Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Framework for Migrating Deployed Serverless Applications

Ayhan Kaplan

Course of Study:	Informatik
Examiner:	Prof. Dr. Dr. h. c. Frank Leymann
Supervisor:	Vladimir Yussupov M.Sc.
Commenced:	2019-04-18
Completed:	2019-10-18

Abstract

Serverless computing has reached a degree of maturity that leads to many providers establishing themselves on the market with their own platform offerings. Although companies would benefit from using these offerings, the risk of locking applications into the provider's ecosystem is still a crucial aspect that influences the final decision on whether to adopt serverless to their business. Consequently, it is desirable to support developers in moving applications, which are already deployed on one serverless platform, to another in order to reduce the restraint of using these platform offerings. In this work, we develop methods on how to support developers in porting existing serverless applications from one provider to another and conceptualize them into a common framework. Furthermore, we describe a conceptual system architecture for implementing the introduced concepts and validate it by means of a prototypical implementation. Finally, we conduct a case study which showcases the prototypical framework in a real world scenario.

Kurzfassung

Das Serverless Computing Paradigma hat einen Reifegrad erreicht, der zahlreiche Cloud Provider dazu veranlasst hat, sich mit eigenen Plattformangeboten auf dem Markt zu etablieren. Obwohl viele Unternehmen von der Nutzung dieser Angebote profitieren würden, ist das entstehende Risiko des Vendor-Lock-Ins, nach wie vor ein zentraler Aspekt, der die Entscheidung über die Integration von Serverless in die Unternehmensprozesse beeinflusst. Folglich ist es erstrebenswert, Entwickler bei der Portierung von Anwendungen, die bereits auf einer Serverless-Plattform eingesetzt werden, zu unterstützen, um somit einen Rückgang der defensiven Haltung gegenüber dieser Plattformen zu erwirken. Diese Arbeit beschäftigt sich mit der Erforschung und Entwicklung von Methoden, die Entwickler bei der Portierung bestehender Serverless-Anwendungen von einem Anbieter auf einen anderen unterstützen. Darüber hinaus umfasst diese Arbeit die Konzeption eines aus diesen Methoden bestehenden, umfassenden Frameworks und den Entwurf einer konzeptionellen Systemarchitektur zur Umsetzung der vorgestellten Konzepte. Auf Grundlage dieser Systemarchitektur wird eine prototypische Implementierung des Frameworks beschrieben und diese anschließend im Rahmen einer Fallstudie validiert.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Over the past decades, the evolution of software delivery has gone through several major paradigm changes. Starting with monolithic application architecture, in which all system components are developed on a single codebase [VGC15], the trend has moved towards increasingly distributed architectures of service-oriented application systems (SOA) [Sch11]. While SOA paved the path for novel approaches, the next step in the evolution of application architecture has emerged with the adoption of microservices [LF15], in which systems are split into modular and fine-grained compositions of business capabilities.

In the modern era of computing, the trend for hosting applications is increasingly moving in the direction of cloud-based solutions [BLS11]. While cloud computing introduces numerous novel advantages, such as, fine-grained payment models, dynamic scaling of resources, and managed administration of the underlying infrastructure [MG11], it also introduces trade offs which come with the adoption of managed cloud services, inter alia, locking applications into the cloud provider's ecosystem.

Serverless computing, often referred to as serverless, is an emerging cloud computing paradigm for deploying event-driven, cloud-native applications. The areas which are described by the term *serverless* are inconclusive: while serverless is commonly referred to Function as a Service (FaaS), the term was initially used to describe applications for which a major part of components depends on third-party, cloud-hosted services [RF18]. FaaS refers to an aspiring service model for hosting small-factor applications, commonly called *functions*, inside ephemeral containers of managed cloud services. Serverless Computing by no means implicates that there is no demand for servers, however, serverless platforms abstract away the actual infrastructure on which the code runs from the tenant. Developers focus on writing functions which implement the business logic and can be triggered by numerous types of events of the platform.

One significant innovation that comes with cloud computing is that cloud providers are able to support elastic scaling of resources as managed service for their customers. While the majority of traditional cloud service models are based on a subscription-based payment model, e.g., charging for dedicated infrastructure, even for idle times [BCC17], serverless marks the next step in the evolution of elastic scalability. Function invocations are executed in ephemeral containers that are destroyed after termination which results in the possibility of scaling the servers to zero for idling serverless functions.

A plethora of FaaS platforms arose in the recent years, such as, AWS Lambda, Microsoft Azure Functions, and IBM Cloud Functions. Certainly, different providers built their platforms using different design decisions regarding several aspects, such as, programming models for functions, or supported types of event sources and services which can be incorporated to FaaS-based applications. As a result, there is a strong tendency for vendor lock-in in FaaS platforms.

In this work, we capture methods and concepts on how to support developers in analyzing existing serverless applications and moving them to another platform. The investigated methods are conceptualized into a common framework which provides serverless migration support for developers.

In accordance with the introduced framework, we describe a conceptual system architecture and provide technical details on its prototypical implementation. Finally, a preliminary case study is conducted on a real world scenario to validate the migration support concept by showcasing the framework's prototypical implementation.

Outline

The remainder of this work is structured as follows:

Chapter 2 – Fundamentals: introduces fundamental concepts and provides background knowledge to the research field of this work.

Chapter 3 – Motivation: discusses the fundamentals and provides the motivation background to our research.

Chapter 4 – Concepts and Design: presents the concepts and design of a conceptual support framework for porting already deployed serverless applications across providers.

Chapter 5 – Implementation: describes the framework's conceptual system architecture and provides technical details on its prototypical implementation. Further, the prototype is validated through a case study on a real world scenario.

Chapter 6 – Related Work: discusses related scientific work addressing the problem domain and approaches which are relevant to the design decisions made for our concept.

Chapter 7 – Conclusion and Outlook: summarizes the work. The findings of our research are concluded and an outlook to potential future work is provided.

2 Fundamentals

Serverless computing, like many other emerging paradigms of the past, is an outcome of demands which have not been satisfied before it arose. Traditional approaches of software development rely on monolithic application architecture. Over the past decades software architecture has been reinvented and refined which resulted in *service-oriented architecture* (SOA), and *microservice architecture*. This chapter introduces fundamentals of architectural styles, the concepts of cloud computing and serverless computing.

2.1 Monolithic Applications

In the era before cloud computing, traditional software solutions were typically developed using monolithic architectural style and on-premise infrastructure hosts. Monoliths are single-tiered applications, or systems, which enable clients to communicate with a single application. Typically, the application logic is separated into three layers, namely, *presentation*, *application logic*, and *resource layer*. All components are developed on a single codebase [VGC15], that is, the components are tightly coupled to each other which consequently results in a *single point-of-failure*. In order to avoid having a single point of failure, modern applications typically are scaling out, which means, that additional machines with an identical instance of the application are provisioned to increase resiliency for, e.g., server crashes or overflow of incoming requests for one machine. In complex software, there is most probably the demand for some components to be scaled independently from other due to varying workload that needs to be managed and the *service level agreements* (SLA) they must be compliant to. Monolithic applications are scaled by replicating the application on multiple machines including all the components it consists of (Figure 2.1). Certainly, scaling a monolith leads to under- and over-provisioning which can be attributed to the lack of dynamic scalability [Gra06]. Although it is desirable to replace traditional systems for reasons like inflexibility and maintenance costs, there are still many legacy systems that run critical business processes for which introducing serious bugs while re-engineering the system for a new platform is too high of a risk [KH07]. Researchers have spent big effort and time in figuring out *enterprise architecture patterns* for transforming big monolithic applications into modern systems [Ern08; Fow02; PI13].

2.2 SOA and Microservices

While monolithic applications struggle with issues such as inflexibility, fault-tolerance and rapidly increasing complexity proportional to the number of components, service-oriented architecture emerged as a new solution to these issues with a component based distributed architecture [Sch11]. Architectural components are divided into more fine-grained *services*, which are largely decoupled from each other in their communication and choreography to perform business processes. While components are units of software which are independently replaceable and upgradeable [HC01], services are out-of-process components which are typically invoked via some form of remote or in-procedure calls [NS14]. Krafzig et al. [KBS05] classify four key elements of SOA:

- *Application Frontends.* Web frontend with graphical user interface (GUI) that initiates business processes and receives the results.
- *Services.* Software components which encapsulate high-level business concepts.
- *Service Repository.* Enables Service-Discovery for service-consumers and provides additional information, such as physical location, information about the provider, technical constraints, available service levels, etc.
- *Service Bus.* Enables communication between services and application frontends in a decoupled manner. Conceptually, the service bus is similar to the concept of a software bus as it is defined in the context of *Common Object Request Broker Architecture* [ZM95] (CORBA).

Provided that these services are independent from each other, critical aspects like resiliency and service lifecycle management greatly benefit from designing system in a service-oriented manner [XWQ16]. However, services typically strongly vary in scale; services in SOA may represent components that range from small business processes to large applications or even subsystems. While SOA has been a step away from monolithic application architecture with the implementation of modular, distributed application functionality [FLR13], microservice architecture emerged as a novel architectural design pattern with the intent to refine the service-oriented approach. A microservice is described as a small, self-contained application which is implemented independently from any other service it communicates with. Microservices typically implement an independent, single-purpose functionality which belongs to a composition of additional services and forms a microservice architecture-based application. Unlike in SOA, microservices aim to share as little as possible across the composition of services, that is, in microservice architecture the governance is decentralized [LF15] with respect to assumptions that two or more communicating services must

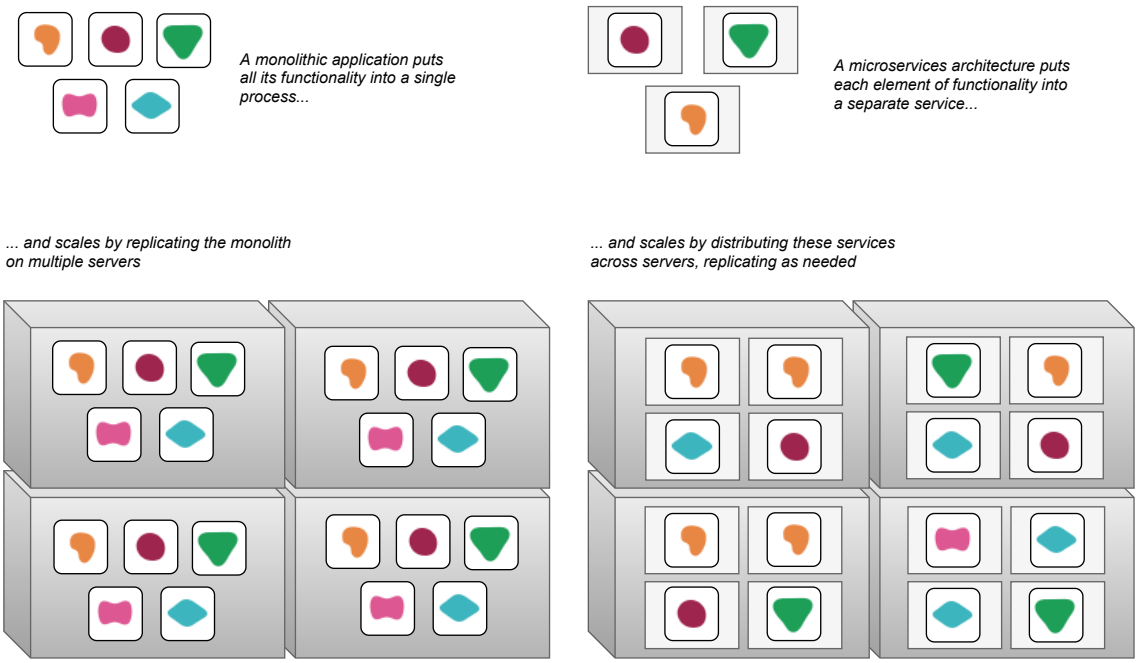


Figure 2.1: Architecture of monolithic and microservice applications [LF15]

make about each other. This leads to a loosely-coupled architectural style [Kay03] where services communicate with each other using lightweight protocols with uniform, predefined sets of stateless operations [UHM11].

Microservice architectures highly benefit from the modular and fine-grained composition of business capabilities into microservices in terms of flexibility in form of module boundaries which allow to implement each service independently in any programming language and using any dependencies and libraries exclusively inside of this particular bounded context. Furthermore, using microservice architecture enables applications to dynamically scale services independently from each other, which diminishes the risk of under- and over-provisioning (Figure 2.1). Compared to monolithic applications, the resilience of the overall system increases significantly by avoiding the single-point-of-failure [VGC15]. However, developers must implement microservices applying appropriate design patterns to enable fault-tolerance (Design for Failure [Han13; Nyg18]).

2.3 Cloud Computing

Cloud computing is a paradigm that focuses on enabling ubiquitous, 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 efforts or service provider interaction [MG11]. Mell et al. describe the five essential characteristics of cloud computing:

On-demand self-service. A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service provider.

Broad network access. Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, tablets, laptops, and workstations).

Resource pooling. The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is a sense of location independence in that the customer generally has no control or knowledge over the exact location of the provided resources but may be able to specify location at a higher level of abstraction (e.g., country, state, or datacenter). Examples of resources include storage, processing, memory, and network bandwidth.

Rapid elasticity. Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time.

Measured service. Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service.

2.3.1 Service Models

Traditional service models offered by cloud providers include infrastructure, platform-based offerings or even complete software stacks as managed service. In the following, the traditional cloud service models are introduced:

Infrastructure as a Service (IaaS). Consumers are able to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. Consumers do not manage or control the underlying cloud infrastructure but has control over operating systems, storage, and deployed applications; and possibly limited control of selected networking components (e.g., host firewalls). [MG11]

Platform as a Service (PaaS). Consumers can deploy applications onto the cloud infrastructure using programming languages, libraries, services, and tools supported by the provider. Consumers do 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. [MG11]

Software as a Service (SaaS). Consumers are enabled to use the provider's applications running on a cloud infrastructure. The applications are accessible from various client devices through either a thin client interface, such as a web browser (e.g., web-based email), or a program interface. Consumers do not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings. [MG11]

The three models can be considered as levels of abstraction with respect to the depth of services that are managed by the cloud provider in the application stack. Starting with IaaS, the corresponding cloud provider manages the infrastructure of the virtual servers, whereas in PaaS-offerings, the consumer is provided a pre-configured platform on which the consumer deploys the desired application on the provided runtime. At last, SaaS-offerings provide a managed service to the consumer in which the complete application stack is managed by the provider [MG11]. Exemplary SaaS-offerings may be web-based E-Mail applications or streaming services. Figure 2.2 depicts a comparison of cloud service models with a traditional on-premise application stack. Beyond the general classification of cloud service models, there are numerous additional cloud service models such as *Database as a Service* (DBaaS), *Containers as a Service* (CaaS), or *Function as a Service* (FaaS).

2.3.2 Virtualization in the Cloud

There are different layers of virtualization corresponding to the previously described cloud service models. While dedicated machine architecture has no virtualization which results in strong dependencies on the underlying hardware, cloud computing relies on the subdivision of physical resources [AZ18]. Starting with IaaS, the cloud provider manages the infrastructure of the virtual servers which are often referred to as *virtual machines* (VM). This type of virtualization is called *server virtualization* which come with *hypervisors*. Hypervisors abstract the hardware of a physical, shared server into virtualized hardware [FLR14] and can be classified in two types: *Type I hypervisors* which run directly on top of the underlying hardware, e.g., in server virtualization, and *Type II*

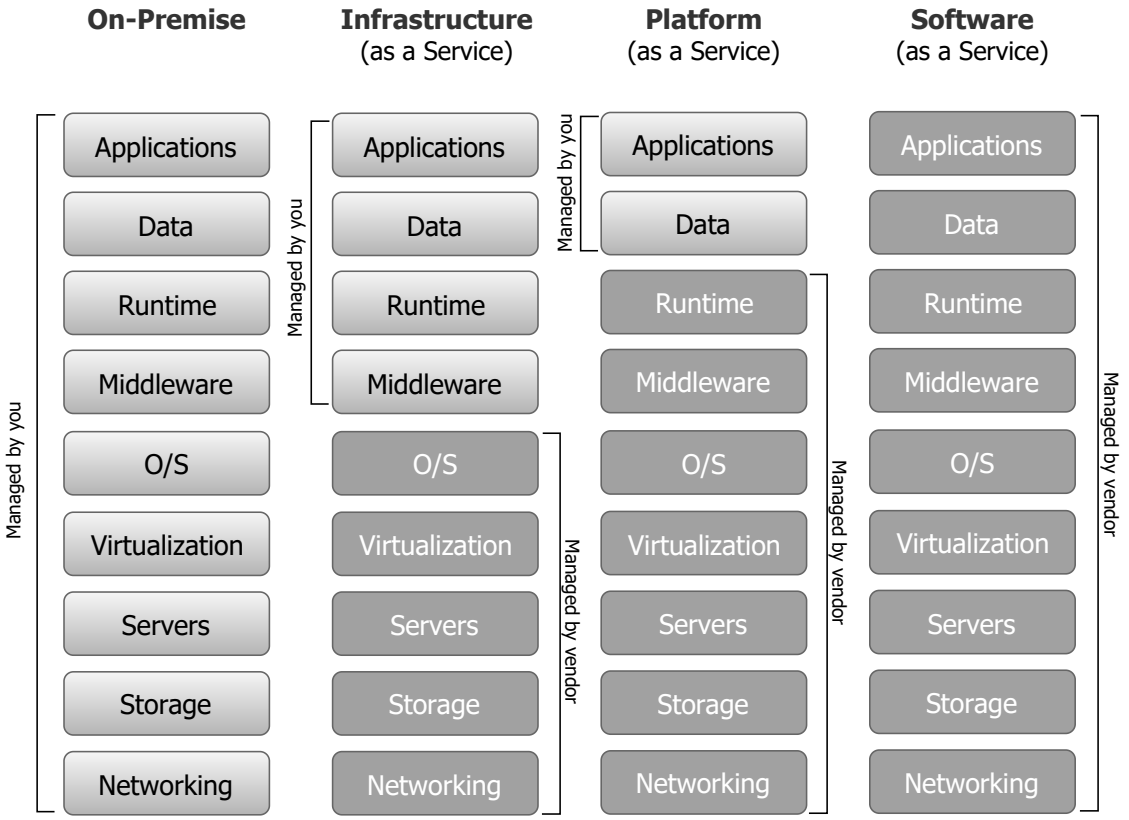


Figure 2.2: The general Cloud Computing service models compared with a traditional on-premise application stack [Cho09].

hypervisors which run as applications in a operating system [DOSP13]. Type II hypervisors are typically used in *OS-level Virtualization*, which allows the existence of multiple isolated user-space instances on the same kernel. The corresponding service model layer for this virtualization type is CaaS which can be classified as a layer between IaaS and PaaS [CNC18]. Finally, PaaS-offerings can be considered as virtualization on application-level by providing a virtual environment for an application within an OS (e.g. Java Virtual Machine [TPA14]) and consequently enables the application to run on different operating systems [Sab11].

Containers as a Service (CaaS). Applications are *cloud-native* if they are developed explicitly for running in cloud environments and are designed to benefit from the advantages of cloud environments [FLR14]. One common approach for deploying cloud-native applications is to build them upon container technology in order to have isolated, packaged features. Typically, FaaS and PaaS offerings are built on top of *container orchestration platforms* [CNC18; PBSJ17], such as *Kubernetes* [The19] and *Docker Swarm* [Doc19] to exploit the reusability and portability of container technologies while maintaining full control over the infrastructure [CNC18].

