage: Azure storage account with blob storage for saving images of objects.
- Computer Vision: Azure custom vision, to identify objects in images. The advantage of using a custom vision, is that is can be trained for any specific use case.

The architectural layout for the hosting and deployment of the application is depicted in figure 5.1. The subdivision of frontend and backend into smaller microservices was not put into effect, due to the quite small volume of functions implemented in the application. A subdivision would only make sense if the applications' functionality was further extended.

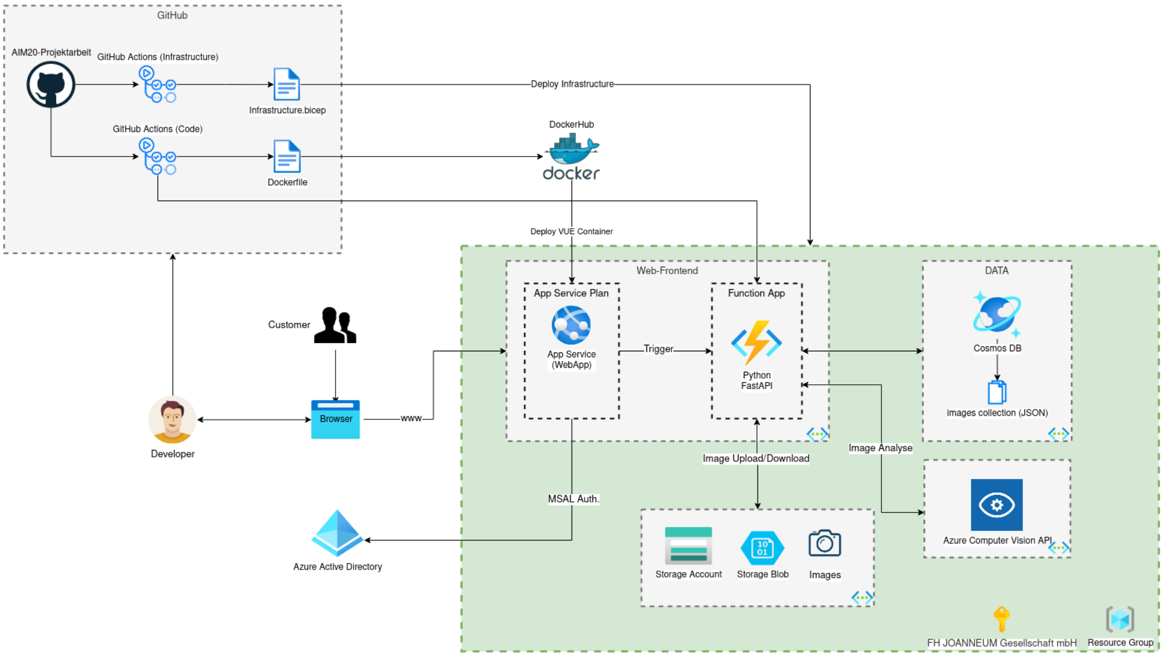
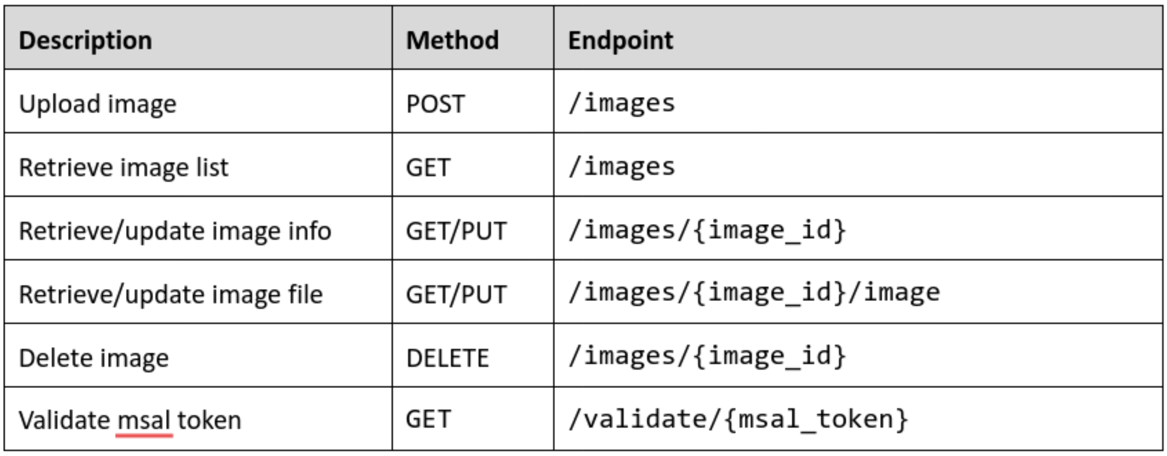


Figure 5.1: Architecture plan of the original application, to be converted for multi cloud hosting.

The whole architecture is written as IaC using the Microsoft proprietary definition language Azure Bicep. The code for the application, as well as the architecture, was hosted in a GitHub repository and deployed using GitHub Actions. The frontend was dockerised and hosted using an App Service Plan, while the backend was deployed serverless as a Function App. Identity management for the application is handled by Azure Active Directory (AD) using Microsoft Authentication Library (MSAL).

The Create, Read, Update, Delete (CRUD) functionality of the application is based around the following API endpoints:



Description	Method	Endpoint
Upload image	POST	`/images`
Retrieve image list	GET	`/images`
Retrieve/update image info	GET/PUT	`/images/{image_id}`
Retrieve/update image file	GET/PUT	`/images/{image_id}/image`
Delete image	DELETE	`/images/{image_id}`
Validate msal token	GET	`/validate/{msal_token}`

Figure 5.2: Application backend API endpoints.

5.2 Application Rework

Due to the initial conception of the application for the Azure cloud stack, there were a few proprietary products in use that made the application itself not easily portable between providers. Since the idea of dynamically moving services between providers should be a competitive advantage to the end user or a cost saving measure, having to rework source code every time a service is moved to another provider was not an option. Therefore, most proprietary technologies had to be removed from the application. Overall, there were three major changes to the original design of the application.

Backend Deployment

The Backend of the application was originally deployed as a serverless function app. While AWS has a comparable service to Function Apps with AWS Lambda, there is a lack of standardisation for the deployment of both services. Deploying a FastAPI backend to Azure functions was not very straight forward in the first place and required altering the backend source code. Therefore, the decision was made to move away from this particular serverless approach and instead containerise the backend and host it in a Docker container. This also has the advantage that the service could easily be hosted on local infrastructure, as long as the API is publicly accessible. There is still the option to deploy the container in a serverless manner on Azure and AWS, using Azure Container Apps or AWS Fargate respectively. These serverless deployment options are usually a little more costly compared to the use of self managed clusters, but allow for an even more hands-off experience.

Database

The initial project in which the application was developed required the use of Microsofts proprietary NoSQL database service Azure Cosmos DB. Once again, there is a comparable service from AWS, the DynamoDB, however, since they are not standardised, interacting with with both Database (DB)s would require altering the application source code. Therefore, the move to the open-source Structured Query Language (SQL) Relational Database Management System (RDBMS) MariaDB. This has the main advantage of allowing for a simple switch between database instances simply by changing the connection string.

Storage

Originally, the application interfaced only with Azure blob storage using the library “azure-storage-blob”, which is primarily maintained by Microsoft itself. [30] The application itself was therefore not yet fully portable. Since there is no shared storage standard between Azure and AWS a library able to interface with multiple standards had to be used. The Python libcloud library represents such a solution and was used for all storage operations. Libcloud, developed by the Apache Software Foundation, abstracts over 50 cloud APIs from different providers and provides access to them in a single unified library. The library is open-sourced and actively maintained by the open-source community. [18] All changes needed to the application code to switch between providers, is the definition of the correct provider. This is achieved using environment variables, which can be set using Terraform. Listing 5.1 shows how environment variables can be accessed in Python.

Listing 5.1: Accessing environment variables in Python.

```python
# Accessing environment variables in Python
ACCOUNT_NAME = os.getenv("STORAGE_ACCOUNT_NAME")
ACCESS_KEY = os.getenv("STORAGE_ACCESS_KEY")
CONTAINER_NAME = os.getenv("STORAGE_CONTAINER")
PROVIDER = os.getenv("STORAGE_PROVIDER")

# Storage account config azure
```

```
cls = get_driver(PROVIDER)
storage_driver = cls(key=ACCOUNT_NAME, secret=ACCESS_KEY)
storage_container = storage_driver.get_container(CONTAINER_NAME)
```

Custom Vision

The portability of the Azure custom vision service was omitted from the scope of the practical application. Since the application was previously dependent on a custom vision endpoint to run, this dependence had to be removed to make sure the application is operational with only the resources declared in the Terraform configuration files. Therefore, the application now checks for the presence of a custom vision endpoint URL in the environment variables and leave out the automatic classification in case the variable is not set. To run the application without a trained custom vision, the variable CUSTOM_VISION_URL in the Terraform configuration needs to be defined as an empty string.

5.3 Problems and Challenges

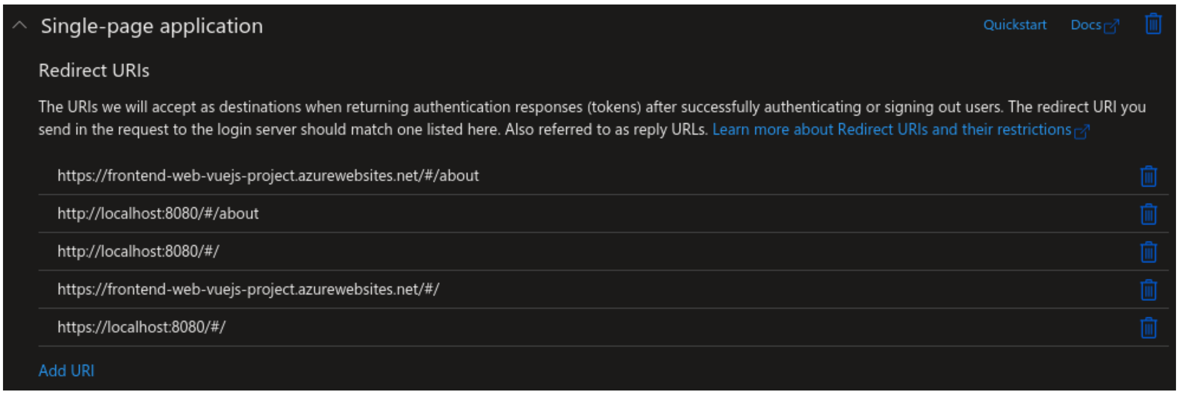
Database

Deploying a database which can be migrated from one provider to another limits options for managed databases. The goal for the first configuration was to use Amazons Relational Database Service (RDS) for a fully managed MariaDB instance. This would be the minimal effort configuration for a database, especially considering features like the creation from a backed up image. Since there is the possibility to automatically create a snapshot before the database is deleted, this would be a good solution to keep the state of the database when resources are destroyed. Unfortunately, these backups are AWS specific and can therefore not be imported in a managed MariaDB instance in the Azure cloud. This functionality had to be manually configured using Terraform.

Another problem surfaced, when changing the connection string from an AWS hosted instance to an Azure hosted instance using the SQLArchemy python library. The official SQLAlchemy documentation indicates to use the following connection strings to access database instances:
dialect+driver://username:password@host:port/database
This worked fine for when the database was hosted on AWS. When hosted on Azure, a @dbname needed to be added to the username. The @ needs to be URL encoded as %40, resulting in the following connection string:
dialect+driver://username%40hostname:password@host:port/database
Unfortunately, adding the @dbname to the user when connecting to an AWS hosted database does not work. Therefore, additional code had to be added to the database.py file to make these adjustments in the connection string based on the database endpoint name.

Identity Provider and HTTPS

The use of Azure AD as identity provider forces the use of Hypertext Transfer Protocol Secure (HTTPS), otherwise users will not be able to log into the application. This is because addresses which are allowed to redirect to Microsoft need to be listed under the app registration entry in Azure. These URLs are required to be HTTPS (apart from local host domains for testing purposes). Figure 5.1 shows the redirect URLs of an application registered with Azure AD.



Single-page application

Redirect URIs

The URIs we will accept as destinations when returning authentication responses (tokens) after successfully authenticating or signing out users. The redirect URI you send in the request to the login server should match one listed here. Also referred to as reply URLs. Learn more about Redirect URIs and their restrictions

https://frontend-web-vuejs-project.azurewebsites.net/#/about
http://localhost:8080/#/about
http://localhost:8080/#/
https://frontend-web-vuejs-project.azurewebsites.net/#/
https://localhost:8080/#/

Figure 5.3: Azure app registration URL configuration.

An additional problem is that, unless the application is registered under a static public domain, the domains that are dynamically assigned by AWS are based on the IP address of the underlying EC2 instance. This means that the new DNS name would need to be updated in the Azure app registration for every deployment. This is no the case with Azure, where the dynamic hostnames are based on the app service instance name and therefore stay consistent between deployments. Furthermore, Azure provides SSL certificates for their automatically assigned DNS names, this is not the case with AWS. The best solution for this problem would be to register the application under a public domain and request or import a certificate for that domain, so that it can be accessed using HTTPS. Since the AWS provided IP addresses are not consistent between deployments, one would additionally need to reserve a static public IP and associate it with the frontend service. It is to be noted that this is not a problem when replacing the identity provider with a solution that allows logins over HTTP.

Passing Environment Variables to vue.js Applications at Runtime

Connection strings to entities in the architecture are subject to change between deployments. It is therefore impractical to hardcode these connections in the application source code, since the code would need to be altered for every deployment. The way this problem was solved for the backend is to use environment variables which are passed to the application via Terraform and can be accessed by the Python application at runtime. In the first configuration the backend was hosted on Azure, where web apps get static DNS hostnames, so there was no need to change the backend connection string in the frontend application code. However, when moving the backend to AWS, the static DNS name is no longer guaranteed. Unfortunately, vue.js compiles all environment variables to static strings when compiling the application with vue-cli-service build, which happens when building the container image of our frontend application. This means that environment variables that will be passed to the application at runtime will be ignored.

There are two potential solutions to this problem. The first is to work with static IPs or DNS names. However, this again implies reserving a static IP with AWS, for which one will be charged whenever the IP is not in use. Another option is to use a workaround to force vue.js to update its environment variables. Two solutions to achieve this can be found in the following Stack Overflow post: https://stackoverflow.com/questions/53010064/pass-environment-variable-into-a-vue-app-at-runtime. Alternatively, both the frontend and backend could simply be hosted with a provider that provides static DNS names, like is the case wtih Azure. After all, this is one of the main advantages of using such a multi cloud set up. If a service from provider A becomes too expensive or is lacking a certain feature, it is quite easy to simply move it over to provider B. This is why the initially conceived configurations 1 and 2 do not actually provide a working solution out of the box, but using parts of both configurations as building blocks it was very easy to achieve a working solution (as is the case with configuration 3 and configuration 4), without the need to invest in a static IP reservation with AWS.

Direct Access to Static Content

The original application retrieved all images served by the application directly from the storage container. To do this the URL for the images is saved in the database entry that describes an image. The problem with this multi cloud set up is that this URL is subject to change. While this is not a big problem in itself and can be solved quite easily, it is one of those considerations that are easy to forget when reworking an application that was initially running in a single static configuration. To solve this, the download of images was simply also moved to an API endpoint in the backend service.

Maintaining Multiple Configurations

This is less of a problem and more of a challenge. Having a final toll of four configurations to maintain can be tedious and challenging. It becomes easy to lose track of which configuration is in what state and where new changes need to be applied. Especially, when working on the final two configurations, this became apparent. Often, changes made during the development of one configuration were not immediately replicated to the other configurations. This led to problems when switching back to older configurations and resulted in extra troubleshooting time. The best solution is to propagate any changes made to one configuration to all others immediately.

5.4 Deployment Plans and Configurations

The overall idea of the implantation was to be able to move individual services dynamically from one provider to another. The tool chosen to achieve this is Terraform, since it is one of the most well known IaC solutions on the market, that supports multiple providers. The goal was to draw up an initial configuration defining the complete application architecture using both providers and subsequently inverting that particular configuration, so that all services that were previously on Azure would be hosted on AWS and vice versa. This should serve as a solid basis for creating any number of architectural combinations by using sections of these configurations as building blocks and combining them in a new file. The use of both providers in the initial configuration will also exemplify how interactions between providers, like the transfer of connection strings, can be achieved in Terraform. The transfer of information between providers and the application should ensure application code does not need to be altered after service redistribution. The next step was to create scripts that make sure that data is not lost when moving between configurations. This includes the transfer of storage and database contents from one cloud provider to another.

