DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

High-Level Cloud Architectures for Platform-Independent Serverless Applications

Michael Lohr

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

High-Level Cloud Architectures for Platform-Independent Serverless Applications

High-Level Cloud Architekturen für Plattformunabhängige Serverless Anwendungen

Author:	Michael Lohr, B.Sc
Supervisor:	Prof. Michael Gerndt, Ph.D
Advisor:	Anshul Jindal, M.Sc
Submission Date:	2022-08-31

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

Munich, 2022-08-31 Michael Lohr

Abstract

Cloud computing enables companies to quickly build scalable distributed systems without large initial capital expenses. Cloud service providers offer computing resources on-demand with a pay-as-you-go pricing strategy. Those resources can be used in a broad range of different fields, from factory automation and monitoring to complex shopping websites.

Some of the cloud services are considered “serverless”, meaning the underlying infrastructure is hidden from the user and managed by the cloud platform. Those services are typically unique to their cloud platform and require platform-specific knowledge and configuration. However, limiting an application deployment to only one cloud platform has various disadvantages, such as vendor lock-in.

There is no well-established universal standard for describing a cloud architecture that uses serverless services and can be deployed to multiple cloud platforms. The offerings of different cloud platforms often contain services that are similar in function but vastly different in usage and configuration, which is especially true with serverless products. This abstraction barrier is the reason why developing the same application for different platforms is expensive and time-consuming.

We propose a solution to this problem by providing a way to describe high-level cloud architectures in a generic, platform-independent way. Furthermore, a software tool that translates this generic architecture into platform-specific architectures is presented. These contributions enable developers to build applications that target multiple cloud platforms more efficiently.

Contents

1. Introduction

Cloud computing gained a lot of attention in the last few years, which has led to a significant part of the internet being built on cloud services. This technology is increasingly used to power a broad range of diverse services, from factory automation and monitoring to complex shopping websites. Two hundred seventy billion US dollars were spent in 2020 for cloud services, and Gartner predicts this number to increase over the next years [1].

With increasing popularity, new cloud platforms and services appear regularly. This leads to a problem that researchers and cloud developers have been trying to solve for over ten years [2]: How can multiple cloud platforms' services be used simultaneously for the same use case without significant development overhead?

This introductory chapter first explains the aspects that motivate our solution in Section 1.1. Following, in Section 1.2, is an enumeration of our contributions. Lastly, in Section 1.3, we give a short outline of the following chapters.

1.1. Motivation

"Serverless Computing" describes software services that are managed by a Cloud Service Provider (CSP) and can be used to build distributed applications running in the cloud [3]. They are similar to Platform as a Service (PaaS) and Software as a Service (SaaS) products and do not require any infrastructure setup. The advantages of serverless computing are described in Section 2.3. An example of such services are Function as a Service (FaaS) functions, which execute code on-demand without managing the underlying runtime environment, Operating System (OS) or hardware.

Using serverless products lowers the development and maintenance costs and allows for rapid development and deployment [4]. Developers use them to build complex software or distributed systems that are fault-tolerant and scale automatically [3]. They can spend more time developing the actual software instead of actively managing cloud infrastructure. Costs typically occur only for the resources that are actually used on a pay-per-use basis.

However, there are some difficulties when developing cloud applications using serverless products:

Many different configuration parameters have to be set while configuring cloud services. Since serverless products are designed with flexibility in mind, there are many parameters and often multiple cloud resources that have to be set up in order to employ a serverless

service in a cloud deployment. In most architectures, the same services are reused multiple times in a similar manner, with certain properties not used at all or with the same set of cloud resources together. A high-level abstraction could simplify the architecture and speed up the development process.

Platform-specific knowledge about the product and the cloud platform is required because serverless products are tightly integrated into their cloud platform. Most serverless concepts (for example, object storage offerings) can be found on every cloud platform as a service variant. They often hide behind different and unintuitive names like "Amazon S3" and "Microsoft Storage Container". Most service variants have different configuration parameters and interfaces.

Vendor lock-in is a major concern [5] of companies and hinders the adoption of cloud computing [4]. The high-level view of the same architecture on different platforms is often similar but vastly differs in the low-level details. The architecture must be re-engineered to deploy the same application on other cloud platforms. A platform-independent architecture specification could solve this issue by abstracting the small details and implementing sensitive platform-specific defaults for the specific use case.

1.2. Contributions

We propose a solution to the aforementioned issues. This solution solves the problem of specifying serverless cloud architectures without platform-specific knowledge while preventing vendor lock-in. It allows for high-lever abstractions and custom component definitions. The goal is to design an Infrastructure as Code (IaC)-based tool that translates a high-level, generic modeling language into deployment configurations for different cloud platforms.

The key contribution of this thesis is a software tool that transpiles a generic platform-independent architecture to platform-specific Terraform configurations. This tool is open-source, well-documented, and written in an accessible way using a widely known programming language. We also discuss multiple related approaches and suggest further areas of improvement.

Other contributions that supported the development of the transpiler tool are:

- IaC templates for multiple cloud platforms to deploy a serverless web application. This is used to compare and analyze the differences between each cloud platform.
- A Generic Architecture Modeling Language (GAML) that can be used to define generic high-level cloud architectures using IaC.
- Configurations for the transpiler to generate the platform-specific IaC templates for the serverless web application.

1.3. Outline

After introducing the motivation and contributions of this thesis, we provide a short overview of the chapters to come: Chapter 2 provides a more profound introduction to important topics of cloud computing together with problems they impose, relevant for the remaining part of this thesis. Based on those problems, Chapter 3 presents selected related work that approaches the cloud interoperability problem. Chapter 4 introduces the methodology we used to develop our approach. The implementation details of our core contribution, the transpiler, are shown in Chapter 5. Following, in Chapter 6, we present a similar solution developed by a startup during the same time as our approach. Next, our transpiler tool is evaluated in Chapter 7 based on a demo application. After this, Chapter 8 discusses possible future work to improve our contributions. Finally, we present our conclusion in Chapter 9.

2. Background

After introducing our motivation and contributions, this chapter gives a more detailed introduction to some specific cloud topics. In Section 2.1, we provide a quick overview of the history of cloud computing. Following in Section 2.2, we define cloud computing and explain its different aspects. The trends to employ serverless computing (Section 2.3) and multi-cloud environments (Section 2.4) are explained afterward. They impose a problem known as "cloud interoperability", as discussed in Section 2.5. Finally, Section 2.6 introduces a way to define infrastructure in a human-readable text-based format.

2.1. Historical Context

In 1961, John McCarthy, a pioneer in the field of artificial intelligence, already predicted that computing would eventually be implemented as a service: "Each subscriber needs to pay only for the capacity that he actually uses, but he has access to all programming languages characteristic of a very large system." [6]

However, the first notion of cloud computing, utility computing started with High-Performance Computing (HPC) in the 1980s, only accessible to the HPC research community [6]. With the emergence of the internet came the need for services that allow scaling quickly. Companies running services on the internet had to build their own data centers and design complex distributed systems to cope with the load [6]. Building a new service on the internet required large initial investments that most companies could not afford [6].

This changed when Amazon launched its object storage and compute platforms with their revolutionary "pay-as-you-go" pricing in 2016 [6]. It enables businesses to build reliable software without huge initial capital expenses for the required infrastructure. This headstart leads to the Amazon Web Services (AWS) platform dominating the market in the early years of cloud computing [6].

However, other platforms quickly caught up with AWS. The most popular CSPs nowadays are AWS[1], Microsoft Azure (Azure)[2], and Google Cloud Platform (GCP)[3] [7]. This competition in the cloud service market has led to lower prices and a wider range of different cloud-based services and products [6]. But this led to cloud differentiation, where every cloud platform implements its own services that are incompatible with others. AWS offers over 200 services

[1] https://aws.amazon.com/
[2] https://azure.microsoft.com/
[3] https://cloud.google.com/

to millions of customers [8]; while some of these services leverage open-source software that supports standardized formats, most of them are proprietary software built by Amazon specifically for their cloud platform. This led to new challenges in the current era of cloud computing.

2.2. Cloud Computing

Cloud computing describes an environment offering services for developers that can be provisioned on-demand while only paying for the resources actually consumed. Such cloud services are typically tools or infrastructure services that provide or are based on computing, networking, and storage resources. Developers use these services to build fault-tolerant and highly available distributed systems and applications.

By leveraging a cloud platform and its services, developers build systems that can scale to any workload without managing the underlying infrastructure. The resources are often cheap, available in small quantities, and can be provisioned whenever needed. Only the demanded resources are billed, and resource requirements can be changed at any time. The National Institute of Standards and Technology (NIST) describes cloud computing as "[...] a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." [9]. CSPs are able to provide this offering because of the economies of scale and ability to share resources between different consumers.

More and more companies invest in the migration of their existing infrastructures to the cloud because it allows them to build and maintain reliable products and services. Cloud services typically implement or help to deploy software that implements the following aspects:

- *Agility*: New resources can be deployed quickly
- *Scalability*: Complex infrastructure can be scaled up and down
- *Elasticity*: Applications scale up or down automatically to any workload
- *High Availability*: Services are available without service interruptions
- *Fault-Tolerance*: Systems remain up during a fault
- *Cost-Effectiveness*: Low prices; Only consumed resources are billed

The services offered available on cloud platforms can be categorized as one of the three service models Infrastructure as a Service (IaaS), PaaS, or SaaS [9]. Cloud services that offer fundamental computing, network, and storage resources to consumers on-demand are called IaaS [10]. The user controls the OS, storage, and software but not the underlying hardware infrastructure [9]. Cloud platforms enable consumers to build their own virtual data center in the cloud worldwide without up-front capital expenditure.

PaaS runs applications developed by the consumers, so they only manage the deployed

software as well as configurations [10]. The CSP manages the underlying infrastructure, like server configuration, and the OS [10]. NIST defines it as "The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure, including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment" [9].

Software applications that run in the cloud and can be accessed on-demand over the internet are categorized as SaaS. Here, the consumer only uses the software and does not develop and maintain the software himself. IaaS, PaaS and SaaS offerings have different levels of management overheads and control as can be seen in Figure 2.1.

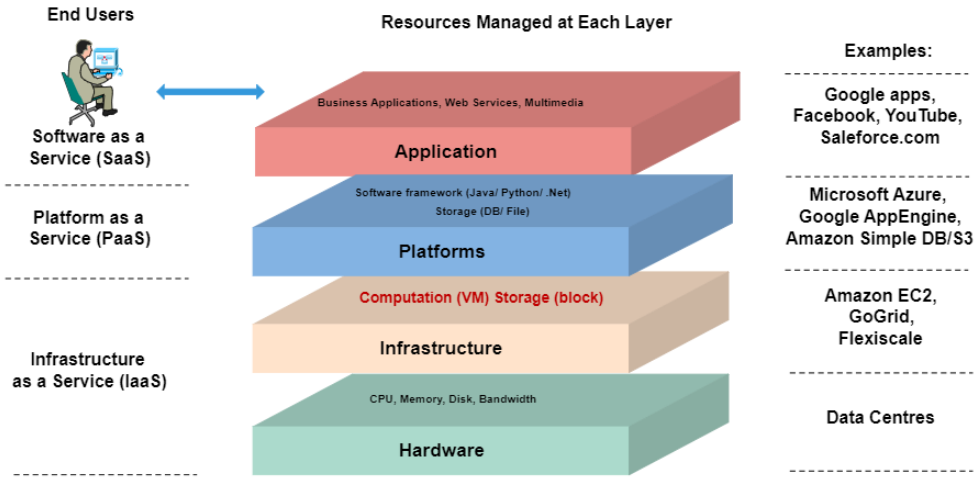


Figure 2.1.: The different aspects the consumer has to manage when using traditional IaaS, PaaS, SaaS offerings. Figure taken from [11] based on [12].

When deploying an application to an IaaS offering (e.g., a simple web application), the consumer provisions a Virtual Machine (VM), configures its storage and network, and finally installs the application. This deployment and setup process differs on each cloud platform; however, the rough steps will remain the same. The disadvantage of this approach is that consumers have to maintain the VM (e.g., patching and updating the OS) themselves. However, with PaaS and SaaS solutions, consumers do not have to update, patch, or maintain the machines. While the process to deploy IaaS resources is similar on different cloud platforms, for PaaS and SaaS services it is vastly different.

2.3. Serverless Computing

"Serverless Computing" (or simply "serverless") describes a concept similar to PaaS; however, it does not describe an application deployment platform but rather a set of cloud services

that can be used to build applications. Those services are fully managed like PaaS and SaaS products and do not require the consumer to perform any server setup or infrastructure management.

Before cloud computing, the term "serverless" was used in the context of certain network architectures [13]. In 2014, during the yearly AWS conference, Amazon introduced the roots of their serverless platform (the AWS Lambda product), which back then, was categorized as PaaS [3]. But after more fully managed services were developed, the term "serverless" gained popularity, as is reflected in the interest of the search term visualized in Figure 2.2. Nowadays, the term is sometimes described as an evolution of PaaS [3].

Figure 2.2.: *Google trends* measures the search popularity for certain search terms. This chart displays the history of the interest (Y-axis) in serverless computing over time (X-axis) [14]. "Interest" is a relative measure that describes how often the term was searched relatively to the maximum search count.

The term serverless computing is often used to describe FaaS products. But while FaaS is an important offering that categorizes as serverless, it is not the only one anymore. Today, services that are FaaS or Backend as a Service (BaaS) are categorized as serverless [15]. BaaS describes tools and services that run in the cloud where the consumer does not manage the backend as illustrated in Figure 2.3. Those services are managed serverless products (e.g., FaaS, message queues, databases, authentication systems, and other building blocks) that integrate tightly with other services on their platform.

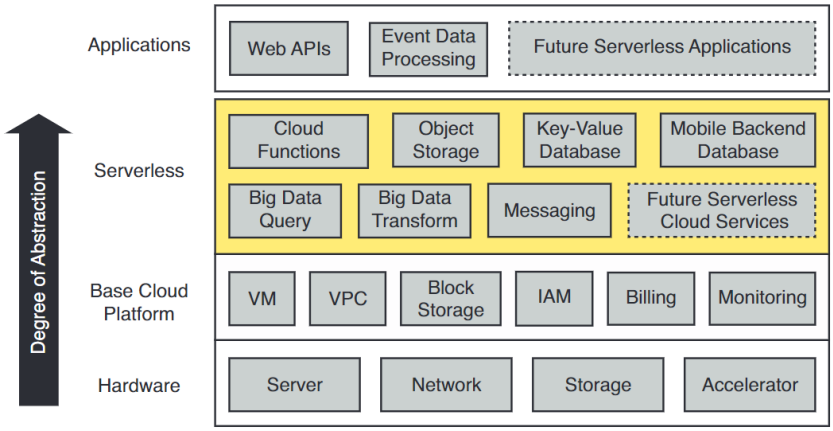


Figure 2.3.: The architecture of a serverless cloud with the different abstraction layers from [15]. Serverless products are built using regular cloud services like VMs and block storage and hide those deeper layers from the consumer. Cloud applications are often built using serverless products and therefore are placed in a higher abstraction layer than serverless.