2.4 Serverless Computing and Function as a Service

Serverless computing, often referred to as serverless, is an emerging cloud computing paradigm for deploying event-driven cloud-native applications. The term serverless by no means implies that there are no demand for servers anymore, however, developers of serverless applications do not have to deal with any operational management with respect to provisioning, monitoring, maintenance, scalability and resilience of the servers the application is deployed on [BCC17]. One might argue that these characterstics are also given in PaaS-offerings, therefore there is a demand for clarification of the difference between serverless and PaaS. While PaaS also abstracts away the server management, the deployment and payment models are different to those of serverless applications. Both platform models are usually based on containerization with additional managed layers on top of the containerized environments which abstracts away the management of the containers and operating systems. In PaaS, pre-packaged applications are deployed on a provided runtime of the corresponding platform, hence, developers still need to take care of the application structure [Wig17]. Observe that there are two cloud service models that fit nicely into the serverless computing paradigm [WBK18]:

(Mobile) Backend as a Service (BaaS). Originally, the term serverless was used to describe applications for which the majority of features partly or fully depend on third-party services in order to manage state and server-side logic. These applications are also often referred to as Backing Services (Twelve-Factor App [WBFL17]).

Function as a Service (FaaS). In FaaS, developers write server-side logic and deploy it on a platform where it runs in event-driven, stateless, ephemeral containers which are managed by a third party [RF18]. This paradigm aligns well with the idea of serverless computing and represents a significant part of the serverless paradigm in the cloud context. In the remainder of this work, the term *serverless application*, if not explicitly defined otherwise, refers to FaaS-based applications which use additional provider-managed services as their sub-components.

While PaaS-hosted applications are more of traditional nature with respect to application architecture, serverless provides a platform for rapidly deploying small pieces of cloud-native code that responds for events, for instance, to coordinate mircoservice compositions that would otherwise run on the client or on dedicated middleware [BCC17]. Similar to PaaS, FaaS-based applications consist of stateless code, referred to as functions, for handling incoming events, however, the composition of these functions within an application is not restricted to one single programming language. More specifically, each function can be written in another language out of a set of supported languages the FaaS platform provides. Certainly, this is possible since FaaS-based applications consist of n functions and m event sources $(n, m \geq 1)$ from which each of the functions is deployed on a separate runtime and the functions are decoupled from the implementation of the different event sources. Essentially, serverless functions are single-purpose functions that are part of a workflow which conceptually gives FaaS platforms capabilities similar to a event processing system [BCC17].

One remarkable innovation that comes with FaaS is the ability of scaling the servers down to zero during idle periods of the application and thus no charges for the client of the platform while the application is not used. Although this also requires developers to consider issues like cold

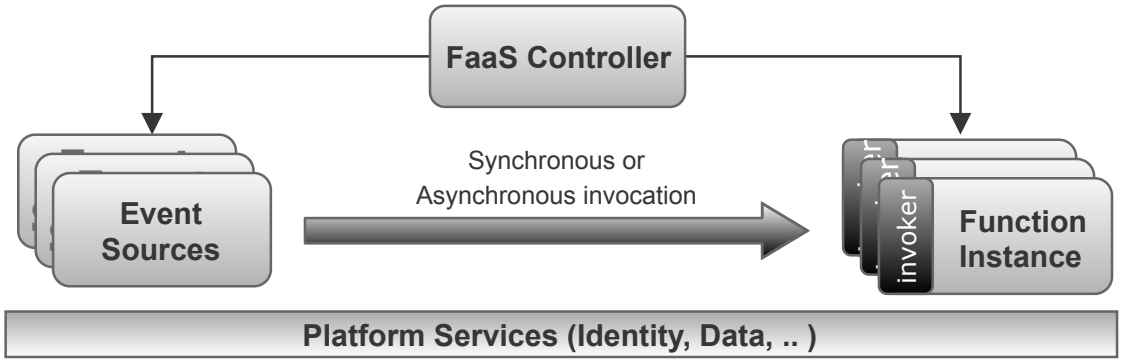


Figure 2.3: Interaction of key elements in the generic architecture of a FaaS-based application [CNC18].

start time, the fact that the deployed applications are fully managed by the provider, more than ever, enables developers to focus on writing business logic instead of dealing with operational concerns [BCC17].

2.4.1 Serverless Processing Model

This section illustrates the generic architecture of FaaS-based applications and summarizes the interaction of the key components within the serverless ecosystem. The generic architecture of a FaaS-based application typically consists of four key components [CNC18]:

- *Function instances* - Single-purpose functions or microservices which can be scaled with demand
- *Event Sources* - Events which invoke one or more function instances. Such events may be trigger or stream events.
- *FaaS Controller* - Responsible for deployment, monitoring and control of function instances and their corresponding sources.
- *Platform services* - General cluster or cloud services used by the FaaS application (often referred as BaaS). In most cases, these services are dependent from the serverless platform provider and tightly coupled into their ecosystem.

Unlike in monolithic application architecture, serverless applications align with the architectural model of microservices. The server-side logic of the application is divided into many small, short-lived functions which preferably implement one single and simple functionality. Observe that using this modern approach of application architecture consisting of fine-grained stateless functions, the application highly benefits in scalability when compared with traditional application architecture. FaaS-based applications incorporate event sources for the invocation of one or more functions which execute specific parts of the business logic. These *trigger events* may be synchronous as well as asynchronous which is dependent on the type of event source that handles incoming requests or reacts on modification-based events, for instance, of a database service. There are different types of possible event sources which include, but are not limited to *event steaming* and *messaging*, such as *message queueing* or *publish-and-subscribe*, *storage services*, such as object storage or

NoSQL databases (DBaaS), *endpoint services*, such as HTTP Gateways or IoT [CNC18], and other serverless functions, as well. Note that the catalogue of the supported event source services depends from the platform provider and can vary strongly. The following section introduces the most relevant FaaS platforms and analyzes similarities between their offerings.

2.5 FaaS Platforms

Cloud providers offer infrastructure, ready-to-use software solutions, and runtimes for the special needs of their customers. Equally, the majority of global cloud providers offer a platform for Function as a Service. Cloud vendors provide FaaS platforms on which developers can deploy functions and build FaaS-based applications. Beyond providing a runtime for FaaS, each of them offer an ecosystem of services which can be used in order to implement an event-driven application consisting of easy-to-integrate vendor-specific cloud services. The cloud provider is responsible for automated scaling, fault tolerance, and operational concerns with respect to the underlying infrastructure. In the following, we introduce the most well-established cloud providers [Eva19] which offer platforms and other services around the development of FaaS-based applications.

2.5.1 Commercial Platforms

Although *Amazon Web Services* was not the first vendor to provide a managed service for FaaS [Rao19], AWS Lambda [Ama19k] is known as the first serverless platform with outstanding commercial success and thus is considered as an early adopter of the serverless paradigm. *Microsoft Azure* can be considered as a direct competitor to AWS. Microsoft introduced their own serverless platform including the FaaS service *Azure Functions* [Mic19b]. The origin of Azure Functions lies in a technology called *WebJobs*[1] which is a service that enables event-driven programming for .NET console applications. With the growing interest of the market in FaaS, the support for multiple languages was added to WebJobs which subsequently formed the Azure Functions platform [BA18]. *IBM Cloud* provides a holistic solution of cloud services including IaaS, PaaS, storage offerings, etc., which makes IBM one of the potential competitors of AWS and Azure and even surpasses Google Cloud [Eva19]. *IBM Cloud Functions* [IBM19c] is IBM's managed service for FaaS which can be considered equivalent to Lambda and Azure Functions, however, there is one significant aspect in which the platform differs from the other offerings, namely, that Cloud Functions built on top of the open-source serverless platform *Apache OpenWhisk* [Apa19d].

In the following, the introduced platforms are analyzed to provide a detailed overview of how serverless platforms are set up. The providers are analyzed for their *programming model* for the written functions, *deployment methodology*, and for their *application models* and how FaaS-based applications are integrated into the provider's ecosystem.

Programming Model

Typically, serverless functions are written in a stateless, short-lived style and follow the platforms respective programming model. The following discusses the FaaS programming model for the introduced platforms.

[1] https://docs.microsoft.com/azure/app-service/webjobs-create

Handler. The *handler* represents a function which will be invoked by the platforms runtime. Functions implement the business logic which gets triggered as a response to incoming events. Typically, the event data is passed to the handler through the runtime, e.g., AWS Lambda, Azure Functions, or IBM Cloud Functions. The way in which the data, often referred to as *context*, is passed to a handler depends on the language it is written in. As an example, handlers written in JavaScript typically get passed a JSON-formatted object as first parameter of the function signature [Ama19n; Mic19c]. The context object enables the communication between handlers and runtime. Each platform handle this communication differently, e.g., context in Azure Functions are used both for input and output of a handler [Mic19c], whereas in AWS Lambda, an event data object is passed to the handler with the payload of the request, as well as, a separate context object is passed to the handler which provides information about the caller and can be used for returning values via callbacks [Ama19n]. In IBM Cloud Functions, the function gets passed a set of key-value-pairs which represent the event data [IBM19d]. The handler processes incoming event data and may invoke any other function or service in the serverless application. By convention, the handler code needs to be written in a stateless style with isolation of the logic from the underlying compute infrastructure [Ama19n].

Logging. Logging in distributed computing is typically a bigger challenge as it is in traditional systems [KR09]. This also applies in FaaS-based applications in which functions typically rely on proprietary logging solutions. Lambda functions can contain logging methods which will be written to AWS' logging service CloudWatch [Ama19c]. Similarly, functions on Azure can create metrics by passing logs to the context object in order to write trace output to the console. In contrast to these active design decisions, IBM Cloud Functions have logging automatically enabled in the runtime for activities including event data flow, function invocation, and the corresponding output. The log trace is available via provider-native logging services, e.g., IBM Cloud Log Analysis [IBM19e].

Exception Handling. Whenever a function communicates with other services, errors may raise which are originated from the underlying service's API or similar. Hence, a function execution may terminate successfully or result in an error. In both cases, the outcome is ideally communicated with the serverless runtime [Ama19n]. Different platforms provide different ways to handle exceptions, including managed services, best practices, or, requiring developers to write the functionality inside the handler themselves. In Azure Functions, some of the provided services have built-in retry support, which means that the request will be repeated until the invocation is successful or a pre-defined limit of retries is reached [Mic19d]. Otherwise, exception handling can be performed using integrated error monitoring service, e.g., through *Grafana* [Gra19] dashboards in IBM Cloud Functions, or writing exception handling and logging methods in the handler code directly.

Concurrency. Typically, serverless functions handle one request at a time [Ama19n]. Whenever multiple requests are sent to a function which still processes another request, the stateless style of the deployed function is exploited for scaling the respective service by running additional container instances. To be more precise, a container is created for a function invocation which can only handle a single function at a time, that is, a new container is created for each request and consequently enables scaling out and high availability. FaaS platforms typically utilize an optimization mechanism, called warm start, in which container instances are reusable for further invocations of the corresponding function [AL19]. Consequently, existing container instances for a specific function can be reused for subsequent processing of queued requests to this specific function before being destroyed [Thö16].

FaaS Platform	Synchronous triggers	Asynchronous triggers	Resource Bindings
AWS Lambda	Amazon API Gateway	Amazon S3	Amazon S3
	Amazon Cognito	Amazon SNS	Amazon Kinesis
	Amazon Alexa	Amazon SES	Amazon DynamoDB
	Amazon CloudFront	AWS CloudFormation	
	Elastic Load Balancing	Amazon CloudWatch	
	Amazon Lex	AWS CodeCommit	
	Kinesis Data Firehose	AWS Config	

Table 2.1: Native supported AWS Services which can be integrated into applications based on AWS Lambda

Application Model

The introduced platforms provide a runtime for functions which set up FaaS-based applications along with other incorporated services. FaaS-based applications comprise one or more serverless functions, which typically can be written in different languages, event sources, and other service offerings. For commercial FaaS platforms, these incorporated event sources and services are offerings from a set of supported services of the provider's cloud ecosystem, e.g., from AWS (Table 2.1). The interaction between provider services and serverless functions includes function invocations to resource lifecycle events, e.g., updates on a storage service, explicit triggers, such as HTTP requests, consumption of events from a queue, or recurring triggers based on timed events. These events can follow a synchronous, as well as, an asynchronous style or rely on an active resource binding to the function.

One common approach to deploy FaaS-based applications across the platforms is to use a declarative modeling by utilizing the concept of *infrastructure as code* [Mor16] (IaC). The majority of well-established cloud providers provide tools for deployment modeling of resources within their cloud ecosystem, e.g., AWS *CloudFormation* [Ama19i] or *Azure Resource Manager* [Mic19f] (ARM). In more traditional cloud-native applications, access from a particular service, or application, to another service has been handled via hardcoded API calls in the source code. Typically, the interaction between components of a FaaS-based application are modeled via manifest files to avoid hardcoded access to other services inside the code of a function which improves the transparency of the application architecture. Amazon provides a proprietary *serverless application model* [Ama19l] (SAM) which is an extension of CloudFormation for Lambda-specific application models. The main focus lies on SAM templates, which represent the application architecture written in the providers *domain specific language* (DSL). Furthermore, SAM provides a *command line interface* (CLI) which enables developers to test locally and to deploy the application to Lambda. Similarly, Azure Functions allows developers to deploy FaaS-based applications via Azure Functions *Core Tools* which provides the Azure Functions runtime to locally develop and test the applications and deploy afterwards [Mic19h]. Applications based on IBM Cloud Functions typically are developed similar to native OpenWhisk-based applications, hence, the usage of *package manifest* files is utilized to deploy FaaS-based applications on IBM's FaaS platform [Apa19a].

2.5.2 Open-Source FaaS Platforms

The current trend in the field of deploying cloud-native applications moves to the direction of container orchestration using Kubernetes clusters [PBSJ17]. Kubeless [Bit19] is an open-source framework for developing serverless applications which run on top of Kubernetes. The framework represents a Kubernetes-native approach for adding FaaS support to standard Kubernetes clusters. That is, serverless functions are common custom resource definitions (CRD) in the cluster and the interaction not bound to a third-party CLI but handled with the common `kubectl` tool. One significant advantage of using a Kubernetes-native approach is that the platform is extensible with respect to runtime and event implementation plugins for the functions. Further FaaS platforms on Kubernetes clusters are Oracle's Fn Project [Ora16], Fission [Pla17], Riff [Piv19b] and Knative [Kna19]. The developed FaaS-based applications are in an isolated, provider-agnostic architecture which can easily be deployed on any cloud provider within a Kubernetes offering or on-premise and thus avoiding the risk of being locked into the vendors ecosystem. On the other hand, the application is locked into a specific technology which can be considered as an appropriate trade-off in certain scenarios.

Alternatively, developers can create FaaS-based applications on top of *OpenFaaS* [Ell17]. OpenFaaS is a lightweight open-source framework which enables to build a serverless platform on top of Kubernetes and Docker Swarm. Serverless applications can be deployed on-premise or on any private and public cloud. OpenFaaS consists of an API Gateway that can be exposed to any kind of event, such as services from AWS, Azure or open-source services, such as Minio [Min19] or MongoDB Atlas [Mon19a]. Furthermore, the deployment can be performed manually through the UI portal or automated using the OpenFaaS-CLI with manifest files for templating and defining functions in YAML format.

2.5.3 Frameworks for FaaS development

There are various frameworks which support developers in building and deploying FaaS-based applications on existing platforms. These *enablers* [WLZ18] can be compared to provider-specific tools, such as, SAM, Azure Function Core Tools, or OpenWhisk packages. There are plenty of task-specific frameworks for developing serverless FaaS-based applications, such as *Zappa* [gun19], and *Claudia.js* [Adz19] which support developers in simplifying the process of deploying *Django*[2] apps and NodeJS APIs, respectively.

General purpose Frameworks

There are several frameworks that support developers in the deployment of serverless applications to the most prominent platform providers. In the following, we will introduce two of these frameworks which cover the development and deployment of serverless applications to both commercial and open-source platform providers.

Terraform HashiCorp Terraform [Has19] is an open-source DevOps tool that provides a holistic solution for infrastructure provisioning across the majority of cloud providers (currently over 100 providers are supported) covering nearly every service offering within the providers ecosystems.

[2] https://www.djangoproject.com/

Terraform allows to define infrastructure as code [Mor16] through a declarative programming model that enables developers to deploy and manage infrastructure across the supported providers without the necessity of using provider-specific deployment tools for each provider, separately. Along a very broad range of cloud services of the various platforms, Terraform also supports the user in the deployment of serverless applications for the corresponding providers. Embracing the style of IaC, Terraform allows to define manifest files for declarative modeling of the target infrastructure similar to the way it would have been done through the vendor-specific tools, such as AWS CloudFormation or Azure Resource Manager. This configuration file is written in the *HashiCorp configuration language* (HCL) syntax which consists of several types of configuration *blocks* which are associated with a particular entity in the infrastructure. The following will describe a small excerpt of the most relevant blocks:

- *Provider.* Configuration of the provider with all necessary information, for instance, the region (Listing 2.1).
- *Resource.* Description of one or more infrastructure objects. The header consists of the resource type and its name, whereas the type is a composition of the provider name as prefix and the actual resource type as suffix (Listing 2.1).
- *Module.* Configuration of a container for multiple resources which represent a logical group.

The Terraform workflow is divided into three steps. First, a declarative modeling including relevant resources and parameters is created. Since Terraform uses a plugin-based architecture, the providers are encapsulated in their own binaries. This configuration will be validated via the Terraform CLI using `terraform init` and download the needed binaries in order to create an *execution plan*. Next, the execution plan is displayed to the user in the style of a *git diff-format* by running `terraform apply`. Finally, after verification of the execution plan, the execution will be confirmed and subsequently deployed on the configured cloud provider. Providing a holistic solution for fine-grained infrastructure management, Terraform does require to configure necessary IAM policies manually by the user. This brings both advantages and disadvantages with it, depending on the complexity of the project that needs to be managed. Generally, the usage of Terraform is hard to learn and thus developers need to invest time into this learning phase.

Listing 2.1 Example resource configuration of AWS Lambda functions using Terraform

```
# Configure the AWS Provider and create a Lambda function with pre-defined IAM policies
provider "aws" {
  version = "~> 2.0"
  region  = "us-east-1"
}

resource "aws_lambda_function" "lambda_broadcast_function" {
  filename      = "lambda_broadcast_payload.zip"
  runtime = "nodejs8.10"
  ...
}
```

Serverless Framework The Serverless Framework [Ser19] is an open-source project which enables users to develop and deploy serverless architectures on various existing platforms, such as AWS Lambda, Azure Functions, IBM Cloud Functions, and Kubeless. Compared with Terraform, Serverless also provides a generic deployment tool for different existing providers, however, the focus is clearly on developing FaaS-based applications and included resources are restricted to native supported services within the provider's ecosystem. A CLI is provided for deploying serverless applications which are modeled inside manifest files using Serverless' DSL for serverless architectures (Listing A.1).

Functions can be written in different languages and dependent on different runtimes, as long as the target platform supports this. Generally, the built-in set of supported event service types are relatively limited compared to Terraform. However, in some cases, like Azure Functions, there is an option to define additional bindings which are supported by Azure Functions. The deployment models are deployed through the SLS CLI when running `serverless deploy` which will translate the manifest file to calls of the corresponding platform API to dynamically define the resources.

Serverless Inc. asserts that their tool enables developers to deploy serverless applications in a provider-agnostic manner, however, this is only partially accurate. Since the manifest files are translated to provider-specific API calls, the definition of deployment models varies strongly between the providers. For instance, when defining a deployment for AWS, additional resources are declared inside a *resources* key which accepts raw CloudFormation syntax. On the other hand, Azure Functions handles function outputs in a completely different way by declaring them directly inside a function as events with an output direction. Furthermore, the types of events are specific for each provider with respect to the supported services and also the properties inside those events are tightly coupled to those of the vendor-specific programming model. Consequently, it is not possible to mix up multiple providers per deployment model to compose different serverless functions over multiple providers.

2.6 Portability and Interoperability

Section 2.5.1 describes how commercial providers tend to provide native support for incorporation of services into their serverless application model. These services typically comprise a subset of the rich ecosystem of managed cloud services which constitutes the provider's serverless offering, e.g., *AWS Serverless* [Ama19o]. While the usage of such rich service ecosystems is attractive for developers, it most often leads to a dependence on the provider's proprietary solutions, hence, brings a risk of locking in the application to the providers platform [BCC17]. Conceptually, there is a set of cloud services that the cloud providers have in common, however, they are likely implemented in different ways and most often offer distinct kinds of features inside the cloud providers ecosystem. This particularity of commercial FaaS platforms are convenient for developers because of the simplified development of event-driven applications by augmenting the user's functions with different service offerings but more so leads to *vendor lock-in*. Cloud vendors provide non-standardized solutions for all kinds of services in order to ensure an optimized performance for their service models which consequently locks applications consisting of these solutions to the platform [OST16]. In particular, the migration of such an application to another platform requires intense re-engineering effort which makes it even harder to switch the provider.