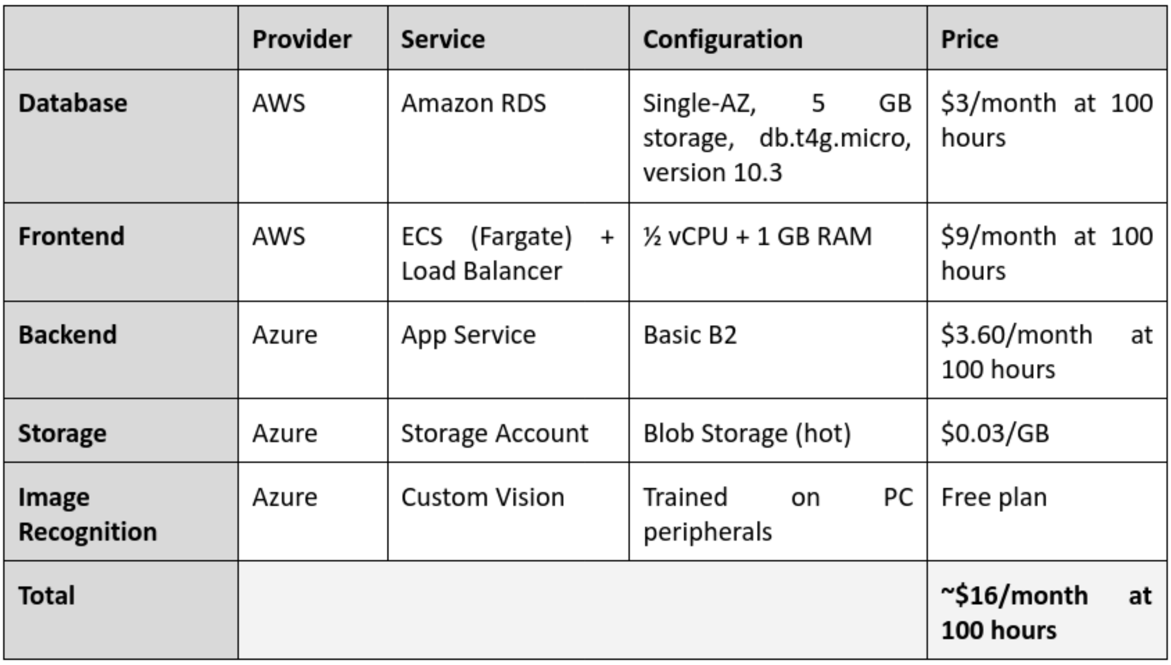
The source code for the application, as well as Terraform configurations and deployment instructions can be found in the following GitHub repository: https://github.com/tiemtom/-Masterarbeit/tree/master/code

All Terraform configurations use the same set of input variables for the configuration of values that need to be altered to deploy the configuration to a custom environment. These variables are set in the terraform.tfvars file. This file is automatically loaded by the Terraform engine when the configuration is applied. The values in that file can either be directly changed, or a custom variable file can be specified using:
terraform apply -var-file=”customfilename.tfvars”

Changing certain variables will require additional changes in the configuration files themselves, this can be gathered from the code documentation.

Configuration 1

The initial configuration was developed using the following paid services from Azure and AWS:



	Provider	Service	Configuration	Price
Database	AWS	Amazon RDS	Single-AZ, 5 GB storage, db.t4g.micro, version 10.3	$3/month at 100 hours
Frontend	AWS	ECS (Fargate) + Load Balancer	½ vCPU + 1 GB RAM	$9/month at 100 hours
Backend	Azure	App Service	Basic B2	$3.60/month at 100 hours
Storage	Azure	Storage Account	Blob Storage (hot)	$0.03/GB
Image Recognition	Azure	Custom Vision	Trained on PC peripherals	Free plan
Total				**~$16/month at 100 hours**

Figure 5.4: Service overview and pricing of configuration 1.

The database is a MariaDB instance using the Amazon RDS service in its cheapest configuration. The frontend was initially deployed using an EC2 cluster using an autoscaling group. The reason for this is that this configuration is a cheaper than the serverless Fargate solution. The EC2 solution can be found in the config_1_ec2 folder. The configuration was changed to Fargate since the frontend container will run well with a smaller resource definition (0.5 vCPUs and 1 GiB of RAM) than provided by a t3.medium EC2 instance (2 vCPUs and 4 GiB of RAM). At this time, t3.medium is the smallest EC2 instance supported for auto-provisioned EC2 Elastic Computer Service (ECS) clusters. When scaling up the frontend service, the EC2 configuration should be considerably cheaper.

To complement the Fargate cluster, there is an application load balancer that accepts incoming traffic from the internet and forwards it to the docker containers. While this is not strictly necessary, unless scaling the service is desired, it makes it much easier to retrieve the DNS name for the frontend with Terraform. The backend is hosted using an Azure App Service using the basic B2 hosting plan, which provides 100 Azure Compute Unit (ACU)s and 1.75 GB of RAM. ACUs are Azures standardised way of comparing computing power to their own services. 100 ACUs should roughly equate to 1 vCPU. [31]

Finally, there is a storage blob container, which is charged through the storage account and the image recognition, which was ported over from the previous project and runs in a free configuration to avoid costs. Especially training costs for the image recognition service can ramp up quickly, therefore it is not meaningful to recreate and retrain the service with every deployment and is left as a static instance. It is however possible to train the service through an API, making a fully automated deployment possible. Alternatively, to avoid training altogether, the Azure Computer Vision service could be used, which offers pretrained API endpoints for image analysis, this would however require altering the application code. The Architecture of configuration 1 is depicted in figure 5.5.

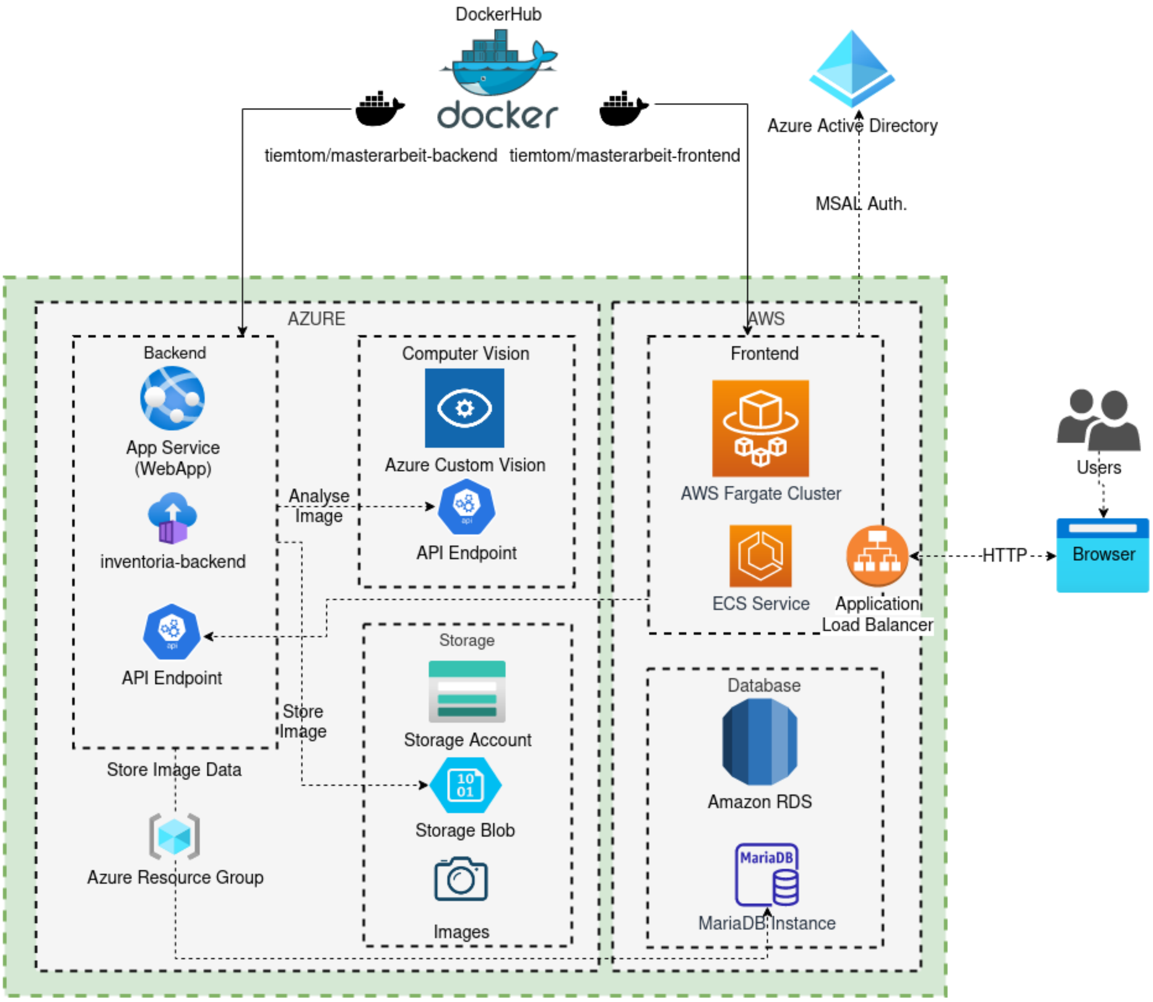


Figure 5.5: Architecture of configuration 1.

Figure 5.5 also shows the dataflow of a user accessing the Inventoria application. One can see a user communicating with the frontend service, hosted using as an ECS Service. The frontend page that is loaded then contains additional requests to the backend API, which handles most data transfers. The backend will upload images to the storage account and then call the custom vision API with the blob data. The response returned from that API, along with user entered information, is then committed to the MariaDB instance. A convenient way to relieve some traffic from the backend would be to directly get all image data from the storage container, however, since the URL to retrieve the image is stored in the database, when the application storage is migrated, all URL value would have to be altered.

Also, If the container that stores the image data is to be private, the retrieval of the image would have to be handled by the backend as well. There is an API endpoint in place that handles image downloads from the storage container.

Not depicted in the configuration is the network configuration of the application. Since the configuration distributes services amongst multiple providers, there is no way to isolate components from the public internet by putting them in a single subnet. While theoretically it would be possible to run all communications through the frontend and pair it to the backend services using a site-to-site VPN, this is not the security approach that was chosen for this application. Already in the preceding project, the idea was to keep APIs public, but harden them to make unauthorised access impossible. This can be achieved using the MSAL token validation function available in the backend. While, this has not been fully implemented, to make testing during the development of the cloud architecture easier, simply calling the function before every API request would make backend API calls impossible without login. Currently, the validation function is only used to secure frontend access. Therefore, on the Azure side, there is no need for any vnet configurations. On the AWS side, the implementation and visibility of the default network is different. Unless configured otherwise, the resources will be deployed in a default virtual private network Virtual Private Cloud (VPC), which can be associated with a security group that will restrict access to these resources. Therefore, if this network is already in use, it might not be a good idea to modify security groups associated with the default network. Hence, a new VPC is deployed to host all AWS resources needed for this application. When creating VPCs using Terraform, the creation of routing information, that is automatically handled by AWS when creating a VPC through their portal, needs to be defined as well. The configuration therefor also creates a gateway, routing table and the routes themselves. The network definition can be seen in listing 5.2.

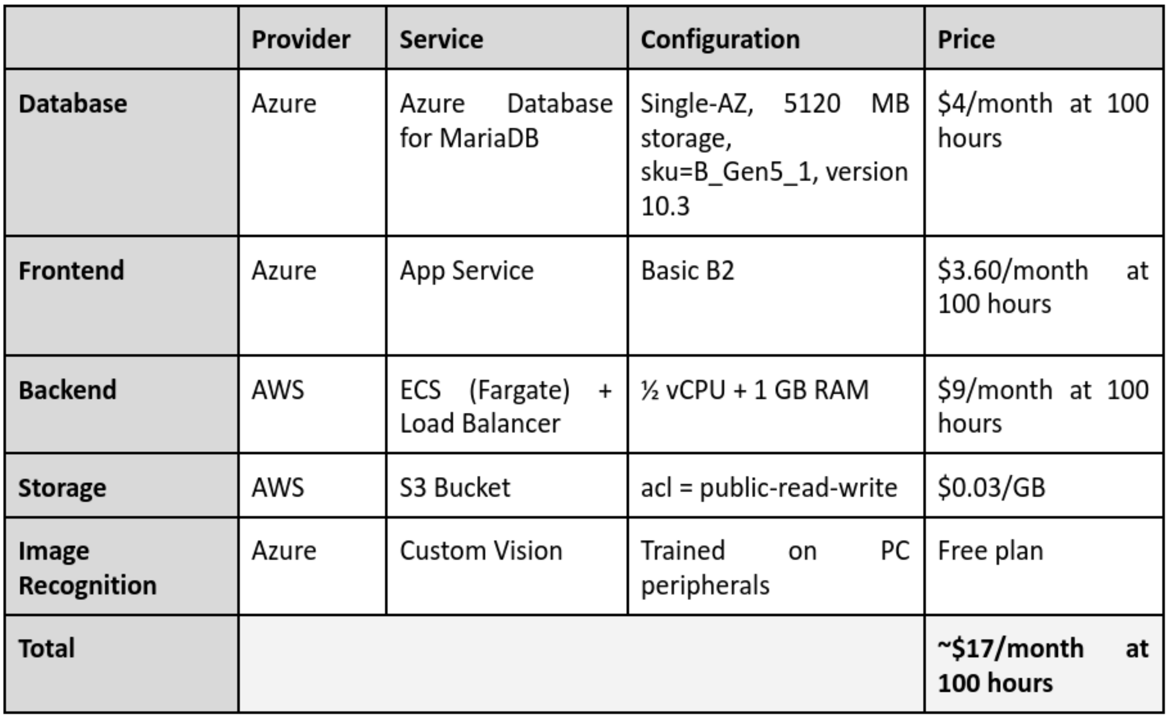
Listing 5.2: Network definition in config 1 for AWS.

```
# vpn defition
resource "aws_vpc" "main" {
 cidr_block = "10.0.0.0/16"
 enable_dns_hostnames = true
 enable_dns_support = true
}

# internet gateway
resource "aws_internet_gateway" "main" {
 vpc_id = aws_vpc.main.id
}

# subnet 1
resource "aws_subnet" "public" {
 vpc_id              = aws_vpc.main.id
 cidr_block = "10.0.0.0/24"
 map_public_ip_on_launch = true
 availability_zone = "eu-central-1a"
}

# subnet 2, additional subnet with a different availability zone is necessary for a db-subnet-group
resource "aws_subnet" "subnet2" {
 vpc_id              = aws_vpc.main.id
 cidr_block = "10.0.1.0/24"
 map_public_ip_on_launch = true
 availability_zone = "eu-central-1b"
}

# routing table for vpc
resource "aws_route_table" "public" {
 vpc_id = aws_vpc.main.id
}

# add route to internet gateway
resource "aws_route" "public" {
 route_table_id     = aws_route_table.public.id
 destination_cidr_block = "0.0.0.0/0"
 gateway_id         = aws_internet_gateway.main.id
}

# route to subnet1
resource "aws_route_table_association" "public" {
 subnet_id    = aws_subnet.public.id
 route_table_id = aws_route_table.public.id
}

# route to subnet2
resource "aws_route_table_association" "subnet2" {
 subnet_id    = aws_subnet.subnet2.id
 route_table_id = aws_route_table.public.id
}
```

Unfortunately, due to inconveniences with acquiring SSL certificates, to be associated with the frontend service, especially the additional monetary investments needed for static IP reservation, usage of the application in this configuration is severely limited. The exact causes behind this problem are detailed in chapter 5.3. It is possible to access the frontend, but the login will not work out of the box. Therefore, the backend API functions can only be tested directly, the easiest way to do this is to navigate to the /docs site on the backend endpoint, which serves an auto generated API documentation, which enable testing for the different endpoints. A fully functional version of the application is available in configuration 3 and configuration 4.

Configuration 2

The second configuration is an exact inversion of configuration 1, all services that were previously hosted on AWS are moved to Azure and vice versa. Configuration 2 uses the following services:



	Provider	Service	Configuration	Price
Database	Azure	Azure Database for MariaDB	Single-AZ, 5120 MB storage, sku=B_Gen5_1, version 10.3	$4/month at 100 hours
Frontend	Azure	App Service	Basic B2	$3.60/month at 100 hours
Backend	AWS	ECS (Fargate) + Load Balancer	½ vCPU + 1 GB RAM	$9/month at 100 hours
Storage	AWS	S3 Bucket	acl = public-read-write	$0.03/GB
Image Recognition	Azure	Custom Vision	Trained on PC peripherals	Free plan
Total				**~$17/month at 100 hours**

Figure 5.6: Service overview and pricing of configuration 2.

The dataflow is partly omitted, in figure 5.7 to improve readability, it is the same as in configuration 1.