There are three key distinctions between serverless and conventional computing according to [15]:

1. *Decoupled computation and storage*: Storage and compute services scale independently and are billed separately.
2. *Executing code without managing resource allocation*: The cloud platform automatically provisions the resources required to execute the code.
3. *Paying in proportion to resources used instead of for resources allocated*: Only the resources/dimensions that are actually used (e.g., execution time) for the specific workload are billed, not the required base infrastructure (like VMs) that has to be provisioned.

Serverless services can typically be used without worrying about scalability, elasticity, availability, and fault tolerance because those aspects are managed by the CSP. This makes it easier and more cost-effective to develop applications for the cloud. The history of serverless and the current trend to employ them in modern cloud architectures leads to the prediction that the adaption of serverless will increase drastically over the next few years [15]. It is, therefore, important to make serverless more accessible.

2.4. Multi-Cloud

Cloud deployments employing multiple CSPs are typically described by the umbrella term "multi-cloud". Businesses with workloads in the cloud will often arrive at a multi-cloud setup eventually [16]. There are two common use cases [16]: The first use case is mirroring a similar architecture with the same functionality across CSPs for backup or redundancy and preventing vendor lock-in by building applications "cloud-agnostic" [17], [18]. The second

use case is to use different services from multiple CSPs together to build one application to expand the available service landscape [11].

Often the term "multi-cloud" is used to describe one of the two uses cases. This thesis focuses on the first use case and uses the term "multi-cloud" to describe scenarios where an application is written in a cloud-agnostic way where it supports multiple cloud platforms.

Applications built with tools from the cloud services are typically fault-tolerant and have high availability, but outages can still happen [19]. Multi-cloud approaches are therefore popular because they protect the software from global outages of one CSP [16]. If some CSP has an outage affecting all instances of a certain service, it makes sense to just switch the workload over to a similar service from a different CSP. Similar services of the same category that exist on different cloud platforms are named "service variants" (e.g., Amazon S3 and Azure Storage Container for object storage).

The elasticity aspect, which describes quickly scaling to the current demand, can be viewed in a similar manner. However, Paraiso, Merle, and Seinturier differentiate between the fine-grained and coarse-grained approach [20]. Elasticity on the fine-grained level is achieved by vertical or horizontal scaling [20]. Vertical scaling means upgrading existing machines (like adding more memory units or disks), while horizontal scaling describes adding more machines to the compute pool. The coarse-grained approach manages elasticity by switching CSP or using multiple CSPs at the same time [20]. This can also help with outages and improve high availability.

Those approaches also make it easier to prevent vendor lock-in and switching CSPs later (for example, because of financial or legal reasons) since only part of the architecture or a copy of it runs on one platform [16]. Businesses can also save money or profit from higher service quality since they are free to choose which CSP to use for which functions and with which capacity. [16]. Also, by deploying service variants across CSPs, service or scaling limits can be stretched.

However, implementing a multi-cloud approach is expensive because several cloud platforms have to be managed at the same time [16]. This is especially the case when serverless or PaaS products are used since those are often very specific to their cloud platform. In such a scenario, a system is required that makes the different interfaces between the cloud service variants compatible. There are different approaches to solving these interoperability challenges, which are discussed in Section 2.5.

This leads to the conclusion that a multi-cloud approach using the CSP's specific serverless products is desired but a challenging undertaking. To ease the development and lower the initial costs, this thesis proposes a high-level abstraction layer that is based on IaC to transpile a generic cloud architecture to multiple platform-specific architectures. With this strategy, users do not need specific knowledge of the different cloud platforms to deploy their multi-cloud workload. This allows for a multi-cloud architecture with the advantages of managed cloud-native products without relying on an additional abstraction layer running in the cloud.

2.5. Cloud Interoperability

Cloud interoperability is understood as the ability to combine cloud resources from multiple CSPs that can work together, hence taking advantage of specific features provided by each CSP [21]. However, this ability is also useful for multi-cloud deployments since consumers can keep their options open and prevent vendor lock-in by building interoperable applications. The major CSPs use their monopoly to introduce their own formats and standards, which makes it hard to switch CSP later [22], [23]. Since there is currently little to no standardization effort from the major CSPs, it is necessary to build an abstraction to provide platform-independent cloud resources [4]. This concept is often called "sky computing" since it is an abstraction layer higher than conventional cloud platforms [6], [24].

Because the workloads of products in the IaaS service model are often represented as self-contained VM images and storage units, they can benefit from standardization [4]. In theory, those basic building blocks can be moved if there would be a tool to export the VM image and network configuration to a standardized format as well as access the storage Application Programming Interface (API) using a standardized protocol [4]. There are a lot of different solutions that tackle these problems as presented in [5], [25], [26].

Consumers must commit to one platform to fully profit from serverless or PaaS-based products. This is because serverless products typically are technologically diverse and tightly integrate into their cloud platform. While there might be some cloud offerings that support open standards (such as OpenAPI[4] or SQL[5]), most solutions are proprietary and specific to the cloud platform.

SaaS products, on the other hand, are specific applications. Internally they might use very different data models and process the workloads with unique logic. The only area where the consumer would benefit from a standardized interface is the data export in applications of the same kind (e.g., bookkeeping software) [4]. It is not uncommon that SaaS software already offers such export functions[6]. Without such an interface, users have to download the data from one application, transform it into a different format and then upload it to the other application.

In practice, there are two approaches to implementing cloud-agnostic applications for the IaaS and PaaS service models: service brokering and standardized interfaces [27]. Service brokers are an additional layer that runs between the cloud and the workload to translate the communication between different cloud interfaces. The more straightforward solution is standardized interfaces, which have to be implemented by the CSPs. This is unlikely to happen because of the following barriers [2]:

- Vendor lock-in prevents existing customers from migrating to their competitors

[4] https://swagger.io/specification/

[5] SQL is a query language used to interact with relational databases.

[6] For example, SaaS bookkeeping software often supports the export and import of its data using the DATEV format.

- CSPs offer differentiated services with their own unique selling points to attract costumers
- CSPs might not agree on certain standards
- Developing and agreeing on such standards takes a lot of time
- There are already multiple standards - consensus on which to adopt may be difficult
- Each service model (e.g. IaaS, PaaS, SaaS) would need different standards

While the CSPs will likely not implement standardized interfaces for serverless products in the next few years, there are some advances in terms of IaaS: By deploying a minimal architecture to each cloud platform, a system can then run services that are packaged using a standardized application format. Containers (such as Docker containers[7]) standardize software deployments that include all dependencies and configurations. There are approaches like [28] that use containers together with a custom orchestrator and adapters that run on each cloud platform. However, the orchestrator and adapter, as well as the infrastructure they are deployed to, have to be managed by the consumer.

A cloud broker forms an additional middleware layer that provides a platform-independent interface and typically enables the communication between applications hosted on different cloud platforms. NIST classifies the services of a cloud broker into three activities: arbitration, aggregation, and intermediation [29]. Service intermediation describes a broker that enhances an existing cloud service [29]. Service aggregation describes when a new service is created by combining other existing cloud services [29]. Service arbitrage is similar to but more flexible than service aggregation because it allows the broker to select services on-demand based on the specific workload [29].

Cloud brokers can help with interoperability issues by providing a platform-independent interface that can be implemented for serverless and PaaS services. However, they are difficult to realize such services are particularly specific to their platform and have different properties and processes. Containers can be used to abstract the infrastructure but cannot generalize serverless offerings. Cloud brokers solve the interoperability problem by translating the requests from the consumer to the platform-specific API of the CSP. They often use the lowest common denominator approach, where only the features that all supported cloud platforms implement are supported. However, more established approaches will provide the capability to override platform-specific parameters.

2.6. Infrastructure as Code

NIST defines Infrastructure as Code as “the process of managing and provisioning an organization’s IT infrastructure using machine-readable configuration files, rather than employing physical hardware configuration or interactive configuration tools.” [30]. IaC allows to provision systems and to change their configuration by using consistent, repeatable routines [31].

[7]Docker is a Linux-based platform for running applications through OS-level virtualization: https://docker.com

The files of IaC projects are typically maintained by multiple developers and are put under version control. This enables to post-process infrastructure changes and the use of automation tooling to test and apply those changes automatically [31]. It is mostly used with IaaS and PaaS offerings.

The benefits of using IaC to manage cloud infrastructure are manifold and include, but are not limited to [31]:

- Enables rapid delivery of value
- Reducing effort and risk of changing the infrastructure
- Providing common tooling across development, operations, and other stakeholders
- Creating systems that are reliable, secure, and cost-effective
- Make governance, security and compliance controls visible
- Improve the speed of troubleshooting and resolving failures

In order to use an IaC approach successfully, three core practices should be considered. The first practice is to define every aspect of infrastructure as code, which allows for making changes rapidly and reliably [31]. This ensures that individual components can be reused and or adapted to different situations [31]. It also makes the architecture more consistent because code ensures a certain structure and syntax [31]. The system will also be more transparent since it is easy to inspect and audit the code as well as to suggest improvements [31]. The second practice is to continuously test and deliver all work in progress by implementing Continuous Integration (CI) and Continuous Delivery (CD). The last practice is to build small, simple pieces that can be changed individually.

According to Brikman, IaC tools can be put into five broad categories [32]:

- *Ad hoc scripts* that use a scripting language to automate individual steps
- *Configuration management tools* are designed to install and manage software on existing servers
- *Server templating tools* are tools like Docker that create self-contained “images”
- *Orchestration tools* like Kubernetes that manage VMs and containers
- *Provisioning tools* that deploy resources (for example a VM) in the cloud

Such provisioning tools can be used to deploy most cloud resources and are not limited to IaaS services. Most CSPs offer their own tool specific to their platform. However, there are also tools by third parties that work with multiple CSPs. They can be used to deploy serverless services, but they are not platform-independent and require specific knowledge about a serverless product. However, similar to cloud brokers, provisioning tools (that support multiple platforms) provide a homogeneous interface that can be used to deploy resources on different platforms using the same language. This makes them a great tool to use in a multi-cloud scenario, even though they do not solve the interoperability challenges.

3. Related Work

Different approaches to solving the aforementioned interoperability problem have already been developed: Section 3.2 presents a multi-platform modeling language used to specify cloud application architectures. The approach presented in Section 3.3 uses Kubernetes to provide cloud provider abstraction. Virtual Serverless Providers (VSPs) (Section 3.4) is an approach that provides a custom platform to hide the differences by implementing an abstraction layer and automatically choosing the best CSP for a specific workload. Section 3.7 presents a multi-layered solution for the whole lifecycle of a cloud application: It allows the deployment, management, and migration of PaaS offerings. Following in Section 3.6 is the Open-Source API and Platform for Multiple Clouds (mOSAIC) project, which introduces its own abstraction for computing services that are dynamically executed on the best CSP for that task. Finally, we present a software library that provides high-level cloud service abstractions for the programming language Python in Section 3.5.

3.1. OCCI

The Open Grid Forum[1] launched the Open Cloud Computing Interface (OCCI) in 2010 [33]. It is described as "a RESTful Protocol and API for all kinds of Management tasks." [34], since it defines a standard for a web-based interface that can be used to manage the deployment, autonomic scaling, and management of various cloud resources of the IaaS [35], PaaS [36] and SaaS service model [34]. OCCI is used as a platform-independent abstraction layer in front of the different CSP APIs as can be seen in Figure 3.1.

The supported PaaS resources are limited to only a few resources: containers, databases, and routers are currently available [36]. This can be used to build small serverless applications but does not allow to use more platform-specific cloud products.

There are multiple platforms implementing this standard. However, none of the big CSPs support it. Nevertheless, there are OCCI servers like rOCCI[2] that implement a bridge between the AWS API and the OCCI interface [37].

[1]The Open Grid Forum (OGF) is an open global community that works on standardizing grid computing. Website: https://ogf.org/ogf/doku.php

[2]https://github.com/the-rocci-project/rOCCI-core

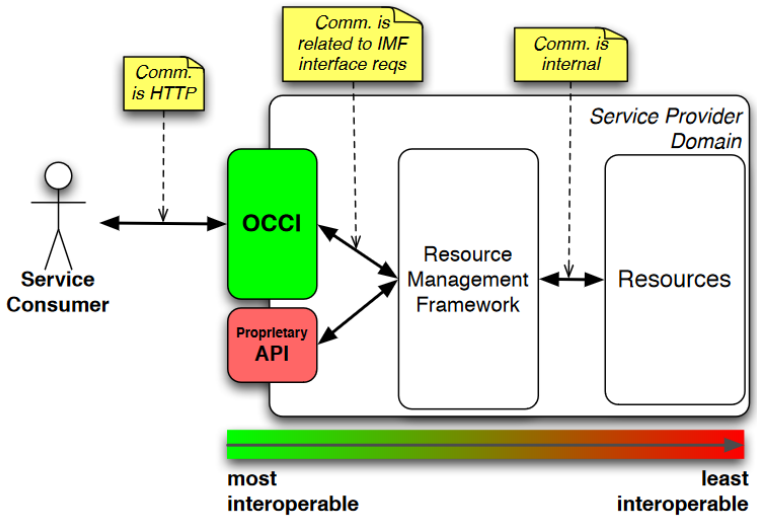


Figure 3.1.: The place of OCCI in a CSP's architecture from [34] showing the different interoperability levels: OCCI provides a fully interoperable interface; The actual cloud resources are located at data centers of CSPs with their own proprietary APIs.

3.2. TOSCA

The OASIS[3] Topology and Orchestration Specification for Cloud Applications (TOSCA) provides a language to describe and manage complex cloud applications in a portable and CSP-agnostic way [38]. It proposes a modeling language that can be used to specify the topology and management tasks of a cloud application [39]. Applications defined with TOSCA can operate in different cloud environments if these environments support the specified feature set (for example, programming language runtimes) required by the application [40].

The TOSCA modeling language is multi-platform, cross-technology as well as human and machine-readable [39]. It aims to address automated deployments, portability, and management of application descriptions, as well as interoperability and reusability of components [38]. This open standard removes the need to work with CSP platform-specific APIs and protocols [39]. Nevertheless, it does not implement the "lowest common denominator approach" [39]. Only features that all cloud platforms implement can be specified in such an approach. However, TOSCA applications can specify requirements that only apply to certain environments.