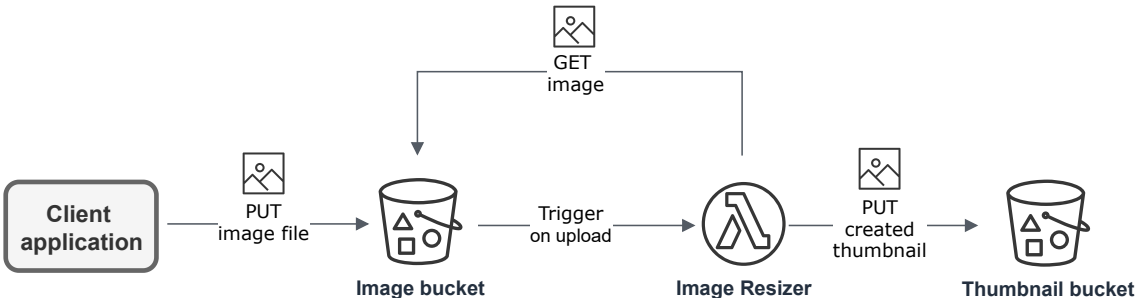
The vendor lock-in applies as soon as a FaaS-based application needs to be moved to another platform or partly migrate features across different platforms for the application, e.g., integrate third-party services from other cloud platforms [OST16]. These scenarios can be classified to two concepts: (1) *portability* and *interoperability* of cloud components. Cloud interoperability denotes the ability to communication between cloud platforms or components, e.g., replacing a service offering of one providers ecosystem with a similar service of another platform [Pet11]. For instance, a customer may decide to deploy a hybrid cloud application in which services from different cloud platforms exchange information. Cloud portability describes the degree to which an application, or parts of it, can be moved to another platform as-is with little to no modification [KW14]. As an example, migrating an application, which is deployed on AWS and incorporates services from the providers ecosystem, to Microsoft Azure or some open source platform requires portability of the incorporated services and component of the application from AWS to the selected target platform.

3 Motivation

While the serverless computing paradigm comes with numerous benefits, there are also issues that the adoption of this event-driven approach introduces which need to be considered before implementing a serverless application on one of the providers' FaaS platforms. The most notable issue is the strong tendency for vendor lock-in. While factors, such as, potentially locking an application into the providers ecosystem can be evaluated in prior to deployment, moving an already existing application turns out to be even harder. Certainly, many potential challenges arise in the context of porting existing FaaS applications across providers.

There are several well founded reasons for moving applications from one platform to another. First, there is a constant competition between cloud providers to improve their services with respect to pricing of commodity hardware, availability, or quality of service [BCC17]. Customers are interested in using the offering which delivery the best performance possible for their investments. Further reasons for moving away from a provider may be that the business model has a major shift in technology trends or underlies legal issues, that is, moving away from a provider's platform may be mandatory for legal reasons like compliance concerns [BBSR13]. One significant factor for switching providers can be attributed to licensing for the implementation of services, such as, non-relational storage offerings that are closed-source and hence the customer cannot have full knowledge about what exactly the corresponding provider is doing with the, most often sensible, data. Furthermore, one common considered risk is that the provider stops offering one or more of the services their customers rely on, however, this represents an exceptional scenario [Eic19].

Figure 3.1 illustrates the architectural outline of two exemplary FaaS applications which are deployed on AWS Lambda and may be ported to another platform. Figure 3.1a showcases *ThumbNailer*, a common serverless use case, in which a client, e.g., some web application exposing a *graphical user interface* (GUI), puts an image into a bucket. This bucket serves as event source for a Lambda function which gets triggered as soon as a new image file is uploaded to the respective bucket and gets the newly uploaded image file by using the meta data provided in the event data. The function generates a thumbnail of the image via third-party library usage and puts the resulting image file into a *thumbnail bucket*. Another common use case is illustrated in Figure 3.1b. *MailNotifier* represents an event processing application which exposes a public API via *API Gateway* and triggers the *Event Transmitter* function while passing the event data as payload to the function handler. The event transmitter transforms the request into a message format and sends it to a *Message Queue* which subsequently passes the message to the *Notifier* function. The notifier processes the message and sends a notification to a list of e-mail subscriptions via a *Publish-and-Subscribe channel*. The potential issues that might arise in the introduced example applications are incompatible service offerings at different providers, such as, having message queues as event sources for functions, or having different types of supported triggers for changes inside of object storage offerings, e.g., lacking the possibility to filter the trigger events exclusively on changes in terms of uploads into a data store.



(a) ThumbNailer

(b) MailNotifier

Figure 3.1: Architectural outline of two common FaaS application architectures. ThumbNailer is a thumbnail generation application which generates and persists a resized version of every newly added image file in a particular bucket. MailNotifier represents an event processing application which notifies a list of subscribed e-mail addresses about an occurring event.

By the time of this work, several approaches have been developed and proposed in order to evade vendor lock-in by enabling inter-cloud communication for using composite serverless architectures across multiple platforms. Serverless Inc. introduce *Event Gateway*; an open-source event router designed to fill the missing piece of FaaS-based application architecture. It simplifies the wrapping of functions with HTTP endpoints, react to business events and to share event subscriptions in order to enable a multi-cloud solution for serverless applications. Event Gateway is comparable with event routing services, such as Azure Event Grid[1], however, works with platform-independent events. Furthermore, CNCF Serverless Working Group [Clo19a] works collaboratively on *CloudEvents* [Clo19c], a specification for describing event data of cloud services in a common way to normalize the schemas for events. The concept of normalized schemas aims to achieve compatibility for common types of events in heterogeneous systems. CloudEvents and Event Gateway support developers of cloud applications in means of interoperability of cloud components between different providers. While these approaches aim to provide a solution for the vendor lock-in by scattering applications over multiple cloud platforms, there are several reasons why applying a multi-cloud approach is not applicable to any migration use case. Scattering a serverless application over multiple cloud platforms will inevitably cause higher costs since the pricing is based on the computation time of the instance and the usage of services which will include factors, such as latency between cross-provider calls or cold-starts for chained functions within another provider. Finally, many users still fear data leakage even when using DBaaS-systems inside the same cloud provider, which are deployed near the computational infrastructure [YBD08].

[1] https://docs.microsoft.com/azure/event-grid/overview

In our context, the term migration can be referred to as the interoperability of a set of the applications components with services from another provider, as well as, to the portability of the application, e.g., moving it to another provider's platform. Portability of applications raises non-trivial challenges for added abstraction layers between the developer and the operational level of the platform. For instance, IaaS cloud services typically stick to standards which enable portability of applications, such as OVF [DMT19] and the provision of commonly used operating systems like Linux [BDE14]. Starting with PaaS, environments can vary widely between the vendors, hence, typically require an extensive amount of reengineering ranging from service discovery to source code refactoring [BBSR13]. FaaS platforms represent an even higher abstraction layer when compared with PaaS. As a result, FaaS applications typically comprise multiple artifacts which are specific to the vendors platform and consequently leads to an increasing amount of reengineering when moving them to another platform.

In this work, we capture methods and concepts on analyzing existing serverless applications and moving them to another platform. The introduced applications in this chapter can be considered as motivational use cases for moving serverless applications from one provider's platform to another. We aim to answer the following research question:

> *"How can we support developers in moving already existing serverless applications to another platform?"*

4 Concepts and Design

This chapter presents the concepts and design of a support framework for porting serverless applications which are hosted on FaaS platforms. Beginning with a coarse-grained overview of the migration phases for porting existing FaaS-based applications to another provider, the conceptual phases are introduced in brevity. The remaining subsections focus on the specific phases of the framework and describe them in detail.

A Support Framework for Porting Serverless Applications

Typically, the process of porting an already deployed FaaS-based application is cumbersome and comprises a significant amount of manual work. We propose a conceptual support framework for porting serverless applications from one provider's platform to another by taking over considerable parts of the migration. Figure 4.1 presents an overview of the concept, which comprises six migration phases: (1) *Discovery*, (2) *Intermediate Transformation*, (3) *Platform Evaluation*, (4) *Migration Model Generation*, (5) *Preparation & Refinement*, and (6) *Automatic Deployment*. In the following, the migration phases are described in brevity before going into detail for each phase in subsequent sections. In the remainder of this work, we use the terms *porting* and *migration* interchangeably. Note that the focus of our approach is on porting FaaS-based applications between provider's platforms.

Discovery The first step is to investigate an existing application to get a high level representation of its constituents. That is, the information of the components that comprise the application, e.g., deployment models, components' source codes, incorporated services, etc., must be retrieved at an architectural level. The application is analyzed with respect to the way it has been deployed on the source provider's platform (Section 4.1.1). The outcome of this analysis determines whether the necessary information needs to be extracted or if there is already a *deployment package* available which contains the necessary knowledge about the application. In this context, a deployment package contains a set of necessary files and information, such as, source code of the functions and corresponding configuration files for dependencies, deployment model files which present a declarative modeling of the application's architecture, and other resources which are needed to have a package that allows to successfully deploy the application. In Chapter 3, we introduced two example applications, ThumbNailer and MailNotifier. An example of a sufficient deployment package for ThumbNailer may be a set of source files which describe the application's architecture including knowledge about *incorporated services*, e.g., the S3 buckets for images and thumbnails with their types of bindings to the Lambda function. The deployment package typically contains an application model which describes the application's architecture. Depending on the way ThumbNailer has been deployed to AWS, there may be SAM templates or other deployment models available, e.g., Terraform or Serverless Framework models. Furthermore, the deployment package contains the function *handlers* and additional dependencies, corresponding to the used language, e.g., `node_modules` for functions written in JavaScript.

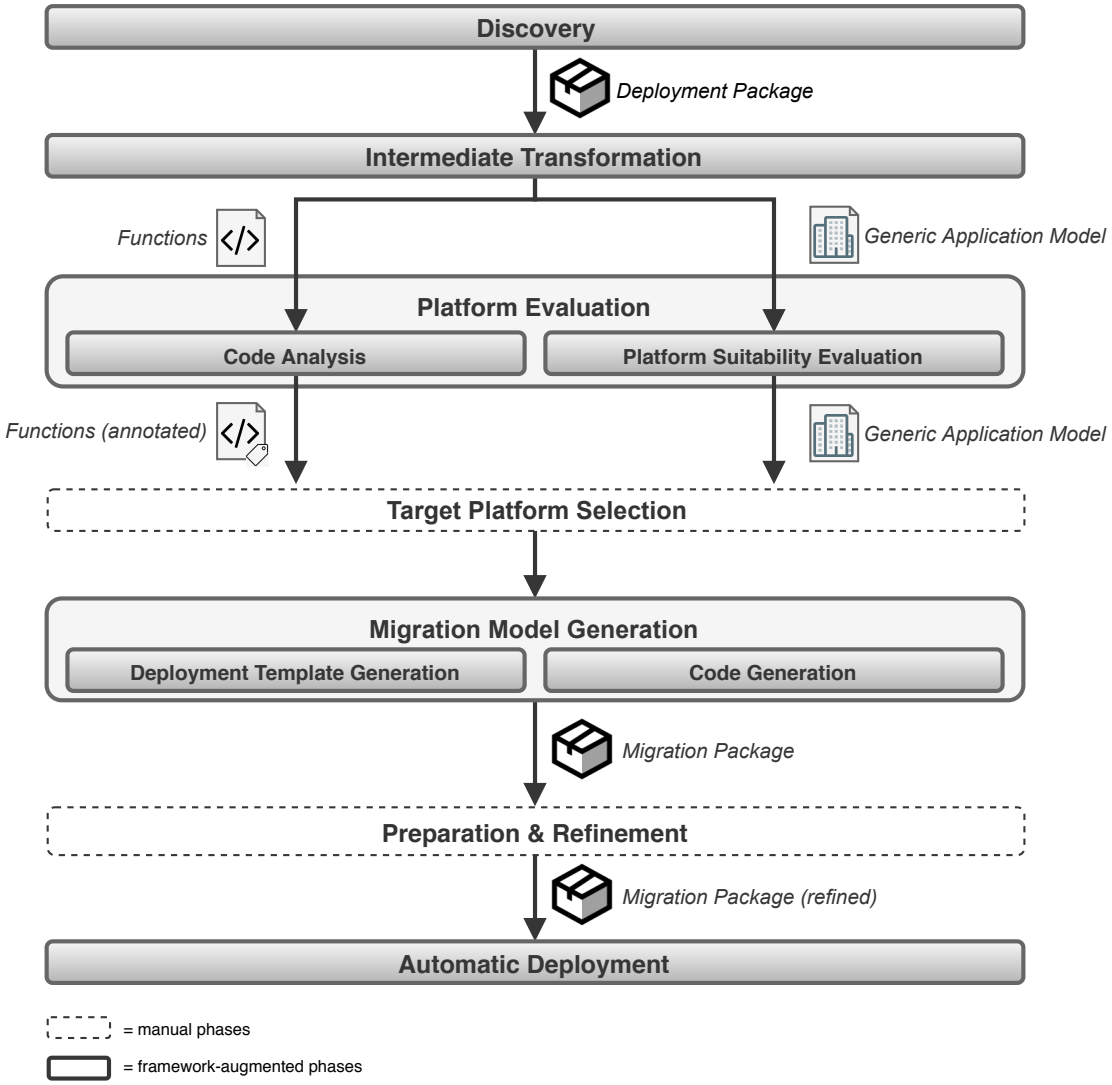


Figure 4.1: A stepwise diagram of the serverless porting roadmap

Intermediate Transformation After the needed information is extracted, the application model is transformed into an intermediate format with generic descriptions for the resources in the source application, e.g., S3 buckets or API gateways. The intermediate transformation is achieved by performing model mapping using a generic meta model and a repository for service resource mapping. The result of this phase an updated version of the application model in a more generic format.

Platform Evaluation After discovery of the applications architecture, depending on the task's context, the next step is either to discover qualified target providers for porting the application or to evaluate the suitability of an already chosen provider's platform. The evaluation phase deals with the suitability of either one chosen target platform or a range of possible *platform candidates* for migrating the extracted application. Each provider has a corresponding instance of the generic application model which describes the set of its supported service resources in a generic manner that is uniform with the mapped model of the extracted application. The evaluation comprises two sub-

phases: (1) *platform suitability evaluation* and (2) *code analysis*. In platform suitability evaluation, the incorporated services of the source application are checked for compatible counterparts within the target platform's set of supported services. In the code exploration phase, the deployment packages is analyzed for included functions. This is done interactively by supporting users in annotation tasks for available source codes. The evaluation provides metrics which will be used to classify the target platform with respect to the relative amount of effort for the migration task and support users in the decision making process of selecting a suitable target platform.

Migration Model Generation Subsequently to the selection of the desired target platform, a *migration package* is generated. The migration package presents a baseline for an automatically deployable deployment package for the chosen target platform's runtime. This includes generating a suitable architecture which conforms the target platform's application model. In addition, implementation artifacts are generated with boilerplate codes for the functions of the source application's deployment package to align the functions to the target platform's programming model.

Preparation & Refinement The generated migration package contains an equivalent instance of the source application's deployment package, which is aligned with the target platforms model. Typically, there is still manual work to do for preparing the migration package for the deployment. Developers must deal with refinements, such as, access management, refactoring the generated function templates. Depending on the context, it might be needed to manually provision the incorporated resources. Finally, a *refined migration package* is produced which can be deployed automatically to the target platform.

Automatic Deployment The goal of the migration package refinement is to facilitate developers to have an automatically deployable deployment package for the chosen target platform. Having such a deployment package, developers can test the application for correctness and compliance concerns using sandbox environments or virtual test scenarios on the target platform. Finally, the application can be deployed on the target platform.

The described migration phases provide an overview of the proposed concept. The following sections provide detailed insights into these phases and further concepts which augment the framework.

4.1 Discovery

Typically, the architecture of traditional software systems is documented and modeled before development and deployment. Most often these documentations represent the initial envisioned architecture and differ from the actual implementation of the existing piece of software. Therefore, the discovery of traditional software architecture is potentially time consuming and requires an extensive amount of manual verification aligned with complex tools, such as KDM [Obj16] and SMM [Obj18]. FaaS-based applications, on the other hand, typically consist of a single deployment package which contains only a few files that are needed to deploy a functioning application. Accordingly, embracing the usage of IaC and available deployment packages is most often feasible in the retrieval of all relevant components for an existing application.

One crucial requirement for supporting the decision process on porting existing applications is that the relevant sources are accessible. This is mandatory since the existing application needs to be analyzed in order to facilitate support to developers in the migration task. Accessible sources

may either be already available sources, e.g., deployment packages which were used to deploy the application, or denote that while the sources are not available yet, the developer has sufficient permissions to extract them from the platform. The extraction rises some challenges. Either the developer must own the same role or permission credentials which were used to deploy the application or the original account for the deployment must be handed over.

Several scenarios exist for the deployment of FaaS-based applications which are typically supported for the majority of platforms: *Manual deployment*, *Deployment via vendor-specific interfaces*, and *Deployment via third-party frameworks*. Each of the analyzed providers supports, and, sometimes encourages, developing serverless applications manually using the platforms browser-based GUI. Following this approach, the underlying application model is almost completely abstracted away from the tenant, thus, provides basically no transparency regarding the structural properties of the components. Alternatively, the application can be developed within a local project which will then be bundled into a deployment package and remotely deployed through the provider CLI, e.g., using SAM templates. The majority of vendors provide proprietary interfaces for remote deploying FaaS projects. Typically, the application infrastructure is defined via declarative deployment models which rely on the concept of IaC. Remote deployments are also possible using third-party deployment tools, e.g., Serverless Framework and Terraform (Section 2.5.3).

4.1.1 Discovery Workflow

The discovery of the application's architecture aims to gather all necessary sources that set up a functioning deployment package. The described deployment methods represent decisive factors in the discovery process. Typically, cloud-native applications are version controlled during development and maintenance which makes the collection of necessary sources straightforward for this type of applications. Although FaaS-based applications are part of the cloud-native domain, there are several scenarios that need to be considered when collecting those sources (Figure 4.2). First, an investigation of which deployment method has been used to deploy the application needs to be performed. The deployment methods are classified as follows:

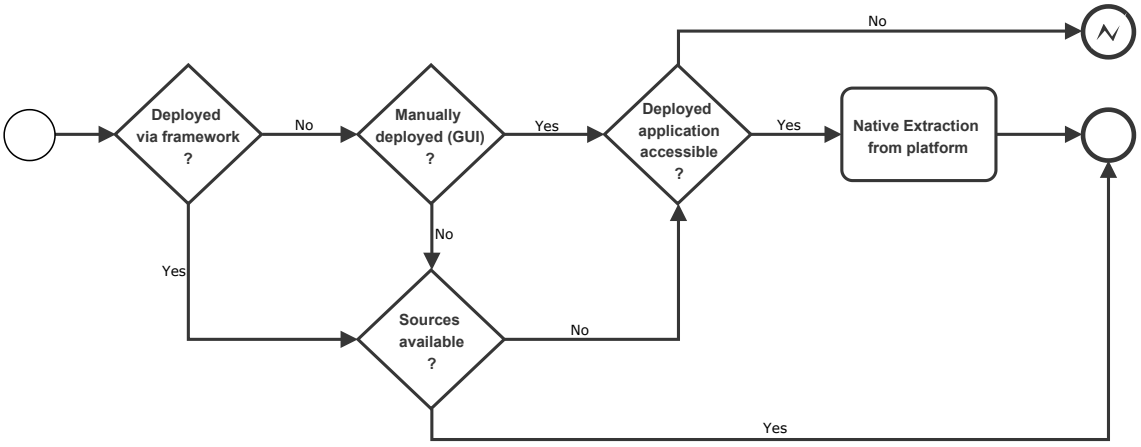


Figure 4.2: Conceptual workflow for the discovery process of existing FaaS-based applications.