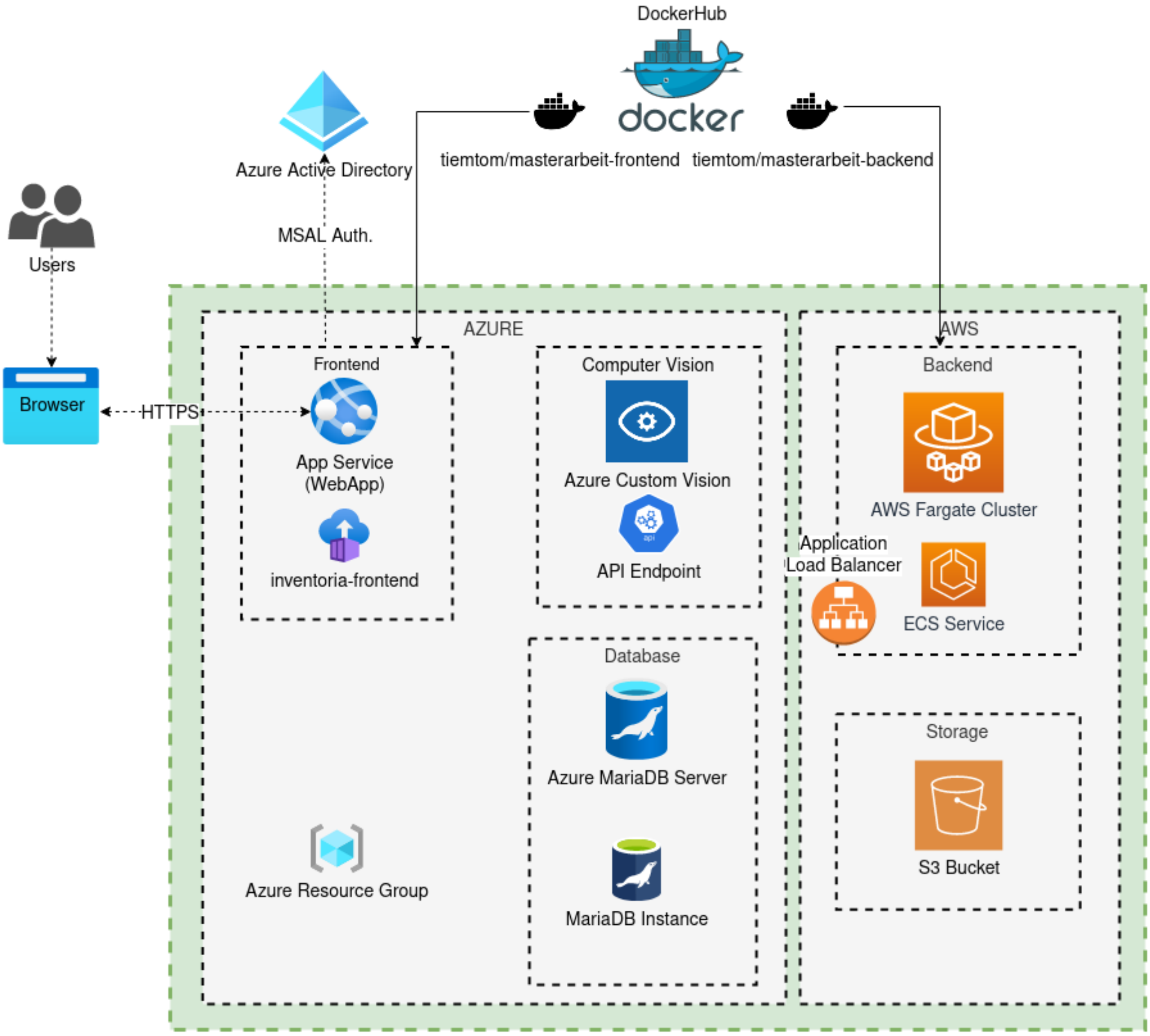
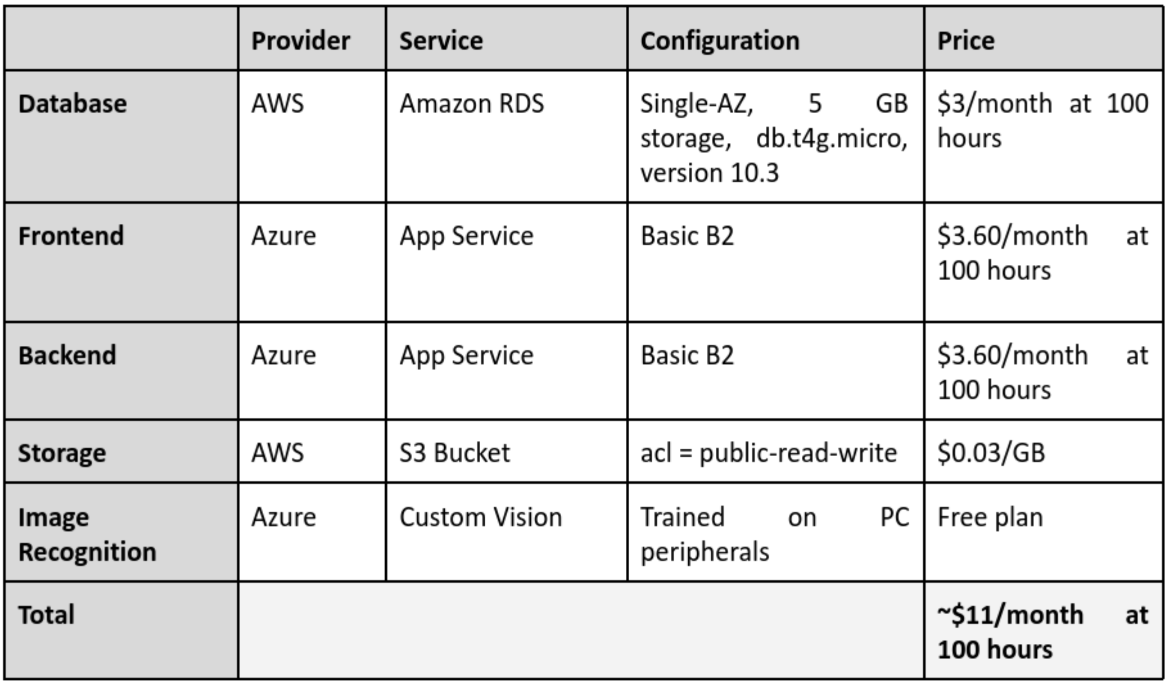


Figure 5.7: Architecture of configuration 2.

Configuration 3

The primary goal for this third configuration, was to have a deployment plan that works out of the box, without the need to purchase and configure a domain and SSL certificate, as is the case in configuration 1 and configuration 2. To achieve this, both the backend and frontend were moved to Azure, since their web app service has DNS names based on the name defined in the template, which stay consistent between deployments. Additionally, Azure provides certificates for their azurewebsites.net domain out of the box. This is primarily necessary due to the constraint that MSAL will only work over HTTPS, the details of this restriction are described in Chapter 5.3. It is by chance also the cheapest configuration, which is due to Azure offering cheaper hosting plans. Howerver, these hosting plans are not intended for productive use and do not contain the same functionality as the ones used in the setup configured for AWS. Most notably, there is no support for a load balancer in these cheap hosting plans.

The creation of the third configuration was very easy and fast compared to the previous configurations. Due to being able to reuse components from the previous configurations, the creation of this one was as simple as copy and pasting the right components and adjusting environment variables. In this particular configuration, it would be possible to run both the backend and frontend service from the same app services plan, since the plan only sets the operating system and the Stock Keeping Unit (SKU) defining the computing power and memory for the web apps that will be hosted on that plan. The SKU also defines the pricing for the services. So, running both front- and backend from the same plan can be more economical. However, this also means that the two web apps will share the resources in that app services plan and run on the same VMs. This takes away the ability to scale the services individually, defying one of the major advantages of a microservice oriented architecture.



	Provider	Service	Configuration	Price
Database	AWS	Amazon RDS	Single-AZ, 5 GB storage, db.t4g.micro, version 10.3	$3/month at 100 hours
Frontend	Azure	App Service	Basic B2	$3.60/month at 100 hours
Backend	Azure	App Service	Basic B2	$3.60/month at 100 hours
Storage	AWS	S3 Bucket	acl = public-read-write	$0.03/GB
Image Recognition	Azure	Custom Vision	Trained on PC peripherals	Free plan
Total				**~$11/month at 100 hours**

Figure 5.8: Service overview and pricing of configuration 3.

Architecture overview for configuration 3:

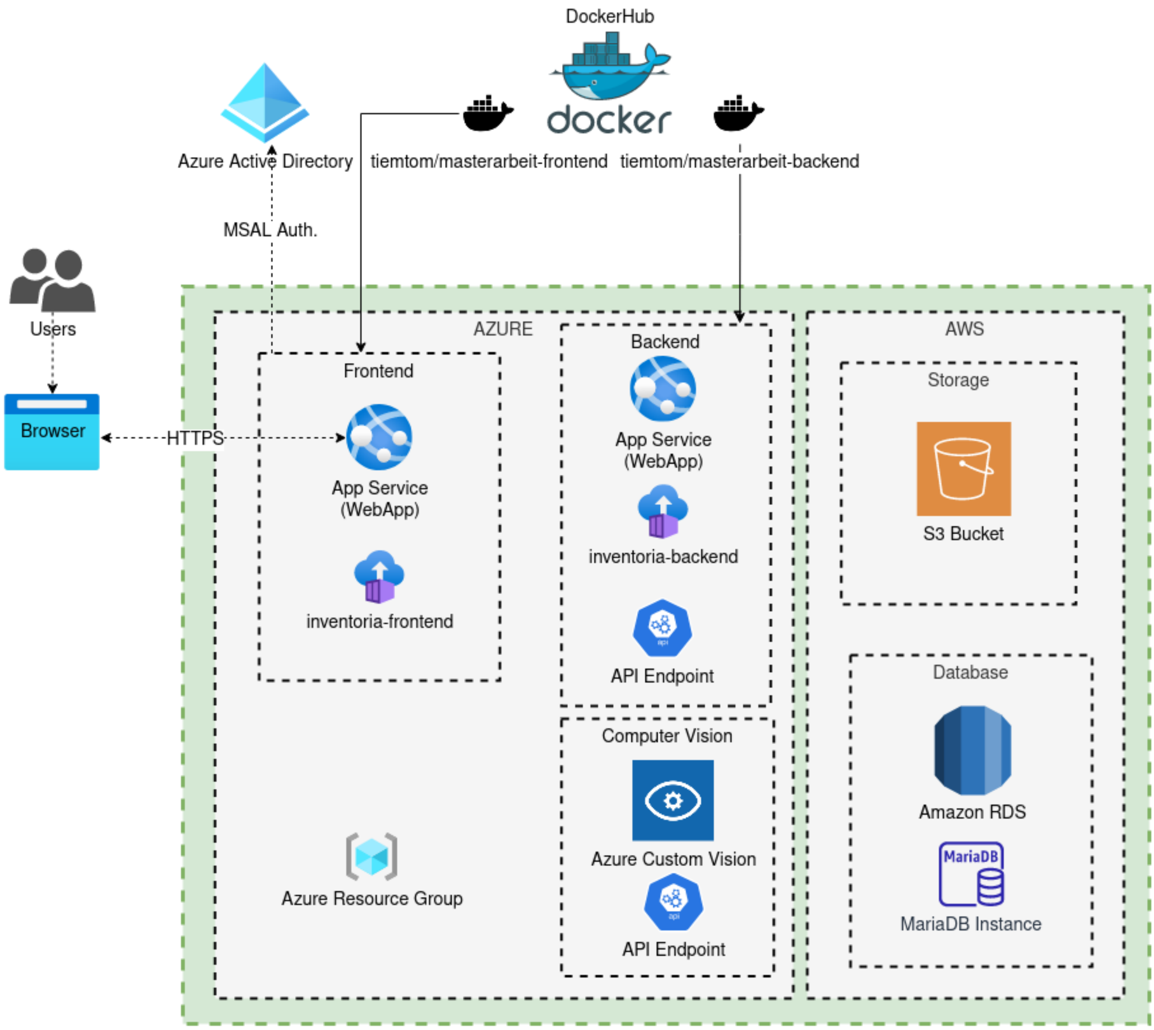
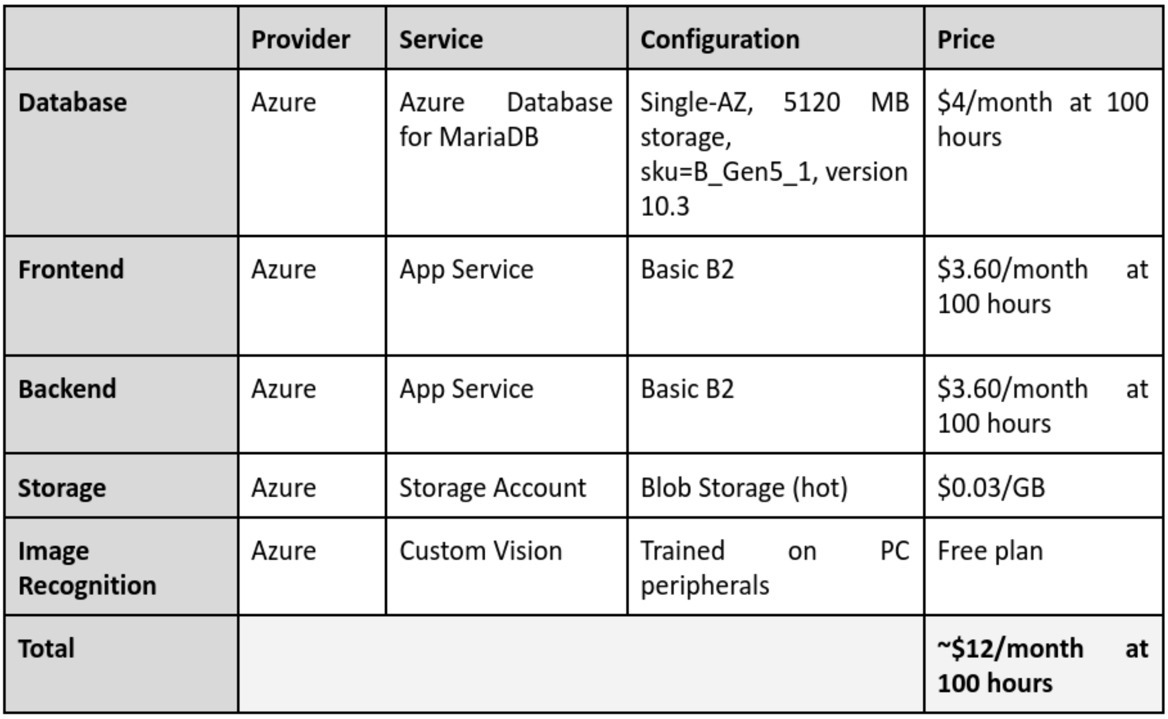


Figure 5.9: Architecture of configuration 3.

Configuration 4

The goal for configuration 4 was to demonstrate the working data migration, between AWS and Azure. This is essentially a single cloud setup using only Azure. This means that the change from configuration 3 to configuration 4 will not require any changes to the frontend code and will therefore work out of the box. But, since the database and blob storage is moved from AWS to Azure, data exports need to be used to transfer the data between providers. Information detailing the synchronisation of data between providers can be found in chapter 5.5.

Again, the creation of new configuration was now just a matter of copy and pasting the right components and not very time-consuming.



	Provider	Service	Configuration	Price
Database	Azure	Azure Database for MariaDB	Single-AZ, 5120 MB storage, sku=B_Gen5_1, version 10.3	$4/month at 100 hours
Frontend	Azure	App Service	Basic B2	$3.60/month at 100 hours
Backend	Azure	App Service	Basic B2	$3.60/month at 100 hours
Storage	Azure	Storage Account	Blob Storage (hot)	$0.03/GB
Image Recognition	Azure	Custom Vision	Trained on PC peripherals	Free plan
Total				**~$12/month at 100 hours**

Figure 5.10: Service overview and pricing of configuration 4.

Architecture overview for configuration 4:

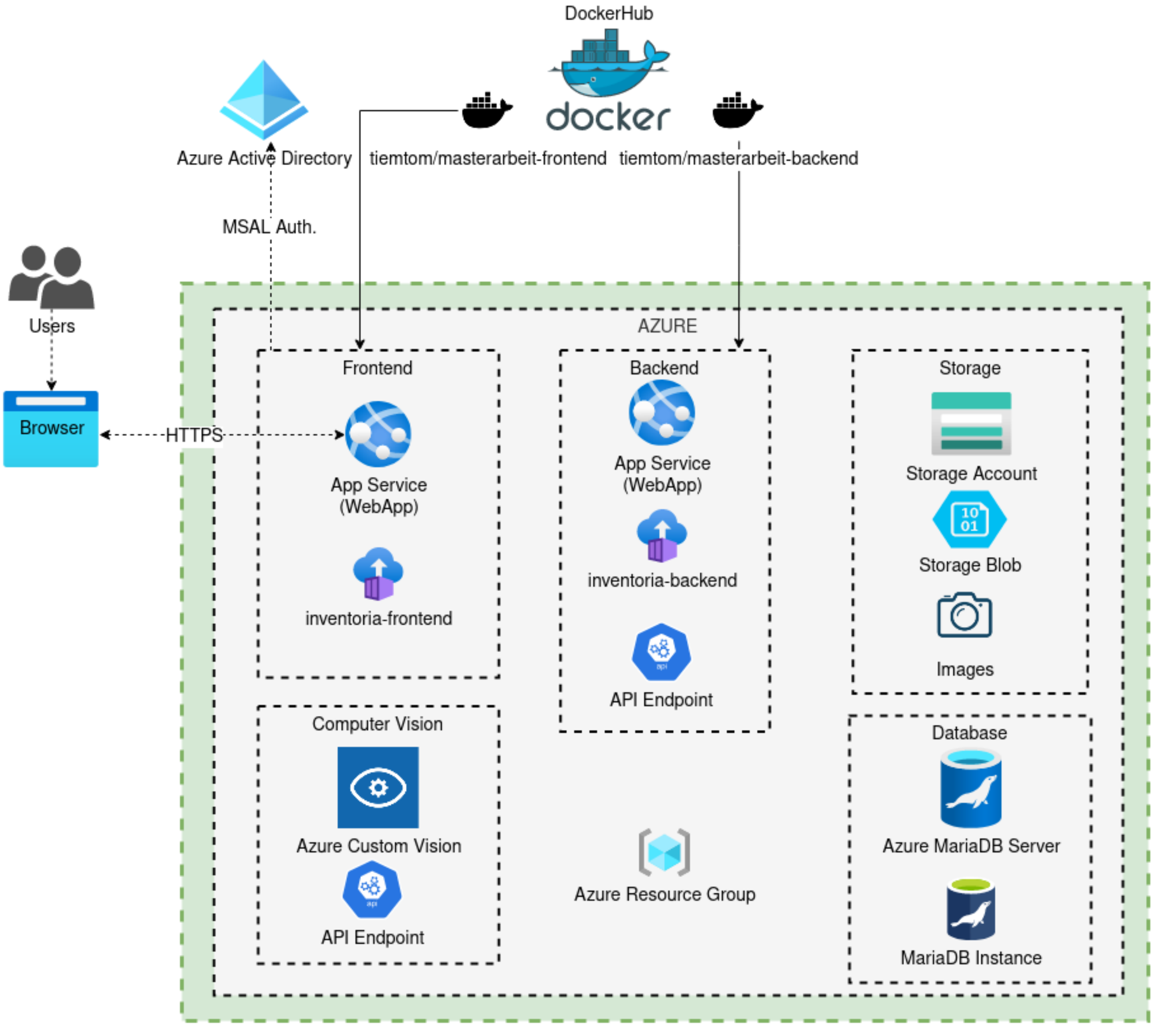


Figure 5.11: Architecture of configuration 4.