TOSCA orchestrators such as OpenTOSCA [41] are runtimes that execute TOSCA applications (similar to OCCI servers). They can run on-premise or on top of a CSP and deploy virtual machines to provide runtime environments as specified by workflows [42]. CSPs can also

[3]OASIS is an internationally recognized consortium that develops open standards. Website: https://oasis-open.org

become TOSCA compliant by allowing the deployment of applications specified with TOSCA.

Those applications are represented by different components as depicted in Figure 3.2. Workflows are often generated by the orchestrator using the component topology. They specify how to set up the service template (which represents an application) in a given environment [42]. Policies allow specifying additional non-functional behavior like monitoring strategy, payment conditions, or scalability [42].

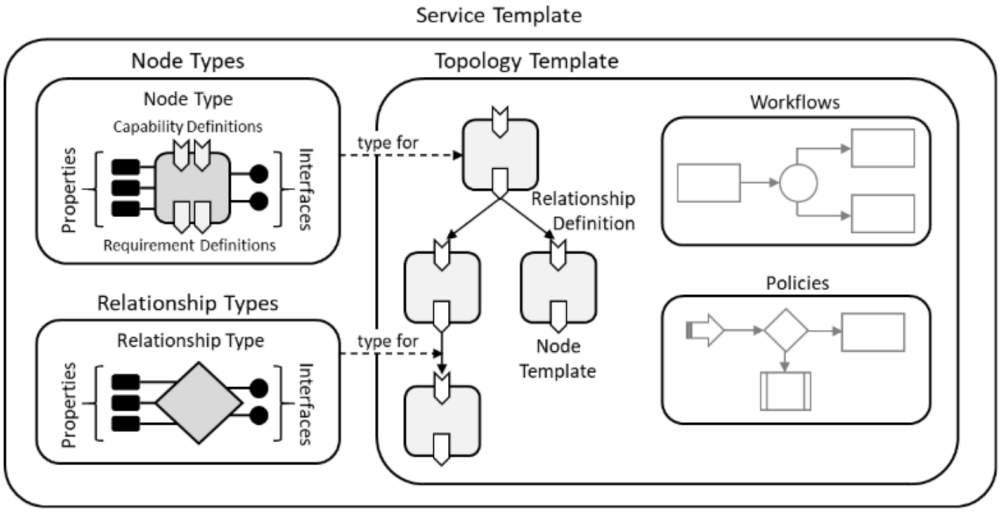


Figure 3.2.: The TOSCA (version 2) service template as presented in [42]. Nodes represent application components that have relationships (for example, dependencies) with other nodes. Workflows specify how to deploy an application. Policies specify non-functional behavior like monitoring or scaling behavior.

The TOSCA modeling language is based on `YAML`, which makes it easy to parse and write. This language characterizes an application (or "service template") as can be seen in the example in Appendix A. The service template is used to define the application's structure and management behavior. The `node_template` section of the `topology_template` defines the different components (or "component nodes") whose category is defined by the `type` [42]. The category or "node type" determines the properties and operations available to manipulate that component [42]. Additionally, nodes can specify requirements for the execution environment or dependencies on other nodes.

TOSCA orchestrators come with a library of built-in components, some of them also containing serverless offerings like containers and object storage. However, we found no library containing platform-specific serverless services that are more integrated into their own platform, like queueing systems. Custom components can be implemented if the orchestrator supports this, but often by inheriting existing components to extend the functionality.

Both OCCI and TOSCA offer platform-independent models and management for cloud resources [33]. They come with a limited set of platform-specific serverless resources and focus more on IaaS. However, TOSCA aims to be a system for interoperable infrastructure,

while OCCI focuses on the management part. They can be used together: TOSCA for the cloud resource model and OCCI to abstract the CSP APIs [33].

3.3. Kubernetes-Based Abstraction Layer

While OpenTOSCA uses VMs to execute cloud resources specified in a standardized format, Pellegrini, Rottmann, and Strieder presents a solution that relies on Docker and Kubernetes to provide an abstraction layer [5].

Kubernetes[4] is a container orchestration platform that can help with the deployment, scaling, and management of containerized workloads. Many cloud providers offer Kubernetes as a managed solution. By using such offerings, the consumer is responsible for configuring his containers and their environment in the cluster but not the Kubernetes cluster itself.

Pellegrini, Rottmann, and Strieder bypass vendor-lockin for IaaS offerings by adding an abstraction layer that is implemented by deploying Kubernetes clusters on VMs of the different cloud platforms [5]. With this layer of abstraction, the consumer's software could be developed for a Kubernetes environment and then be homogeneously deployed to the Kubernetes clusters running on each cloud platform. Unified management of those clusters is achieved by sharing the configuration files and images to a central repository [5]. They conclude that a Kubernetes-based abstraction layer enables portability, interoperability, and automized deployment within a multi-cloud environtment [5].

However, users have to maintain their own containers and cannot take advantage of fully managed serverless offerings.

3.4. Virtual Serverless Provider

Baarzi, Kesidis, Joe-Wong, *et al.* introduce the concept of a VSP, which is a third-party entity that aggregates serverless offerings from multiple CSPs to prevent vendor lock-in while profiting from the advantages of serverless offerings [43].

Combining multiple serverless platforms and being able to choose from them dynamically comes with multiple advantages: Service variants are often provided by multiple cloud platforms. They serve the same function but differ in details, such as pricing schemes or performance, which can be leveraged to the consumer's advantage. For example, a FaaS function might be more expensive for burst operations than long-running workloads, which could be the opposite on a different platform. A VSP can leverage this and choose a specific cloud platform to optimize for cost or performance. With this approach, one also profits from other multi-cloud advantages such as stretched service and scalability limits [43].

Baarzi, Kesidis, Joe-Wong, *et al.* acknowledge that serverless products are often tightly coupled

[4] https://kubernetes.io/

to other services of the same platform [43]. This increases vendor lock-in but can be prevented by using self-hosted alternatives [43]. They are especially useful to be used as bridges between different cloud platforms.

This approach requires controller software that maps resource requests to the different providers according to the optimization results of a scheduler. In their implementation, their VSP only supported FaaS functions across different CSPs where they were able to increase performance while reducing the costs to up to 54% [43].

3.5. Apache Libcloud

Apache Libcloud[5] is a Python library that implements a platform-independent API wrapper. It provides a unified API for several CSPs [44], which works by translating the API calls to the platform-specific. It also follows the lowest common denominator approach and provides high-level platform-independent abstractions [45], as seen in Listing 3.1.

```
from libcloud.dns.types import Provider, RecordType
from libcloud.dns.providers import get_driver

cls = get_driver(Provider.ZERIGO)
driver = cls('<email>', '<api key>')

zones = driver.list_zones()
zone = [zone for zone in zones if zone.domain == 'mydomain.com'][0]

record = zone.create_record(name='www', type=RecordType.A, data='127.0.0.1')
```

Listing 3.1: An example using the Apache Libcloud Python library to create a Domain Name System (DNS) record on the Zerigo DNS platform. The code sets up the required platform credentials and looks for a certain DNS zone object. It then creates an A-record in that zone. Source: [44].

Since Apache Libcloud is a software library, it requires changes in the code to add support for new CSPs or cloud services. It mainly supports service variants from the following four categories [45]:

1. Cloud servers and block storage
2. Cloud object storage and Content Delivery Network (CDN)
3. Managed load balancers
4. Managed DNS services

[5] https://libcloud.apache.org/

Apache jclouds[6] is a similar library for Java also with the objective to provide a high-level API [45], [46]. Both approaches are limited to a specific set of programming languages.

3.6. mOSAIC

The mOSAIC project[7] tries to provide simple access to IaaS- and PaaS-based heterogeneous cloud computing resources as well as tackle the vendor lock-in problem [47]. It was initiated in 2009 [48] and is funded by the European Commission [47].

MOSAIC is a so-called "broker-based" approach. Cloud brokers (or "intercloud brokers") provide a single interface through which multiple CSPs can be managed and share resources [49]. Cloud brokers themselves can be viewed as self-hosted PaaS systems providing access to cloud resources [48]. The main purpose of a cloud broker is twofold: They help the consumer to find the best CSP for the given task in terms of pricing and Service-Level Agreement (SLA); They also provide a uniform interface to manage the lifecycle of the deployed services [50]. The mOSAIC framework consists of two main components that implement these purposes: The semantic engine and the dynamic discovery and mapping system [45].

The semantic engine provides platform-independent representations of cloud resources [45]. It provides a high-level semantic-based abstraction of the CSP APIs [45]. The dynamic discovery service discovers the CSP's resources and aligns them to the mOSAIC API [45].

3.7. Cloud4SOA

Cloud4SOA has the goal of reducing the semantic interoperability barriers between different CSPs by employing a Service Oriented Architecture (SOA) [23]. Like the mOSAIC project from Section 3.6, it is broker-based as well and also partially funded by the European Commission [23]. It introduces an approach to homogenizing the diverse and heterogeneous collection of capabilities offered by PaaS providers [23].

The core capabilities of Cloud4SOA are the management of cloud applications in a homogenized way, migration from already deployed applications, unified platform-independent monitoring, and semantic matchmaking to find the best PaaS offering. By analyzing the requirements of an application, the matchmaking functionality of Cloud4SOA can look for the best service offering that matches the specified requirements [23]. The algorithm implementing this functionality is also able to identify equivalent concepts (like different units to specify CPU resources) in different services [23].

The project's reference architecture consists of five layers [23], [51]:

1. The front-end layer is represented by a web-based user interface that provides a dash-

[6] https://jclouds.apache.org
[7] https://mosaic-cloud.eu/

board for configuration and management

2. The semantic layer contains models representing information about the PaaS offerings.
3. The SOA layer acts as a mediator to the other layers' services, letting the front-end layer access core functionality. It translates information from the semantic layer into a high-level architecture.
4. The governance layer enables lifecycle management. It measures performance and mitigates violations.
5. The distributed repository layers provide a harmonized API that enables the interconnection and management of different PaaS offerings.

While similar to the mOSAIC project, it focuses more on PaaS. The homogenized API, which covers functionalities offered by the majority of PaaS providers uses "adapters" to implement the bridging logic [23]. The API is generic so that it can abstract the differences between the different cloud platforms. However, adding custom components requires programming adapters that contact the CSP's API to manage the resources. It should also be noted that the project seems to be abandoned since the last contribution in their GitHub organization[8] is from 2016.

[8]https://github.com/orgs/SeaCloudsEU/

4. Methodology

The previous section introduces different approaches to solving the interoperability challenges. In this chapter, we want to break the problem down and discuss our approach to solving the problem of building cloud-agnostic applications. In Section 4.1 we present Terraform, an IaC based tool that we employ to abstract the different cloud platform APIs. We use it to specify the cloud infrastructure for an exemplary web application targeting multiple CSPs as described in Section 4.2. This helps to analyze the differences between the different platforms and therefore supports the development of the transpiler tool.

4.1. The Terraform Layer

The challenge of building cloud-agnostic applications can be split into two subproblems or layers, as seen in Figure 4.1. The first layer handles high-level platform-independent resources and will be discussed shortly. The second layer (the "API abstraction layer") abstracts different cloud APIs to provide a homogenized way of requesting resources on the various cloud platforms. A unified API is used to translate the resource requests to the APIs offered by the CSPs.

The first layer abstracts the resource itself and its implementation details to remove the need for platform-specific knowledge of the cloud services. For example, the AWS S3 object storage and Azure storage container are abstracted to an object storage resource that provides the functionality of the lowest common denominator of both resources.

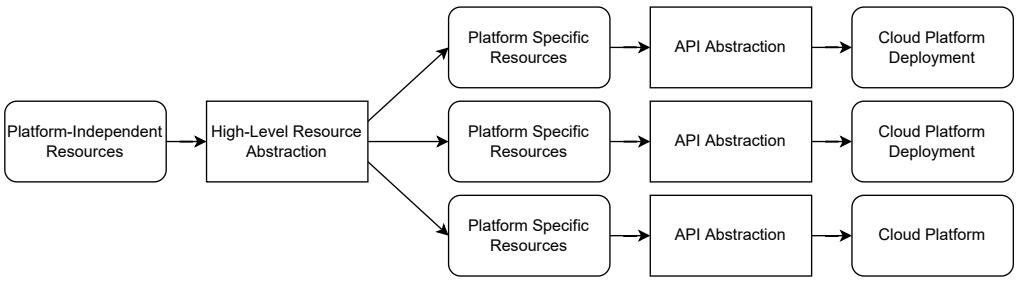


Figure 4.1.: The two abstraction layers of the platform-specific APIs. The first layer ("High-Level Resource Abstraction") is the high-level abstraction layer that transforms platform-independent resource definitions into platform-specific resource requests. Those platform-specific resources are then processed by the second layer ("API Abstraction") that translates those into API calls to the specific platform.

In the following chapters of this thesis, we present our implementation of the first layer. In order to offer a complete solution, we have to take care of the second layer as well. There are many different ways to achieve this API abstraction. Our approach leverages a commonly used software tool to provide a mechanism for the complete deployment process by separating both layers and implementing a solution to the first subproblem that outputs the input to the second layer.

This aforementioned software tool is Terraform[1], which is an open-source IaC tool that allows deploying complex cloud architectures. Terraform configurations are either written in `JSON` or a declarative configuration language named HashiCorp Configuration Language (HCL) [32]. Terraform HCL, as seen in Listing 4.1, can be deployed to different CSPs to provision different resources.

```
resource "aws_instance" "my_instance" {
  ami           = "ami-0e081ed4ce61c1fb2" # Ubuntu 18.04 LTS
  instance_type = "t2.micro"              # AWS EC2 instance type
}

resource "google_dns_record_set" "my_a_record" {
  name         = "demo.example.com"
  managed_zone = "my-zone"
  type         = "A"
  ttl          = 300
  rrdatas      = [aws_instance.my_instance.public_ip]
}
```

Listing 4.1: A simple Terraform example in HCL that provisions a `t2.micro` AWS EC2 instance with an Ubuntu VM image together with an DNS record in the GCP's DNS service linked to the EC2 instance. Example inspired by [32].

A notable difference to other IaC tools (like Chef[2], Puppet[3] and SaltStack[4]) is that Terraform does not require a master server for storing the state of the infrastructure nor an agent software running on each server [32]. Terraform requires "providers" that interface with the different cloud platforms directly.

Terraform providers are extensions that implement abstractions for cloud platform-specific APIs. They are available for a majority of the cloud services available on the well-known CSPs. Those providers expose the functionality of a cloud as Terraform resources so they can be used in Terraform IaC templates.