Using a third-party framework: One possibility is that the application is deployed using a third-party framework, such as Terraform or Serverless Framework, for which the developer needs to create a deployment package including a manifest file to declaratively model the application architecture. Typically, these sources are stored within an accessible repository.

Manually via web portal: Another possibility is that the application is developed and deployed manually over the provider's web portal. In this case, the sources that constitute the deployment package are typically stored in a repository or storage offering instance of the platform.

Provider-specific deployment tools: If none of those two deployment methods are used, we assume that the application has been developed locally and deployed using provider-specific deployment tools, e.g., SAM, Azure Functions Core Tools, or IBM Cloud Functions CLI with OpenWhisk packages. Provided that the deployment package is typically developed locally, these sources may be available outside of the provider's platform in an external repository or storage which is accessible for the developer.

If the deployment package with all necessary configuration files and sources is available, the discovery phase terminates. Otherwise, the deployment either has been performed manually over the web portal or the sources are unavailable and need to be extracted from the platform. Based on the assumption that the deployed application sources are accessible, the deployment package is extracted from the existing application. The extraction of the application's deployment package can be achieved in several ways. Typically, the user can manually download a zipped archive of the deployment package over a management console in the web portal. Alternatively, the analyzed platforms provide APIs from which the application's deployment package can be crawled. This approach depends on the platform's provided proprietary interfaces which can be a provider-specific CLI[1], e.g., SAM CLI, or a GET API call to a REST interface which is implemented at the corresponding applications web location [Mic19i]. In the remainder of this section, we refer to this approach as *native extraction*.

For remotely deployed applications, that is, locally developed projects which have been deployed using the providers deployment tooling, the native extraction of the deployment package is straightforward. This can be attributed to the usage of the same interface for both deployment and extraction, e.g., SAM for AWS Lambda. Typically, native extraction is also feasible in scenarios where the application is deployed over a third-party framework. For instance, both Serverless Framework and Terraform provide domain-specific languages to model the applications infrastructure. As soon as the deployment is performed, the framework translates the specific models to the providers native application model and subsequently deploys it on the corresponding platform. Consequently, the platform is unaware of how the application is developed since the provided deployment model is passed in the provider-compatible format.

4.2 Intermediate Model Transformation

This phase focuses on the transformation of the application model in the retrieved deployment package into a generic format (Figure 4.3). In the remainder of this work, the source application model is referred to as *platform-specific model* (PSM). One significant part of the transformation phase is the static analysis of the PSM with respect to the source application's infrastructure

[1] https://docs.aws.amazon.com/cli/latest/reference/lambda/get-function.html

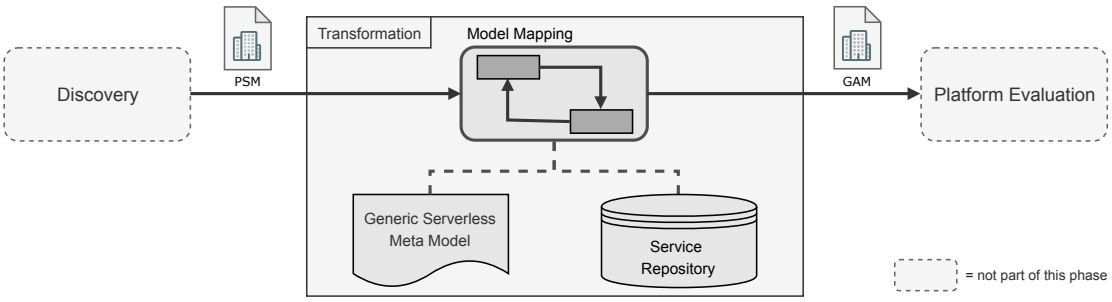


Figure 4.3: The PSM gets mapped into a generic application model using the generic serverless meta model and a service repository for mapping the provider-specific services to generic service categories.

including the interconnections of functions and incorporated services, e.g., object storage offerings. Incorporated services denote all cloud service offerings that are integrated into the application architecture, whereas they may be event sources or resources that get invoked by FaaS-hosted serverless functions. Since different platforms typically provide proprietary DSLs for defining these interconnections, it is desirable to transform the PSM into a intermediate format with a generic way to model the relations between components. The PSM is transformed into a *generic application model* (GAM), which modifies the PSM in two ways:

- *Generic description of components in application architecture and their interconnections*
- *Generic resources instead of provider-specific services*

In order to perform this transformation, we need a meta model which describes the architecture of FaaS-based applications in a provider-agnostic way. Furthermore, the application model contains provider-specific resources which need to be mapped to corresponding service categories which might represent generic resource identifiers that can be applied across all provider's application models. This mapping task can be performed using a repository which holds information about provider-specific service offerings and maintains knowledge about each service offerings characteristics in order to classify them to a provider-agnostic service category. For instance, ThumbNailer incorporates two S3 buckets which can be classified as instances of a *object storage* service. MailNotifier incorporates an API gateway, SQS, and an SNS topic, which can be classified as *endpoint*, *message queueing*, and *publish-and-subscribe-channel*, respectively.

The knowledge must be aligned with the documentations of the providers and maintained over time, especially because serverless computing is a relatively novel paradigm. Vendors push updates to their platforms in a very fast pace, such as new supported event sources from their existing service ecosystem or proprietary solutions, specific to their serverless offering. This leads to new possibilities of mappings between the provider's service offerings or suitable service alternatives for migration of self-hosted applications to commercial platforms - and vice versa. It is conceivable that this repository is handled like a domain-specific knowledge base which is kept alive by community-based contribution, similar to *CloudEvents* [Clo19c] and *CloudComparer* [Fll19].

4.3 Generic Serverless Meta Model

Section 4.2 points out the necessity of a meta model which describes serverless application topology in a provider-agnostic way. This section introduces the concept of a *generic serverless meta model* (GSMM) which enables a description of serverless application topology across the providers.

4.3.1 Application model comparison

In Section 2.5, the most relevant cloud providers are analyzed for their FaaS platform offerings. In the following, these platforms are compared with respect to their application model for FaaS-based applications in order to derive a conceptual platform-independent description for serverless FaaS application architecture.

One common ground across modern serverless platforms is the usage of declarative deployment models embracing the concept of IaC [Mor16]. Typically, these deployment models are materialized in form of one or multiple manifest files. This approach enables developers to avoid hardcoded access from FaaS-hosted functions to other service instances, hence, significantly improves the transparency of the application architecture. Although the architecture of the analyzed application models is comparable in their basic concepts, they differ format-wise in the declaration of incorporated services and how they model the inter-relationships between different entities, respectively.

SAM relies on the declaration of CloudFormation stacks which represent collections of AWS resources that are generated within one resource collection bundle. Furthermore, SAM simplifies the development of resource stacks for FaaS-based applications by adding several new declaration types to the traditional resources of a CloudFormation template, e.g., a simplified instance model for Lambda functions. Typically, CloudFormation resource instances consist of a `Type` and a set of `Properties`, whereas the content of the properties depends on the type of the resource. For Lambda functions, the corresponding set of properties contains a general configuration of the function, e.g., handler location, runtime, memory size, timeout, code-URI, and a set of `Events` and `Policies`. `Events` describes a set of *event source mappings* which trigger the respective function. This set comprises resources which invoke one or more Lambda functions through synchronous or asynchronous events. `Policies` describe a set of *execution roles* that grant the respective function access to other incorporated services. For instance, creating a document inside a database with given processed data from the function invocation or reading from a database with a query that has been passed to the function through an HTTP trigger.

Similar to AWS, Azure follows the concept of IaC for deployment of serverless applications using ARM templates. Although ARM templates are used to deploy a complete application stack with provisioning of the incorporated services, the actual binding of these services to the FaaS-hosted functions is defined in a dedicated configuration file for each function. Each configuration contains a list of resource bindings for the corresponding function, whereas each binding instance consists of the `type` of the binding, the binding `direction` and a `name` for the binding entity (Section 2.5) as well as some specific properties depending of the binding type. Essentially, for functions on Azure, there is no explicit property that divides incorporated services into invoked services and event sources. Therefore, the way how the relation of an incorporated service to a function is defined, differs significantly from SAM.

The application architecture in IBM Cloud Functions is declared inside a manifest file utilizing the *wskdeploy* model in which the functions are defined as `actions` and the application is a composition of action sequences inside an OpenWhisk package. These action sequences consist of self-written

functions and other actions from incorporated packages which can be pre-installed for interaction with IBM Cloud services or other packages written by developers. Conceptually, packages for interaction with IBM's cloud services contain actions which implement the API calls of the respective service. This very simplistic approach is functionally equivalent to the hardcoded access from the self-written serverless functions, however, slims down the code which has to be written to integrate these services into the application. Action sequences are handled like self-contained actions which are mapped to `triggers` within `rule` definitions in the manifest file. This makes IBM Cloud Functions' definition of trigger events for the implemented functions unique across the evaluated providers. Essentially, this approach is much more generic regarding integration of services which can be attributed to the fact that IBM Cloud Functions is based on the open source platform OpenWhisk.

Typically, open source FaaS platforms, such as OpenWhisk and OpenFaaS, provide basic functionality to deploy serverless functions. Consequently, third-party services can be integrated in a generic fashion without leveraging services from a specific provider over another by adding built-in plugins for convenient integration. For instance, the application modeling approach for OpenFaaS represents a very simplistic declaration of functions and their exposed API gateways. In most cases, function invocations from external services are HTTP-based via the gateway proxy, however, there are also event connector pattern[2] based event source mappings. These mappings create a separate broker or microservices which map functions to topics and invokes them accordingly. Nevertheless, tracing services and resources which are invoked by the deployed functions proves to be much harder because in most cases the communication from function to these services is implemented in form of hardcoded API calls inside the function body.

4.3.2 Conceptual generic meta model

The most noticeable difference between the analyzed application models lies in the way in which bindings of the incoming and outgoing event streams are defined inside the ecosystem of functions and the incorporated services. In the following, a generic serverless meta model for FaaS-based applications is introduced which aims to simplify the comparison of the platform offerings. The GSMM describes the interconnections between functions and incorporated services in a standardized, provider-agnostic manner. For this, we use the terminology of the *Pipes and Filters* pattern [Meu95] since the concept of filters which are connected via pipes can be considered a suitable terminology for event-driven architectures.

Analogous to the traditional Pipes and Filters pattern, functions and services are referred to as *Filters* which process or transform event data and pass it to the next filter through one or more *Pipe* connections. This pipelining of the event data can be considered as event bindings. Filters of type *Function* have n_{f_x} input pipes and m_{f_x} output pipes ($n_{f_x} \geq 1$, $m_{f_x} \geq 0$) whereas filters of type *Service* have n_s input pipes and m_s output pipes ($n_s, m_s \geq 0$).

GSMM describes the architecture of serverless applications in a form of a generic schema to provide transparency over the inter-relationships between the components that constitute the building blocks of the application (Figure 4.4). An application comprises a set of *filters*. Each filter has *properties*, a set of *inputs* and *outputs*, and a *type*. A filter's set of inputs and outputs, comprise the pipes for which the filter is the *target* or *source*, respectively. The type of a filter determines the content of its

[2] https://docs.openfaas.com/reference/triggers/other-event-sources

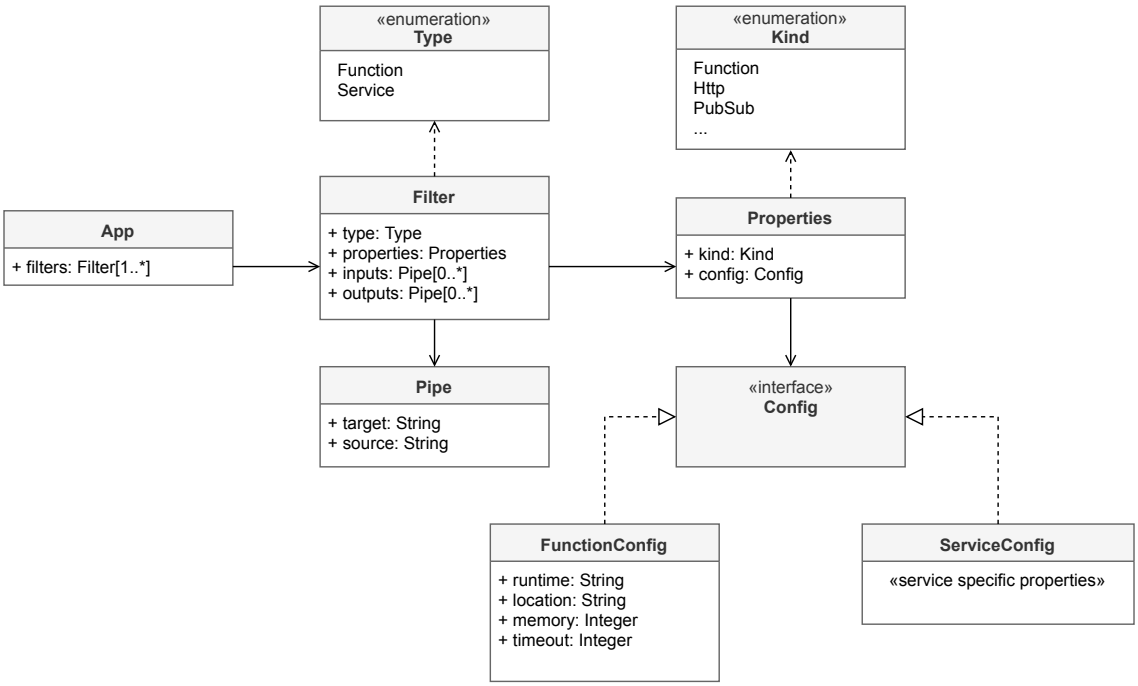


Figure 4.4: Generic meta model based on the contemplated serverless application models

properties object and its set of pipes which connects to other filters, e.g., a filter of type *function* must have at least one *input*. Note that some platform application models, e.g. Azure Functions, distinguish between input bindings and actual triggers of a function. In GSMM, input pipes of a filter with type *function* are equivalent to event sources which invoke functions by passing event data through the pipe, whereas output pipes exclusively describe the output of processed data from the filter component. Practically, a connection between two components comprises both an output pipe from the caller perspective and an input pipe from the callee point of view which assemble to a interconnection of these two filters. The properties of each filter define the *kind* of the filter, e.g., function, endpoint, or event stream, which essentially determines the body of the *config* property. This config property contains the corresponding resource configuration, e.g., topic name and batch size for event streams or host configuration for a serverless function, such as runtime, memory size, and timeout. The GSMM is designed in an extensible fashion for handling the strongly varying configuration properties of third-party service interfaces which is further discussed in Section 4.4. In addition, the GSMM is designed in a uniform format in order to enable semantic mapping to the analyzed platforms application deployment models in a plugin-based manner.

4.4 Service Classification

In serverless computing, applications consist of a collection of multiple building blocks which can be event-driven functions or third-party services. Commercial platform providers encourage developers to integrate their own service resources into the serverless application by supporting an easy-to-integrate pluggable architecture with a set of supported services from their cloud ecosystem. On the other hand, open source platforms do not come with a set of commercial service offerings but most often focus on the deployment of serverless functions and provide basic embedded functionality, such

as scheduled invocation and endpoints. Therefore, migrating a self-hosted serverless application to another platform raises much more effort in looking for alternative services to replace integrated third-party triggers in the open source based application. Typically, incorporated services and resources in serverless applications either serve as *event sources* or as *invoked services* which the functions communicate with. In the following subsections, these two types of incorporated services in serverless applications are discussed and compared across the analyzed providers.

4.4.1 Event Sources

FaaS-hosted functions can be invoked by events originating from different event sources. As described in Section 2.4, function invocations can be synchronous, asynchronous (messaging, queueing), in form of event streams, or run on a schedule. Different providers might support different types of services as event sources for their serverless platform. This represents one of the biggest challenges for porting existing FaaS-based applications since the platform's provided services might rely on different technologies or consist of different types of proprietary interfaces. For self-hosted platforms, e.g. using OpenFaaS (Section 2.5.2) or OpenWhisk, it is even harder to find suitable event sources since there is no definite set of services that are supported as function triggers. In the following subsection, the most relevant event source types are classified. One generic approach to classify event source types is introduced in [CNC18]. Our concept comprises a more detailed categorization of services across commercial and open source providers in Table 4.1.

Note that there are many more supported event sources for each of the vendors' serverless platforms which represent specific services from the cloud platform of the corresponding provider. For instance, IBM has a big focus on artificial intelligence including natural language processing and IBM Watson-based services[3]. Furthermore, Microsoft supports the complete *MS Graph* suite [Mic19g] for invoking functions by interacting with MS services. Likewise, AWS supports event sources from platform-specific services, such as *AWSConfig* [Ama19j] or *Amazon Alexa* [Ama19a]. Certainly, the migration of a FaaS-based application that relies on one or more of those platform-specific services will require a complete reengineering of the existing system or is even impossible without building a multi-cloud solution. Observe that we claim to be as accurate as possible with Table 4.1, however, there are still ambiguities between each providers services which impede a direct assignment of event triggers for a category and therefore hinders us to guarantee that the services are mappable between the platforms. In particular, the assignment of a suitable service turns out to be more difficult when considering the services' underlying technologies. For example, IBM *Event Streams* is a fully-managed Kafka as a Service offering while Amazon *Kinesis* [Ama19f] relies on a different technology which is based on another programming model and event format. Currently, AWS also offers a fully-managed service for Kafka, however it is not supported yet as a direct event source for Lambda functions [Ama19g]. Note that almost any service offering of the providers ecosystems can transiently be used as event source by integrating them through each platforms supported event streaming or queueing services for FaaS-based applications. In the following section, explicitly supported event sources of commercial platforms are compared for the most relevant categories. Additionally, an exemplary set of event sources is provided which can be considered suitable for open source alternatives, such as, OpenFaaS.

[3] https://cloud.ibm.com/docs/openwhisk?topic=cloud-functions-pkg_watson_assistant

Category	AWS Lambda	IBM Cloud Functions	Azure Functions	OpenFaaS
Endpoint	API Gateway	API Gateway Websocket	HTTP Webhook	HTTP Webhook
Schedule	Schedule	Alarm	Timer	Cron
Object Storage	S3	Object Storage	Blob	Minio
NoSQL	DynamoDB	Cloudant	CosmosDB	Cassandra MongoDB
Publish-and-Subscribe	SNS	Event Streams	Event Grid	Redis
Event Streaming	Kinesis	Event Streams	Event Hubs	Kafka
Point-to-Point	SQS	Event Streams	Queue Storage	RabbitMQ

Table 4.1: Comparison of supported services from AWS Lambda, IBM Cloud Functions, Azure Functions and OpenFaaS for commonly used event source categories.

Endpoint Services

Every serverless platform provider offers fully-managed endpoint services for serverless functions. Endpoint services most often are implemented as static HTTP APIs that are accessed via a *uniform resource identifier* (URI) to which HTTP requests are sent and responses are returned from. Typically, endpoint services fire synchronous requests which result in a blocking call. That is, the client issues a request to an endpoint and waits for an immediate response [CNC18]. AWS Lambda supports Amazon's fully-managed API Gateway [Ama19b] as event source for function invocations. Azure Functions can be triggered by the built-in HTTP triggers which can also be customized to respond to webhooks for both function invocation and output bindings [Mic19e]. IBM Cloud Functions can be triggered via OpenWhisk's built-in API Gateway and *WebSockets* [Apa19f]. Open source platforms, such as OpenFaaS, typically also provide embedded API endpoints for deployed functions. We propose a set of properties for the description of HTTP endpoints that comprises:

- *Method:* Configurable request methods for particular endpoints
- *Path:* Configurable paths for requests to trigger particular functions
- *Auth:* Optional configuration of authentication policies

Each of the described endpoint service offerings fulfill these requirements with more or less compatible naming conventions for the configuration properties which makes the usage of endpoints easy to migrate between different platforms. Solely, OpenWhisk's package APIs have a proprietary way to define the path of the requests for a given function (Listing 4.1). AWS SAM provides a Serverless API [Ama19m] which is a simplified version of the holistic API Gateway offering. These properties represent a baseline for the analyzed providers, however, to make the concept robust for adding further providers and platforms, the set of properties must be extensible by additional properties, if necessary.