5.5 Synchronising Application Data

When moving application services from one provider to another, they are not actually transferred, but rather deleted and recreated. This means that during the deletion of application services like storage and database, all data is lost. So, in order to grant full portability, we are also required to synchronise all our data between providers. There are two main approaches to this problem. Either, the application is recreated first in its new configuration and data is then synced between the old and the new services. Or, when the application is deleted, all application data is synced to a local folder and synced to the new configuration when it is deployed. In this project, the second option was chosen, primarily because it meshes well with Terraform and allows for a fully automated data download and upload without any user interaction.

The synchronisation of data is handled by shell commands, which can be scheduled and called by Terraform. Terraform supports so-called provisioners, which can be declared inside of resources. These provisioners can be used as local-exec provisioners or remote-exec provisioners to invoke a script or shell command locally or on a remote resource, either when the resource is deployed or destroyed. Listing 5.3 shows how local-exec provisioners are used in the S3 storage configuration to automatically synchronise data upon resource destruction and creation.

Listing 5.3: Example of S3 bucket synchronisation using local-exec provisioners.

```
# -------------------------STORAGE------------------------
resource "aws_s3_bucket" "storage" {
 bucket = "inventoriastorage"
 force_destroy = true # allow deletion of non empty bucket
 # will run a shell command before the resource is destroyed
 # downloads all images the local backup directory
 provisioner "local-exec" {
   when = destroy
   command = "aws s3 sync s3://inventoriastorage/ ../../backup/images"
  }
}

resource "aws_s3_bucket_acl" "bucket_acl" {
 bucket = aws_s3_bucket.storage.id
 acl    = "public-read-write"

   # will run a shell command after the resource is deployed
   # uploads image data saved in the backup folder to the S3 bucket
 provisioner "local-exec" {
   command = "aws s3 sync ../../backup/images/ s3://inventoriastorage/"
 }
}
```

The same can be done to execute a remote database dump on the MariaDB instance. However, since the database is in a custom VPC, that is also managed by Terraform, we need to make sure that the internet gateway for that network is not deleted before trying to access the database to produce the database dump. Using a local-exec provisioner on the database itself will therefore not work without adding manual dependencies, since there is no visible implicit dependency for Terraform between the database and the internet gateway. This can be fixed by adding explicit dependencies for the routing table, the route to the gateway and the route table association to the subnet. The data is saved to the backup folder, from where the data is then also imported by other configurations.

Listing 5.4: Database import and export using local-exec provisioners.

```
resource "aws_db_instance" "db" {
  allocated_storage = 5
  engine           = "mariadb"
  engine_version   = "10.3" # aws rds describe-db-engine-versions
      --engine mariadb
  instance_class   = "db.t2.micro" # "db.t4g.micro"
  name             = "inventoriadb"
  identifier       = "inventoriadb"
  username         = var.db_user
  password         = var.db_passwd
  skip_final_snapshot = true # will not create a snapshot of DB when
      instance id deleted
  # final_snapshot_identifier = "inventoria_backup"
  publicly_accessible = true
  vpc_security_group_ids = [aws_security_group.aws_sec_group.id]
  db_subnet_group_name = aws_db_subnet_group.db_subnet.name

  # Tag for resource group
  tags = {
    "rg" = "kleinhapl"
  }

  # will run a shell command before the resource is destroyed
  # import db backup
  provisioner "local-exec" {
    command = "mysql -h
        inventoriadb.cccqkewazeyi.eu-central-1.rds.amazonaws.com -u tom
        -pPa55w.rd inventoriadb < ../../backup/db/backup.sql"
  }

  # will execute a command locally when the resource is destroyed
  provisioner "local-exec" {
    when = destroy
    command = "mysqldump -h
        inventoriadb.cccqkewazeyi.eu-central-1.rds.amazonaws.com -u tom
        -pPa55w.rd inventoriadb > ../../backup/db/backup.sql"
  }
  depends_on = [
    aws_route_table.public,
    aws_route.public,
    aws_route_table_association.public
  ]
}
```

Since the database is a standardised MariaDB instance, the import and export of data is exactly the same with an Azure hosted DB as with an AWS hosted DB. The storage migration, due to the lack of a common standards employed by both providers, is implementation is slightly differently for each provider. The download and upload of image data is also effectuated using CLI commands, as can be seen in listing 5.5. If the storage account was to be private, the download and upload needs to be effectuated using an SAS token, that can be generated through Terraform and appended to the download / upload command.

Listing 5.5: Uploading and downloading application data from an Azure storage account.

```
resource "azurerm_storage_container" "container" {
  name                  = "images"
  storage_account_name = azurerm_storage_account.storage.name
  container_access_type = "container"

  # will execute a command locally before the resource is destroyed
  # db data dump (needs to be done here because once the gateway is
      deleted there is no way to reach the db)
  provisioner "local-exec" {
    when = destroy
    command = "az storage blob directory download -c images
        --account-name inventoriastorage -s '*' -d
        '../../backup/images/'"
  }

  # will run a shell command after the resource is deployed
  provisioner "local-exec" {
    command = "az storage blob directory upload -c images --account-name
        inventoriastorage -s '../../backup/images/*' -d '.'"
  }
}
```

5.6 Resulting Thoughts

The development of the Terraform files, defining the individual configurations, took more time and effort than anticipated. That being said, with better knowledge of the cloud environments for which the architecture is developed, the workflow of converting a finished architecture to a Terraform configuration is not very difficult. Therefore, the initial configuration, where half the services remained very similar to their original configuration in the preceding project, was straight forward for the Azure hosted services and more time intensive for the AWS hosted services. This was to be expected, as I am very familiar with the Azure cloud from my studies but barely worked with AWS before. While the process is not quite as fast and straight forward as when using the native IaC languages from Microsoft and Amazon, which make it possible to export a running infrastructure from their respective portals, the good documentation and help online to fill possible gaps make this development an overall pleasant experience. Also, the use of resources from previous configurations as building blocks for new configurations makes is possible to create a new configuration in a couple of hours, as was the case for configuration 3 and 4.

Where Terraform shines is when having to transfer information generated by one provider configuration to another. For example, when generating a database on the AWS cloud using Amazons RDS, the connection endpoint for this database is generated using a random string. In order to avoid hard coding the endpoint into the application, it needs be read from the host operating system using environment variables. If the backend is hosted on Azure and the frontend on AWS, the environment variable needs to be exported from AWS upon generation and then configured in the Azure app service environment. Since Terraform is able to interface with both providers, it can simply create a reference to that string inside the same configuration file. An example from config 1 is shown in listing 5.6. In the app settings block, we can simply reference the resource defined elsewhere in the file to extract the endpoint connection string. This string is then accessed in the applications' database configuration (database.py) to build the complete connection string.

Listing 5.6: Configuring an environment variable using references to resources from another provider.

```
# ------------------------DATABASE AWS------------------------
resource "aws_db_instance" "db" {
  allocated_storage = 5
  engine            = "mariadb"
  engine_version    = "10.6.7" # aws rds describe-db-engine-versions
      --engine mariadb
  instance_class    = "db.t2.micro" # "db.t4g.micro"
  name              = "inventoriadb"
  identifier        = "inventoriadb"
  username          = "tom"
  password          = "Pa55w.rd"
  skip_final_snapshot = true
  # final_snapshot_identifier = "inventoria_backup"
  publicly_accessible = true
  # security group
  vpc_security_group_ids = [aws_security_group.aws_sec_group.id]

  # Tag for resource group
  tags = {
    "rg" = "kleinhapl"
  }

# ------------------------WEB APP AZURE------------------------
}
# web app service
resource "azurerm_linux_web_app" "inventoria_backend" {
  name                = "inventoria-backend"
  resource_group_name = "masterarbeit_kleinhapl"
  location            = "northeurope"
  service_plan_id     = azurerm_service_plan.app_service_plan_backend.id

  # container config
  site_config {
    application_stack {
      docker_image  = "tiemtom/masterarbeit-backend"
      docker_image_tag = "latest"
    }
  }

  # environment variables which can be accessed from code
  app_settings = {
    "DB_ENDPOINT" = aws_db_instance.db.endpoint
  }
}
```

While the overall experience with Terraform was quite positive, there are still a few bugs and hiccups here and there, that require manual intervention. This is particularly true during the development of the deployment configurations. A number of times Terraform seemed to lose parts of the state of the infrastructure, which is saved in the .tfstate file, and fail to acquire certain elements when refreshing the state. This was usually the case when a deployment execution did not completely run through due to a misconfiguration. This can lead to Terraform being unable to destroy or deploy parts of the infrastructure, since it is unaware of the presence of a certain service. Upon manual deletion of the service, Terraform is once again able to assess the complete state of the infrastructure.

Furthermore, while Terraform supports most cloud providers on the market, working with them still has a learning curve, that, even if already familiar with other providers, can be quite steep. Specifically, when working on hosting the docker containers, the transition from Azure to AWS was quite time demanding, since the implementation between providers differs in many ways. The same can be said for the implementation of storage, with AWS buckets and Azure storage accounts. Theoretically, orchestration engines adhering to a unified standard, such as TOSCA which is covered in chapter 6, should abstract away these differences between providers and make the use of different providers seamless for the end user. One such abstraction is also in use in this project, not in the architecture configuration, but in the application source code, for storage access. The Apache libcloud library keeps its promises of offering storage access to different providers using a unified interface. Only the provider and the access information need to be altered for the code to work, which is done using environment variables set in the Terraform configuration files.

Finally, the maintenance of multiple configurations makes the adaptation and minor changes made to one configuration time-consuming. Especially, not immediately propagated to other configurations can lead to issues. This was also experienced in this implementation. Adjustments to the application were made after the first couple of configurations were already finished and required adjustments in the configuration template. This led to problems when switching back to an older configuration, which had not yet been adjusted for these application changes. In the end, it would have been less time-consuming to immediately propagate changes from newer configurations to older ones, instead of having to troubleshoot old configurations at a later time.

All in all, the use of Terraform as a development tool represents a large amount of upfront effort, but once configurations are defined, the deployments and resource management across providers becomes very straight forward and time saving. The use of a common DSL for the declaration of resources for both providers, does not take away much of the complexity of working with different providers, due to the underlying APIs being different and Terraform not providing a simplifying abstraction. However, the possibility to access resource information generated by one provider in the resource of another provider does take away some complexity by solving the problem of transferring endpoint URLs and access keys in an elegant and convenient way. Still, a tool like Terraform is far away from representing a perfect solution for true seamless portability of resources across cloud providers, but it does give one the ability to create this portability to a certain extent.

5.7 Shortcomings and Possible Improvements

Computer Vision

The automatic training of the computer vision services was omitted from in this project. This was primarily due to cost management reasons. Furthermore, the custom vision portals and APIs from AWS and Azure differ a lot, with no training support in terraform. But there is Azure CLI and AWS CLI support for the training of the respective computer vision services, which could be integrated into Terraform. The application is currently designed to run specifically with an Azure custom vision endpoint, using AWS computer vision will require altering the backend code. However, the application is not dependent on this service for operation, but uploaded images cannot be automatically classified without a custom vision endpoint.

Identity Management

In its current state, the Inventoria application requires a Microsoft AD domain to function. Changing this requirement to a self-hosted solution would make the deployment more versatile and straightforward to try out. That being said, the deployment of the application does not require the domain and is possible with an Azure and AWS subscription. The integration of MSAL into the application is also not completely isolated to the backend or frontend, like it could be expected from a microservice oriented application. It consists of frontend components handling the login and configuration of the Azure AD tenant, found in the App.vue and main.js files. As well as a backend function, that checks the validity of the MSAL Token received after logging into the application. This function can be found in the main.py file and is referenced in the frontend routers index.js.

Additional Providers

It would be interesting to look at other providers supported by Terraform and see whether the problems that are encountered during development are similar to the ones experienced with Azure and AWS. AWS and Azure are clearly the two biggest Infrastructure as a Service (IaaS) and Platform as a Service (PaaS) cloud providers on the market, with a 33% and 21% market share respectively. [9] It would therefore be especially interesting to take a look at incorporating services from smaller providers and see whether Terraform through the use of these provider APIs is able to offer the same level of operationality and whether the services themselves are on par with the offers from the market leaders.

Open Database Port

In terms of security, the application and the configuration of the infrastructure could benefit from several improvements. Particularly problematic is the database port, that is completely open to the internet. The easiest fix for this would be to restrict access to that port to the backend IP address. However, to initially populate the database and export the data before deletion, the machine running the deployment scripts also requires access to the database. There is no meaningful way to automate this process, and the IP would have to be manually granted access in the security group. Since security was not a primary objective of the implementation and to minimise manual alterations to the deployments, this was not implemented.

Easily Adjustable Frontend Deployment

Due to problems with autogenerated DNS names described in chapter 5.3, there is the need to adjust a backend connection string manually whenever the backend DNS name changes. Since the application is deployed from a container registry, this means that the container needs to be rebuilt and uploaded to the repository, from where it will be pulled by the hosting cloud service, every time a connection string adjustment is needed. As a result, someone deploying the application in a configuration that requires alterations to the application source code, is also required to create their own docker repository and change the repository name in the deployment plans accordingly. Choosing a different deployment model could make the process of altering application code less cumbersome, but possibly at the loss of some automation.

6 The Cloud Infrastructure Landscape

This chapter examines the current cloud infrastructure landscape, with a focus on standardisation and development efforts that aim to improve working with multi cloud infrastructures in the cloud. The first chapter specifically looks at standards that provide theoretical solutions for multi cloud deployment and management. The subsequent chapter inspects projects that aim to provide practical solutions, using some standards described in the first chapter. This will give some perspective on the practicability of some of the standards analysed in the first chapter.

6.1 Standardisation

TOSCA

There are efforts to create standards that allow for a common method of interacting with cloud resources across providers. One of the most prominent standards trying to achieve this is the Topology and Orchestration Specification for Cloud Applications (TOSCA) developed by the non-profit consortium Organization for the Advancement of Structured Information Standards (OASIS). TOSCA is designed as an open standard DSL that enables portability and automated management of cloud applications regardless of the underlying cloud platform. Using machine and human-readable models, the TOSCA standard can drive the orchestration of applications, services and resources throughout their entire lifecycle. As such, TOSCA goes beyond the provisioning of services and offers support for automated management as well. [32]

To achieve this, TOSCA uses a metamodel for the definition of IT services, which describe the service structure and how to manage it. The original specification uses the XML schema, however, there is also a YAML rendering of TOSCA called the TOSCA Simple Profile, which aims to provide a more accessible syntax. In TOSCA, services are defined in service templates, the elements making up such a template are depicted in figure 6.1. The topology of a service is described through nodes and relationships. Plans describe, in form of process models, functions to create, terminate and manage the service. Process models can be designed using existing modelling languages, such as BPMN (Business Process Model and Notation) or BPEL (Business Process Execution Language). The abstraction described by the TOSCA metamodel makes no assumptions about the hosting environment, this is instrumental in enabling complete portability. [5]

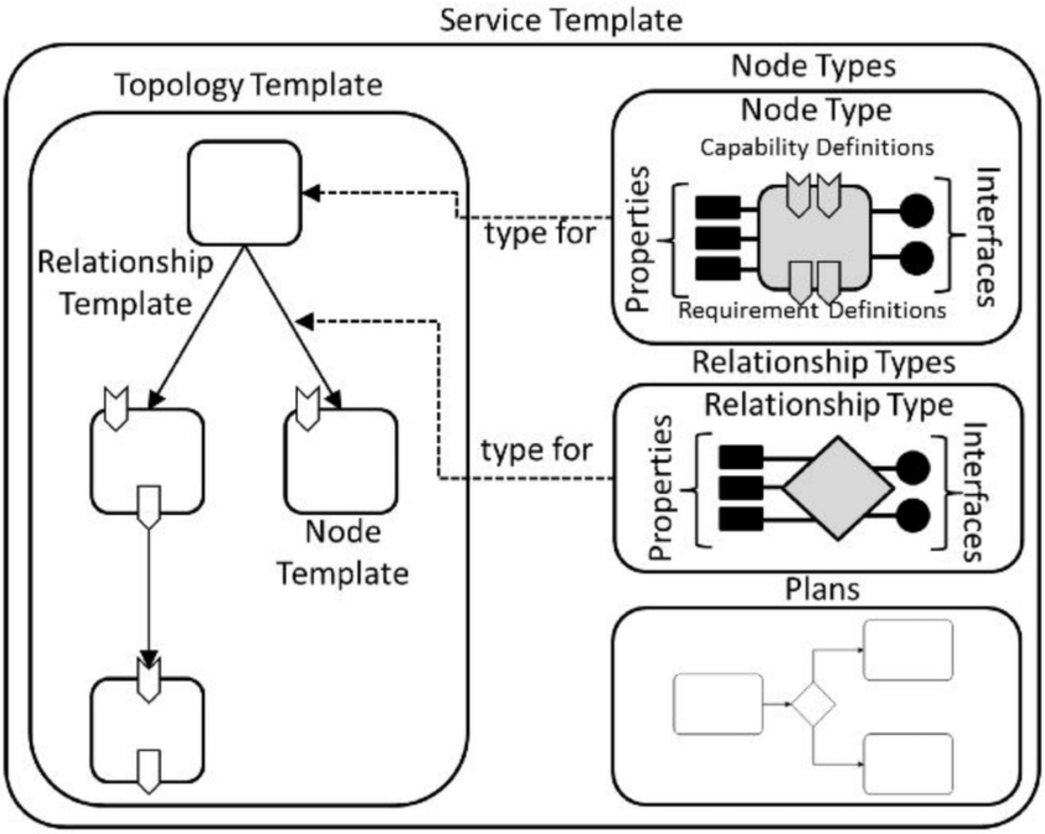


Figure 6.1: TOSCA service template overview. [5]