Templates contain "blocks", whose structure can be seen in Listing 4.2, that can be of different types such as resources, input variables, output values, and data sources [52]. The resource

[1]https://terraform.io/
[2]https://chef.io/
[3]https://puppet.com/
[4]https://saltproject.io/

type is used to interface with the aforementioned providers. Depending on the block and resource, some properties are required, and some are optional, which can be looked up in the corresponding documentation. The properties of other blocks can be referenced with the syntax `<BLOCK TYPE>.<BLOCK LABEL>.<IDENTIFIER>`.

```
<BLOCK TYPE> "<BLOCK LABEL>" "<IDENTIFIER>" {
  # Block body
  <IDENTIFIER> = <EXPRESSION> # Argument
}
```

Listing 4.2: The structure of Terraform blocks [52].

Terraform templates are applied by executing the `terraform apply` command, which tells the `terraform` binary to parse and deploy the infrastructure described by the IaC files [32]. Terraform will contact the specified providers and make sure that the resources are modified according to an "execution plan" that is created by Terraform. This plan describes how the current "state" of the deployment should be modified. The current state stores information about all resources, their properties, and how they map to the deployed cloud resources in `JSON` files either locally or on some remote storage [32], [52]. The Terraform HCL configurations are typically stored in some version control system, which makes collaboration possible and captures the entire infrastructure history [32].

While Terraform can be used to manage multiple clouds at the same time, it does not provide abstractions of different cloud platforms APIs to enable building cloud-agnostic applications. Such abstractions tend to require a "lowest common denomitor" approach [53]. However, Terraform exposes the full functionality, including platform-specific features, without hiding them behind one unified interface [53], similarly to Apache jclouds [45]. While it is possible to build such abstractions to a certain extent using Terraform modules, they are limited to one platform at a time and are limited in functionality.

Even though Terraform requires platform-specific knowledge of the CSPs one wants to deploy to, it can be seen as a unified cloud API that provides one interface (HCL) to communicate with multiple CSPs. This makes it a great intermediate layer for a system that uses multiple clouds, similar to how OCCI (Section 3.1) works but based on IaC.

4.2. Multicloud Terraform Configuration

The transpiler tool generates cloud platform-specific Terraform HCL configurations. In order to design the transpiler, it is required to know what the resulting configurations should look like. These configurations are also used to analyze the differences as well as similarities between the different cloud platforms. This is why Terraform HCL definitions for each cloud platform are written. Those definitions deploy an exemplary web application that runs in the cloud. Web applications are a typical use case for the cloud, which is why many cloud

services support web protocols and services.

To have a realistic web application that can be used to write Terraform HCL for and test the transpiler, a web application that analyzes text is implemented. It analyzes the user-given input string in terms of statistics about the characters and words, as can be seen in Figure 4.2. The application should only use serverless services to minimize the management and infrastructure effort. The different architectures should be as homogenous as possible, and high-level abstractions (like OpenAPI definitions) should be used instead of cloud platform-specific solutions. To keep the project organized, the different Terraform deployments should be self-contained (infrastructure as well as application resources), but common application parts should be shared.

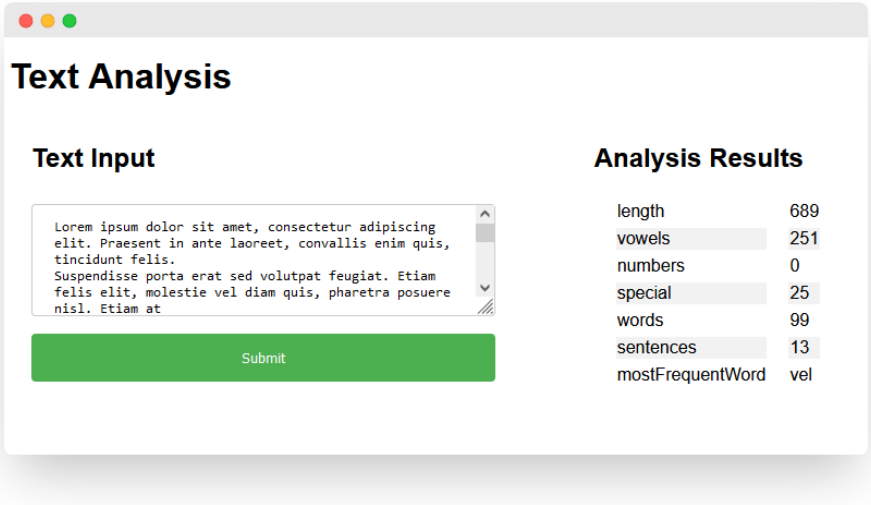


Figure 4.2.: The frontend of the example application that analyzes the given input string in terms of the occurrence of characters and words.

The application's backend provides an API that calls a `JavaScript` function that returns a `JSON` message. The function counts characters, vowels, words, and more word statistics. The frontend is implemented as a static website using `HTML`, `CSS`, and `JavaScript`. It uses `JavaScript` to send a request to the backend when a button is pressed. After receiving the response, it updates the data on the website.

The Terraform configurations are implemented for the most well-known cloud providers AWS, GCP and Azure using their respective official Terraform providers[5]. Those platforms were chosen due to their popularity, free tiers, and availability of educational credits.

The `JavaScript` function, forming the backend, should run as a FaaS function written in `JavaScript`. The core logic of the function is put into its own file, which is shared across

[5]Terraform providers are the interface between Terraform and a specific cloud platform and contain bindings for the different cloud services and their properties.

all platform configurations. Since every FaaS service requires the `JavaScript` entry point, input parameters, and response to be defined in a different way, every platform has its own function entry point that then calls the shared code. We also implement a small abstraction for the response parameters so that the shared code returns the expected result for every FaaS service. The source code is uploaded to the platform's object storage service, from where it is loaded into the FaaS service. Object storage is a serverless cloud offering that provides data storage for unstructured data ("objects"). This data is stored in "buckets", which are replicated to multiple regions.

The API gateway sits in front of the FaaS function, which should not be reachable directly. They provide features to manage invalid requests, load balancing, caching, and rate limiting. They are typically a fully managed solution that is secure and scales automatically. In order to make the specification of the backend API platform-independent, the OpenAPI standard is used.

The frontend consists of the static website that serves the `JavaScript`, `HTML`, and `CSS`, which are uploaded to a public "bucket" [6]. The storage bucket is served by a CDN to provide good availability with low latency from around the world. This architecture is summarized in Figure 4.3.

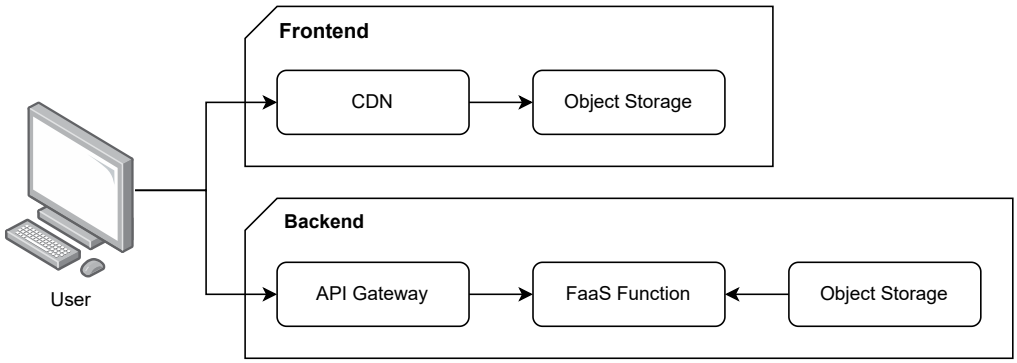


Figure 4.3.: The architecture graph of the example application, with the arrows indicating the data flow. The user visits the static webpage by requesting a resource from the CDN, which returns the contents originally stored in the object storage. The web browser calls the website's `JavaScript` code, which initializes a request to the backend. The API gateway handles the connection and calls the FaaS function stored in the object storage.

Every CSP provides a different service offering and often presents multiple solutions for a given problem. This makes choosing the right service variant of a specific category for each platform complicated. Additionally, the services should fit the abovementioned requirements (homogenous, serverless, etc.). Our service selection is based on the most typically used service for that use case and its similarity to the other platform's services. Table 4.1 lists the type of service together with the platform-specific serverless service variant that we chose. Architecture graphs of the cloud services for the example application deployed to the three

[6] Azure uses the name "container".

cloud platforms are presented in Figure 4.4.

Service	AWS	Azure	GCP
Object Storage	Amazon S3	Storage Container	Cloud Storage
CDN	Amazon CloudFront	Microsoft CDN	Cloud CDN
FaaS	AWS Lambda	Function Apps	Cloud Functions
API Gateway	Amazon API Gateway	API Management Services	Cloud Endpoints

Table 4.1.: The different serverless cloud services, sorted by their type and CSP, that are used for the example application.

AWS has the most flexible and comprehensive cloud offering. Feature-wise it provides everything that is required to implement the aforementioned example application. The AWS Terraform provider covers all the features and is up-to-date with the cloud platform.

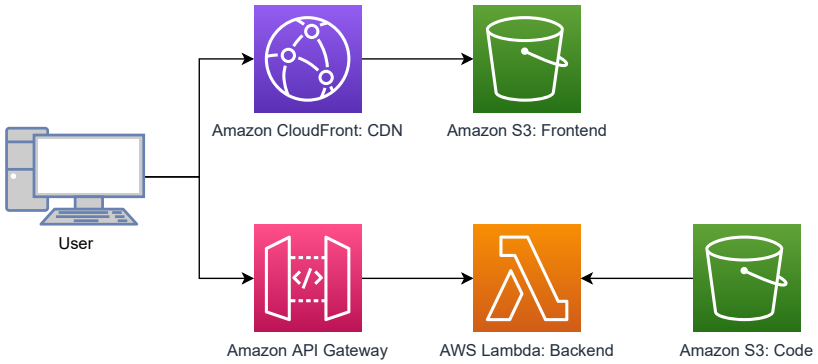
Azure offers a similarly comprehensive offering, with the most common features implemented. However, it is possible to bypass the API gateway and to call the FaaS function directly, which could incur additional costs when bypassing caches. The Microsoft Azure Function Apps also do not support ES6-style `JavaScript` modules yet. The Terraform provider is lacking some features that are possible on the Azure platform, like multiple origins per API gateway.

GCP services often work differently than similar services on other cloud platforms. Their service offering is smaller and offers fewer features. The CDN requires a load balancer to function, which increases cost. Also, it does not provide a domain or certificate, which means the user has to provision an IPv4 to access the website. Furthermore, the API gateway only supports OpenAPI version two and not version three as AWS and Azure support. The API gateway requires the `hashicorp/google-beta` provider as this resource is still in beta.

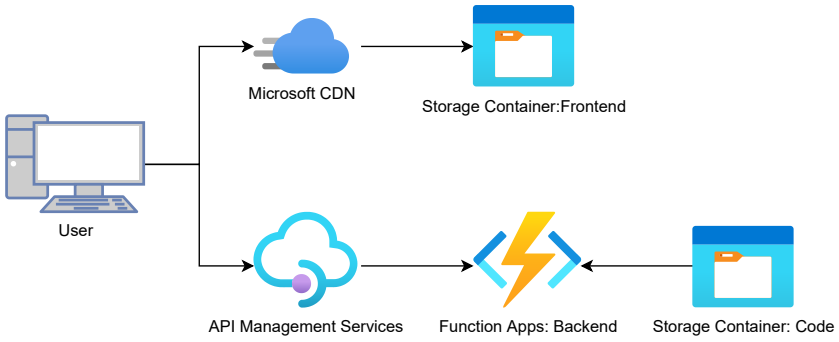
It can be concluded that each CSP offers similar services to implement the aforementioned architecture. For each service/problem type and platform, there is exactly one main cloud service that solves the problem. Some services, like the API gateway from GCP internally, require sub-services that are automatically created and therefore do not have to be modeled. The services of one category are relatively similar in terms of functionality and configuration.

The `git` repository containing all the code and Terraform configuration can be found on GitHub[7].

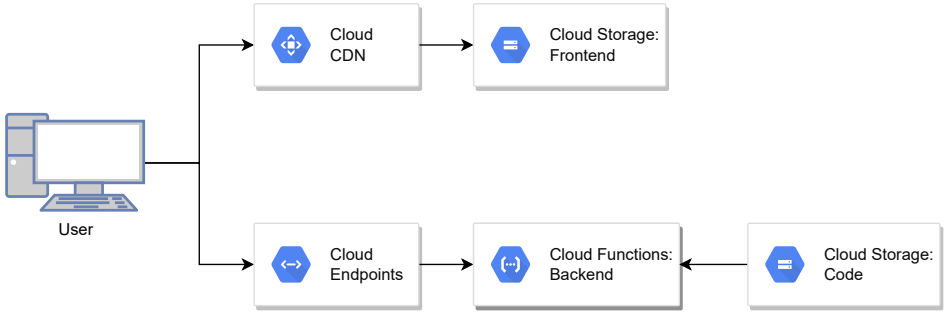
[7] https://github.com/michidk/serverless-webapp



(a) The serverless web application on the AWS platform.



(b) The serverless web application on Azure platform.



(c) The serverless web application on the GCP platform.

Figure 4.4.: The different web application architectures for the AWS, Azure and GCP cloud platforms.

5. Transpiler

The transpiler[1] translates (or "transpiles") the platform-independent architecture to Terraform configurations. This transpiling approach leverages Terraform as an intermediate layer that provides a unified cloud API abstraction, as explained in Chapter 4.1. This allows us to focus on the abstraction problem rather than on interfacing with the APIs of different CSPs.

The transpiler requires a template library, which defines the different components available to use in the generic architecture specified using GAML. Together with that specification, the transpiler applies the templates and outputs the different HCL files for each cloud platform, as illustrated in Figure 5.1. However, there are also non-functional requirements. The transpiler should perform its tasks within an acceptable amount of time.

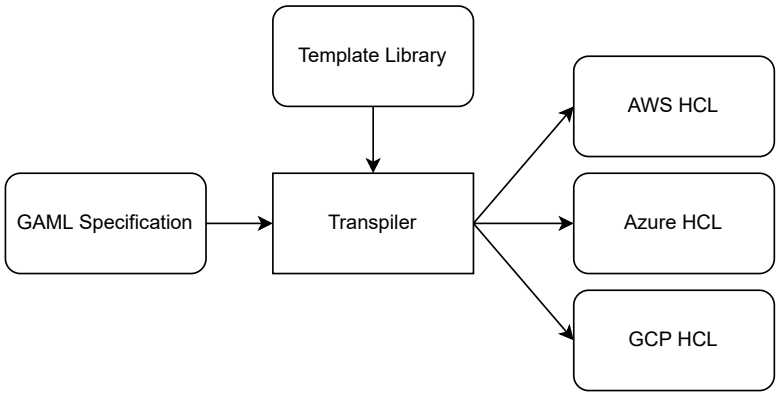


Figure 5.1.: The transpiler requires the GAML specification along with the template library as input. It outputs the Terraform HCL configurations for the different platforms.