Listing 4.1 OpenWhisk package API declaration for a GET request to a request path `/function/path`.

```
packages:
  function_package:
    ...
    apis:
      function-path:
        function:
          path:
            my-function:
              method: GET
              response: http
```

Object Storage

The majority of serverless platforms support the invocation of functions for events in object storage services. The function is bound to one or more storage instances and gets triggered asynchronously for events within that storage instance, such as adding a document or deletion of some data. Probably the most prominent example use-case is to create a thumbnail for each newly added image of a particular bucket with a serverless function and store it inside another bucket with a certain naming convention [BCC17]. Amazon's *Simple Storage Service* (S3) [Ama19h] is known as most commonly used cloud object storage. In Table 4.1, comparable storage offerings for IBM Cloud and Azure Functions are presented. The following characteristics can be considered a suitable example set of key properties for object storage offerings:

- *Bucket ID:* Resource identifier of the object storage instance
- *Event types:* Possible configuration on triggering functions when an object is *added*, *modified*, *deleted*, *restored*, etc.
- *API:* Supported interfaces

Cloud Object Storage can be considered as IBM's counterpart for S3, especially, this service aims to make the migration from AWS Lambda to IBM Cloud Functions much easier since the offering provides an S3 compatible API [IBM19a]. In Azure Functions, the model for resource bindings is fundamentally different from AWS Lambda and IBM Cloud Functions in means of API usage to interact with service resources. Interaction with bindings between functions and services are handled through the context object which will be passed to the function or from the function to the resource. Resulting from this, the difference between the object storage offerings is only in the naming convention of buckets and Azure Storage's *containers*. However, for scenarios in which a function passes a context object to a container, the executed logic will be declaratively included in the `function.json` file. When considering open source platforms, there are several third-party service offerings that are compatible with the commercial offerings on the market. OpenFaaS provides an event-connector pattern which maps the functions API gateways to topics and get triggered by integrated third-party services. *Minio* [Min19] is an S3 compatible open source object storage service which can be integrated via webhooks or by publishing events to Apache Kafka and represents a suitable alternative for migration-related service mappings.

Non-relational Databases (NoSQL)

With the rise of cloud computing and distributed systems, the demand for scalable and distributed non-relational databases emerged. Certainly, these NoSQL databases represent a significant component of many cloud applications, thus, are relevant for a pletora of serverless use-cases. The most prominent provider-specific NoSQL offerings include *DynamoDB* [Ama19e], *Cloudant* [IBM19f] and *CosmosDB* [Mic19a]. Compared with object storage offerings (Section 4.4.1), database services are much more sophisticated with respect to implementation and interaction through provided interfaces. IBM Cloudant is a hosted cloud document store which is based on *Apache CouchDB* [Apa19c]. Amazon DynamoDB is a hosted, scalable DBaaS offering which provides the functionality of a document store and key-value storage. CosmosDB provides functionality of document stores, key-value stores, wide column stores and can also be used as an Graph Database Management System.

While all of them provide a REST API, CosmosDB also provides APIs for a range of commonly-used database offerings, e.g., *DocumentDB* [Ama19d], *MongoDB* [Mon19b], *Cassandra* [Apa19b], *etcd* [Clo19b] and *Spark* [Apa19e]. Clearly, migrating from Azure Functions with usage of CosmosDB with one of these technology-specific APIs is potentially harder than porting between two of the other providers with simple usage of REST APIs. Additionally, DynamoDB and Cloudant do not support structured query language (SQL) while CosmosDB provides an SQL-like query language. There are also differences in pre-defined typing of the database services, namely, DynamoDB and CosmosDB support pre-defined types whereas Cloudant has no typing in its default state. Observe that Amazon's service suite includes services which are compatible with DocumentDB or MongoDB [Ama19d], however, does not support it as a trigger for Lambda functions. Furthermore, an option exists to generate a self-hosted CouchDB using AMIs which can also be integrated into AWS Lambda applications, however, this is not a subject for our comparison since having self-managed components violates the overall idea of serverless applications. According to the feature set of compatible APIs in CosmosDB and Cloudant, suitable open source alternatives can be on-premise database implementations, e.g., DocumentDB, Cassandra, MongoDB, or CouchDB. This comparison leads to the following example set of key properties for non-relational database offerings which can be considered suitable for the analyzed providers:

- *Database ID:* Resource identifier for the database instance/partition
- *Type:* Supported database models
- *API:* Supported interfaces
- *Query language:* Supported types of query languages

Note that this set of recommended properties provides a baseline for future iterations and therefore needs to be extensible for additional properties, if necessary. Furthermore, the proposed properties can be divided in actual feature-based descriptions of service offerings and more general concerns, e.g., having a *database ID*, which is most certainly available at any provider's service offering.

Scheduled Events

One common use-case of serverless applications is the execution of short-lived tasks in a periodic manner. Timed events, such as *cron jobs*[4], are asynchronous requests which invoke functions on a fixed schedule. For instance, a function for backup creation gets triggered in a recurrent manner, e.g., every third day at 12PM. Typically, cron expressions are normalized[5], however the provided scheduling event sources of the analyzed providers have differences in their used formats (Table 4.2). Conceptually, the differences are relatively small so that an overall schema can be applied to schedule cron events for each provider which is more or less in the same format with little to no refactoring. The following listing presents an exemplary set of key properties which can be used to compare scheduling-based service offerings:

- *Cron:* Supports cron jobs
- *Cron type:* Supported cron format, e.g., crontab(5), cron for .NET, etc.
- *Interval:* Supports recurrent jobs based on time intervals
- *Once:* Supports once-in-a-lifetime triggers

The analyzed services differ in proprietary solutions for *Interval* schedules. AWS Lambda allows developers to decide which unit (minutes, hours, days, etc.) should be taken for a recurring event timing, whereas IBM's event scheduler binds the interval scheduling on *minute* values. Azure *Timer* has a proprietary format in which developers configure an interval in `hh:mm:ss` if the first two digits are smaller than 24 and `dd:hh:mm` otherwise. Certainly, this feature has been implemented as a convenience for developers, however, it can also be achieved using traditional cron expressions and consequently increase the portability the application. Solely, IBM Cloud provides a scheduled event for events to fire *once-in-a-lifetime* which can be scheduled in *Date Time String Format*[6]. The majority of open source FaaS platforms support cron-based schedules [7] which can be attributed to the fact that most of these platforms are native deployments on top of Kubernetes. Likewise, OpenFaaS provides a cron event connector[8] which is based on Kubernetes' cron jobs. Note that, conceptually, the set of key properties is also extensible for scheduled events, however, there will most certainly be no huge gaps between the providers' offerings since schedules are typically implemented by using the standardized cron formats.

Event Streaming and Messaging Events

Messaging is an essential part for the majority of cloud-native applications which, typically, gives developers a standardized component for handling informational streams. The field of messaging includes multiple paradigms, each comparable to one another but serving for different purposes. Fundamentally, there are two different types of channels: *point-to-point* (message queues) and *publish-subscribe* (pub-sub) [HW03]. While most of the technologies that implement these paradigms are based on the basic principles of messaging, they differ in architectural concerns, like

[4] http://man.openbsd.org/cron.8
[5] http://crontab.org/
[6] http://www.ecma-international.org/ecma-262/5.1/sec-15.9.1.15
[7] https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/
[8] https://github.com/zeerorg/cron-connector

Schedule Types		AWS	Azure	IBM
CRON	Format	Min Hr DoM M DoW Y	S Min Hr D M DoW	Min Hr DoM M DoW
	All	* * * * ? *	* * * * * *	* * * * *
	Range	X-Y	X-Y	X-Y
	Set	X, Y, Z	X, Y, Z	X, Y, Z
	Interval	0/X	*/X	*/X
INTERVAL		<Value><Unit> (e.g. 1 hour, 2 minutes, ...)	hh:mm:ss dd:hh:mm	minutes <Value> (e.g. minutes 3)
ONCE		N/A	N/A	ISO 8601 format

Table 4.2: Comparison of scheduled events for AWS, Azure and IBM with respect to the formatting in seconds (S), minutes (Min), hours (Hr), days (D), day-of-month (DoM), months (M) and day-of-week (DoW).

consisting of topics and queues or relying on a sequence of topics and subscriptions. Table 4.1 demonstrates that providers' service offerings are sometimes ambiguous and it might be cumbersome to assign them to one of the described paradigms. For instance, Amazon's *Simple Notification Service* (SNS) enables tenants to implement channels in the style of pub-sub or point-to-point, and can also be used in combination with *Simple Queueing Service* (SQS) to form an alternative for *Kinesis*. Likewise, Azure Functions supports different kinds of services as event sources for the defined messaging categories which may also be used interchangeably, depending on the context and the requirements an application. IBM Cloud, on the other hand, supports only one general service offering for messaging concerns, namely, *Event Streams* which provides a fully-managed Kafka service [IBM19g]. For open source alternatives such as OpenFaaS, there are numerous third-party offerings for event streaming and messaging. Apache Kafka can be considered as the most widely-used alternative for general event streaming tasks. Furthermore, OpenFaaS supports integration of *Redis* [Red19] for pub-sub notifications, *RabbitMQ* for point-to-point messaging [Piv19a], and several hosted services, such as, SNS and Azure Event Grid. An exemplary set of properties for messaging and event streaming services could look like follows: The following set of properties can be considered as an extensible first iteration of characteristics for which messaging- and event-based service offerings may be compared:

- *Queue / Topic:* Location or resource name of topic/queue
- *Batch size:* Configures how many events are delivered in a bundle
- *Filter:* Policy for filtering in order to get only the subset of notifications/events which are relevant for the application context

4.4.2 Invoked Services

Apart from event sources, FaaS-based applications typically contain service resources which are accessed, or *invoked*, by functions. For instance, in ThumbNailer, the function gets triggered by an object storage event of a S3 bucket instance, gets the corresponding files, modifies them, and puts the resulting files into a second bucket instance. The latter S3 bucket can be considered as an *invoked service* within a FaaS-hosted application. Unlike event sources, virtually any available cloud service can potentially be an invoked service. This can be achieved through hardcoded API calls inside the

function code whereas the discovery of such service bindings would require cumbersome analysis on source code level. Most FaaS platforms provide ways to make such interactions between hosted functions and services transparent in the applications' deployment model.

Accordingly, Lambda and Azure Functions follow a transparent approach for modeling the communication of functions with invoked services. While Azure Functions define these services as output bindings in each functions configuration file for bindings (`function.json`), in AWS Lambda a set of *execution roles* can be defined for granting functions permission to access other cloud service resources from the providers' ecosystem. When integrating invoked services in FaaS-based applications, the differences of the several platforms become more explicit, e.g., while in Azure Functions, service bindings follow the concept of pipelining event data in the defined direction, Lambda functions still contain API calls inside the function code for accessing the services. In comparison, IBM Cloud Functions rely on the usage of pre-installed or manually-installable packages for enabling hosted functions to communicate with the providers' cloud services. These packages are basically function templates with implemented API calls to the service interface which ensures to keep the self-written function code cleaner and safer with respect to necessary credentials and access management. The invocation of services can implicitly traced down through the set of defined package dependencies in the applications package, e.g., `/whisk.system/cloudant/write`.

In conclusion, the analysis of invoked services in serverless applications with FaaS-hosted functions turns out to be much harder than event source discovery. Consequently, it is typically necessary to include an according amount of manual verification when porting FaaS-based applications which incorporate a set of provider-specific services that includes both event sources and invoked services.

4.5 Platform Evaluation

The evaluation phase focuses on the analysis of the source application's GAM with respect to incorporated services. There are two common scenarios which can be considered for application migration:

S1 - The application is intended to be ported to one particular desired provider or platform.

S2 - The application is intended to be ported from the current platform to another provider or platform without concrete preferences.

Essentially, the procedure is identical for both scenarios. The GAM is compared with one or more other platforms to which it potentially can be ported. In the remainder of this work, we will denote these target platforms as *platform candidates*. Depending on the scenario, the set of platform candidates comprises one or more platforms. One crucial evaluation task is to compare the services in the GAM with the supported service offerings of each platform candidate. Based on this comparison, developers can be provided transparency over the compatibility of the platforms and the potential effort for the migration. Figure 4.5 shows the main steps of the evaluation phase.

First, the incorporated services in the GAM are compared with the supported service offerings at the platform candidates. The overlap of these service offerings results in a coverage score for the corresponding platform candidate. The details are described in Section 4.5.1. Subsequently, this coverage score is settled with a source code analysis of the functions in the source application's

deployment package which is assessed by exploring the functions for provider-specific logic. Finally, the coverage scores form the foundation classifying the platform candidates according to the *Serverless Platform Suitability* hierarchy (Section 4.5.2).

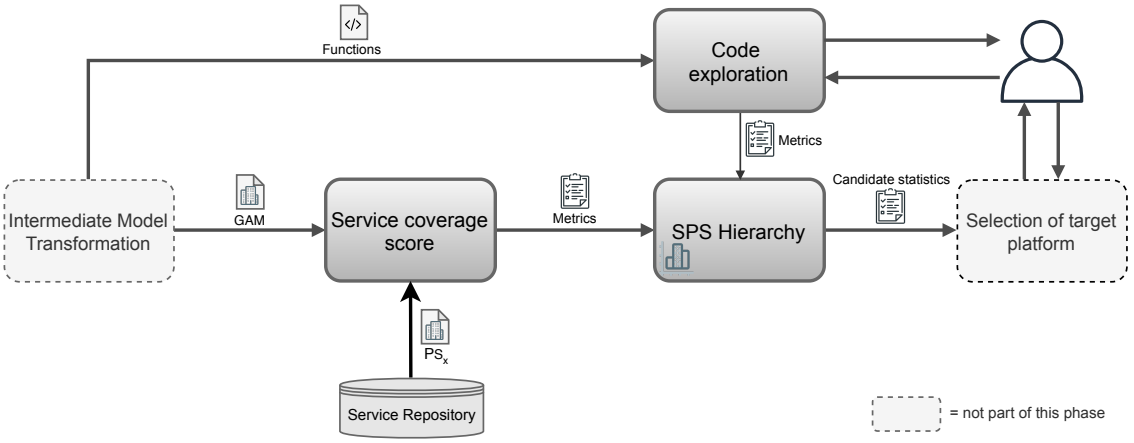


Figure 4.5: Outline of the evaluation process for the suitability of platform candidates.

4.5.1 Platform Suitability Evaluation

The GAM is an instance of the GSMM which describes the architecture of the source application. When porting an application to another provider's platform, the incorporated services from the source platform s need to be compared with the supported service offerings at the target platform x. This comparison is performed over the generic resource identifiers of the incorporated services. In Section 4.2, the mapping of provider-specific services to generic resources is described to be achieved under usage of a suitable *service repository* which maintains knowledge about each providers service offerings and maps them to suitable generic service categories. Section 4.4 describes the architecture of such a repository in a conceptual manner. The generation of the GAM can be attributed to having such a service repository which maintains a super set of several provider's service offerings. Consequently, the service repository can be used to generate a set of generic service categories PS_x for the supported service offerings of each platform candidate x. It is important to make the service repository extensible in order to add further supported providers to the support framework. Note that, while PS_x comprises the complete set of supported service categories for platform x, the set of distinct service categories in the GAM is limited to the services that are actually incorporated in the source application. That is, GAM comprises a subset of the set of supported services at the source platform. The following subsection introduces a model comparison approach for the incorporated services in the GAM with one or more potential platform candidates regarding the coverage of the supported service offerings. Note that the approach is described for the second scenario, however, works analogously for scenarios where a concrete provider is chosen as a desired target platform.

Coverage Score

There are numerous methods for computing set (vector) similarity. We consider the *Jaccard coefficient* [Jac01], as the measure that comes closest to our approach. The main difference between our method and the traditional Jaccard coefficient is that the latter defines set similarity by dividing the size of the intersection of two sets A and B by the size of their union:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{4.1}$$

The set of incorporated services in the GAM gets compared with each platform candidates' set of supported services PS_x by assessment of the suitability through a *coverage score* C_x ($x \in X, X$: platform candidates). Note that the comparison is restricted to the service categories which are actually incorporated in the GAM. While a coefficient is a constant factor by design, our coverage score measurement depends strongly on the set size of the GAM's service categories. The reason behind this decision is that services that are supported by a platform candidate but are not used in the source application can be omitted since they are not of relevance when porting the application from the source to the target platform. This would result in a modified, directional Jaccard measure:

$$J_d(A,B) = \frac{|A \cap B|}{|A|} \tag{4.2}$$

$$A, B = \textit{set of distinct service categories in GAM}, PS_t, \textit{ respectively.} \tag{4.3}$$

The modified Jaccard measure serves as inspiration for a more suitable measurement which is described in the remainder of this section. For simplicity, all service categories are weighted with a similar relevance factor $w_\sigma = \frac{1}{|\Sigma|}$ (Σ : *set of distinct service categories in GAM*). Each platforms coverage score is assessed as follows:

$$C_x = \sum_{\sigma \in \Sigma} x_\sigma \cdot w_\sigma \tag{4.4}$$

$$x_\sigma = \begin{cases} 1 & \textit{service category } \sigma \textit{ is available in } PS_t \\ 0 & \textit{else} \end{cases} \tag{4.5}$$

The resulting coverage scores C_x can be considered as an initial coarse-grained representation of the platform candidates suitability. This approach, however, represents a superficial analysis of the actual effort that have to be evaluated when porting an application to the target platform. Conceptually, this approach works with any coverage score measures which can be adapted in a plugin-based manner. In the following, a more refined measurement is described for demonstration concerns of refined assessment approaches.

Section 4.4 describes various service offerings and shows that service that correspond to the same service category may still rely on completely different technologies or provide proprietary interfaces. Factors like these can make the migration process almost as cumbersome as the absence of a comparable service offering at a platform candidate. In Chapter 3, MailNotifier is introduced as an exemplary FaaS-based application use-case. MailNotifier incorporates Lambda's pub-sub service offering, SNS, to publish and send an e-mail to a list of subscribers. Porting this application to Azure's FaaS platform, the counterpart of SNS, as described in Section 4.4, is Azure Event Grid. Event Grid, however, does neither support the e-mail protocol, nor can be used as an invoked service in applications hosted on Azure Functions. In Thumbnailer, the function gets triggered

on each newly uploaded file in the image bucket. While it is not in particular impossible to port the application with this functionality to another provider, e.g., Azure, there is a difference in the conceptual model of object storage triggers. Azure blob storages trigger functions on every change event, e.g., including deletions, which would lead to errors when trying to get the respective image file on each function invokation. Consequently, it is necessary to provide a more fine-grained evaluation of the incorporated services and the corresponding counterparts at the target platform.

Similar to C_x, the refined score C^* assesses the coverage of incorporated service categories. In contrast to the plain checking for the availability of corresponding service offerings at the platform candidate, the similarity of service offerings can be compared with respect to each services' properties, as described in Section 4.4. Each service category is still weighted similar with relevance factor w_σ, however, we refine x_σ to a non-binary value range ($0 \leq x_\sigma \leq 1$) which depends on the number of relevant properties that will be compared when evaluating the similarity of two service offerings which belong to the same service category:

$$x^*_\sigma = \sum_{p \in P_\sigma} x_p \cdot w_p \tag{4.6}$$

$$w_p = \frac{1}{|P_\sigma|} \tag{4.7}$$

$$P_\sigma : \textit{set of properties for service category } \sigma \tag{4.8}$$

$$x_p = \begin{cases} 1 & \textit{property p is similar in both platforms service offerings of service category } \sigma \\ 0 & \textit{else} \end{cases} \tag{4.9}$$

It is important to mention, that the comparison of two services' general set of properties can distort the actual migration effort. This can be attributed to the fact that the source application may only use a subset of the possible properties from that service offering which could be fully covered by the target platforms service. For instance, MailNotifier uses the e-mail protocol of SNS which makes the incorporated service usage fully incompatible with Azure's Event Grid. Assuming that the source application could use only the basic pub-sub functionality to enable event streaming via SNS, Event Grid would have a fully coverage of the incorporated functionality of the application's pub-sub service offering. Consequently, P_σ is limited to the service properties that the incorporated service implementations in the GAM actually depend on:

$$P_\sigma : \textit{set of properties for service category } \sigma \textit{ which are actually used in GAM} \tag{4.10}$$

This leads to our refined coverage score:

$$C^*_x = \sum_{\sigma \in \Sigma} x^*_\sigma \cdot w_\sigma \tag{4.11}$$

As we cannot predict how important certain incorporated services are for the given context, the coverage score is introduced using default weighting. Sophisticated variants of coverage evaluation for platform candidates may include a dynamic weighting of service filter components in the GAM which may be realized through user interaction or a community-based experience repository for migration patterns. The latter could amplify the user-based input of weighting factor adjustment by

providing recommendation for the weight factors based on experiences of other developers who faced similar migration scenarios, e.g., migrating a simple object storage trigger from AWS *S3* to Azure *blob*, or moving an application from AWS to IBM Cloud Functions which is using SQS as an event source.