In order to execute TOSCA templates, an orchestration engine is required. This TOSCA processor must be capable of parsing and interpreting service templates in order to instantiate, deploy and manage the described service in a certain environment. To achieve this, orchestrators may make use of existing tools, like Terraform or Ansible as well as environment-specific APIs, in their implementation. TOSCA node types, which can be seen as frames for general services (e.g. Compute), are also depicted in figure 3.2. They are defined separately from service templates, so that they can be reused in other templates. There is also a normative list of pre-defined node types, that are expected to be known by all TOSCA orchestrators. This will be the case for most common use cases, like a compute instance node. The idea is that additional node types will be defined by the community to make the creation of service templates easier. In listing 6.1 a simple example using the TOSCA Simple Profile in YAML, to model a single server. [33]

Listing 6.1: A service template for provisioning a single server using TOSCA Simple Profile. [33]

```
tosca_definitions_version: tosca_simple_yaml_1_3tosca_simple_yaml_1_3

description: Template for deploying a single server with predefined
    properties.

topology_template:
  node_templates:
    db_server:
      type: tosca.nodes.Compute
      capabilities:
        # Host container properties
      host:
       properties:
        num_cpus: 1
        disk_size: 10 GB
        mem_size: 4096 MB
      # Guest Operating System properties
      os:
        properties:
         # host Operating System image properties
         architecture: x86_64
         type: linux
         distribution: rhel
         version: 6.5
```

All node and relationship types that are defined in the Simple Profile specification are designed to offer portability out of the box. This means that every TOSCA compliant orchestrator is required to know how these nodes or relationships can be deployed. Therefore, the example in listing 6.1 requires no additional plans containing the logic for the deployment and implementation of the service. To install software on the compute node, the service template can simply be extended by another node template. In listing 6.2 a node template of type "tosca.nodes.DBMS.MySQL" is used to configure a MySQL instance on the compute node. The relationship to the compute node is defined in the requirements section, indicating where the MySQL instance should be hosted. The database will then be instantiated using the default set of operational scripts that contain basic commands for managing the service, such as starting and stopping the application. It is also possible to override these default management plans by providing custom configuration scripts. Additional compute resources can be added the same way, making the definition of whole server architectures possible. [33]

Listing 6.2: Adding a MySQL instance to a TOSCA service template. [33]

```
mysql:
    type: tosca.nodes.DBMS.MySQL
    properties:
      Root_password: admin
      port: 3306
    requirements:
      - host: db_server
```

In the example from listing 6.2 a product specific node is used to declare the database, this service template will always deploy a MySQL database. Using TOSCA it is also possible to define product-agnostic templates, by modelling a specific functionality without defining a particular product or vendor. This allows the end-user to decide at runtime what product he would like to use to fulfil a particular function. [33]

The conception of TOSCA as a metamodel, one layer above provider APIs, allows for the creation of portable, platform-independent service templates, without the need of knowing platform-specific APIs. However, this also means that different orchestrators are needed for every cloud environment. While such an API implemented by every cloud provider would represent an attractive solution to end-users, providers themselves might be deterred from an implementation, as it will counteract vendor lock-in. Still, if the standard becomes popular enough amongst consumers that TOSCA compliance and interoperability becomes a notable decision criteria in choosing a cloud provider, providers might be driven to support efforts to build such orchestrators. But TOSCA does not rely on cloud providers to implement a standardised API, orchestrators can also be built using third party tools. Version 1.0 of TOSCA was released in November 2013, to this date there is no official support for TOSCA orchestrators from any of the leading cloud providers. Nonetheless, there are a number of third party orchestrators and orchestration frameworks that make use of the standard. One of the more prominent solutions was represented by the SeaClouds project, which is described in more detail in chapter 6.2.

CAMP

The Cloud Application Management for Platforms (CAMP) standard, which stands for "Cloud Application Management for Platforms" is another cloud standard developed by OASIS, that defines a generic API for deploying and managing applications in PaaS cloud environments. The development of the standard began in 2012, spearheaded by a consortium of technology vendors including Oracle and Red Hat. [34] Essentially, the goal was to create a common application management API used to deploy and manage application that are deployed to the cloud. This would mean that someone who was to deploy an application to provider A could deploy the application to provider B using the exact same tools and processes. Typical application lifecycle tasks like uploading, customizing, deploying, instantiating, suspending, stopping, destroying and extracting can be controlled through the CAMP API. [35]

While some projects have leveraged the standard, overall support with providers is quite limited, with many rather opting for proprietary solutions. The last version of the standard was published in 2018 and the working groups has been closed in April 2021. The lack of adoption for CAMP might also be attributed to the popularity of containerisation of applications, which circumvent classical application deployments and provides an alternative, now standardised approach that was already widely adopted at its time of standardisation. But, the downfall of CAMP can also in large part be attributed to the lack of incentives for leading providers to implement such a standardised API. [36]

Open Container Initiative

One of the most important innovations for portability of applications in the cloud was the virtualisation of resources on the OS level, brought about with the introduction of container technologies. It is important to note that containers ensure the portability of the workload, not the application architecture, as is the goal of the practical implementation in chapter 5. While hardware virtualisation opened up the possibility to migrate virtual machines from one host to another, interoperability between providers is not always a given. Migrating from one virtualisation environment to another is often possible, but not nearly as straightforward and effortless as moving containers. Both Microsoft and Amazon offer solutions to migrate VMs from the respective other to their infrastructure. Microsoft with a custom migration tool developed in-house [37] and AWS promoting the use of another cloud computing company, CloudEndurance, which specialises in the domain of disaster recovery, backups and live migrations. [38] The company, whose commercial solutions are also used by other big players on the cloud market, like Google and Cisco, was acquired by Amazon in 2019 for over $200 million, reportedly outbidding Google for the acquisition. [39] With containers, there is no need for additional tools to migrate from one environment to another.

Containers allow one to pack software along with its dependencies into a single image, that can then be deployed to a container runtime environment. This means that a container image is built once and can then be deployed any number of times to different container runtime environments. An instance of such an image is then referred to as a container. What makes these containers so versatile, is that they are built around essential functionality already present in the Linux kernel, namely control groups and namespaces. Linux kernel name spaces make it possible to completely isolate processes from one another. Linux control groups, also referred to as cgroups, are used for resource management. They are used to share resources with other processes running on the same kernel, without the processes knowing about eachother. With cgroups a certain amount of computational resources can be assigned to a group of processes. [40]

Standardisation of container technologies began in 2015 with the launch of the Open Container Registry (OCI) by the Linux Foundation. This initiative was spearheaded by Docker, the most famous and widespread implementation of containerisation, and quickly gained support from many big brand names. [41] Unfortunately, OCI is not an example of a standard leading to wider adoption of a technology. Widespread support for containers is largely independent to the standardisation of the technology. Most cloud providers offered support for the hosting of containers well before standardisation efforts began. AWS, for instance, even launched their own services centred around container technology before the creation of the OCI, with their ECS service. [42] It is difficult to say whether the standardisation efforts of container technologies had an impact on the adoption of the technology in the cloud. However, what the standardisation does ensure is that developments of the technology are not bound to one company like Docker, but the technology can be further developed and improved by anyone, while maintaining compatibility by adhering to the open standard. A good example for this is Podman, a container management engine that fully supports docker containers but addresses some of Dockers shortcomings, mainly in terms of security. [43]

Containers are a perfect example of a technology that severely reduces vendor lock-in and is supported by almost every cloud provider out there. Unfortunately, it also represents a recipe for success that is difficult to reproduce. The keys for success for containers are on the one hand the ease of implementation and on the other a dramatical simplification of end-user workflows, that made the technology a must-have for most customers. Compared to the TOSCA standard, which would also greatly improve portability of applications in the cloud, there is a much higher incentive on the provider side to implement the container standard, which is one of the reasons why TOSCA is struggling to find traction.

Storage

The lack of standardised file storage implementations was one of the biggest surprises discovered during the practical implementation detailed in chapter 5. There are standardised storage APIs, such CDMI, an open standard published by the Networking Industry Association (SNIA). But, the adoption of this standard, as with TOSCA and CAMP, is lacking. Especially bigger providers, such as Azure and AWS often choose to rather implement proprietary solutions. Even the publishing of guides to use CDMI to implement functionality similar to S3, did not suffice to encourage AWS to implement it in their cloud environment. [44] The standard itself is still in development, with version 2.0 being published in 2020. [45]

There are many more interoperability standards that have at some point been created for the cloud. The ones described above are simply the ones that are associated the closest to the practical work done in chapter 5. Unfortunately, what most of these standards for the cloud have in common is their low adoption rate with providers. What can be taken from the one extremely successful example, OCI, is that simplicity on both the end user and provider side is indispensable for widespread adoption. Another way to force adoption of a standard is through policy. For instance, this could be a mandatory standardised API, that is able to control all common services in the cloud and has to be implemented by all cloud service providers. This would mean that there is portability around the core services that make up most applications in the cloud, but still room for providers to work on specialised proprietary services that are not necessarily openly standardised, so that innovation is no inhibited.

6.2 Multi Cloud

In itself, multi cloud scenarios have been around for a while and the inconveniences and problems that are caused by such cross provider solutions (as documented in chapter 2.2) have been pondered and worked on in several projects, that aim to make managing multiple cloud providers easier. Today, there are a number of tools out there, that will indeed make it possible to address particular pain points of multi cloud management, but no practical solution is able to offer complete portability on a large scale.

Among multi cloud management platforms, there are generally two main approaches for the integration of multiple providers into a single unified interface. The first is to use third party multi cloud toolkits such as Apache Libcloud, which was also used in the practical implementation documented in chapter 5, to have unified access to multiple providers through an abstracting API. One such example is Scalr, one of the more popular cloud management platforms. Scalr promises to offer unified cost visibility and control, central service monitoring and central policy administration. To achieve this, Scalr uses agents, that are installed on resources that are then managed by its core server, the corresponding architecture is depicted in figure 6.2. Scalr also makes the definition of infrastructure agnostic infrastructure templates possible. While this functionality initially sounds like this feature might hide a TOSCA like orchestration engine, the functionality is much more basic and closer resembles a specced down implementation of Terraform. Custom resource templates can be created in a JSON format. Making them provider-agnostic means either leaving out parameters that differ between providers and filling them in upon execution, or defining different defaults for the providers. The services that can be defined through the Scalr platform are also more limited than Terraforms offerings. The list of providers supported by Scalr is also rather short. [6]

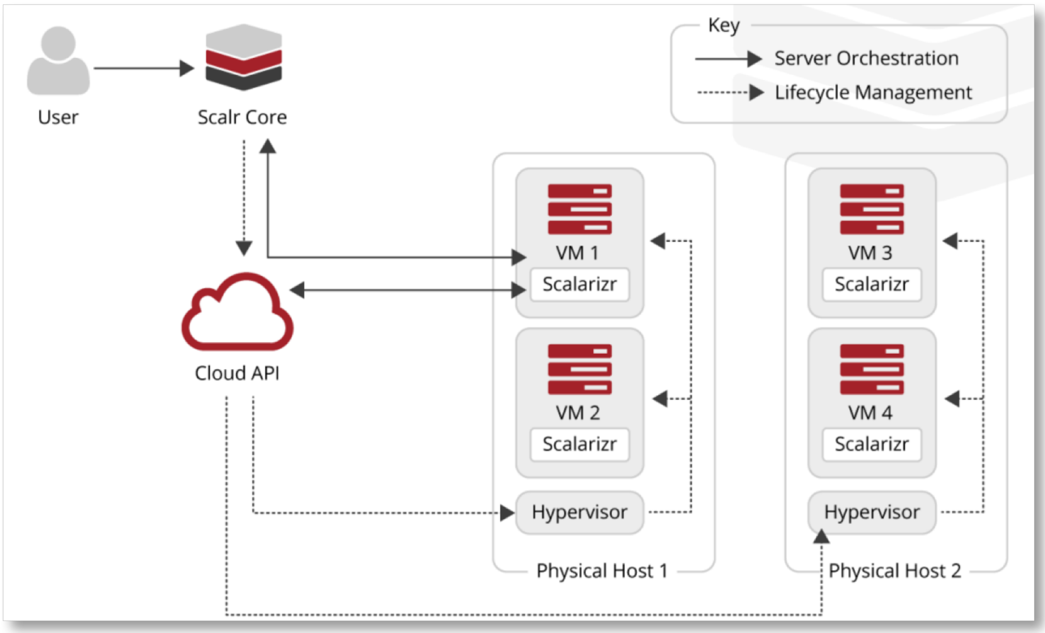


Figure 6.2: Scalr software architecture. [6]

The other approach to the integration of multiple providers consists of deploying a common technology stack. This is, for example, how OpenStack works. OpenStack is really a bundle of fully open sourced projects, initially developed by NASA and Rackspace, with the goal of creating an open IaaS stack that is easy to implement. [7] It is debatable whether this should be referred to as a multi cloud solution, since the cloud environment is always the same and all available providers run the same technology stack. With solutions like OpenStack, where all providers support a common standard, there is no need for cloud orchestrators that are able to translate an infrastructure from one provider to another, portability is guaranteed by the standardised technology stack. By abstracting underlying hardware to common infrastructure components, that are shared by all their providers, OpenStack is able to construct a homogeneous architecture across providers. For migrations, OpenStack provides their own orchestrator called Heat, through which infrastructure templates can be defined and deployed to OpenStack providers. [46] In figure 6.3 one can see the different infrastructure components that are part of OpenStack and what roles they fulfil.

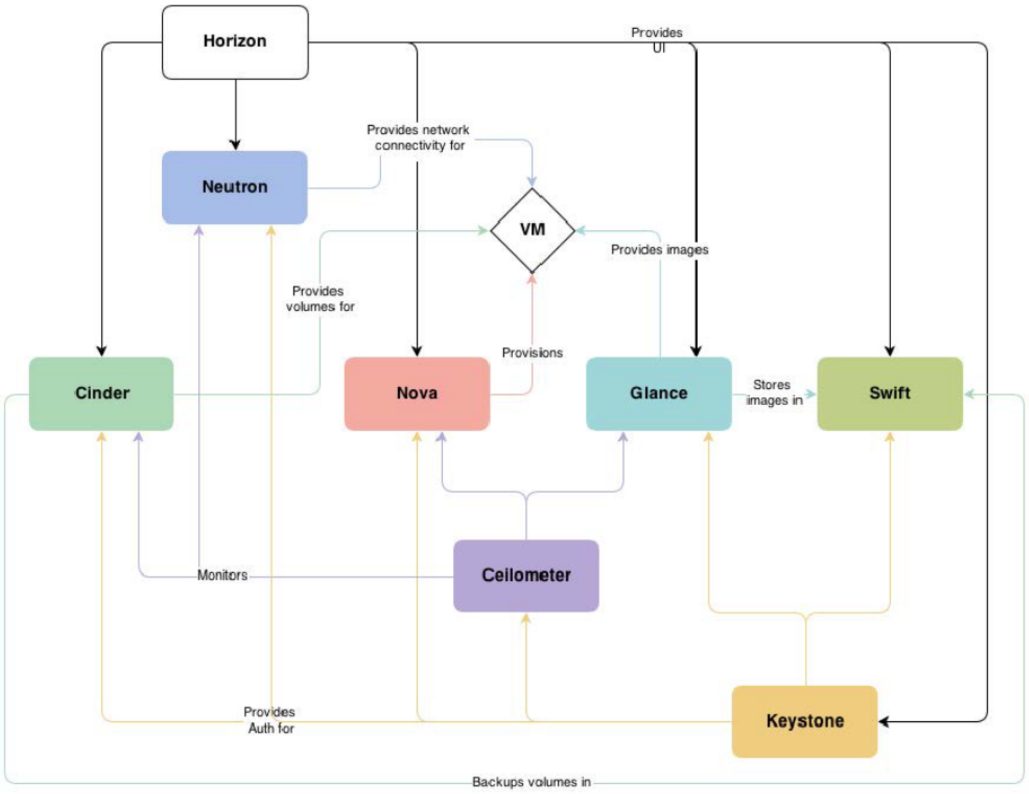


Figure 6.3: Conceptual architecture of OpenStack, showing how their services interact. [7]

Each infrastructure component in OpenStack provides its own API that is independent of other services. This enables OpenStack providers to offer only specific OpenStack Services, without the need to implement the whole stack. The Services depicted in figure 6.3 are just a few of the services defined by OpenStack. They offer the following functionality:

- Nova: Compute module providing virtual servers. The servers can be provided using several supported hypervisors, which is another benefit of OpenStack, having little requirements to the underlying hardware or software.
- Glance: OS image lookup and retrieval service for virtual machines.
- Neutron: Virtual network service.
- Swift: Object/Blob storage.
- Cinder: Block storage, mainly to provide storage volumes to nova compute instances.
- Keystone: Identity management service, integral to any OpenStack infrastructure.
- Horizon: Cloud management dashboard.
- Ceilometer: Monitoring solution that is also used as basis for billing systems.

[7]

These are just a few basic services in the OpenStack portfolio. By now, the service portfolio has more than tripled. [47] The support for a large range of hardware, makes OpenStack a perfect candidate for a hybrid cloud solution. The modular approach makes it possible to deploy OpenStack modules on local hardware or have it hosted in a private cloud.

There are also some more ambitious projects centred around multi cloud management. One such project is the SeaClouds or seamless adaptive multi cloud management of service-based applications project, funded by the EU in 2014. The project started with four major goals:

1. Orchestration and adaptation of services distributed over different cloud providers.

2. Monitoring and run-time reconfiguration of services distributed over multiple heterogeneous cloud providers.
3. Providing unified application management of services distributed over different cloud providers.
4. Compliance with major standards for cloud interoperability.