This chapter is structured as follows. First, we present our modeling language GAML in Section 5.1, which is a generic high-level IaC-based language for cloud architectures. The template library explained in Section 5.2 contains templates that instruct the transpiler on how to generate the platform-specific Terraform configurations. Next, Section 5.3 describes the transpilation process that translates the GAML into concrete Terraform infrastructure definitions, targeting different cloud platforms. Finally, Section 5.4 points out some important aspects of how the code of the transpiler is designed.

[1]The term transpiler indicates that source code is translated to source code in a different format, as opposed to compilers which build machine code.

5.1. Generic Architecture Modelling Language

The software design of the transpiler depends on the modeling language that is used to specify the input architecture. The findings from the previous Chapter help to derive the language specification so that it fits the following requirements.

Previous work suggests using an abstract modeling language representing high-level objects to describe the input architecture and its functional aspects in a platform-agnostic manner [18]. This input architecture language, called GAML, needs to be platform-independent or "cloud-agnostic". It should abstract the differences of cloud platforms away to provide a single interface that is flexible enough, so it can represent complex architectures across CSPs. This architecture modeling language should be formatted as a human-readable text file similar to Terraform HCL files and the configuration languages of other IaC tools [54], [55].

HCL, as introduced in Chapter 4.1, would fit this use case. However, as previously mentioned, Terraform itself was developed for the opposite use case, where the user has control over the full functionality of each platform [53]. Furthermore, no software libraries that implement functionality to parse HCL files and resolve parsed references are available. However, it would be possible to implement a custom Terraform provider that provides resources for generic components.

Other approaches often use the `YAML` text format to specify human-readable non-code IaC. Since that format is a general-purpose configuration language, there are many libraries and tools available to parse, modify and extend the `YAML` functionality. It can represent complex objects and their properties but is easy to read, understand, and write. However, it natively has no notion of references, which is needed because components have to reference each other, as shown in Figure 4.3. However, such functionality can be implemented using node tags, as explained later. Other IaC tools like AWS CloudFormation choose this `YAML`-based approach as well [55]. In conclusion, `YAML` is a flexible but easy-to-use format that fits this use case.

The GAML has to be able to cover the complete model of generic architectures. Since Section 4.2 has shown that one service for each service type and cloud platform is required, such architecture can be modeled using a list of different components with a set of properties. It was also shown that the different cloud services of one type typically offer similar settings or that their settings can be abstracted to the same properties across platforms. Most properties are specified with various data types, such as strings and boolean values. However, references to other components are also essential. Components are defined by their type, a name, and a list of properties that depend on the type. That model also has to define the platforms it can be deployed to. Most cloud platforms also require platform-specific arguments to specify the deployment environment (such as the targeted data center, country, or region).

The `YAML` architecture definition file consists of a metadata section and a specification section. The specification includes a list of platforms, and a list of components, as seen in Listing 5.1. Platforms require different properties: The AWS platform only needs a region as input; GCP also requires a project identifier and a country code. Each component has a name, a

type, and type-specific properties that might contain references. The YAML format can be described as Cerberus schema, which is used to validate the input architecture but also serves as documentation. This schema is shown in Appendix B.

```yaml
kind: Architecture
metadata:
  name: web-service
spec:
  platforms:
    - name: aws
      properties:
        region: us-east-1
  components:
    - name: backend-code
      type: object-storage
      properties:
        uniqueName: faas-files
    - name: backend-faas
      type: function
      properties:
        uniqueName: faas-backend
        language: javascript
        source:
          bucket: !ref backend-code
          object: function.zip
```

Listing 5.1: The input YAML file defines an example architecture by listing the components as well as their properties and references. This architecture would deploy a FaaS function, whose code is stored in some object storage.

The Python PyYAML[2] library allows defining custom tags that execute custom functionality. This feature can be used to implement a `!ref <component name>` directive that handles references to other components. Those references form a type-annotated dependency graph that can be resolved and validated.

5.2. Template Library

The user-provided template library contains the definitions and templates that determine how the generic architecture is translated to the platform-specific Terraform configurations. The template library is a directory containing component definition files and component templates.

A template root file (`root.yaml`) specifies a list of template folders, each defining one

[2]https://pypi.org/project/PyYAML

component. It also specifies the template folders for the `main.tf` and `versions.tf` files that will be generated. The "main" and "version" files are defined separately since they have to exist exactly once per Terraform configuration and cannot be influenced by the architecture definition. An example of the template root file can be seen in Listing 5.2.

```yaml
kind: TemplateRoot
spec:
  main: main/
  versions: versions/
  templates:
    - api-gateway/
    - cdn/
    - function/
    - object-storage/
```

Listing 5.2: A template root `YAML` file defining a template library by specifying the `main.tf` template, the `versions.tf` template and the component templates.

The component definitions, written in `YAML`, are part of the component directory that is referenced by the template root file. Each component definition is defined in terms of the supported platforms and the properties, as can be seen in Listing 5.3. The properties section is a Cerberus schema that is used to validate the corresponding properties of the component's instances in the architecture definition file. References to other components can be specified using the custom `reference` type, which also allows specifying a `ref_type`, which is the name of the component definition that is expected as a reference.

Each configured platform requires a component template named `<platform>.tf.j2` to be present in the same directory, as can be seen in Figure 5.2. During the transpilation process, these templates are transformed to Terraform HCL configurations by replacing the template instructions with the component values.

The transpiler is based on the Jinja2[3] templating engine. This Python library provides a tiny programming language that can be used inside "templates" to fill in placeholder values. It is commonly used in web development and is well-documented. Using such an established system, our approach profits from the extensive ecosystem of extensions and documentation.

The aforementioned Terraform component templates are HCL configurations that use the Jinja2 syntax, as shown in Listing 5.4. The properties of the component instance in the architecture definition file are available as Jinja2 variables to use in these component templates. The variable `resourceId`, which contains the name of the current component, and `resourceType`, which contains the name of the component type, are available as well.

Variables can be of the type `integer`, `float`, `string`, `list`, `tuple` or `dictionary`. They can be inserted anywhere into the Terraform Jinja2 template using the `{{ variable }}` instruction. `if` and other typical programming control structures are also possible and allow

[3] https://palletsprojects.com/p/jinja/

```yaml
kind: TemplateDefinition
metadata:
  displayName: Function
spec:
  platforms:
    - aws
    - gcp
  properties:
    language:
      type: string
      required: true
      allowed:
        - javascript
        - python
        - java
    source:
      type: dict
      required: false
      schema:
        bucket:
          type: reference
          ref_type: object-storage
          required: true
        object:
          type: string
          required: true
```

Listing 5.3: A component `YAML` file defining a template for a componenent by specifying the the supported platforms and the template properties.

```
template/
├── api-gateway/
├── cdn/
├── function/
├── main/
├── object-storage/
│   ├── aws.tf.j2
│   ├── definition.yaml
│   └── gcp.tf.j2
├── versions/
└── root.yaml
```

Figure 5.2.: The file tree of an exemplary template directory. The template library is defined in the `root.yaml` file and the folders (indicated by a `/` suffix) contain the actual templates (`*.tf.j2`). The `object-storage` folder is the only one that is expanded due to space reasons, but the other folders are structured similarly.

```
resource "aws_s3_bucket" "{{ resourceId }}" {
  bucket        = "{{ uniqueName }}-bucket"
  force_destroy = "false"
}
```

Listing 5.4: The Terraform Jinja2 template for an AWS S3 bucket. The template defines the Terraform resource to have the same name as the component and uses the variable `uniqueName` from the architecture definition file to set a property.

```
{% if language | lower == "javascript" or language | lower == "typescript"  %}
runtime = "nodejs14.x"
{% elif language == "python" %}
runtime = "python3.9"
{% else %}
runtime = "none"
{% endif %}
```

Listing 5.5: An example showing the Jinja2 syntax for an `if` statement. In the condition, the `lower` filter is used to convert the variable to lowercase letters to make the comparison case-insensitive. If the first branch does not evaluate to true, then the second branch (`elif`) is tested. If the second branch is not matched as well, the third branch (`else`) is taken.

for complex templates[4]. `if` instructions, in particular, have shown to be very useful to design more high-level abstractions that behave differently depending on the environment, as can be seen in Figure 5.5.

Our custom references resolve to `string` (containing the name/`resourceId` of the targeted component) while transpiling, which allows referencing other Terraform components which use `resourceId` as their name. There is no further processing needed in order to work with references since Terraform supports referencing.

5.3. Transpilation Process

The transpiler exposes a Command-Line Interface (CLI) that is used to configure and start the transpilation process. The `--verbosity` flag allows setting the log level for the console output. There are two subcommands available: `transpile` and `plot`. The latter can be used to generate a graph of the dependency relations between the different components, which is demonstrated in Section 7.3. This graph is generated by evaluating component references by recursively iterating through all properties. The pygraphviz[5] library is used to offer a variety

[4]A complete documentation of the syntax is available here: https://jinja.palletsprojects.com/en/3.0.x/templates/
[5]https://pygraphviz.github.io/

of graph export formats[6] (which can be chosen by changing the file extension of the output file by using the `out` flag).

The `transpile` subcommand starts the transpilation process. The `--architecture` flag has to be set to a file path pointing to the architecture definition file. Similarly, the `--templates` flag points to the directory containing the component templates, and the `--output` flag to the output folder. A `--debug` flag will increase the debug output and add more details to the generated report that can be enabled with the `--report` flag.

The transpilation process requires multiple steps, as visualized in Algorithm 5.1. First, the template library and input architecture are parsed and validated. Afterwards, the `main.tf` and `versions.tf` files for each platform are generated according to their templates. Following this, the process is repeated for every component. The final step is to generate a report.

Algorithm 5.1: The transpilation processs

```
Parse and validate the template library;
Parse and validate the input architecture (GAML);
for each platform do
    Generate the main.tf and versions.tf files according to the template;
    for each component do
        Generate a component HCL file according to the template;
Write report;
```

To validate the structure and types of the components as well as the input architecture definition file, the Cerberus[7] Python library is used. It enables the validation of data with schemas written in `YAML`. As an example, the schema for the architecture definition model from Section 5.1 can be found in Appendix B. It also serves as documentation because it lists the required fields together with the expected values. As a final validation step, references are checked to ensure that the target component exists and is an instance of the required type, which the template library specifies.

Each platform, configured in the architecture definition file, will result in one output directory containing a `main.tf`, `versions.tf` and other Terraform HCL files. The `versions.tf` defines the Terraform version and which Terraform providers to use. The `main.tf` file configures the Terraform provider settings. The other Terraform IaC files are built from the user-provided template library.

Finally, a report is generated that, depending on the configuration, provides a mapping between the components of the architecture definition file together with their platform and the resulting Terraform file, as well as debug information and statistics. This provides metadata that can be used to further process the resulting files with other tools or to validate the functionality.

[6]The supported output formats are listed in the `graphviz` documentation: https://graphviz.org/docs/outputs/
[7]https://docs.python-cerberus.org/en/stable/

5.4. Source Code

The transpiler application is written in the programming language Python[8] version 3.10.5. Python was chosen due to its simplicity and vast ecosystem. It allows for quick prototyping but often leads to code that is difficult to read. We, therefore, set ourselves some constraints to implement the following best practices:

1. *Use object-oriented programming* to keep the code clean and satisfy the DRY (don't repeat yourself) principle.
2. *Document everything* to enable others to work with the code. Every non-magic method/function[9] should be documented.
3. *Annotate the code with Python typings*[10] to improve documentation and improve reliability by making the code easier to use and prevent mistakes.

Table 5.1 lists the Python libraries[11] that the transpiler depends on. Other libraries like `Pylint` are used during development to ensure good code quality.

Dependency	Explanation
`loguru`	Handles exception formatting and logging
`Jinja2`	Templating engine
`MarkupSafe`	Character escaping library; Required be Jinja2
`Cerberus`	Data validation library
`PyYAML`	`YAML` parser
`pygraphviz`	Graph visualization library

Table 5.1.: The Python dependencies required by the transpiler project.

The source code of the transpiler is organized in multiple Python files, as can be seen in Figure 5.3. The main entry point of the application is the `main.py` file, which is responsible for setting up the CLI. It is also responsible for calling either the `graph.py` or `transpiler.py` depending on whether the user wants to plot the dependency graph or transpiler the given architecture. The `transpiler.py` file implements the logic presented in Algorithm 5.1. The `common.py` contains code that is needed in both branches, such as initializing the templating system and loading the architecture file. The `utils.py` file offers various little helper scripts like loading and validating `YAML`.

The Cerberus schemas defined in the `schemas/` folder, contain the `YAML` structure definitions for the `Architecture`, `TemplateRoot`, and `TemplateDefinition` file kinds as presented in

[8] https://python.org/

[9] Magic methods are functions like `__init__` that the compiler has special logic for.

[10] Typings are annotations that provide hints on the type of properties, variables, or return values: https://docs.python.org/3/library/typing.html

[11] The libraries can be found in the Python package index: https://pypi.org/

```
src/
├── schemas/
│   ├── architecture.yaml
│   ├── templateDefinition.yaml
│   └── templateRoot.yaml
├── architecture.py
├── common.py
├── config.py
├── main.py
├── schema.py
├── tags.py
├── template.py
├── transpiler.py
├── utils.py
└── validator.py
```

Figure 5.3.: The directory structure of the `src/` folder containing the source code of the transpiler project. Some files that are not contributing to the application logic are omitted from this visualization to increase readability.

Figure 5.4. They map to Python objects in the `template.py` source file, which contain their parsing logic and hold the data. It contains Python objects that inherit from the `TemplateConfig` object, which is responsible for parsing the user-provided data. Itself inherits from `YamlConfig`, defined in `config.py`, which defines the general structure (the `metadata` and `spec` sections) of the `YAML` files used in this project. The `ArchitectureConfig` object defined in `architecture.py` parses, checks and hold the data from the architecture definition files and inherits `YamlConfig` as well.

The `tags.py` file defines the logic needed to parse `YAML` files that use custom tags with `PyYAML`. We use custom `YAML` tags to specify references to other components in the architecture definition file as presented in Subsection 5.2. The `validator.py` source file contains the necessary code to validate architecture definition files that contain custom references. It checks references and searches the other components for the referenced name and validates the component type.