Certainly, analyzing the service offerings for certain key properties of a corresponding service category is a non-trivial task and therefore requires manual verification. Typically, such information is collected to form a knowledge base which in this context can contain service categories with mappings to provider-specific services and contains key properties that can be used as a standardized measure for feature similarity of two distinct service offerings. Note that properties can stand for both provided features, e.g., supported protocols for pub-sub services, and more general properties, e.g., availability in specific regions or SLAs.

4.5.2 Serverless Platform Suitability Hierarchy

Assuming a platform candidate has full coverage of the incorporated services and provides all necessary features within these service offerings, an effortless migration still cannot be guaranteed. It is not uncommon that service offerings on different platforms support the same functionality, yet, require further implementation artifacts of provider-specific logic inside the functions. Refactoring provider-specific pieces of code can quickly become a time consuming and verbose task. In the following, the *Serverless Platform Suitability* (SPS) hierarchy is introduced. The SPS hierarchy classifies platform candidates regarding the required effort for the migration task based on coverage scores and measures of further source code analysis of the functions inside the source application's deployment package (Figure 4.6). The hierarchy comprises four levels:

Level 0 - Partially portable The source application incorporates at least one service category for which the platform candidate provides a fully incompatible service or no offering at all:

- *An application is only 'partially portable' to a platform candidate x, if* $x^*_\sigma = 0$ *for at least one incorporated service category in the GAM*

Platform candidates for which the source application is partially portable, the implementation of a similar application will require an noteworthy reengineering effort. Furthermore, there must be at least one service category that is covered by the platform candidate. Otherwise, the candidate is *incompatible* for the migration task.

Level 1 - Migration ready All incorporated services in the GAM have similar counterparts at the platform candidate. Furthermore, the overall similarity of the services including their properties must be greater or equal than a specified lower bound ω.

- *A candidate is 'migration ready' if* $x^*_\sigma > 0$ *for all incorporated services in the GAM and* $C^*_x \geq \omega$.

For platforms which are migration ready, the user can expect the application to be portable, however, it might require not only refactoring in the source code but also to find workarounds for particular provider-specific logic.

Level 2 - Migration aligned For every provider-specific piece of code in the functions of the source application, there is an alternative code pattern in at the platform candidate (Section 4.5.3). For candidates which are migration aligned, the user can expect that the source application is portable with respect to incorporated services and without finding workarounds for provider-specific logic in the functions.

Level 3 - Migration optimized The platform candidate provides a holistic coverage of incorporated services and code patterns in the source application. Another possible scenario is that the source application contains no unique provider-specific dependencies. That is, the application is deployed in a provider-agnostic manner, thus does not contain any provider-specific API calls in its functions and is not triggered by non-standardized event sources within the providers ecosystem. For platforms which are migration optimized, the user can expect a seamless migration with little to no complications.

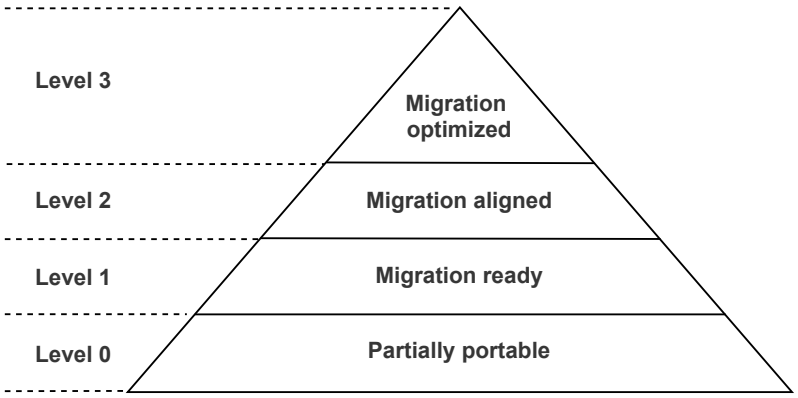


Figure 4.6: The SPS hierarchy constitutes of four levels of platform candidate suitability classes: (1) *Partially portable*, (2) *Migration ready*, (3) *Migration aligned*, and (4) *Migration optimized*.

The SPS hierarchy is *constructive* in its levels 1-3, that is, a platform candidate which is *migration optimized* must also be *migration aligned* and *migration ready*. The first two levels of the hierarchy can easily be checked using the coverage scores, as introduced in Section 4.5.1. We assume that the source application has at least one service incorporated, namely an endpoint service, which can be considered to be the minimal setup of any FaaS-based application. Otherwise, the introduced coverage score measurement would need to be refined to initially check for an empty set of services since this would break the coverage assessment when setting the relevance factor w_σ by throwing an arithmetic exception (Equation (4.7)).

4.5.3 Code Analysis

As described in Section 4.1.1, we assume that the retrieved deployment package contains sources of the serverless functions that execute the business logic of the application. Although it is a critical factor for serverless migration, simply mapping the architectural components is most often not enough to ensure a seamless migration for developers. The incorporation of certain service instances may also require to add provider-specific lines of code in the written functions which will most probably differ between the platforms, e.g., for object storage services. For instance, accessing the actual data based on a event in S3 buckets needs to be implemented explicitly in the function

whereas the data pipelining approach in Azure Functions passes the data body directly to the function. Consequently, tracing provider-specific logic with meaningful semantic inside the functions is a significant aspect of supporting engineers in the decision making process while migration to another platform. Further, depending on the platform, functions may also include provider-specific logic which is not coupled to the usage of external services, such as non-standardized logging or exception handling interfaces that are part of the underlying programming model (Section 2.5). This section introduces a concept of interactive code exploration for retrieved source files, as illustrated in Figure 4.7.

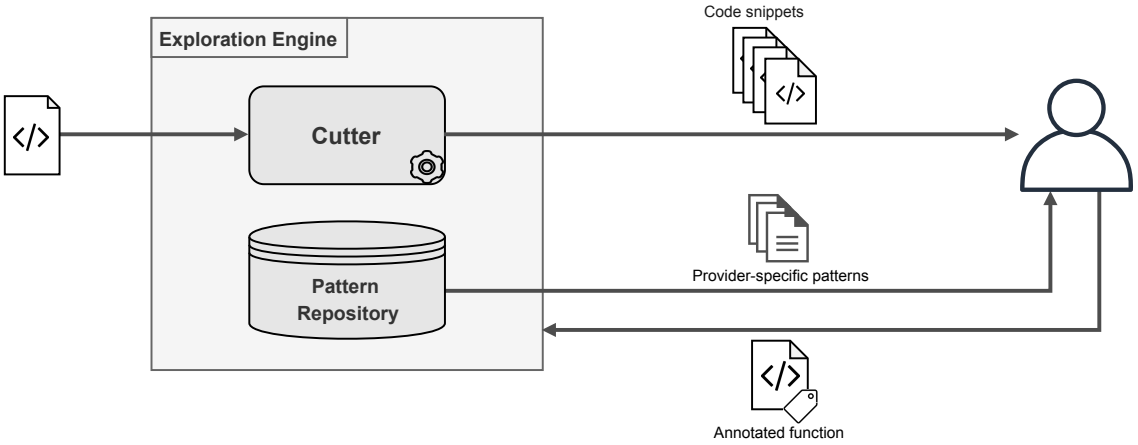


Figure 4.7: A diagram of the interactive code exploration workflow.

The exploration approach comprises several components which engage user interaction for the code analysis. First, the function's source code is divided into semantic blocks of methods, API calls, etc. Typically, such tasks are performed using a domain-specific code parser or lexer, which splits the source code of a function into *snippets*. The existence of provider-specific code snippets can be checked implicitly by analyzing the source files for included dependencies or libraries of provider SDKs and related interfaces. In cases when no provider-specific libraries or SDK imports can be found in the function, the code exploration terminates. Otherwise, the user annotates the snippets according to definitions of suitable *code patterns*.

Pattern Repository Many FaaS-hosted functions handle similar kinds of business processes which in many cases leads to boilerplate code for similar semantics, e.g., interaction with certain incorporated platform services. Section 2.4 described good practices for writing serverless functions, such as, implementation of single-purpose services for simple business processes. Thus, functions ideally have only few lines of code which comprise business logic and provider-specific code for implementing required interfaces. The latter most often represents recurring boilerplate code so the functions potentially differ only in the code portions with business-specific logic. The idea is to maintain a community-driven *pattern repository* which holds a collection of provider-specific code patterns and is maintained by an active cloud developer community, similar to approaches like CloudComparer or CloudEvents [Clo19c; FIl19]. Examples for code patterns might include recurring tasks, such as, GET requests to retrieve files from object storages: This knowledge base can be utilized for identifying patterns within the code snippets and annotate them accordingly. Having a static code analysis in form of a parser or lexer, developers can be supported by having a

Listing 4.2 Exemplary code pattern for AWS-specific usage of the `ObjectStorage_GET` pattern.

```
// @Pattern("AWS_ObjectStorage_GET")
var params = {
    Bucket: "",
    Key: ""
   };
   s3.getObject(params, function(err, data) {
     if (err) console.log(err, err.stack);
     else     console.log(data);
   });
```

recommendation system for potential pattern matches for the currently inspected code snippet. For instance, a code snippet can be recommended to be tagged as an object storage get request, if there are method calls containing strings like `get`, or initializing objects containing properties like `bucket`, or `key`. Such a pattern repository may be extensible in order to support updates in case of new discovered patterns. Conceptually, the repository holds a collection of code patterns for each platform. Analogous to the concept of a service repository, each of these code pattern entities can be assigned to exactly one *generic pattern type* which represents a meta-model for the code pattern instances. For instance, the `AWS_ObjectStorage_GET` pattern can be mapped over its generic pattern type `ObjectStorage_GET` to another candidate platform's pattern instance, e.g., `IBM_ObjectStorage_GET`.

There are several reasons to design this approach in an interactive manner by involving human reasoning instead of an automatic code exploration algorithm. Although not good practice, the source files could contain hardcoded access to particular provider services [IBM19b]. In this case, an automatic code exploring algorithm would annotate this code portion as provider-specific logic, however, hardcoded access, e.g. REST API calls, will most likely still work when the application is deployed on another platform. For scenarios like this, the interactive exploration approach proves to make this phase very robust by adding human reasoning to the annotation task and therefore gains performance regarding the solutions quality. On the other hand, for particular big projects for which manual verification for all snippets would be practically infeasible, integrating random forest classification or active learning approaches [CAL94; TK01; TL11] may lead to significant improvements.

Code Pattern Coverage For each pattern in the annotated function, the existence of a corresponding pattern for the platform candidate can be checked using their common generic pattern type. This will be elaborated further to assess the final suitability score for the platform candidates. Similar to the service coverage scores, as introduced in Section 4.5.1, the coverage of corresponding provider-specific code patterns can be measured for each platform candidate:

$$T_x = \sum_{\gamma \in \Gamma} x_\gamma \cdot w_\gamma \tag{4.12}$$

$$w_\gamma = \frac{1}{|\Gamma|} \tag{4.13}$$

$$\Gamma : \textit{set of distinct generic pattern types in the collection of annotated code patterns} \tag{4.14}$$

$$x_\gamma = \begin{cases} 1 & \textit{defined code pattern of type } \gamma \textit{ exists in pattern repository for platform } x \\ 0 & \textit{else} \end{cases} \tag{4.15}$$

SPS classification In Section 4.5.2, platform candidates are classified as migration aligned iff every provider-specific code pattern in the source applications' deployment package has a counterpart at the targeted platform. Provided that each platform candidate has a service coverage score C_x^*, the final candidate score Ω is assessed as follows:

$$\Omega_x = C_x^* \cdot T_x \tag{4.16}$$

This leads to the conclusion that a candidate platform is *migration aligned* iff:

$$x \textit{ is migration ready} \bigwedge T_x = 1 \tag{4.17}$$

Furthermore, a candidate platform is *migration optimized* iff:

$$\Omega_x = 1.0 \tag{4.18}$$

A statistical overview is provided to the developer in which the suitability is illustrated for candidate platforms. Regardless of the scenario, as described in Section 4.5, the suitability is evaluated for a range of candidate platforms and displays the result to the user. In case of a desired target platform, the platform suitability evaluation of the remaining candidate platforms can be considered as advice of a more suitable platform for porting the application. Essentially, the framework provides decision support for moving applications between platforms by guiding engineers in the selection of suitable target platforms.

4.6 Migration Model Generation

This section describes the generation of a migration model from the results of the former phases. The gathered information from preceding phases is utilized to support engineers in the generation of a *migration model*. The migration model essentially represents a *deployment package* for the deployment of the source application architecture to the chosen target platform. The migration model generation comprises two sub-phases: (1) *Deployment Template Generation*, and (2) *Code Generation*. The resulting migration model provides a robust baseline for the deployment of an application with equivalent architecture and functionality on the target platform (Figure 4.8).

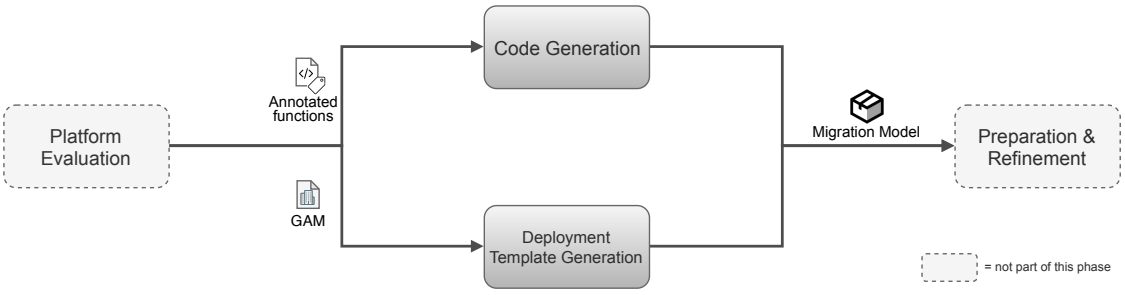


Figure 4.8: Outline of the sub-phases in the migration model generation.

4.6.1 Deployment Template Generation

Given a selected target platform t, it is desirable to utilize the existing intermediate model of the source application for generating an equivalent architecture that conforms the target platforms application model. The GAM can be translated into PSM_t of the target platform via meta model mapping with the GSMM. In addition, the generic services in the GAM are mapped to the respective provider-specific service offerings of the corresponding service category by using the service repository. PSM_t may be an explicit platform-specific application model, e.g., SAM, ARM, or wskdeploy package, for AWS, Azure, and IBM, respectively. However, the differences between the required configuration structures for deployment on the different platforms tend to vary strongly. Especially considering deployment approaches, e.g., for Azure Functions or open source platforms such as OpenFaaS, in which either the infrastructure modeling and provisioning of incorporated resources is distributed over multiple directories, or have to be performed completely manually. Consequently, a more feasible approach my be mapping the GAM into an intermediate format, e.g. EDMM [WBF19], or translating it into a production-ready deployment model of well-supported deployment frameworks, such as, Terraform or Serverless Framework.

Translating the GAM into an intermediate format with broad support of mapping to the provider's application models is beneficial in several ways. Decoupling the GAM from the provider's models is generally a good idea. Having an intermediate layer between GAM and PSM reduces the number of necessary transformation models significantly since it relies on the mapping of GAM to one common syntax which can handle the deployment on all supported platforms (Figure 4.9). On the other hand, the deployment template generation still depends on a third-party model, however, relying on one intermediate format makes the alignment of the GAM less complex, e.g., for syntactical changes on the third-party model. Note that this concept includes, but is not limited to the described intermediate formats represent possible mappings. Additional models can be added in a plugin-based manner to such a migration framework.

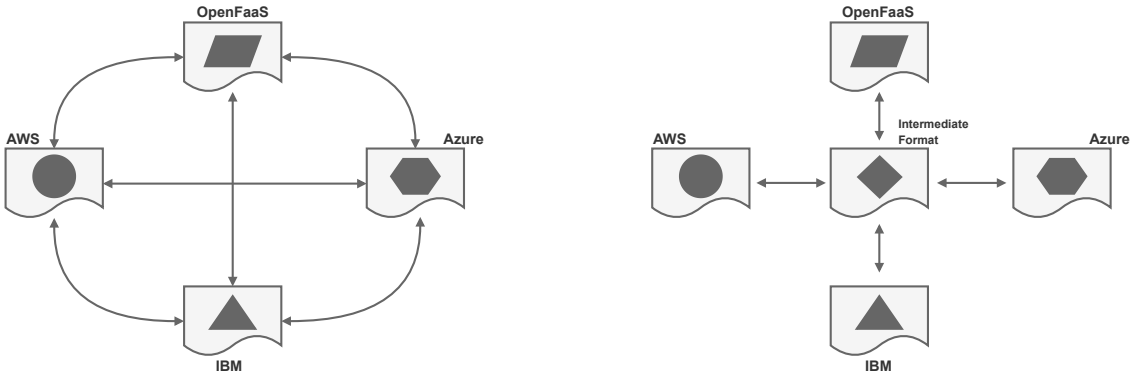


Figure 4.9: Number of transformation models when directly mapping to platform models (left) and with an intermediate format (right).

4.6.2 Code Generation

The annotated functions from Section 4.5.3 are utilized to support developers in generating adequate *function template* substitutions for the FaaS-hosted functions from the source deployment package. Initially, functions need to be generated which conform the target platforms programming model with respect to, e.g., function signatures or naming conventions (Section 2.5.1). Although it is conceptually possible to implement a system which performs code transformation in a completely automated fashion, such a system would require to integrate machine learning models, such as, neural networks for code generation [FK89; Goo10; YN17]. Typically, training neural networks is a verbose and time consuming task. In addition, there is a need for a relatively big training set of code samples on which the network can train, test, and validate. Depending on the context, these samples also may require to get labeled in prior. Consequently, aiming for a completely automated approach on code transformation can be considered infeasible in regard to the amount of required

Listing 4.3 Exemplary boilerplate skeletons of generic pattern type `ObjectStorage_GET`. The above boilerplate skeleton represents a get method of an object out of an S3 bucket whereas the boilerplate skeleton below represents an open source alternative with Minio as object storage service.

```
// @Pattern("AWS_ObjectStorage_GET")
var params = {
    Bucket: "",
    Key: ""
   };
   s3.getObject(params, function(err, data) {
     if (err) console.log(err, err.stack); // an error occurred
     else     console.log(data);           // successful response
   });
```

```
// @Pattern("OS_ObjectStorage_GET")
var size = 0
var Bucket = ""
var Key = ""
minioClient.getObject(Bucket, Key, function(err, dataStream) {
  if (err) {
    return console.log(err)
  }
  dataStream.on('data', function(chunk) {
    size += chunk.length
  })
  dataStream.on('end', function() {
    console.log('End. Total size = ' + size)
  })
  dataStream.on('error', function(err) {
    console.log(err)
  })
})
```

work for creating and maintaining such a system which is performing robust enough to omit manual labor. Instead, the pattern annotations of the functions can be utilized to generate code skeletons for the corresponding code patterns.