[8]

So the goal was to create a platform that would eliminate the need to learn or be familiar with the cloud APIs and management platforms in use by different cloud providers, by providing a single unified interface to the end user that is able to manage services from different providers, while also providing interoperability so that services can be moved from one provider to another. Essentially, this platform would represent an ideal solution to what was aimed to achieved in the practical implementation of this thesis.

To achieve its goals, the project planned to leverage existing standards and solutions and incorporate them into their platform, as can be seen in the SeaClouds architecture overview in figure 6.4. At the heart of the orchestration engines, there are the TOSCA and CAMP standards, used to guarantee the portability of applications. The management of applications utilises Apache Brooklyin, a framework to model, monitor and manage applications through autonomic blueprints. The blueprints are defined in YAML and currently comply to the CAMP standard, with plans to comply with TOSCA as well in the future. [48] Additionally, there are three further open source frameworks in use by SeaClouds project. Apache Whirr, which uses Apache jClouds a multi cloud toolkit, similar to the Libcloud framework used in the practical application, used to interact with cloud services. And finally, Cloud4SOA, a monitoring solution that abstracts provider-specific APIs to offer a unified monitoring interface. [8]

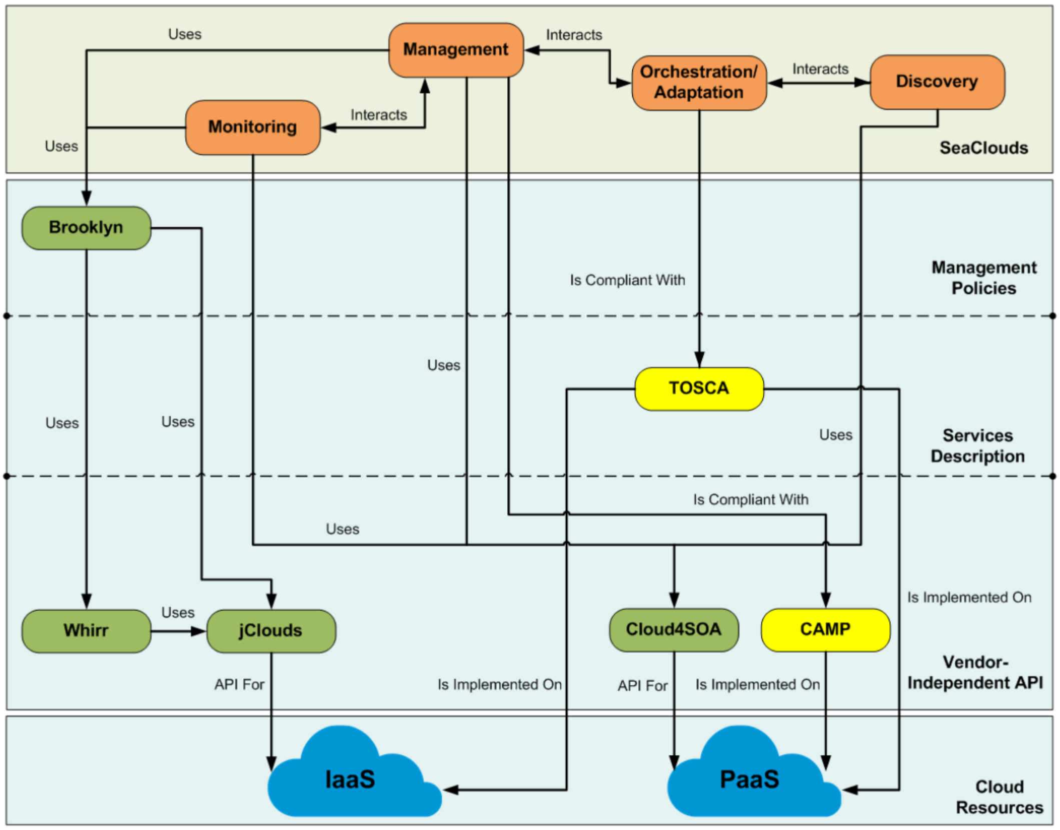


Figure 6.4: Architecture overview of the SeaClouds platform. [8]

Unfortunately, the SeaClouds projects has been mostly inactive since 2016 and has not managed to produce many of the results it wanted to. Some of the projects that were to be leveraged within the SeaClouds platform have also ceased to be updated. One goal that was achieved, is the contribution to the development of the cloud standards TOSCA and CAMP, but without the implementation of the standards in a working prototype. With CAMP already being discontinued and TOSCA having very few actual implementations, it seems like the goal of having a widely supported common standard for infrastructure portability in the cloud will not be attained in the near future.

6.3 Cloud Infrastructure Markets and Competition

The cloud infrastructure market is dominated by very few big players. As can be taken from the infographic in figure 6.5, the big cloud providers are primarily American and Chinese companies. While, provider exclusive features are far and few between, overall portability in the cloud is still quite difficult, which solidifies the position of established players. Once a provider is in use by a consumer, it can be very cumbersome to make the switch to another provider with the same feature set. This conclusion can be taken from the implementation of a single application in multiple configurations involving two cloud providers, described in chapter 5. This means that once a sizable infrastructure is deployed with a particular provider, the retention rate for these clients will be very high.

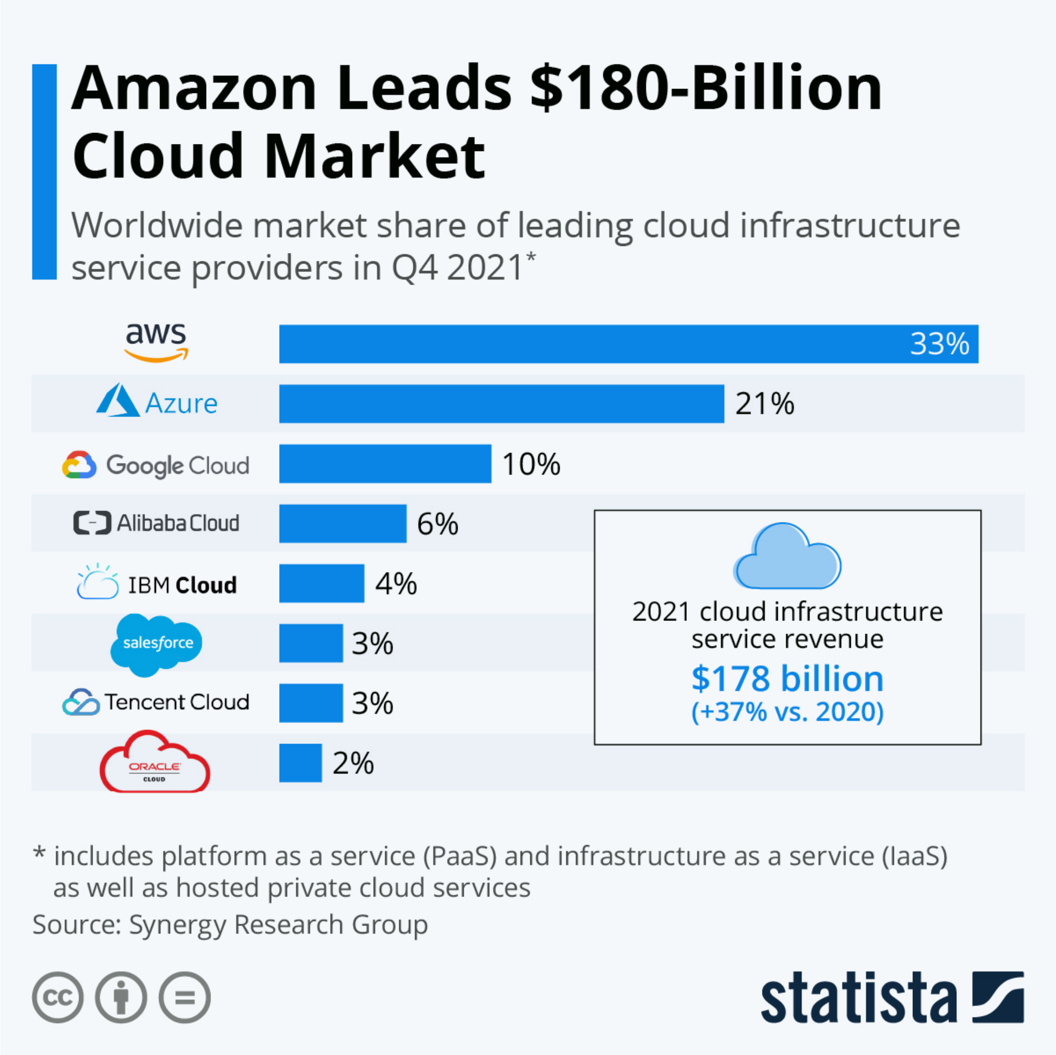
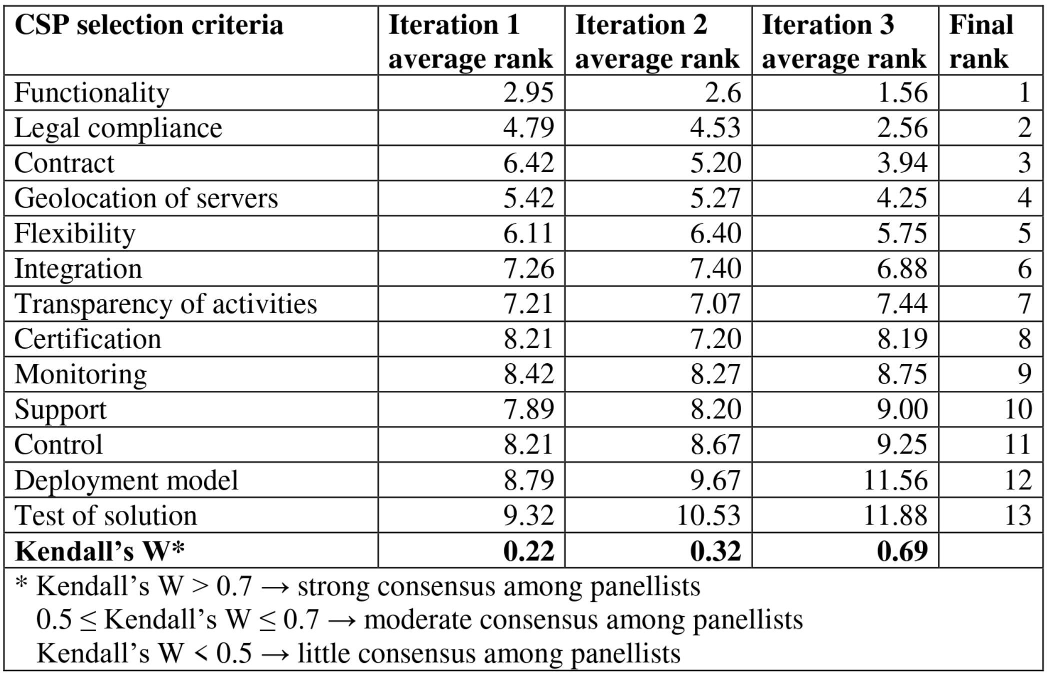


Figure 6.5: The cloud market is dominated by American and Chinese companies. [9]

A study conducted in 2016, using the Delphi method to find consensus in a group of 16 selected cloud experts, concluded that the most important aspect for the choice of a cloud provider is the feature set they offer. [10] The results of the study are depicted in figure 6.6. Consequently, it is very difficult for smaller providers to find traction and grow to a size where they can compete with the giants dominating the market. While common features like compute and storage are usually comparable between smaller and larger providers, it is not possible for smaller players in the market to offer the same range of additional functionality as the market leaders.



CSP selection criteria	Iteration 1 average rank	Iteration 2 average rank	Iteration 3 average rank	Final rank
Functionality	2.95	2.6	1.56	1
Legal compliance	4.79	4.53	2.56	2
Contract	6.42	5.20	3.94	3
Geolocation of servers	5.42	5.27	4.25	4
Flexibility	6.11	6.40	5.75	5
Integration	7.26	7.40	6.88	6
Transparency of activities	7.21	7.07	7.44	7
Certification	8.21	7.20	8.19	8
Monitoring	8.42	8.27	8.75	9
Support	7.89	8.20	9.00	10
Control	8.21	8.67	9.25	11
Deployment model	8.79	9.67	11.56	12
Test of solution	9.32	10.53	11.88	13
Kendall's W*	**0.22**	**0.32**	**0.69**	

* Kendall's W > 0.7 → strong consensus among panellists
0.5 ≤ Kendall's W ≤ 0.7 → moderate consensus among panellists
Kendall's W < 0.5 → little consensus among panellists

Figure 6.6: A ranking of the most important criteria for cloud provider choice, based on expert opinions. [10]

This lack of competitiveness and difficulty to regulate offshore providers has prompted the EU to find a way to try and even the playing field, at least on European ground. In 2019 the Gaia-X project was launched collaboratively by several EU countries with the following goal:

> *Gaia-X is an initiative that develops a software framework of control and governance and implements a common set of policies and rules that can be applied to any existing cloud/ edge technology stack to obtain transparency, controllability, portability and interoperability across data and services. [49]*

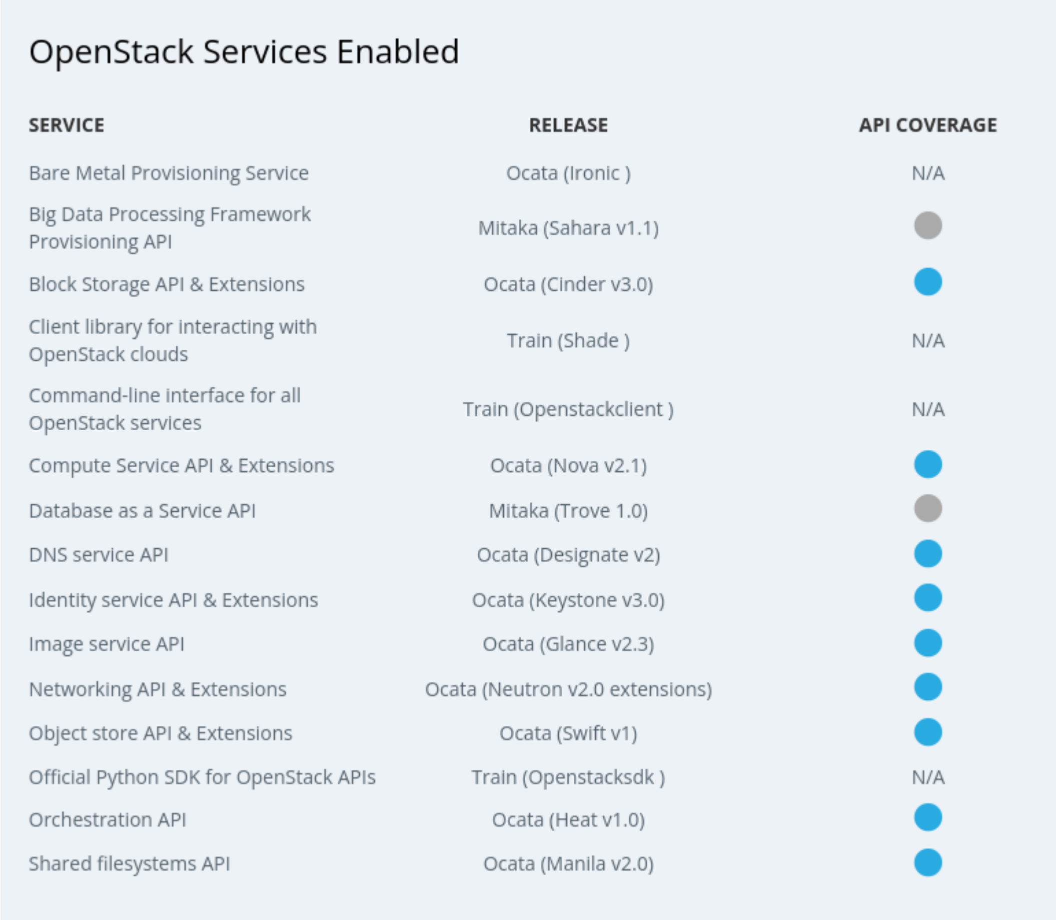
The keywords in this goal definition mesh quite well with some of the criteria important to customers from the study depicted in figure 6.6. Legal compliance, transparency and data control are all factors that are important to costumers and could therefore improve provider competitiveness by adhering to the standards defined by Gaia-X. Furthermore, the fact that portability and interoperability are central goals for the project should make it possible to simply substitute services from one Gaia-X conforming provider to another. This might be especially interesting, with the somewhat controversial integration of major cloud providers like AWS and Azure into the Gaia-X project.

The way Gaia-X tries to improve competition on the market is by offering all of its federated services through a self-service portal, that will show solutions from different providers side by side. Interoperability between services is guaranteed by the platform. This allows end-users to pick and combine services as they need to fulfil their demands. It should easily be possible to exchange a service for another. The approach that Gaia-X takes to ensure portability between providers is very different to a platform like SeaClouds. With SeaClouds, the portability is created by the platform or framework itself, Gaia-X on the other hand forces all of its participants to play by their rules, one of which makes sure that providers implement or support tools that make the transfer of data to other providers possible. As can be taken from their latest policy rules document:

> *The provider shall implement practices for facilitating the switching of providers and the porting of data in a structured, commonly used, and machine-readable*

> *format including open standard formats where required or requested by the provider receiving the data. [49]*

There have also been other initiatives that manage to even the playing field a little more and allow for small providers to compete with the big players. One of them is OpenStack, which offers a common, open source and open standard technology stack to end users, but can be implemented using different providers. At the time of writing, there are 15 public cloud providers. This not only allows OpenStack to be deployed to different locations all over the globe, that are provided by the different providers, but also means that due to the modular composition of the service stack (for a more detailed explanation see chapter 6.2), providers are not forced to offer all services that OpenStack theoretically supports, but can specialise in certain domains. OpenStack provides an infographic for every provider listed in their portal, that shows exactly which services are available with a particular provider, as can be seen in figure 6.7. This allows providers that do not offer a wide range of services to still be competitive and find their place in the market.