One reason for the previously mentioned constraints (object-oriented programming, extensive documentation, and typing annotations) is that we want to contribute this project to the open-source community. We think that other developers can profit from the project's code and hope that they will contribute to it. The transpiler project was named "Multiform", a mix of the name of Terraform and the term multi-cloud. The source code can be found on GitHub[12].

[12]https://github.com/michidk/multiform/

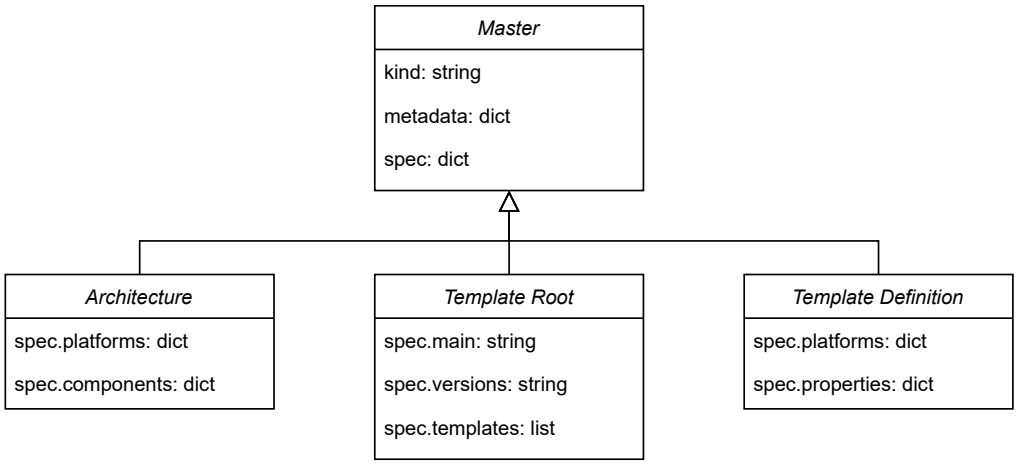


Figure 5.4.: The schemas used to validate the `YAML` files. Every schema file has a kind, which defines it's class or type (`kind` contains the name of the type they define; for example "Architecture" or "TemplateRoot"), a dictionary for metadata (`metadata`) and the specification (`spec`) that uses Cerberus validation rules. Architecture definitions, the template root, and template definitions inherit those properties and further specify the structure of the specification section that is then used to validate instances of that type. Therefore, the master schema's `spec` section is also used to validate the other schemas.

6. The Multy Approach

Having our templating-based transpiler explained, we now present a different approach that is based on a similar idea.

During the final steps of writing this thesis, we discovered an unknown[1] GitHub repository called Multy[2] that tries to solve the same problem with a similar strategy. They started the initial work in January of 2022, but the majority of the work happened at the same time our approach was developed. While it can already be used in production, it is still under active development. The startup behind the project is small[3] and currently in the pre-seed phase.

Their approach uses a transpiler to translate a generic architecture into cloud-specific ones, as well [56]. Furthermore, they also use Terraform as an abstraction layer to deploy to the different cloud platforms [56]. However, they employ the Terraform HCL language as well as the Terraform tooling to execute the transpilation process itself. They implement a custom Terraform provider[4] in Go[5], which provides generic cloud resources that are translated to the platform-specific equivalents [56]. Because of this, their approach integrates better into the Terraform workflow since only one CLI command is required to transpile and deploy the resources without any additional steps.

By using a Terraform provider this way, developers can leverage the well-known HCL language to define platform-independent resources, as seen in Listing 6.1. As opposed to our approach, theirs is not based on templating: Their software provides a `gRCP`[6] interface that transforms the resource requests into Terraform HCL templates. The translation logic is not based on templates but programmed into the application, which allows for more detailed platform-dependent validation logic and additional sanity checks[7] of the input data. They use their "encoder" project to write out the Terraform configurations to the disk, which are then automatically deployed.

At this time, only AWS, Azure, and GCP are supported. Since the translation logic is programmed into the software, adding new platforms or cloud resources is complicated. Consumers depend on the project contributors to add new resource types. Only a few of the

[1] The GitHub repository of the Multy project currently (2022-07-23) counts 200 stars (and six users watching). Registered GitHub users can "star" a repository if they like it. This number indicates how popular a GitHub project is.
[2] https://github.com/multycloud/multy
[3] As can be seen on LinkedIn: https://linkedin.com/company/multy-cloud/people/
[4] https://registry.terraform.io/providers/multycloud/multy/latest/docs
[5] https://go.dev/
[6] gRPC is a "Remote Procedure Call" framework based on `HTTP/2` and protocol buffers: https://grpc.io/
[7] https://stackoverflow.com/a/4069450/3453182

```
variable "clouds" {
  type    = set(string)
  default = ["aws", "azure"]
}

resource "multy_virtual_network" "vn" {
  for_each = var.clouds
  cloud    = each.key

  name       = "multy_vn"
  cidr_block = "10.0.0.0/16"
  location   = "eu_west_1"
}

resource "multy_subnet" "subnet" {
  for_each = var.clouds

  name               = "multy_subnet"
  cidr_block         = "10.0.10.0/24"
  virtual_network_id = multy_virtual_network.vn[each.key].id
}

resource "multy_virtual_machine" "vm" {
  for_each = var.clouds

  name = "test_vm"
  size = "general_micro"
  image_reference = {
    os      = "ubuntu"
    version = "20.04"
  }
  subnet_id = multy_subnet.subnet[each.key].id
  cloud     = each.key
  location  = "eu_west_1"
}
```

Listing 6.1: A possible Terraform HCL configuration to deploy a virtual machine with the Multy provider. The `terraform` and provider statement are omitted for readability. This example only provisions resources in the AWS and Azure cloud; however, GCP could also be configured. The platform's names are stored in a list, and every resource is created once for each of the items in that list using Terraform's `for_each` property.

typical IaaS resources are currently supported. However, some PaaS services (such as object storage and managed Kubernetes clusters) can also be deployed.

The approach that Multy takes is more user-friendly than ours: They simplify the deployment process by leveraging the built-in Terraform provider interface. Multy is configured using the well-known and well-documented HCL, which is parsed and validated by Terraform. Since Multy does not depend on the features of a templating system but rather uses a real programming language to translate the architecture, developers have more freedom in implementing the validation and transpilation logic. Finally, the HCL configuration of the generic architecture can integrate with other Terraform tools and providers.

The Multy developers chose a reasonable compromise to the common lowest denominator approach: While it abstracts some properties (for example, the virtual machine sizes[8]), it provides pre-defined overrides for the most important platform-specific component properties [57]. Additionally, it is possible to use the platform-specific providers in conjunction with the Multy provider to configure more platform-specific resource behaviour [57].

In general, Multy only supports service variants that are very similar on all platforms. Since serverless cloud services often have a lot of differences on different platforms, they would have to have a lot of platform-specific overrides.

While a minimal level of abstraction is required because of the nature of the project, it is still possible to write platform-specific overrides, as discussed above. However, high-level abstractions, like combining complex components into one resource or simplifying complex components, are currently not supported and are not a goal of the project.

[8] https://docs.multy.dev/vm_sizes

7. Evaluation

Next, we evaluate our approach of transforming the generic input architecture into platform-specific Terraform configurations, as presented in the previous chapter. In Section 7.1, we discuss the three main questions we chose to evaluate. The example application is used to test the transpiler and verify the output, which is presented in Section 7.2. The generated Terraform configurations are analyzed and validated to ensure the transpiler works as intended in Section 7.3. In this section, we also present some additional tests we have performed.

7.1. Metrics

We want to show that the transpiler tool can be used to ease the deployment of real serverless applications. Therefore we deploy a serverless web application using our transpiler in Section 7.2. We show how this was realized and discuss some implementation considerations. However, there are more aspects to evaluate beyond generating the correct output, as shown in this chapter.

We want to ensure that the resulting Terraform HCL configuration files are generated correctly. The Terraform software comes with a CLI command that validates configurations. This command checks the syntax and arguments of those configurations. If the template library contains mistakes that are not checked by the transpiler or there is an error in the transpilation logic, this command will report an error.

The resulting configurations can be tested further by deploying them to the cloud platforms. Some errors might only occur when the Terraform provider tries to actually provision the resources by contacting the cloud platform's API. The transpiling process is successful if the web application works.

Another aspect is to benchmark the transpiling process in terms of processing time. While generally, the processing time for IaC tools is not very important, we still want to ensure short transpilation durations. We also want to make sure that the time spent processing does not increase exponentially with an increasing component count.

Important is whether our transpiler reduces the work needed to deploy cloud applications to multiple CSPs. While the quality and usefulness of our implementation of the high-level abstractions, as well as the required knowledge, are subjective, we can measure the development effort in terms of required Source Lines of Code (SLOC). SLOC is a metric that is determined by counting the lines of source code (not comments and empty lines) that have

to be written.

7.2. Example Application

The multi-cloud web application from Section 4.2 can be formulated using the GAML presented in Section 5.1 as seen in Listing 7.1. This input architecture requires a template library containing the following component definitions to be able to transpile them to a working Terraform project:

- `main`: The provider configuration
- `versions`: Specifies the provider versions
- `object-storage`: Templates for object storage
- `cdn`: Templates for a CDN
- `function`: Templates for a FaaS function
- `api-gateway`: Templates for an API gateway

Due to time constraints and availability of credits, we decided to only implement the templates for the AWS and GCP platforms. The source code of these component templates can be found in the `examples/` folder of the transpiler project, which is available on GitHub[1].

This example shows how the transpiler can be used to generate Terraform configurations for multiple platforms using component templates. However, since components require templates, only architectures incorporating the existing component templates can be built. It would be possible to build component templates for every cloud offering type that has a similar product on all required platforms. This would result in a huge template library that enables various architectures. Alternatively, component libraries can be project-specific, similar to the aforementioned example. This would allow the templates of the components to be less generic and more specific to the project.

It is also possible to define different components that use the same cloud products but with different configurations and exposed properties. As opposed to this, one component template can also contain complex Terraform configurations that incorporate multiple cloud products. Similar to how the AWS CDN component of the example architecture combines a bucket policy together with the resources required for the CloudFront service (distribution, access identity, origin request policy, and cache policy).

7.3. Results

The previous section showed that the transpiler can be used to generate platform-specific Terraform configurations from a generic architecture provided in GAML. To validate the

[1] https://github.com/michidk/multiform/

```yaml
kind: Architecture
metadata:
  name: simple-web-service
  version: v1.0.0
spec:
  platforms:
    - name: aws
      properties:
        region: us-east-1
    - name: gcp
      properties:
        region: us-central1
        location: US
        project: sample-project-123
  components:
    - name: frontend
      type: object-storage
      properties:
        uniqueName: 123-static-web-files
    - name: cdn
      type: cdn
      properties:
        uniqueName: 123-cdn
        target: !ref frontend
    - name: backend-code
      type: object-storage
      properties:
        uniqueName: 123-faas-files
    - name: backend-faas
      type: function
      properties:
        uniqueName: 123-faas-backend
        language: javascript
        source:
          bucket: !ref backend-code
          object: function.zip
    - name: api
      type: api-gateway
      properties:
        uniqueName: 123-api-gateway
        function: !ref backend-faas
        openapiFile: openapi.yaml
        swaggerFile: swagger.yaml
```

Listing 7.1: The architecture definition according to Section 5.1 of the serverless web application presented in Section 4.2.

```
/out/aws $ terraform state list
data.aws_iam_policy.backend-faas_execution_role
data.aws_iam_policy_document.cdn
aws_apigatewayv2_api.api
aws_apigatewayv2_stage.api
aws_cloudfront_cache_policy.cdn
aws_cloudfront_distribution.cdn
aws_cloudfront_origin_access_identity.cdn
aws_cloudfront_origin_request_policy.cdn
aws_iam_role.backend-faas
aws_iam_role_policy_attachment.backend-
  faas_execution_role_attachment
aws_lambda_function.backend-faas
aws_lambda_permission.api
aws_s3_bucket.backend-code
aws_s3_bucket.frontend
aws_s3_bucket_policy.cdn
aws_s3_bucket_public_access_block.backend-code
aws_s3_bucket_public_access_block.frontend
```

```
/out/gcp $ terraform state list
google_api_gateway_api.api
google_api_gateway_api_config.api
google_api_gateway_gateway.api
google_cloudfunctions_function.backend-faas
google_cloudfunctions_function_iam_member.api
google_compute_backend_bucket.cdn
google_compute_global_address.cdn
google_compute_global_forwarding_rule.cdn
google_compute_project_default_network_tier.cdn
google_compute_target_http_proxy.cdn
google_compute_url_map.cdn
google_project_service.api
google_project_service.backend-faas_build
google_project_service.backend-faas_functions
google_service_account.api
google_storage_bucket.backend-code
google_storage_bucket.frontend
```

Listing 7.2: The output of the command `terraform state list` after deploying to the platforms AWS (left) and GCP (right), which lists the currently deployed Terraform components.

resulting Terraform HCL files, the `terraform validate` command is used, which succeeded in our case. After running `terraform apply` for the AWS and GCP HCL files, the Terraform components listed in Listing 7.2 are deployed successfully to the cloud. After uploading the files required for the FaaS function and the static website to the corresponding object storage of each platform, the web application previously presented in Section 4.2 will be available on the internet.

The transpiler CLI provides a sub-command to generate a dependency graph, as explained in Chapter 5. The output of this subcommand on the example application can be seen in Figure 7.1.

To change the architecture, for example, by adding a second `object-storage` component instance, the architecture definition file can be adjusted before running the transpiler again. Since the transpilation process is deterministic, the other untouched components will not change, but the new component instance will also be reflected in the Terraform output.

The deployment process of such architecture depends on Terraform and the CSPs since actually deploying the application by interacting with cloud platform APIs and waiting for the resource creation takes up most of the time. Our transpiler will have an insignificant effect on the duration of the whole process: While the actual Terraform deployment can take up to several minutes, the transpilation of the example application takes 0.343*s* on our test setup, which we empirically categorize as an average developer machine. Figure 7.2 shows that the processing duration does not grow exponentially or worse with an increasing component count.