The pattern repository provides *boilerplate skeletons* for available generic pattern types at a target platform which will be placed at the position of the pattern annotation in the generated function template. Boilerplate skeletons are methods, native to the used programming language, which implement boilerplate code in the method body for the corresponding code pattern and are further refined by the engineer. For instance, the deployed application may contain a function which retrieves an object from an object storage resource (Listing 4.3). The boilerplate skeleton provides developers best practices for the SDK usage which further needs to be refined with business logic or credentials. Although it is not time-expensive in particular to look up small refactoring tasks in the providers documentation, this part may quickly get tedious when migrating applications with numerous functions involved. The concept of such pattern repository could rely on a community-driven knowledge base, e.g., with cloud developers contributing to it and aligning the repository with given changes in the providers SDK or similar. This is especially important to keep the developers' freedom to align the functions to the different event formats for different providers. For instance, for functions which are written in an untyped language, e.g., JavaScript, events are passed as objects, hence, the business logic is locked into this specific format which tends to differ between the analyzed providers (Section 2.5).

Provided that such a repository is based on community contributions, there can also be more sophisticated patterns which implement best-practices for recurring use-cases. For instance, the deployed application may implement a code pattern in the function which retrieves an object from one object storage, modifies it in some way and stores the result in another object storage instance. The way in which the retrieved data is modified in particular can be considered as business-specific logic so the implementation template for this code pattern will consist of a boilerplate skeleton for the download of the first entity, an empty method in which the user can add business logic inside the method's body, and another boilerplate skeleton for storing the resulting object in the target object storage (Listing A.2). For instance, ThumbNailer may be an exemplary application for the usage of this pattern. The code portion, which takes the image file object, resizes it using a third-party library, and returns the resized image to the target bucket.

4.7 Preparation and Refinement

After generation of the migration model, there is typically still manual work to do for preparing the deployment package. Developers must deal with refinements, such as, handling incompatible service offerings, access management, and refactoring the generated function templates. Depending on the context, the incorporated resources, e.g., object storage offerings, must be provisioned manually. This section introduces several refinement tasks, the user must deal with to ensure that the deployment package produces a functioning application on the target platform.

4.7.1 Refinement of the Deployment Template

In Section 4.5.1, the platform candidates are evaluated with respect to compatible service offerings which can be considered as counterparts for the incorporated services in the source application. However, the selection of a target platform is in the developers responsibility, which means, that the developer can still select a target platform which is *migration ready* or even *partially portable*.

Consequently, the target platform possibly has one or more incompatible services with the source application. In these cases, the deployment template generation, as described in Section 4.6.1, generates a deployment model which cannot establish a functioning application deployment as-is so the developer has to refine the deployment model by investigating the service offerings of the target provider or considering on-premise solutions for generating a workaround. In addition, this workaround may be contributed to the introduced knowledge base for service mappings to share the experiences in this porting approach and provide solutions for future migration attempts.

4.7.2 Refinement of Boilerplate Code

In Section 4.6.2, the generation of boilerplates is described in a conceptual manner. One significant task that needs to be dealt with is to go through the generated function templates and finalize them. Listing A.2 illustrates that boilerplates still need manual refinement in regards of adding the business logic and providing configuration information, e.g., credentials, bucket names, and properties to search for, such as, keys. Furthermore, it is important to keep the boilerplates as generic as possible since different providers rely on different programming models with respect to naming conventions, e.g., for passed parameters in the function signature, or event data formats on which the handling of passed objects may be locked-in and need to be refactored, accordingly. For instance, while in the function signature of Lambda functions, the event data is passed through an `event` object for which the object structure is dependent from the event source[9], in Azure Functions, there is much more freedom in regards to naming of passed parameters[10]. Such erroneous tasks typically require human reasoning and therefore are handled by the developer as a refinement step for the generated migration model.

4.7.3 Resource Provisioning

It is not uncommon that the deployment model in an extracted deployment package only describes the interaction of the implemented functions with certain types of resources, however, these resources most likely were provisioned manually through a browser-based user interface or via scripts which are not contained in the extracted deployment package. For instance, a Lambda-hosted application may be deployed using SAM templates for interaction with several S3 buckets (cf. Section 4.6.2), yet, the actual instantiating of these buckets were not part of the template. We define two scenarios: (1) *Manually deployed resources*, or (2) *usage of declarative resource deployment technology*. The latter

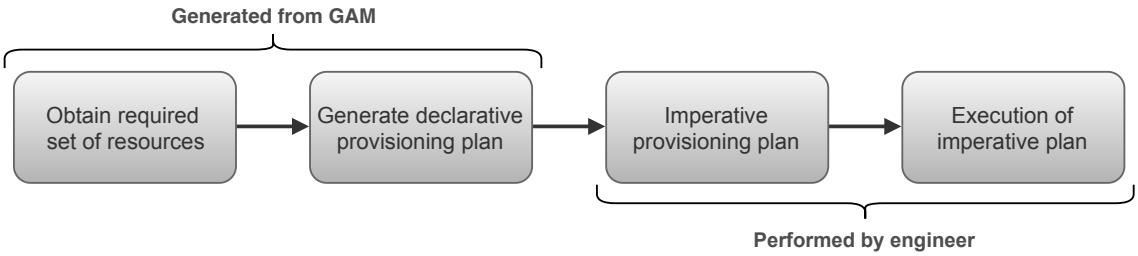


Figure 4.10: Interactive method for provisioning of incorporated resources.

[9] https://docs.aws.amazon.com/lambda/latest/dg/lambda-services.html
[10] https://docs.microsoft.com/azure/azure-functions/functions-reference-node#exporting-a-function

leads to the assumption that a technology-specific DSL is available for analysis which facilitates transformations using meta-models [WBF19]. Otherwise, the set of required resources need to be obtained and provisioned manually. Breitenbücher [Bre16] introduces the *PALMA method*; a hybrid approach for pattern-based management of running applications. For resource provisioning, developers can go through similar steps (Figure 4.10). First, the required resources are obtained from the existing application structure which is transformed into a declarative plan of provisioning tasks. The acquired knowledge about the incorporated services in the GAM can be used to obtain a collection of required resources and generate a declarative plan for resource provisioning. As the approach aims to be decoupled from available technologies, the developer is responsible for forming an imperative provisioning plan from the resulting output. This may encompass generating automation scripts, e.g., for provisioning tasks based on the providers API, or manually provisioning the required resources.

4.8 Automatic Deployment

After generating a migration model and refining it for the actual deployment, ideally the outcome is a production-ready deployment package which can be used to remotely deploy an application on the chosen target platform that is equivalent to the source application. Prior to deployment, the developer optionally can use the deployment model to set up sandbox environments to test the application for correctness and further requirements, e.g., through performance monitoring and end-to-end tests. Finally, the application can be deployed to the target platform and removed from the source platform. Alternatively, the source application can be set in an inactive state for safety reasons instead of being fully removed from the source provider's platform. In any case, it is advantageous to keep the original source deployment package for handling possible difficulties with the ported application by persisting it in some version control system or data storage. This way, the capability for performing a *rollback* [BBSR13] to the original application is given at any point in time by re-deploying the application using the source deployment package.

5 Implementation

This chapter discusses the details of the serverless migration support framework's conceptual system architecture and its prototypical implementation. The prototype allows us to validate the concept, introduced in Chapter 4, and demonstrates a variety of features which support developers in porting FaaS-based applications from one provider to another. Finally, a case study is conducted which demonstrates the prototypical implementation for a real world scenario. In the following, we present the framework using the C4 model-oriented approach [Bro18] for describing the software architecture, starting from a high-level perspective and iteratively introducing detailed views of the prototype's architectural components.

5.1 System Architecture

This section describes the conceptual architecture of a support system for porting serverless applications. Figure 5.1 illustrates a three-tier application architecture consisting of *presentation layer*, *business logic layer*, and *resource layer*. The presentation layer contains components to enable the communication with the business logic layer through API calls, which can be realized on multiple ways, such as, providing a general REST API or a web-based or rich client to enable interaction over a graphical user interface. The business logic layer contains a mainframe with a pluggable architecture which comprises three major components: (1) *Discovery*, (2) *Evaluation*, and (3) *Generation*. The system's architecture is designed in an extensible fashion to enable robustness for changing requirements, e.g., changes in the providers' application model, or novel migration use cases, such as, the demand for support of new FaaS providers. The resource layer exposes interfaces which establish the communication to the business logic layer. Further, it contains repositories for resource artifacts, in which objects for interaction and data exchange tasks are stored, such as, meta data for deployment packages, or application models. In addition to that, knowledge bases, such as, service repository (cf. Section 4.4), are located in the resource layer. The system's core functionality is located in the business logic layer, hence, the major components of the business layer and their constituents are described in more detail:

Discovery Manager This component of the system is responsible for discovery-related tasks, such as, adding existing FaaS-based applications' deployment models, or crawling the web for extracting the relevant information directly from the platform, on which the application is currently deployed (Section 4.1). In both cases, the retrieved deployment models must be persisted in the resource layer's artifacts repository to make the data available for future tasks, such as, evaluation and generation of generic application models.

Evaluation Manager The evaluation component of the framework can be divided in two main domains: (1) Application Model Evaluation, and (2) Code Analysis. The evaluation engine takes a generic application model and a target platform as an input and performs a suitability evaluation of the target platform for the application which is described by the passed model (Section 4.5.1). In

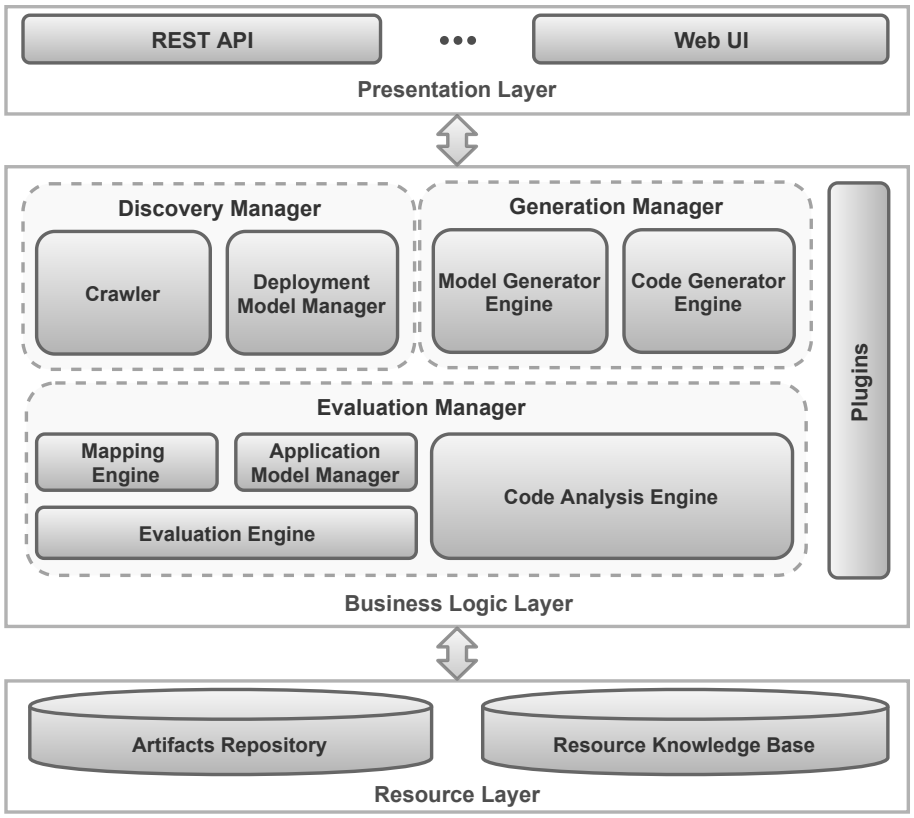


Figure 5.1: Conceptual architecture of a support system for porting serverless applications.

order to map a given deployment model to a generic application model, the respective mapping model must be specified. To keep the system extensible, we decided to design a dynamic approach for this specification by developing a *mapping engine* which consumes *mapping modules*. These mapping modules describe where to find the necessary information for the components of a generic application model, such as, event sources, function declarations, and invoked services of the FaaS-based application. Developers are able to extend the framework to support any given deployment template format, e.g., manifest files for Terraform, AWS SAM, or Serverless Framework, by adding a mapping module written in a simplistic, query-like syntax. The Code Analysis consists of a code lexer for analyzing the functions inside a deployment package for lines of code with usage of provider-specific libraries and annotates the functions, accordingly.

Generation Manager After retrieving all relevant resources and evaluation of an application's portability to a selected platform, an initial migration model is generated for the selected target platform (Section 4.6). This includes the generation of boilerplates and the translation of the application's architecture from the generic application model to the provider-specific model of the target platform.

Certainly, many of these tasks are coupled to the technology-stack or languages, which were used to develop the applications. This leads to our decision to contribute a conceptual system architecture, in which the executing components can be extended via plugins to ensure a robust conceptual framework, which can be maintained and adapted to unseen, challenging scenarios. One possible

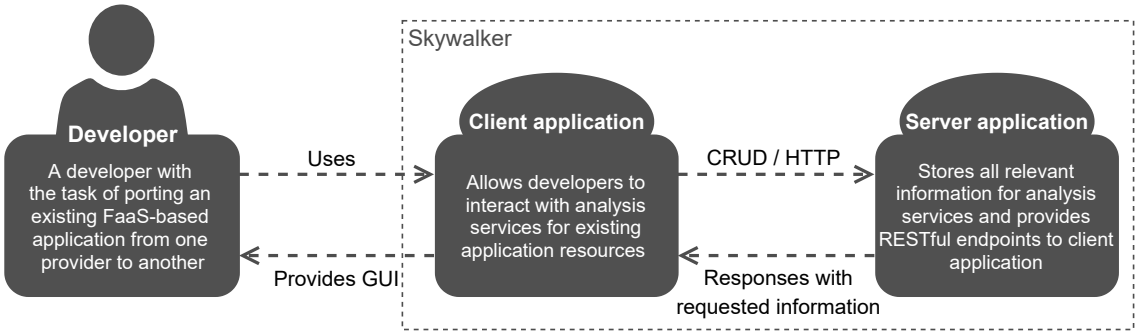


Figure 5.2: The prototypical framework's system context view for the coarse-grained interaction model of developers, web client and server application

implementation of such a system may be realized in form of a client-server application, as illustrated in Figure 5.2. The web client provides a graphical user interface for developers to interact with the system's services, e.g., to perform CRUD operations [Mar83] on files within the extracted application's deployment package and those stored in the system's repositories.

Mapping Engine The number of possible formats which can be used to model and deploy FaaS-based applications is not limited to the utilities offered by the providers themselves but grows with the number of third-party frameworks, e.g., Terraform or Serverless Framework, and intermediate modeling approaches, e.g., EDMM [WBF19]. Among other reasons, such as, potentially changing model formats for each of these deployment modeling approaches, and continuously adding frameworks and FaaS platforms, the necessary effort to create a knowledge base inside a migration support system which covers all these mapping scenarios is practically infeasible. The assumptions lead to the decision to design a dynamic approach for the mapping of provider-specific deployment models into a generic application model. The mapping engine is responsible to find the relevant data inside a given deployment model template and map it into a generic application model. Developers are able to define a set of instructions which are consumed by the mapping engine in order to crawl the corresponding deployment model format. These instructions are materialized in form of mapping modules, written in a conditional, query-like syntax which can be manifested in any human readable format, e.g., markup languages like JSON or YAML. These instructions can range from very simple parsing specifications to more complex structured queries which rely on conditional checks. If there are no complex constraints at all, developers can simply pass the *root* path for crawling the corresponding sources:

```
Function:
  select:
    root:
      - <path-to-the-resources>
```

In some cases, conditional checks must be performed distinguish potential matches from other resource declarations, such as, `Resources` in SAM templates, which can be both Lambda functions and other resource declarations. In this particular scenario, the `Type` property inside these resource declarations are the decisive factor to make a difference between function declarations and other resources. The following code example illustrates how to define conditional checks for property values in order to get the needed information about the resource type:

```
Function:
  select:
    root:
      - <path-to-the-resources>
  where:
    path:
      - <path-to-property-of-interest>
    value: <value that the property must have to fulfill the requirement>
```

After passing the root path for retrieving the list of resources, the path of the `where` statement is set. The where-path is propagated beginning from the root path to the property of interest and checks if its value is equal to the predefined `value` in the mapping module. For equal values, the investigated object is identified as a *function* and extracted from the template. Listing 5.1 presents an excerpt of a mapping module configuration for crawling incorporated event sources inside a Lambda-hosted application which is modeled using AWS SAM. The module specifies that objects, which are describing event sources, can be found inside the `Events` property of list entries inside the `Resources` list for which the `Type` value is equal to *AWS::Serverless::Function* (cf. Listing A.3). The mapping engine is decoupled from other components in the system which enables us to support adapter-based plugins for mapping engine extensions, e.g., for handling deployment models written in different markup languages. For instance, most modern IaC modeling approaches are realized in YAML which results in a demand for a mapping engine based on a YAML parsing library, e.g., SnakeYAML [Som19] or Jackson [Fas19]. However, one possible scenario may be to retrieve a deployment model in JSON format, e.g., Azure Resource Manager or CloudFormation templates, which accordingly needs to be addressed with an alternative implementation of the mapping engine. In conclusion, covering new template formats and potential changes in providers' application models still requires manual work. However, adapting changes inside of simplistic mapping files, which most often only comprise a few lines of code, instead of re-engineering static mapping components inside the system, can be considered a less verbose approach which not only reduces the effort for adapting the mapping process but also decreases the risk of inserting bugs when changing the business logic.

Listing 5.1 Excerpt of a mapping module for mapping event sources of Lambda-hosted functions inside of SAM templates

```
EventSources:
  select:
    root:
      - Resources
    path:
      - Properties
      - Events
  where:
    path:
      - Type
    value: AWS::Serverless::Function
```

5.2 Prototypical Implementation

This section describes *Skywalker*, a prototypical implementation of the migration support system, introduced in Section 5.1. Skywalker is an open source application[1], which is implemented as a self-contained web-application utilizing a REST-based client-server architecture which is illustrated in Figure 5.3. The prototype provides a proof-of-concept for the migration approach, as introduced in Chapter 4. In the following, the prototype's constituents are described in detail.

5.2.1 Skywalker Client

Although Skywalker's core functionality is implemented inside the application layer which exposes its services via REST endpoints, we found that providing a dedicated graphical user interface to developers fits well with the idea of a migration support system by creating transparency over the different data models with which developers work in migration tasks. The client is implemented as a single-page web application, built with Angular [Goo19] and written in TypeScript, which enables developers to perform CRUD operations, such as, viewing and editing all available deployment packages, deployment models, or functions. In addition, the implemented services of Skywalker can be triggered via HTTP requests which are enrobed behind the web interface, e.g., starting the portability evaluation for a given application to a selected target platform, or providing instructions to extract the deployment package of an application from a FaaS platform.

5.2.2 Skywalker Seed

The system's core functionality is implemented inside a server-side Maven application *Skywalker Seed*. Skywalker Seed comprises several services to support developers in migration tasks, as described in 4 and packages correlating services in *feature management groups*, as illustrated in the conceptual architecture model (Figure 5.1). The application is developed using Spring [Piv19c], a widely used framework for rapid prototyping of self-contained Java-based applications in combination with an Angular frontend component. Skywalker's plugin-based architecture allows to extend the logical components in order to adapt to novel platforms, or changing deployment model formats. For instance, we implemented the *Code Analysis* service to analyze and annotate functions which are written in Java, however, the *Code Analysis Engine* (cf. 5.1) is a decoupled, self-contained component of the system. This leads to the capability for extending or even exchanging the component, e.g., for scenarios in which functions need to be analyzed which were written in another language. In the following, the features of Skywalker Seed are described:

API Specification We applied the lessons-learned from a diverse set of related works for efficient design of APIs [MR11; SMS18] and embrace Richardson's maturity model of REST [Fow14] when designing the REST controllers for Skywalker's services. As shown in Table 5.1, basic CRUD interactions are specified following a generic design pattern across the service interfaces for managing the corresponding resources, such as, deployment models, mapping modules, applications models, and deployment packages. Table B.1 and B.2 provide an overview of the complete API specification for the services in the prototypical implementation.