OpenStack Services Enabled

SERVICE	RELEASE	API COVERAGE
Bare Metal Provisioning Service	Ocata (Ironic)	N/A
Big Data Processing Framework Provisioning API	Mitaka (Sahara v1.1)	⚪
Block Storage API & Extensions	Ocata (Cinder v3.0)	🔵
Client library for interacting with OpenStack clouds	Train (Shade)	N/A
Command-line interface for all OpenStack services	Train (Openstackclient)	N/A
Compute Service API & Extensions	Ocata (Nova v2.1)	🔵
Database as a Service API	Mitaka (Trove 1.0)	⚪
DNS service API	Ocata (Designate v2)	🔵
Identity service API & Extensions	Ocata (Keystone v3.0)	🔵
Image service API	Ocata (Glance v2.3)	🔵
Networking API & Extensions	Ocata (Neutron v2.0 extensions)	🔵
Object store API & Extensions	Ocata (Swift v1)	🔵
Official Python SDK for OpenStack APIs	Train (Openstacksdk)	N/A
Orchestration API	Ocata (Heat v1.0)	🔵
Shared filesystems API	Ocata (Manila v2.0)	🔵

Figure 6.7: An overview of OpenStack services available at a provider. [11]

As of now, the most promising solutions that allow for small provider to compete with the American and Chinese cloud giants is OpenStack. There have been quite a few efforts to better overall competitiveness in the cloud, by eliminating vendor lock-in, but the reality is that without support from providers directly, these solutions will likely not gain enough traction to become relevant and have a noticeable impact on the market. Many of the projects examined in this chapter have since ceased to be actively maintained or just do not have enough support to be considered a relevant solution. This is why, from today's perspective, it seems likely that the TOSCA standard will not find widespread adoption and will likely follow the fate of CAMP and fade into obscurity. Even though the standard is technically promising and there are tools out there to build orchestrators around the specification, most tools that are used for these implementations, like Terraform, are only abstracting proprietary APIs. The APIs that are abstracted are not standardised and therefore every abstraction needs to be actively maintained. Essentially, working with a tool like Terraform to implement a multi cloud orchestrator would require abstracting the Terraform API and building another layer onto a technology stack that is already built from another abstraction and seems intricate to maintain as is. In my opinion, the only real solution for unified interaction and true portability in the cloud is to get providers themselves to implement standardised interfaces. Given the amount of effort that has been put into the creation and establishment of such standards over many years, it seems to me this will not happen without policy.

Cloud Brokering

A Cloud Service Broker (CSB) is an intermediary entity between a cloud service provider and a cloud service consumer. A CSB will handle all managerial activities between the customer and the provider, like the negotiation of contracts. The value proposition of a CSB might simply be a more economical contract with a provider, where the CSB is able to offer better conditions to a customer, which it can obtain due to being in a stronger negotiating position with providers, by representing multiple customers. But a CSB might also offer additional benefits, like choosing the right services for a customer from a multitude of providers and facilitating the transition from provider to provider. In a way, CSBs can be seen as an abstraction or simplification of the cloud service landscape for the end user. [50]

Generally, CSBs fulfil up to three general roles: aggregation of services, integration of services and the customisation of cloud services providers. Aggregation of services, means bringing together services from multiple providers and offering them in a central place. Usually these services are offered through a unified portal and include centralised billing and service provisioning. Integration of services refers to the bridging of multiple clouds, this could be private and public or even public and public clouds. The most challenging and important aspect of integrating services from multiple clouds is the security of data exchange. This is also one of the aspects, along with legal expertise, that is often promoted by CSBs. The customisation of cloud service providers goes beyond just managing providers in the name of a customer, but consists of additional added-value services. [50]

There are a multitude of CSBs on the market, usually specialising in a particular direction. One such example is Boomi, formerly known as Dell Boomi, which presents itself as an Integration Platform as a Service (iPaaS) (Integration Platform as a Service), a form of enterprise integration platform specialising in cloud services. Boomi is one of the market leaders in the domain, as can be taken from the Gartner magic quadrant in figure 6.8. Boomi offers services and training aiming to facilitate the integration of services, from architecture conception to service deployment. For example, Boomi offers database connectors, that will interface with many on-premise or cloud databases, essentially providing a low code solution to database integration. [12]

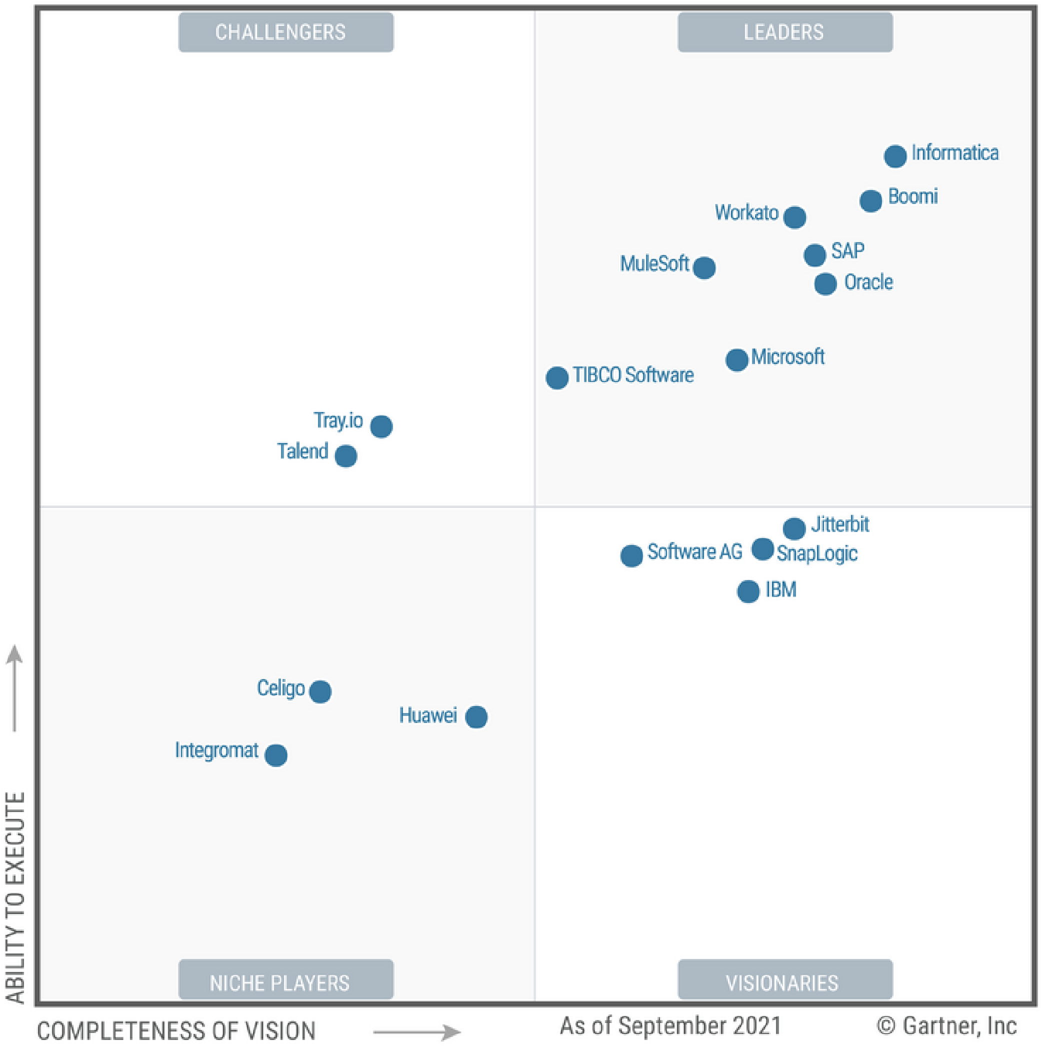


Figure 6.8: Gartner magic quadrant showing CSBs. [12]

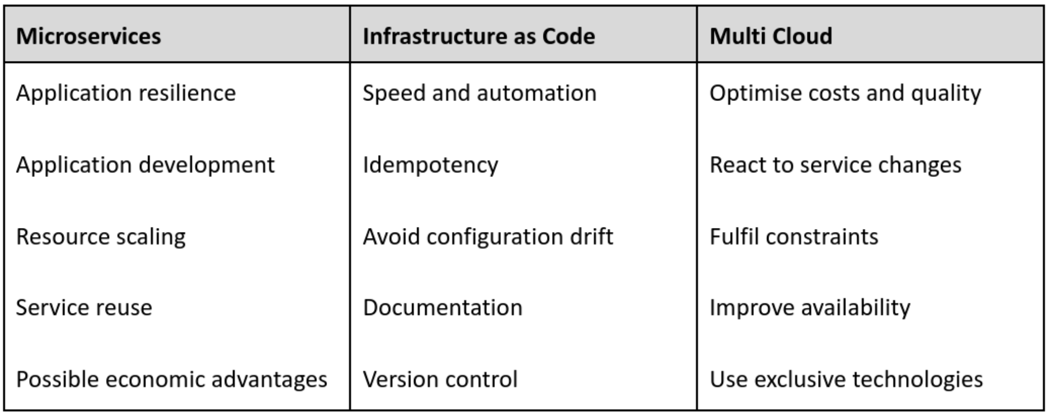
The market for cloud broker services was estimated at $4.96 billion in 2018 and was predicted to grow at a Compound Annual Growth Rate (CAGR) of 17.3% from 2019 to 2025. [51] Given the difficulty of managing multi cloud solutions, even with capable tools, as laid out in chapter 5, the need for such intermediary services is comprehensible, as the expertise needed to integrate multiple cloud environments is considerable. Especially when taking into account security and legal aspects regarding data protection, the added value of going through a brokering services for cloud needs is clearly apparent. The difficulty of establishing widely adopted standards simplifying cloud portability and interconnection furthers the case of CSBs, making them an attractive solution for partly outsourcing multi cloud management.

The impact CSBs have on cloud provider competition is difficult to gauge. While, some CSBs will compare services from different providers for their customers and choose the best fit. The service catalogue will always be based on a limited portfolio of supported providers. Having support for larger, widely known providers, will likely be a priority for most CSBs, as the brand recognition of the supported providers will undoubtedly play a role when competing with other CSBs. Still, the specialisation of a CSB to a certain domain, might bring smaller and lesser known players into the spotlight. As is, they do not really disrupt the established competition amongst the very few major providers.

A development towards a mainly broker oriented market cannot be envisioned without a significantly higher standardisation of cloud services. The current situation, where portability in the cloud is not given out of the box, gives cloud brokers a spot to fill and a way to add value to cloud services, by handling the transition from provider to provider for customers. And as mentioned above, the market size estimation of around $5 billion with a substantial continuous growth prediction shows that an interest for brokering services is definitely given. However, compared to the size of the cloud service market altogether, which lies at around $454.2 billion and has a comparable CAGR to the CSB market [52], there is no reason to expect a transition to a more broker oriented market in the near future.

7 Summary and Conclusion

Combining technology advancements made with IaC, microservices and multi cloud, make it possible to build portable applications, that can be migrated between providers at the press of a button. The most important advantages that can be leveraged by such an application are listed in figure 7.1 and covered in more detail in chapters 2.1, 3.2 and 4.2.

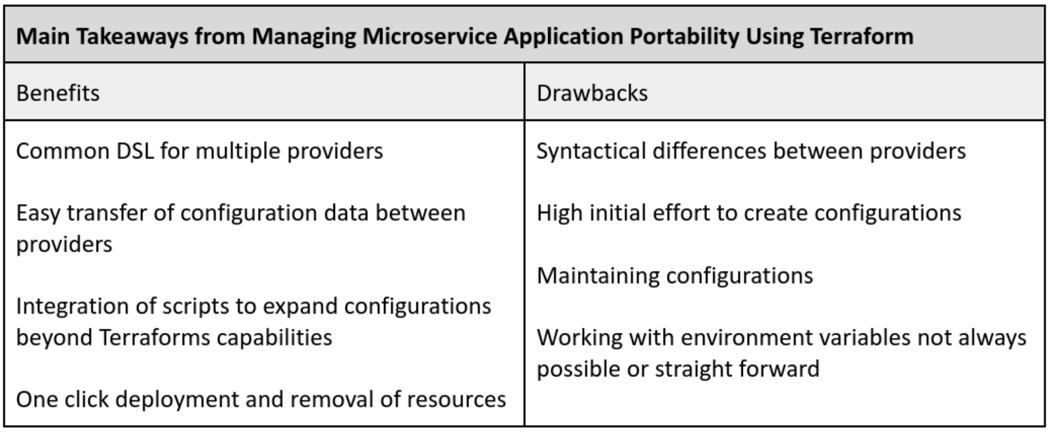


Microservices	Infrastructure as Code	Multi Cloud
Application resilience	Speed and automation	Optimise costs and quality
Application development	Idempotency	React to service changes
Resource scaling	Avoid configuration drift	Fulfil constraints
Service reuse	Documentation	Improve availability
Possible economic advantages	Version control	Use exclusive technologies

Figure 7.1: Advantages of microservices, IaC and multi cloud.

7.1 Managing Portability

The use of a IaC tools for the deployment of microservice architectures is a no-brainer, choosing a tool that supports multiple providers has many added benefits, which are presented in chapter 3.2, and can clearly be endorsed at this point. However, when taking a closer look at the practical implementation, as is done in chapter 5, it has to be said that the portability that can be achieved with a tool like Terraform does come with some inconveniences as well. The main takeaways gained from managing a multi provider, microservice oriented application with Terraform are outlined in figure 7.2 and covered in detail in chapter 5.3 and 5.6. Whether the maintenance of multiple configurations for dynamical redeployments or the use of different providers proves beneficial, has to be evaluated on a case to case basis. Whether an economic advantage can be made depends on the volatility of the prices, on the one hand, and the use case, on the other. Still, the use of a IaC tool supporting multiple providers keeps the option open to expand or translate infrastructures to other providers in the future.



Main Takeaways from Managing Microservice Application Portability Using Terraform	
Benefits	Drawbacks
Common DSL for multiple providers Easy transfer of configuration data between providers Integration of scripts to expand configurations beyond Terraforms capabilities One click deployment and removal of resources	Syntactical differences between providers High initial effort to create configurations Maintaining configurations Working with environment variables not always possible or straight forward

Figure 7.2: The main takeaways from the use of Terraform to manage multi cloud architectures.

Looking specifically at Terraform, the effort of adding support for different providers to an infrastructure should not be underestimated. Especially when working with new providers one is not yet very familiar with, the initial task of converting resources from one provider to another can take hours to days, depending on the level of expertise and complexity of the architecture. One needs to keep in mind that there are slight syntactical differences between services from different providers, even if it is a standardised database service. If components can be copied from other configurations, the creation of new ones is accelerated immensely. Once configurations are in place, switching between can be achieved at the press of a button.

The overall experience using Terraform to manage the deployment and removal of an applications' infrastructure was positive, but there are still a few pitfalls that need to be kept in mind when working with the tool. Missing information in the state file can occur, for instance due to a deployment or removal not running through completely, and require manual intervention with the provider API or portal. Also, dependency inference, which makes sure resources are deployed in the correct order no matter where they are defined in the Terraform configuration file, can fail when components are associated with scripts. This can be fixed by specifying explicit dependencies.

In addition to the infrastructure configuration files and maybe even more important is the conception of the application source code itself. Especially the backend, which will likely be interacting with cloud services like storage and database, needs to be planned out and tested carefully, to ensure it will function in all configurations. In the practical implementation of this thesis, this required the use of a 3rd party library called Libcloud for unified storage access across Azure and AWS, as well as slight adaptations to the database connection establishment, the details of which can be found in chapter 5.2.

7.2 Market and Standardisation

The state of standardisation in the cloud is unfortunately rather bleak, especially regarding portability of cloud infrastructures. While there are initiatives that aim to address this problem, they have struggled to find traction, notably with the largest providers. There are different approaches to alleviate this situation. One is to define standardised APIs, that would need to be implemented by all providers, CAMP is an example of such a standard. The other is to abstract differences between providers into a uniform definition language that can be implemented by orchestrators, using tools like Terraform, the TOSCA standard defines such a standard. Unfortunately, both initiatives have failed to become relevant in the current market. Without enforcement of standards through policy, it is difficult to envision the establishment of a standard, that has such a substantial impact on vendor lock-in, in the near future. This also makes the prospect of a primarily cloud broker driven market an unlikely development in the near future.

On the application code side, the standardisation situation is better. The use of containers to deploy code to different environments quickly gained popularity upon its release. Most container solutions that exist today adhere to the open OCI standard. The popularity of containers can likely be attributed to the ease of implementation for providers, along with the ease of use it provides to the end user. A more detailed look at the state of standardisation, as well as associated projects that make use of them, can be found in chapter 6.1 and 6.2.

There is one cloud infrastructure standard, that plays a role in the current cloud provider market competition. OpenStack has its roots in the multi cloud management domain and defines a set of standards that make up a common technology stack, that can be deployed and used across multiple providers. This allows smaller providers to find a place in the market, even if they only offer very specific services, such as storage. By being fully compatible with all other OpenStack providers, end users can combine multiple providers to build their infrastructures. Accordingly, these infrastructures can also easily be shifted from one provider to another. At the time of writing, this is the most promising solution that makes it possible for smaller providers to compete with the established, dominating cloud providers.

Overall, the cloud market is still largely controlled by a few big players, situated primarily in North America and China. One of the most impactful factors in the choice of a cloud provider is the assortment of services and features on offer. In this department, it is very difficult for small providers to compete with the larger ones. To level the playing field, cloud interoperability needs to be improved, to make it easier to combine services from multiple providers. OpenStack is one way this can be solved. Another, is to force providers to fulfil certain criteria, this is what the Gaia-X project does in the EU, to try and level the playing field and enforce compliance with EU laws. At the time of writing, the Gaia-X project is still in its planning phase and has yet to prove it can achieve its goals.