To compare the amount of work between using the transpiler and developing the Terraform

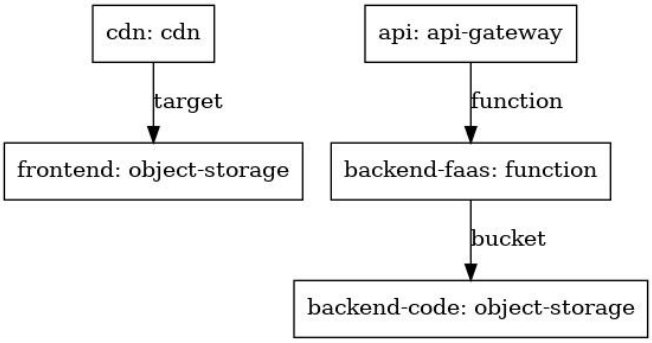


Figure 7.1.: The dependency graph of the example application is generated with the plot utility of the transpiler, by executing the following command: `python3 start.py plot -architecture architecture.yaml -out graph.jpg`. The arrows point to the components being referenced. They are annotated with the name of the variable referencing the component. The node names show the name of the component instance and their component kind.

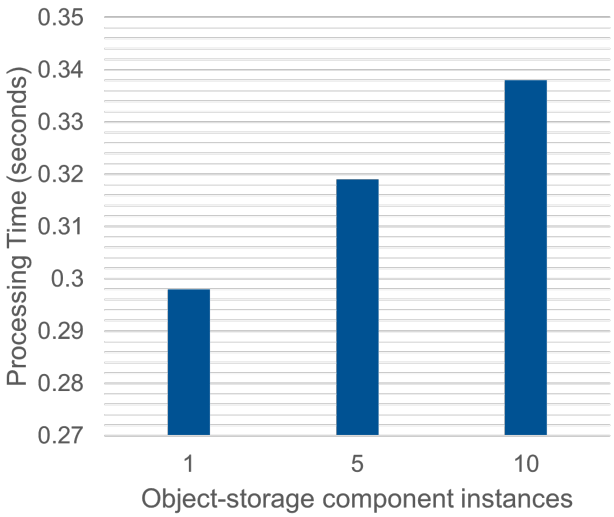


Figure 7.2.: The transpilation durations for architectures with multiple object-storage components, whose count is shown on the x-axis. The duration on the y-axis is the average of 10 executions in seconds. They were measured inside a Docker container running on a system that we empirically categorize as an average developer machine.

configuration directly, the SLOC were measured [2]. The Terraform HCL files that are output by the transpiler contain 251 SLOC across 8 files for the AWS platform and 187 SLOC across 8 files for the GCP platform. This makes 438 SLOC of generated HCL in total. The template library together with the GAML file, contain 520 SLOC across 22 files.

Writing the Terraform HCL configurations by hand results in more work (more SLOC) in the case of the demo application. While the transpiler input files required 520 SLOC, the Terraform configurations for both cloud platforms combined contain 438 SLOC, as can be seen in Table 7.1. However, the GAML file only requires 45 SLOC (or even less without the metadata). If a template library already exists, the SLOC to deploy such architecture is significantly reduced. But even in the absence of a complete template library, the SLOC required for the transpiler can become less than the SLOC of the Terraform files. In our example, only the small object storage component gets reused twice. However, when reusing components more often in larger architectures, the SLOC for the transpiler input will become less compared to the Terraform SLOC. If every component is only used once, the SLOC count with our approach will always be higher since the transpiler templates require extra definition files and Jinja2 instructions. The SLOC required for the input architecture can be further reduced by combining repeatable patterns (for example, a serverless web application) into its own component, similar to how Terraform modules work.

Metric	**SLOC**
GAML definition	45
Template library	475
Transpiler input (total)	520
Transpiler output (HCL)	438

Table 7.1.: The Source Lines of Code (SLOC) measured of the example application from Section 7.2. The total transpiler input is the sum of the GAML definition and the template library.

The GAML proposed in Section 5.1 requires less to no knowledge about specific platforms. Only knowledge about how a component is supposed to be used with the given template library to build a complete architecture is required. While in Terraform configurations the AWS FaaS resource is identified with the term `aws_lambda_function` and for GCP `google_cloudfunctions_function`, in the generic architecture they can be given a generic name like `function`. Furthermore, Terraform does not only have different resource names for each platform, but the properties and correct usage and interaction with other resources are different as well.

This shows that our transpiler approach not only provides a higher-level abstraction but also reduces the amount of work (SLOC) for complex architectures.

[2]Only lines containing actual source code (no empty lines and comments) were counted using `scc`: hrefhttps://github.com/boyter/scchttps://github.com/boyter/scc.

8. Future Work

While the previous chapter showed how our approach could be applied to improve the development of real cloud applications, there is still much opportunity for improvement. Various changes concerning the GAML are suggested in Section 8.1. Following in Section 8.2 are suggestions for improving the transpiler software.

8.1. GAML Improvements

The GAML defined in Section 5.1 is rather limited in its current state. However, the language can easily be improved because it is based on Cerberus schemes and `YAML`. One improvement would be the introduction of "inputs" or input variables. Input variables would be defined in the specification of a generic architecture template and referenced in component properties. They would be set from parameters that are provided by the user before starting the transpilation process. Modeling languages like OASIS TOSCA and Terraform HCL use input variables to make the same architecture template reusable for different instances of an architecture. For example, with input variables for the name, FaaS function, and API definition, the web application from Section 7.2 could be deployed multiple times without changing the architecture definition file itself.

Complex architecture might result in a large architecture definition file which will get confusing to read and work with. Other languages like Terraform HCL therefore have the functionality to isolate and bundle some resource definitions together as "module". With inputs and outputs of modules, they can be used to abstract and reuse functionality with different input properties.

Our architecture model language would benefit from such an approach. For example, one could implement a "serverless web application" module that encompasses the architecture presented in Section 7.2. It would take the OpenAPI definitions and the path of the FaaS source code in the bucket as input. Now the module can be isolated and reused, which allows for a higher level of abstraction as presented in Listing 8.1.

This would be implemented with a new template `kind`. They would be similar to `TemplateDefinition`'s with the difference that they do not use Terraform Jinja2 (`.tf.j2`) templates but rather specify their architecture with references to the input variables.

In its current state, our approach follows the lowest common denominator strategy regarding multi-cloud abstractions. A property defined on a component instance in the architecture is passed to every platform. However, it would be advantageous if some components could allow

```yaml
kind: Architecture
metadata:
  name: multiple-web-apps
spec:
  platforms:
    ...
  components:
    - name: web-app-1
      module: serverless-webapp
      properties:
        uniqueNamePrefix: my-web1
        faas:
          language: javascript
          source:
            object: function1.zip
          api:
            openapiFile: app1/openapi.yaml
            swaggerFile: app1/swagger.yaml
    - name: web-app-2
      module: serverless-webapp
      properties:
        uniqueNamePrefix: my-web2
        faas:
          language: go
          source:
            object: function2.zip
          api:
            openapiFile: app2/openapi.yaml
            swaggerFile: app2/swagger.yaml
```

Listing 8.1: A prototype of what the usage of modules in the GAML could look like.

platform-specific variables or overrides, as they can in TOSCA. This is useful in scenarios where it might be required to expose and configure a setting that only exists on one platform or differs on multiple platforms between component instances.

It is possible to define `labels` in the `metadata` section on an architecture definition, which are simple key-value pairs. They are reserved for further use and are injected into the properties available to use in component templates. In some cloud platforms, resources can be tagged to help identify and distinguish different instances of them. The labels could be used to assign tags to these cloud resources in order to make it possible to inspect which cloud resource belongs to which generic architecture configuration.

8.2. Transpiler Improvements

The transpiler translates the generic architecture into Terraform configurations using templates. The templates of some components require references to different Terraform resources, as the Listing 8.2 illustrates. These references "trust" the template author that the resource always exists. If it does not, the transpilation process will still succeed. However, during the deployment process, Terraform will report an error. To prevent this behavior, the transpiler should detect and validate such references. The validation process should also check whether the type of the referenced resource is the correct one, which would require parsing the Terraform files. Another approach would be to use Jinja2 to apply the templates and then use the `terraform validate` command to validate the resulting configuration automatically.

```
1 resource "aws_s3_bucket_policy" "my_bucket_policy" {
2   bucket = aws_s3_bucket.{{ target }}.id
3   policy = data.aws_iam_policy_document.{{ resourceId }}.json
4 }
```

Listing 8.2: A exemplary AWS S3 bucket policy of a Terraform component template that shows the use of references to other Terraform components.

The target platforms are defined in the specification section of the architecture definition file. They are specified by a name and some properties. Those properties are not further specified and are directly fed into the Jinja2 variables of all component templates. This means that they are also not checked for completeness. To further improve the transpiler, another type of schema has to be introduced that can be used to define cloud platforms in terms of names and properties required by the templates. These platform definitions can then be included along with the component templates and used to validate the properties in an architecture definition file.

Another improvement for the transpiler would be to automatically check the resulting files after transpilation with `terraform validate` to ensure their correctness. Since one Terraform configuration for every platform has to be checked, the Terraform validation result should be

collected, added to the transpiler report, and summarized so that they can be displayed in a structured way indicating from which component template the error originates.

TOSCA, as presented in Section 3.2, provides a modeling language that is based on YAML. The language is quite flexible and similar to our GAML design since it is also based on components (TOSCA nodes) that implement specific types that define the available properties. Since TOSCA is one of the most established systems that provide such generic architecture abstraction, it makes sense to support a subset of their service template format. Other cloud deployment tools have implemented support for the TOSCA specification successfully [58], [59].

The transpiler generates a Terraform configuration containing multiple HCL files for each cloud platform specified in the architecture definition file. To deploy these Terraform configurations, one has to execute commands for each Terraform configuration. By wrapping the Terraform CLI in our transpiler tool CLI, it would be enough to execute one command (e.g., for deploying the architecture) that would automatically run the corresponding Terraform command on every platform's configuration.

In TerraformHCL, modules and components can have "outputs" that contain data after the deployment, which can be displayed to the user. Wrapping the Terraform CLI also allows capturing the output of Terraform commands to collect and parse it. This would enable capturing output variables and allow displaying them in a structured way (for example, a table with the cloud platforms as columns).

A User Interface (UI) could improve the usability of our approach. Instead of modifying the architecture definition YAML file, the user would interact with a web-based application to configure the components, their relationships, and their properties, similarly to the UI tooling available for TOSCA [60], [61]. This would result in the architecture file that can be downloaded afterward.

A more advanced solution incorporating the aforementioned Terraform CLI wrapping could also show the current deployment status across the cloud platforms in the UI.

9. Conclusion

Serverless computing is popular because it simplifies complex cloud deployments and reduces the management overhead. However, challenges such as cloud interoperability and the missing high-level abstractions that arise in multi-cloud environments are yet to be solved.

In Chapter 3, we presented several solutions that suggest abstraction layers to provide an environment for platform-independent applications. OCCI is a standardized interface for cloud management tasks. OASIS TOSCA introduces its own platform-independent application format that can be deployed on compliant platforms. Another approach uses the Kubernetes orchestration software tool running on different cloud platforms to provide a homogeneous cloud environment. The VSP schedules FaaS functions on different CSPs depending on their workload. Apache Libcloud provides a high-level Python interface to provision cloud resources. mOSAIC implements a broker-based approach to provide access to multiple CSPs through one interface. Similarly, Cloud4SOA uses a multi-layered architecture to offer a unified API across different PaaS platforms.

However, those solutions either do not provide an abstraction for serverless services or still require some platform-specific knowledge. Our approach to implementing proper architectures for cloud-agnostic applications is based on Terraform. While Terraform itself does not solve the abstraction challenges, it provides a homogenized API across different CSPs. We present a layer on top of Terraform, implemented as a software tool that uses templates to translate our custom generic architecture format, GAML, into Terraform HCL configurations.

First, Terraform configurations for a simple web application running on three cloud platforms are developed as presented in Section 4.2. This is used as a basis to find the lowest common denominator amongst the architectures. With the similarities derived, a language to model generic architectures is defined as shown in Section 5.1. The transpiler, as presented in Chapter 5, takes the generic architecture specified in GAML, as well as a collection of component templates as input, and transpiles them to one Terraform deployment per platform.

The evaluation in Chapter 7 showed that the transpiling process does work with our exemplary web application. We further showed that it significantly decreases the development time when a template library is already available.

There is a lot of potential for improving the transpiler and architecture language as presented in Chapter 8. We suggested working on the architecture language to implement inputs, modules, and labels. The transpiler could be further improved by implementing an automatic

validation of the architecture file and the resulting Terraform configurations. Furthermore, the Terraform CLI could be leveraged to turn the transpiler into a tool that manages the whole multi-cloud deployment lifecycle. A user interface could further improve the developer experience by enabling easy management of the deployed resources.

The Multy project uses a similar approach as the one presented in this thesis to build cloud-agnostic applications. The Terraform-based language model, together with the Multy transpiler allows using a well-known language model and toolset to specify generic architectures. However, it requires a good understanding of the project as well as programming skills to modify the transpiling behavior and add new components. While a certain level of abstraction is given, it is not possible to use high-level or custom abstractions. In conclusion, Multy shows to be a very promising approach because they simplify the deployment process by leveraging Terraform functionality directly, as explained in Chapter 6.

The core contribution of our work is the transpiler that can be used to translate high-level architecture definitions written in a human-readable text format into platform-specific Terraform configurations. The evaluation showed a real use case by leveraging the transpiler to deploy a multi-cloud web application. While there is a lot of room for expanding and improving our solution, it is ready to be used to deploy cloud-agnostic applications in multi-cloud scenarios.

Acknowledgments

Foremost, I would like to thank my supervisor, Prof. Michael Gerndt, for allowing me to conduct this research at the Chair for Computer Architecture and Parallel Systems.

I would also like to express my sincere gratitude to my advisor, Anshul Jindal, for his close supervision and helpful advice.

Last but not least, I would like to thank all my friends and family members for their continuous support during my studies and the writing of this thesis.

Acronyms

API	Application Programming Interface
AWS	Amazon Web Services
Azure	Microsoft Azure
BaaS	Backend as a Service
CD	Continuous Delivery
CDN	Content Delivery Network
CI	Continuous Integration
CLI	Command-Line Interface
CSP	Cloud Service Provider
DNS	Domain Name System
FaaS	Function as a Service
GAML	Generic Architecture Modeling Language
GCP	Google Cloud Platform
HCL	HashiCorp Configuration Language
HPC	High-Performance Computing
IaaS	Infrastructure as a Service
IaC	Infrastructure as Code
mOSAIC	Open-Source API and Platform for Multiple Clouds
NIST	National Institute of Standards and Technology
OCCI	Open Cloud Computing Interface
OS	Operating System
PaaS	Platform as a Service
SaaS	Software as a Service
SLA	Service-Level Agreement
SLOC	Source Lines of Code

SOA	Service Oriented Architecture
TOSCA	Topology and Orchestration Specification for Cloud Applications
UI	User Interface
VM	Virtual Machine
VSP	Virtual Serverless Provider

List of Figures

List of Tables

List of Listings

Bibliography

[1] Gartner. "Gartner forecasts worldwide public cloud end-user spending to grow 23% in 2021: Cloud spending driven by emerging technologies becoming mainstream". (2021), [Online]. Available: https://www.gartner.com/en/newsroom/press-releases/2021-04-21-gartner-forecasts-worldwide-public-cloud-end-user-spending-to-grow-23-percent-in-2021 (visited on 07/07/2022).