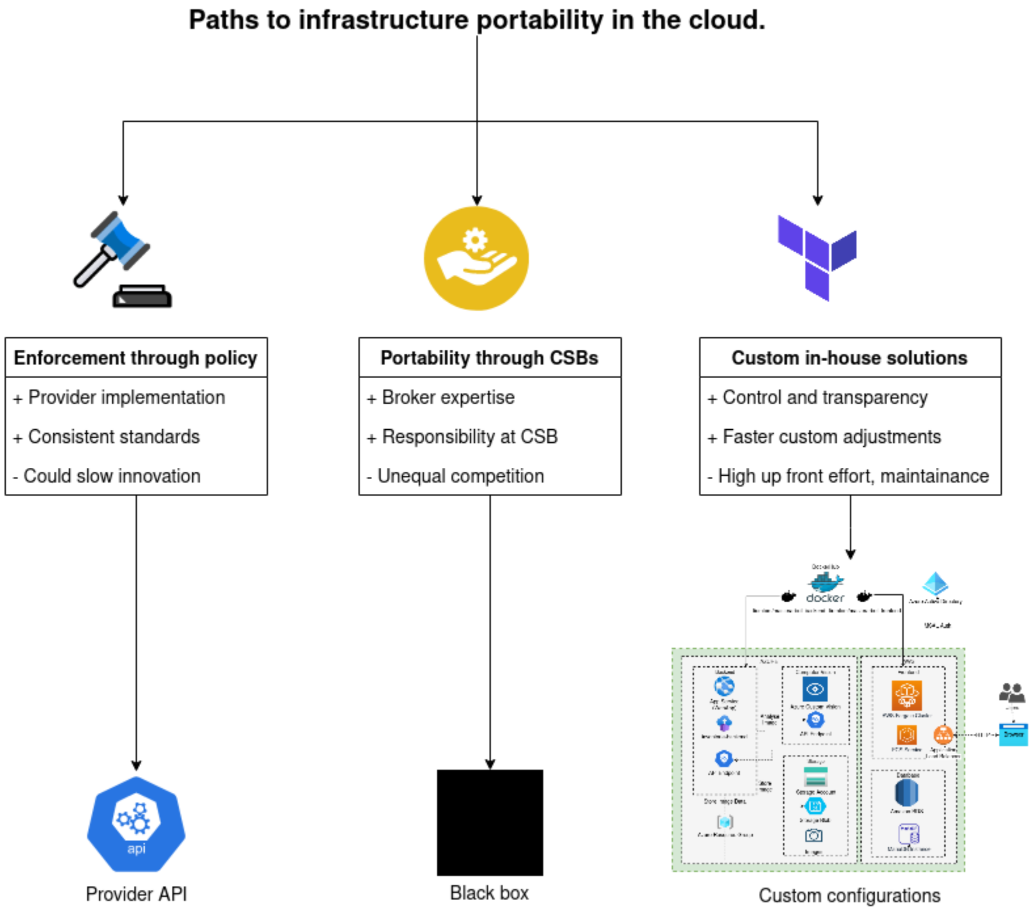

Paths to infrastructure portability in the cloud.
Enforcement through policy
+ Provider implementation
+ Consistent standards
- Could slow innovation
Portability through CSBs
+ Broker expertise
+ Responsibility at CSB
- Unequal competition
Custom in-house solutions
+ Control and transparency
+ Faster custom adjustments
- High up front effort, maintainance
api
Provider API
Black box
docker
Custom configurations

Figure 7.3: The paths that can lead to the portability of cloud infrastructures, identified in this thesis.

In figure 7.3, the paths that can lead to portability of infrastructures in the cloud are summarised. Only two of the proposed paths are possible at the time of writing. That is, on the one hand, the creation of portability manually, as it has been done in the practical part of this thesis. This method has the main advantages of having complete transparency and control over the architecture and being able to make adjustments quickly, without the need to involve another party, at the cost of having a high up front effort and manual maintenance. On the other hand, there is the use of cloud service brokers to manage the portability of an infrastructure. However, due to the current bleak situation regarding standards facilitating portability and the difficulty of custom implementations, offerings from CSBs are also limited. While the use of a CSB means that the responsibility for the implementation and maintenance is outsourced, it also means that the configuration generated by the CSB will likely represent a black box to the end user. Furthermore, the competition amongst providers will likely not be equal, due to the varying difficulty of translating infrastructures from one provider to another. Either there is no support for certain providers, or there might be a difference in pricing. Finally, there is the path of enforced standards through policy. This is arguably the best solution for overall competition, if it is executed in a way that does not stun innovation, a proposition can be found in chapter 6.3. The implementation of a standardised API by providers would take away a lot of the complexity, that currently makes multi-provider architectures so difficult to implement.

To conclude, the main research question “Can single applications be hosted on a multi cloud architecture using a high degree of automation (deployment and configuration) in an economical and feasible way in the current cloud infrastructure landscape?" can definitely be affirmed. While seamless portability, out of the box is still a long way away, the tools to manually create this portability exist. Achieving economic advantages is also possible, evidenced by the different price points of the configurations in chapter 5. Still, the effort that goes into the creation of such an application needs to be evaluated on a case to case basis, to decide whether the benefits of such a complex setup are necessary for the application.

Bibliography

[1] Statista, “Total public cloud providers in use worldwide 2021,” 2021. [Online]. Available: https://www.statista.com/statistics/1259793/public-cloud-providers-use-number-worldwide/ (Accessed 2022-02-21).

[2] HashiCorp, “Command: graph,” 2022. [Online]. Available: https://www.terraform.io/cli/commands/graph (Accessed 2022-04-06).

[3] Google, “Google Trends: Terraform vs. Cloudformation vs. Azure Bicep,” 2022. [Online]. Available: https://trends.google.com/trends/explore?date=today%205-y&q=Terraform,Cloudformation,Azure%20Bicep (Accessed 2022-05-19).

[4] T. Cerny, M. J. Donahoo, and M. Trnka, “Contextual understanding of microservice architecture: current and future directions,” *ACM SIGAPP Applied Computing Review*, vol. 17, no. 4, pp. 29–45, Jan. 2018. [Online]. Available: https://doi.org/10.1145/3183628.3183631 (Accessed 2022-04-20).

[5] OASIS, “Topology and Orchestration Specification for Cloud Applications Version 1.0,” 2013. [Online]. Available: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html#_Toc356403634 (Accessed 2022-04-07).

[6] Scalr, “Overview of Scalr — Scalr 1.0 documentation,” 2018. [Online]. Available: https://cmp.docs.scalr.com/en/latest/introduction/index.html#scalr-architecture (Accessed 2022-05-30).

[7] T. Rosado and J. Bernardino, “An overview of openstack architecture,” in *Proceedings of the 18th International Database Engineering & Applications Symposium*, ser. IDEAS ’14. New York, NY, USA: Association for Computing Machinery, Jul. 2014, pp. 366–367. [Online]. Available: https://doi.org/10.1145/2628194.2628195 (Accessed 2022-05-31).

[8] A. Brogi, A. Ibrahim, J. Soldani, J. Carrasco, J. Cubo, E. Pimentel, and F. D’Andria, “SeaClouds: a European project on seamless management of multi-cloud applications,” *ACM SIGSOFT Software Engineering Notes*, vol. 39, no. 1, pp. 1–4, Feb. 2014. [Online]. Available: https://doi.org/10.1145/2557833.2557844 (Accessed 2022-06-03).

[9] Statista, “Infographic: Amazon Leads $180 Billion Cloud Market,” 2022. [Online]. Available: https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/ (Accessed 2022-05-23).

[10] M. Lang, M. Wiesche, and H. Krcmar, “What are the most Important criteria for Cloud Service Provider Selection? A Delphi Study,” in *ECIS*, 2016.

[11] OpenStack, “Get Started with an OpenStack Public Cloud,” 2022. [Online]. Available: https://www.openstack.org/marketplace/public-clouds/ (Accessed 2022-06-01).

[12] Boomi, “iPaaS Solutions & Tools for Cloud Connected Business,” 2022. [Online]. Available: https://boomi.com/ (Accessed 2022-06-14).

[13] J. Opara-Martins, R. Sahandi, and F. Tian, “Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective,” *Journal of Cloud Computing*, vol. 5, no. 1, p. 4, Apr. 2016. [Online]. Available: https://doi.org/10.1186/s13677-016-0054-z (Accessed 2022-02-21).

[14] Cloudflare, “What is multi-cloud? | Multi-cloud definition,” 2022. [Online]. Available: https://www.cloudflare.com/learning/cloud/what-is-multicloud/ (Accessed 2022-04-11).

[15] "Regulation (EU) 2018/1725 of the European Parliament and of the Council of 23 October 2018 on the protection of natural persons with regard to the processing of personal data by the Union institutions, bodies, offices and agencies and on the free movement of such data, and repealing Regulation (EC) No 45/2001 and Decision No 1247/2002/EC (Text with EEA relevance.)," Oct. 2018, legislative Body: EP, CONSIL. [Online]. Available: http://data.europa.eu/eli/reg/2018/1725/oj/eng (Accessed 2022-08-08).

[16] D. Petcu, "Multi-Cloud: expectations and current approaches," in *Proceedings of the 2013 international workshop on Multi-cloud applications and federated clouds*, ser. MultiCloud '13. New York, NY, USA: Association for Computing Machinery, Apr. 2013, pp. 1–6. [Online]. Available: https://doi.org/10.1145/2462326.2462328 (Accessed 2022-04-11).

[17] SNIA, "Cloud Data Management Interface (CDMI™) | SNIA," 2022. [Online]. Available: https://www.snia.org/cdmi (Accessed 2022-04-13).

[18] The Apache Software Foundation, "Apache Libcloud is a standard Python library that abstracts away differences among multiple cloud provider APIs," 2022. [Online]. Available: https://libcloud.apache.org/ (Accessed 2022-04-13).

[19] WilliamDAssafMSFT, "Azure CLI example: Create a single database - Azure SQL Database," 2022. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-sql/database/scripts/create-and-configure-database-cli (Accessed 2022-03-31).

[20] M. Artac, T. Borovssak, E. Di Nitto, M. Guerriero, and D. A. Tamburri, "DevOps: Introducing Infrastructure-as-Code," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, May 2017, pp. 497–498.

[21] K. Morris, *Infrastructure as Code: Dynamic Systems for the Cloud Age*, 2nd ed. O'Reilly Media, Inc., Dec. 2020, google-Books-ID: UW4NEAAAQBAJ.

[22] HashiCorp, "azurerm_mssql_database | Resources | hashicorp/azurerm | Terraform Registry," 2022. [Online]. Available: https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/resources/mssql_database (Accessed 2022-04-01).

[23] Pulumi, "Pulumi vs. AWS CDK and Troposphere," 2022. [Online]. Available: https://www.pulumi.com/docs/intro/vs/cloud-template-transpilers/ (Accessed 2022-04-05).

[24] Y. Brikman, *Terraform: Up and Running: Writing Infrastructure As Code*. O'Reilly Media, Incorporated, 2019, google-Books-ID: R8FFzQEACAAJ.

[25] mumian, "Templates overview - Azure Resource Manager," 2022. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-resource-manager/templates/overview (Accessed 2022-03-31).

[26] AWS, "Provision Infrastructure As Code – AWS CloudFormation – Amazon Web Services," 2022. [Online]. Available: https://aws.amazon.com/cloudformation/ (Accessed 2022-03-31).

[27] HashiCorp, "What is Terraform," 2022. [Online]. Available: https://www.terraform.io/intro (Accessed 2022-04-05).

[28] Microsoft, "Releases azure bicep," 2022. [Online]. Available: https://github.com/Azure/bicep/releases (Accessed 2022-05-19).

[29] Eberhard Wolff, *Microservices: Flexible Software Architecture*. Addison-Wesley Professional, Oct. 2016, google-Books-ID: zucwDQAAQBAJ.

[30] Microsoft Corporation, "azure-storage-blob: Microsoft Azure Blob Storage Client Library

for Python," Mar. 2022. [Online]. Available: https://github.com/Azure/azure-sdk-for-python/tree/main/sdk/storage/azure-storage-blob (Accessed 2022-04-18).

[31] Microsoft, "Overview of the Azure Compute Unit - Azure Virtual Machines," 2022. [Online]. Available: https://docs.microsoft.com/en-us/azure/virtual-machines/acu (Accessed 2022-05-02).

[32] P. Lipton, D. Palma, M. Rutkowski, and D. A. Tamburri, "TOSCA Solves Big Problems in the Cloud and Beyond!" *IEEE Cloud Computing*, 2018, conference Name: IEEE Cloud Computing.

[33] OASIS, "TOSCA Simple Profile in YAML Version 1.3," 2020. [Online]. Available: https://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.3/os/TOSCA-Simple-Profile-YAML-v1.3-os.html (Accessed 2022-04-07).

[34] Neil McAllister, "Oracle rallies PaaS providers to float cloud interop spec," 2012. [Online]. Available: https://www.theregister.com/2012/08/30/oracle_camp_paas_interop_spec/ (Accessed 2022-06-03).

[35] Anish Karmarkar, "CAMP: a standard for managing applications on a PaaS cloud," in *Proceedings of the 2014 Workshop on Eclipse Technology eXchange*, ser. ETX '14. New York, NY, USA: Association for Computing Machinery, Oct. 2014, pp. 1–2. [Online]. Available: https://doi.org/10.1145/2688130.2688131 (Accessed 2022-06-03).

[36] OASIS, "OASIS Cloud Application Management for Platforms (CAMP) TC | OASIS," 2021. [Online]. Available: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=camp#announcements (Accessed 2022-06-03).

[37] Microsoft, "Discover, assess, and migrate Amazon Web Services (AWS) EC2 VMs to Azure - Azure Migrate," 2022. [Online]. Available: https://docs.microsoft.com/en-us/azure/migrate/tutorial-migrate-aws-virtual-machines (Accessed 2022-05-25).

[38] AWS, "Migrate a Microsoft Azure VM to Amazon EC2 using CloudEndure - AWS Prescriptive Guidance," 2022. [Online]. Available: https://docs.aws.amazon.com/prescriptive-guidance/latest/patterns/migrate-a-microsoft-azure-vm-to-amazon-ec2-using-cloudendure.html (Accessed 2022-05-25).

[39] Julie Bort, "Amazon struck a blow against Google by buying a tiny Israeli cloud company for a reported $200 million-plus," 2019. [Online]. Available: https://www.businessinsider.com/amazon-snatches-cloudendure-from-google-for-a-reported-200-million-2019-1 (Accessed 2022-05-25).

[40] Michael Eder, "Hypervisor-vs . Container-based Virtualization," 2016. [Online]. Available: https://www.semanticscholar.org/paper/Hypervisor-vs-.-Container-based-Virtualization-Eder/7b3752cddfcd4235e0e8f74e4499c8d033817480 (Accessed 2022-05-25).

[41] The Linux Foundation, "Open Container Initiative - Open Container Initiative," 2022. [Online]. Available: https://opencontainers.org/ (Accessed 2022-05-25).

[42] F. Lardinois, "Amazon Announces EC2 Container Service For Managing Docker Containers On AWS," Nov. 2014. [Online]. Available: https://social.techcrunch.com/2014/11/13/amazon-announces-ec2-container-service-for-managing-docker-containers-on-aws/ (Accessed 2022-05-25).

[43] Podman, "What is podman?" 2021. [Online]. Available: https://podman.io/whatis.html (Accessed 2022-07-04).

[44] SNIA, "S3 and CDMI™ A CDMI Guide for S3 Programmers | SNIA," May 2013. [Online]. Available: https://www.snia.org/educational-library/s3-and-cdmi%E2%84%A2-cdmi-guide-s3-programmers-2013 (Accessed 2022-07-05).

[45] SNIA, "CDMI-v2.0.0," Sep. 2020. [Online]. Available: https://www.snia.org/sites/default/files/technical-work/cdmi/release/CDMI-v2.0.0.pdf (Accessed 2022-07-04).

[46] OpenStack, "Heat - OpenStack," 2022. [Online]. Available: https://wiki.openstack.org/wiki/Heat (Accessed 2022-05-31).

[47] OpenStack, "Open Source Cloud Computing Platform Software," 2022. [Online]. Available: https://www.openstack.org/software (Accessed 2022-07-07).

[48] The Apache Software Foundation, "The Theory behind Brooklyn - Apache Brooklyn," 2020. [Online]. Available: https://brooklyn.apache.org/learnmore/theory.html (Accessed 2022-06-03).

[49] Gaia-X, "Gaia-X Home Page," 2022. [Online]. Available: https://www.gaia-x.eu/node/34 (Accessed 2022-06-01).

[50] M. Guzek, A. Gniewek, P. Bouvry, J. Musial, and J. Blazewicz, "Cloud Brokering: Current Practices and Upcoming Challenges," *IEEE Cloud Computing*, vol. 2, no. 2, pp. 40–47, Mar. 2015, conference Name: IEEE Cloud Computing.

[51] Grand View Research, "Cloud Service Brokerage Market Size Report, 2019-2025," 2018. [Online]. Available: https://www.grandviewresearch.com/industry-analysis/cloud-service-brokerage-csb-market (Accessed 2022-06-14).

[52] Precedence Research, "Cloud services market size to surpass 1630 billion usd by 2030," Oct. 2022. [Online]. Available: https://www.globenewswire.com/news-release/2022/06/10/2460529/0/en/Cloud-Services-Market-Size-to-Surpass-US-1630-Billion-By-2030.html (Accessed 2022-06-20).