[2] D. Petcu, "Portability and interoperability between clouds: Challenges and case study", *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6994, pp. 62–74, 2011.

[3] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing", *Communications of the ACM*, vol. 62, no. 12, pp. 44–54, 2019.

[4] G. A. Lewis, "Role of standards in cloud-computing interoperability", *Proceedings of the Annual Hawaii International Conference on System Sciences*, pp. 1652–1661, 2013.

[5] R. Pellegrini, P. Rottmann, and G. Strieder, "Preventing vendor lock-ins via an interoperable multi-cloud deployment approach", *2017 12th International Conference for Internet Technology and Secured Transactions, ICITST 2017*, pp. 382–387, 2018.

[6] I. Stoica and S. Shenker, "From cloud computing to sky computing", *HotOS 2021 - Proceedings of the 2021 Workshop on Hot Topics in Operating Systems*, pp. 26–32, 2021.

[7] Canalys. "Global cloud services spend hits record 49.4 billion usd in q3 2021". (2021), [Online]. Available: https://www.canalys.com/newsroom/global-cloud-services-q3-2021 (visited on 07/07/2022).

[8] Amazon Web Services, Inc. "What is aws". (2022-07-20), [Online]. Available: https://aws.amazon.com/what-is-aws/ (visited on 07/21/2022).

[9] P. Mell and T. Grance, Eds., *The NIST Definition of Cloud Computing: Recommendations of the National Institute*. National Institute of Standards and Technology.

[10] S. R. Goniwada, "Cloud native architecture and design: A handbook for modern day architecture and design with enterprise-grade examples", 2022.

[11] J. Hong, T. Dreibholz, J. A. Schenkel, and J. A. Hu, "An overview of multi-cloud computing", *Advances in Intelligent Systems and Computing*, vol. 927, pp. 1055–1068, 2019.

[12] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: State-of-the-art and research challenges", *Journal of Internet Services and Applications*, vol. 1, no. 1, pp. 7–18, 2010.

[13] W. Ye, A. I. Khan, and E. A. Kendall, "Distributed network file storage for a serverless (p2p) network", in *11th IEEE international conference on networks*, IEEE, 2003, pp. 343–347, ISBN: 0-7803-7788-5.

[14] Google Trends. "Google trends - serverless computing". (2022-07-21), [Online]. Available: `https://trends.google.com/trends/explore?q=Serverless%20computing&date=2016-01-01%202022-01-01#TIMESERIES` (visited on 07/21/2022).

[15] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud programming simplified: A berkeley view on serverless computing", 2019.

[16] Cloudflare. "What is multi-cloud?", [Online]. Available: `https://www.cloudflare.com/learning/cloud/what-is-multicloud/` (visited on 07/07/2022).

[17] D. Baur, D. Seybold, F. Griesinger, H. Masata, and J. Domaschka, "A provider-agnostic approach to multi-cloud orchestration using a constraint language", *Proceedings - 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2018*, pp. 173–182, 2018.

[18] A. Sheth and A. Ranabahu, "Semantic modeling for cloud computing, part 2", *IEEE Internet Computing*, vol. 14, no. 4, pp. 81–84, 2010.

[19] C. Davis, "Realizing software reliability in the face of infrastructure instability", *IEEE Cloud Computing*, vol. 4, no. 5, pp. 34–40, 2017.

[20] F. Paraiso, P. Merle, and L. Seinturier, "Socloud: A service-oriented component-based paas for managing portability, provisioning, elasticity, and high availability across multiple clouds", *Computing*, vol. 98, no. 5, pp. 539–565, 2016.

[21] E. Nogueira, A. Moreira, D. Lucrédio, V. Garcia, and R. Fortes, "Issues on developing interoperable cloud applications: Definitions, concepts, approaches, requirements, characteristics and evaluation models", *Journal of Software Engineering Research and Development 2016 4:1*, vol. 4, no. 1, pp. 1–23, 2016.

[22] The Economist. "The battle of the computing clouds is intensifying". (2021), [Online]. Available: `https://www.economist.com/business/the-battle-of-the-computing-clouds-is-intensifying/21806813` (visited on 07/20/2022).

[23] F. Dandria, S. Bocconi, J. G. Cruz, J. Ahtes, and D. Zeginis, "Cloud4soa: Multi-cloud application management across paas offerings", *Proceedings - 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2012*, pp. 407–414, 2012.

[24] K. Keahey, M. Tsugawa, A. Matsunaga, and J. Fortes, "Sky computing", *IEEE Internet Computing*, vol. 13, no. 5, pp. 43–51, 2009.

[25] Z. Zhang, C. Wu, and D. W. Cheung, "A survey on cloud interoperability", *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 4, pp. 13–22, 2013.

[26] Á. López García, E. Del Fernández Castillo, and P. Orviz Fernández, "Standards for enabling heterogeneous iaas cloud federations", *Computer Standards & Interfaces*, vol. 47, pp. 19–23, 2016.

[27] A. N. Toosi, R. N. Calheiros, and R. Buyya, "Interconnected cloud computing environments", *ACM Computing Surveys (CSUR)*, vol. 47, no. 1, pp. 1–47, 2014.

[28] D. Elliott, C. Otero, M. Ridley, and X. Merino, "A cloud-agnostic container orchestrator for improving interoperability", *IEEE International Conference on Cloud Computing, CLOUD*, vol. 2018-July, pp. 958–961, 2018.

[29] R. B. Bohn, J. Messina, F. Liu, J. Tong, and J. Mao, "Nist cloud computing reference architecture", *Proceedings - 2011 IEEE World Congress on Services, SERVICES 2011*, pp. 594–596, 2011.

[30] R. O. ROSS, V. Pillitteri, G. Guissanie, R. Wagner, R. Graubart, and D. E. BODEAU, *Enhanced security requirements for protecting controlled unclassified information: A supplement to nist special publication 800-171*, Gaithersburg, MD, 2021.

[31] K. Morris, *Infrastructure as code: Dynamic systems for the cloud age / Kief Morris*, Second edition. Sebastopol, CA: O'Reilly, 2020, ISBN: 978-1-098-11467-1.

[32] Y. Brikman, *Terraform Up & Running: Writing Infrastructure as Code*, Second edition. Beijing China and Sebastopol, CA: O'Reilly Media, 2019, ISBN: 978-1-4920-4690-5.

[33] S. Challita, F. Korte, J. Erbel, F. Zalila, J. Grabowski, and P. Merle, *Model-based cloud resource management with tosca and occi*, 2020-01-22.

[34] T. Metsch, A. Edmonds, R. Nyren, and A. Papaspyrou, "Open cloud computing interface – core", in *Open Grid Forum, OCCI-WG, Specification Document. Available at: http://forge. gridforum. org/sf/go/doc16161*, 2010.

[35] T. Metsch, A. Edmonds, and B. Parák, "Open cloud computing interface - infrastructure", in *Standards Track, no. GFD-R in The Open Grid Forum Document Series, Open Cloud Computing Interface (OCCI) Working Group, Muncie (IN)*, 2010.

[36] T. Metsch and M. Mohamed, "Open cloud computing interface - platform", *Recommendation GFD-RP*, vol. 227, 2016.

[37] B. Parák, F. FLORIAN, K. PIOTR, S. MAIK, and Z. Šustr, "The rocci project - providing cloud interoperability with occi 1.1", in *Proceedings of International Symposium on Grids and Clouds (ISGC) 2014*, Proceedings of Science, 2014.

[38] A. Brogi, J. Soldani, and P. Wang, "Tosca in a nutshell: Promises and perspectives", *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7908, pp. 171–186, 2014.

[39] P. Lipton, D. Palma, M. Rutkowski, and D. A. Tamburri, "Tosca solves big problems in the cloud and beyond!", *IEEE Cloud Computing*, 2018.

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

[41] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, and S. Wagner, "Opentosca – a runtime for tosca-based cloud applications", *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6470, pp. 692–695, 2013.

[42] "Tosca version 2.0 specification". (2022), [Online]. Available: https://docs.oasis-open.org/tosca/TOSCA/v2.0/csd04/TOSCA-v2.0-csd04.pdf (visited on 07/07/2022).

[43] A. F. Baarzi, G. Kesidis, C. Joe-Wong, and M. Shahrad, "On merits and viability of multi-cloud serverless", *Proceedings of the ACM Symposium on Cloud Computing*, pp. 600–608, 2021.

[44] The Apache Software Foundation. "Apache libcloud: Apache libcloud is a standard python library that abstracts away differences among multiple cloud provider apis". (2022-05-26), [Online]. Available: https://libcloud.apache.org/ (visited on 07/24/2022).

[45] B. Di Martino, G. Cretella, and A. Esposito, *Cloud Portability and Interoperability: Issues and Current Trends*. Springer, Cham, 2015, ISBN: 978-3-319-13701-8.

[46] The Apache Software Foundation. "Apache jclouds". (2022-04-27), [Online]. Available: https://jclouds.apache.org/ (visited on 07/24/2022).

[47] B. Di Martino, D. Petcu, R. Cossu, P. Goncalves, T. Máhr, and M. Loichate, "Building a mosaic of clouds", *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6586, pp. 571–578, 2011.

[48] D. Petcu, B. Di Martino, S. Venticinque, M. Rak, T. Máhr, G. Lopez, F. Brito, R. Cossu, M. Stopar, S. Šperka, and V. Stankovski, "Experiences in building a mosaic of clouds", *Journal of Cloud Computing*, vol. 2, no. 1, p. 12, 2013.

[49] M. Aazam and E.-N. Huh, "Framework of resource management for intercloud computing", *Mathematical Problems in Engineering*, vol. 2014, pp. 1–9, 2014.

[50] M. Aazam and E.-N. Huh, "Cloud broker service-oriented resource management model", *Transactions on Emerging Telecommunications Technologies*, vol. 28, no. 2, e2937, 2017.

[51] E. Kamateri, N. Loutas, D. Zeginis, J. Ahtes, F. D'Andria, S. Bocconi, P. Gouvas, G. Ledakis, F. Ravagli, O. Lobunets, and K. A. Tarabanis, "Cloud4soa: A semantic-interoperability paas solution for multi-cloud platform management and portability", *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8135 LNCS, pp. 64–78, 2013.

[52] HashiCorp. "Terraform configuration language", [Online]. Available: https://www.terraform.io/language (visited on 07/07/2022).

[53] HashiCorp. "Terraform module composition: Multi-cloud abstractions", [Online]. Available: https://www.terraform.io/language/modules/develop/composition#multi-cloud-abstractions (visited on 07/07/2022).

[54] M. Momberg. "Provision aws services through kubernetes using the aws service broker". (2018), [Online]. Available: `https://aws.amazon.com/blogs/opensource/provision-aws-services-kubernetes-aws-service-broker/` (visited on 07/07/2022).

[55] AWS. "Aws cloudformation template formats", [Online]. Available: `https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/template-formats.html` (visited on 07/07/2022).

[56] Multy. "Multy: Write cloud-agnostic config deployed across multiple clouds", [Online]. Available: `https://multy.dev/` (visited on 07/07/2022).

[57] Multy. "Multy documentation". (2022-07-14), [Online]. Available: `https://docs.multy.dev/` (visited on 07/27/2022).

[58] A. J. Ferrer, D. G. Pérez, and R. S. González, "Multi-cloud platform-as-a-service model, functionalities and approaches", *Procedia Computer Science*, vol. 97, pp. 63–72, 2016.

[59] G. Breiter, M. Behrendt, M. Gupta, S. D. Moser, R. Schulze, I. Sippli, and T. Spatzier, "Software defined environments based on tosca in ibm cloud implementations", *IBM Journal of Research and Development*, vol. 58, no. 2/3, 9:1–9:10, 2014.

[60] O. Kopp, T. Binz, U. Breitenbücher, and F. Leymann, "Winery – a modeling tool for tosca-based cloud applications", in *Service-oriented computing*, ser. LNCS sublibrary. SL 2, Programming and software engineering, P. P. Maglio, Ed., vol. 6470, Springer, 2010, pp. 700–704, ISBN: 978-3-642-17357-8.

[61] U. Breitenbücher, T. Binz, F. Leymann, and D. Schumm, "Vino4tosca: A visual notation for application topologies based on tosca", in *In CoopIS*, Springer, 2012, pp. 416–424.

Appendices

A. TOSCA Template Example

```yaml
tosca_definitions_version: tosca_simple_yaml_1_3

description: Exemplary definition to deploy a MySQL instance

topology_template:
  inputs:
    mysql_rootpw:
      type: string
    mysql_port:
      type: integer

  node_templates:
    db_server:
      type: tosca.nodes.Compute
      capabilities:
        host:
         properties:
           num_cpus: 2
           disk_size: 12 GB
           mem_size: 2048 MB
        os:
          properties:
            architecture: x86_64
            type: linux
            distribution: rhel
            version: 6.5

    mysql:
      type: tosca.nodes.DBMS.MySQL
      properties:
        root_password: { get_input: mysql_rootpw }
        port: { get_input: mysql_port }
      requirements:
        - host: db_server
```

Listing A.1: A TOSCA YAML template that requests a compute resource with MySQL installed.

B. Architecture Definition Schema

```yaml
kind: Schema
metadata:
  name: Architecture
spec:
  metadata:
    type: dict
    required: true
    schema:
      name:
        type: string
        required: true
      version:
        type: string
        required: false
      labels:
        type: dict
        required: false
  spec:
    type: dict
    required: true
    schema:
      platforms:
        type: list
        required: true
        nullable: false
        schema:
          type: dict
          required: true
          schema:
            name:
              type: string
              required: true
            properties:
              type: dict
              required: false
      components:
        type: list
        required: true
        nullable: false
        schema:
```

```
41          type: dict
42          required: true
43          schema:
44            name:
45              type: string
46              required: true
47            type:
48              type: string
49              required: true
50            properties:
51              type: dict
52              required: false
53              nullable: true
54              valuesrules:
55                anyof:
56                  - type: string
57                  - type: reference
58                  - type: dict
```

Listing B.1: The YAML schema of the architecture definition file.