[1] https://github.com/iaas-splab/skywalker-prototype

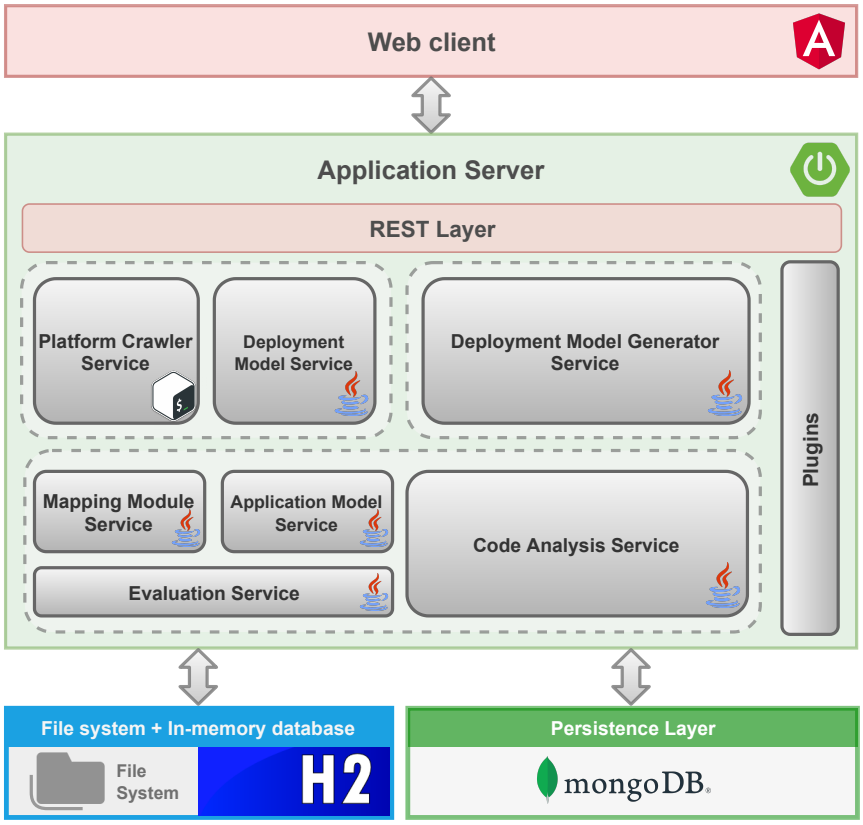


Figure 5.3: The prototypical framework's system architecture consisting of an Angular web client which provides a GUI for developers to interact with the Spring-based application server via HTTP.

Alongside with the generic API specification for managing the different resources, Skywalker Seed implements endpoint services to provide developers decision support when porting a FaaS-based application to another platform. Depending on the component, we defined one or more `POST` request mappings within the services' REST controllers to invoke services which perform the respective business logic.

Platform Crawler The crawling service allows developers to extract the source application's deployment package directly from the platform, on which it is currently deployed on. Section 4.1.1 describes how developers either manually extract the deployment packages, or use the providers' supported API, if available, e.g., using AWS' interfaces[2] or requesting the package from Azure's GET API [Mic19i]. Skywalker integrates an extensible *platform crawling service* to address this extraction from the source platform in an automated fashion by exposing an endpoint in the deployment package service which accepts `POST` requests to **`/packages/crawl/`**`{extractionData}`. The requirements for extracting a deployment package are most often limited to have knowledge about the application's resource identifier on the corresponding platform which is passed as an argument for the command line tool or defines the address of the applications REST API. The

[2] https://docs.aws.amazon.com/cli/latest/reference/lambda/get-function.html

Path	HTTP Method	Description
/{resourceName}	GET	Returns a list of resources
/{resourceName}/{resourceObject}	PUT	Adds a new resource to the repository
/{resourceName}/{id}	DELETE	Removes one or more resources from the repository.

Table 5.1: Generic API specification schema for each service's basic CRUD operations.

extraction data bundles the resource identifier and the platform from which the deployment package should be extracted from [Mic19i]. The REST controller routes the application's identifier to an automation script which conforms the extraction platform in the extraction data. Finally, a subroutine is initiated, which makes API calls to the provider's API and crawls the application's deployment package. Note that this approach requires that the corresponding provider-specific command line tools must be installed on the application server and properly configured with sufficient permissions or access rights to extract the data from the platform, thus, should only be considered when deploying Skywalker Seed on-premise or on private environments.

Generation of Application Models We implemented an endpoint which accepts PUT requests to **/apps/**{config} with a configuration object which contains the id of a deployment model and a suitable mapping module for the transformation into a generic application model. As described in Section 5.1, the mapping engine follows the instructions inside the mapping module to find the information of interest, e.g., about incorporated event sources, or function declarations. Skywalker parses the deployment model file which are materialized in YAML format and extracts the requested information from the resulting tree-structure. The output of the extraction is a set of resources for each of the following resource types:

- Event sources
- Invoked services
- Functions

Next, the retrieved knowledge about the application's architecture is transformed from the provider-specific format into an application model with generic identifiers for resources and their properties (Section 4.2). The *persistence layer* contains two repositories: (1) *Service Mapping Repository* and (2) *Service Property Mapping Repository*, which maintain knowledge about various cloud service from different providers and their generic resource categories. Section 4.4 explains the concept of service classification in detail. The output of the PUT request is an application model which describes the relevant architectural aspects gained from the deployment model in a generic fashion. The resulting application model artifact is stored in the corresponding repository in the resource layer and is ready to be used for the portability evaluation.

Evaluation Service Skywalker integrates an *evaluation service* as part of the evaluation manager component which is exposed via an endpoint that can be reached over a POST request to resource path **/apps/**bundle. The evaluation feature is correlating with the evaluation phase of our concept, as introduced in Section 4.5. The web client allows developers to view the list of generated application models and analyze them with respect to incorporated event sources, functions, and

invoked services (Figure B.2). As described in Section 4.5.1, for each generic application model, the portability to a selected target platform can be evaluated with respect to incorporated event sources and, subsequently, the result is displayed in a transparent overview in the web client's user interface (Figure B.5 and B.6).

Code Analysis For the prototypical implementation, we developed a *code analysis service*, exposed through an endpoint for `POST` requests at **`/packages/`**`functions`, which analyzes serverless functions from the analyzed deployment packages and annotates them to support developers in refactoring provider-specific parts when porting the application to another platform. The service is developed using *JavaParser* [Str19] which is a simple and lightweight library for parsing and analyzing Java code. The functions are analyzed for provider-specific libraries and annotates the lines of code which implement any logic using provider-specific APIs. Finally, the corresponding deployment package is updated with the annotated functions.

Migration Package Generation After evaluating the portability to a chosen target platform and analyzing the serverless functions of the analyzed deployment package, the resulting collection of resources and knowledge can be used to generate initial packages for the migration of the extracted application. Hereby, the annotated functions already serve as a usable starting point to support developers perform the necessary source code level refactoring when porting the application to the target platform. The translation of deployment models is functionally dependent from the result of the portability evaluation, according to the coverage of similar service offerings and their similarity with the source platform services, respectively. Provided that the generation components are pluggable and can be extended with the desired functionality, such as, static translation routines for pre-defined use cases, the framework's basic functionality provides the generation of initial boilerplate templates for the deployment model, conforming the target platform's specific format. The output of this service ranges from automatic deployable resource bundles to packages with boilerplates and metrics, which are consulting the developer on how to reengineer the application to successfully deploy it on the chosen target platform.

5.3 Case Study

This section presents a preliminary case study which showcases how developers can be supported by applying the introduced concepts on a real world scenario. In Chapter 3, we introduced two example applications which present common use cases for FaaS applications. In the following, we focus on the use case of having an application similar to ThumbNailer (Figure 3.1a) as an input for Skywalker.

Application Context

In order to provide a real world example for the implementation of ThumbNailer, we take a working example from the *repository for serverless and FaaS prototypes*[3] developed at University of Stuttgart, Institute of Architecture of Application Systems (IAAS). The project *faas-migration*[4] provides project packages for remote deployments of several common use cases for FaaS applications. Each

[3] https://github.com/iaas-splab
[4] https://github.com/iaas-splab/faas-migration

Listing 5.2 Mapping module instructions which can be used to crawl architectural knowledge about Lambda-hosted applications from deployment models written in the Serverless Framework DSL.

```
EventSources:
  select:
    root:
      - functions
    path:
      - events
InvokedServices:
  select:
    root:
      - provider
      - iamRoleStatements
Function:
  select:
    root:
      - functions
```

application comprises deployment packages for achieving equivalent versions of the application for Lambda, Azure, and IBM, whereas the Lambda-hosted application is deployed using the Serverless Framework [Ser19]. In this case study, we take *ThumbnailGenerator*'s deployment package for Lambda as source application and provide decision support for porting it to Azure's FaaS platform.

Migration Support Procedure

We assume that the deployment package contains the deployment model in format of a domain-specific `serverless.yml` manifest file and the Java functions which are referenced in the function declarations of the model. Using the web client, the deployment package is uploaded to Skywalker Seed and persisted in the database. The saved deployment package can be viewed in the corresponding frontend component of the web client (B.2).

The first step is to analyze the functions for lines of code with usage of provider-specific SDKs or other types of API calls which are related to provider-specific logic. The analyzed functions are annotated with references to the expression or statement that potentially declares provider-specific logic and subsequently replace the plain functions in the stored deployment package (Figure B.7).

Next, the deployment model is analyzed for its portability to Azure Functions. In prior to the evaluation for this, we need a mapping module which defines instructions for crawling the needed information about event sources, invoked services and function declarations, from the Serverless Framework-specific deployment template format. Listing 5.2 presents a suitable mapping module for this scenario.

Subsequently, a generic application model can be generated by selecting the deployment model and the just added mapping module with the desired name for the generated application model instance. As shown in Figure 5.4, the resulting application model contains a set of generic service categories and properties which correlate to the provider-specific service descriptions in the original deployment model.

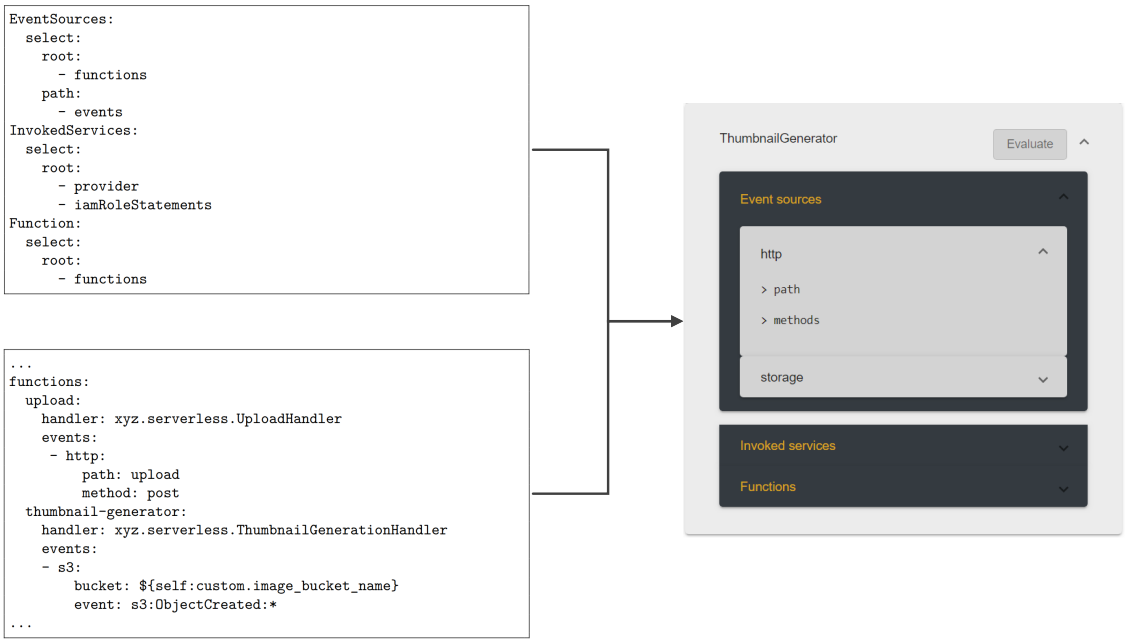


Figure 5.4: The deployment model is analyzed using a pre-defined mapping module with instructions for the resource discovery process. The resulting application model is displayed in the web client.

In the next step, the portability of the generic application model is evaluated with Azure Functions as target platform. In Figure 5.5, the similarity of the service offerings are compared between the generic application model for ThumbnailGenerator, and the service offerings which are supported at Azure's platform (cf. Section 4.4). Note that, although the service similarity is evaluated with respect to the services' properties, we only compare the coverage of those properties, which are explicitly defined in the parsed deployment model from the application model generation phase.

Figure 5.5 shows that the storage offerings' set of similar properties for Azure Functions and the implemented Lambda function is limited to the the `resourceId` which can be considered a minimum configuration of a service declaration (there are exceptions to this, e.g., scheduled jobs). While for Lambda functions, the storage-based trigger can be configured in a fine-grained manner for particular types of changes at the bucket instance, Azure functions get triggered at any event originating from storage-based event sources, thus, the `events` on which the invocation of functions are triggered cannot be configured, e.g., to ignore trigger events caused by deletions. Consequently, this property presents a potential dissimilarity in the way the event sources can be integrated to the application context. Nonetheless, it is the developers decision to evaluate the actual significance of the lack of this particular preference in the application context. For instance, the business logic of `ThumbnailGenerationHandler` might implement an event type checking, which also reacts to deletions by removing the existing thumbnail at the second bucket. Even if this is not the case, the developer may consider modifying the business logic to include such a sub-routine, instead of selecting another platform. This may in particular be the case if the effort to refactor the function code is less than the evaluation of enterprise policies and SLAs for another provider's platform.

Listing 5.3 Translation of existing deployment models (top) into boilerplate templates for the target platform (bottom).

```
# original deployment file for AWS Lambda
service: thumbnail-generator
...
provider:
  name: aws
  runtime: java8
  stage: dev
  region: us-east-1
  iamRoleStatements:
    ...
functions:
  upload:
    handler: xyz.serverless.UploadHandler
    events:
     - http:
         path: upload
         method: post
  thumbnail-generator:
    handler: xyz.serverless.ThumbnailGenerationHandler
    events:
    - s3:
        bucket: ${self:custom.image_bucket_name}
        event: s3:ObjectCreated:*
...
```

```
# boilerplate template of a deployment model for Azure Functions
service: thumbnail-generator
...
provider:
  name: azure
  runtime: java8
  stage: dev
  location: West US
  ...
functions:
  thumbnail-generator:
    handler: xyz.serverless.ThumbnailGenerationHandler
    events:
    - blob:
        path: ''
  upload:
    handler: xyz.serverless.UploadHandler
    events:
    - http:
        route: ''
        methods: ''
```

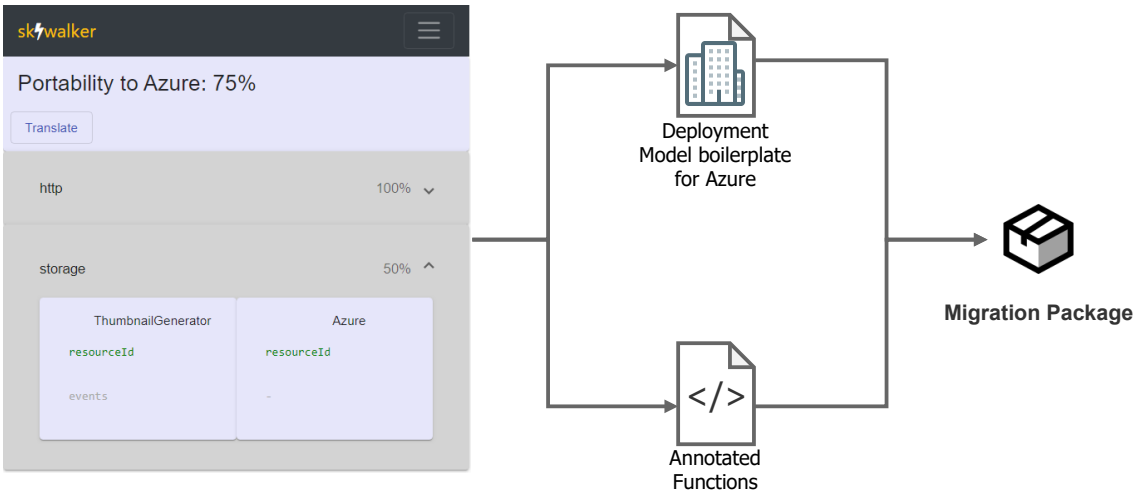


Figure 5.5: After portability evaluation, developers can generate boilerplate templates for provider-specific deployment models and analyze the annotated functions to reveal the needed effort for refactoring.

Finally, the analyzed deployment package can be transformed in a migration model which consists of annotated functions (Figure B.7), and boilerplate templates (Listing 5.3) which serve as a starting point for creating a deployable application model for Azure's FaaS runtime. Essentially, the resulting template could be already ready to get deployed, however, there are platform-dependent restrictions, such as, naming conventions, limitations of namespaces, e.g., `resourceId` for AWS resources must be globally unique[5], whereas in Azure Resource Groups have a user-based scope for namespaces. Furthermore, in most cases, refinement based on the dissimilarities in the interaction between event sources and hosted functions must be considered when porting FaaS-based applications across providers. For instance, Azure Functions directly receive the storage object in the request body of the trigger, whereas in AWS Lambda the business logic must explicitly include API calls with `GET` requests to access the object when the function get invoked. In our case study, these changes can be considered as improvements of code quality by exploiting the target platform's unique features [Cha19; Eic19; Gol19], which may be considered good practice. However, for scenarios in which Azure-hosted serverless applications are ported to AWS Lambda, special paradigms like these rather represents a potential point of failure, than an added value.

[5] https://docs.aws.amazon.com/AmazonS3/latest/dev/UsingBucket.html

6 Related Work

This chapter discusses relevant scientific work from various fields of research which complement our concept, introduced in Chapter 4 and serve as a starting point for our conceptual framework. Each section starts with the work which is closest to our concept followed by a short overview of related works in the respective research field.

6.1 Vendor Lock-in Mitigation

Stigler proposed a provider-agnostic approach to deploy serverless functions on different cloud providers [Sti18]. Using the Serverless Framework as a foundation for this approach, Stigler generated a project structure to enable a provider-agnostic deployment of a function for storing objects in the provider's corresponding Blob storage service (Figure 6.1).

This approach concentrates strongly on the encapsulation of usage of provider-specific SDKs on the source code level of serverless functions. Typically, serverless functions are written in one handler file which handles a single purpose, like getting a message from a POST request and writing the data into a storage offering. Stigler separates provider-specific logic from provider-agnostic code to achieve a solution that is as provider-independent as possible. This is done by dividing the logic into separate files which are located in a normalized schema of directories for the corresponding provider. The application consists of an `index.js` file which contains a variable for the target

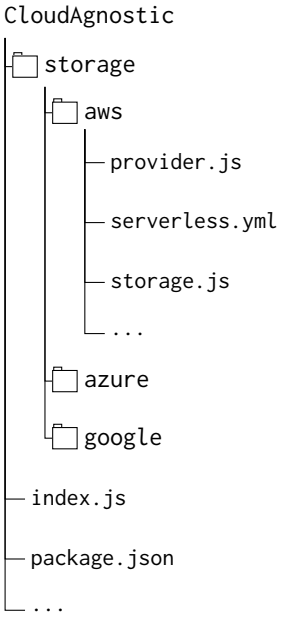


Figure 6.1: Project structure for a provider-agnostic deployable application using the Serverless Framework [Sti18]

Listing 6.1 Schema for encapsulation of provider-specific code from provider-agnostic logic according to the approach of [Sti18]

```
// index.js
var provider = 'aws';
var Storage = require('./storage/' + provider + '/storage');

exports.handler = function (event, context, callback) {
    Storage.saveObject(event, '1');
    ...
}
```

```
// storage.js
var AWS = require('aws-sdk');
...
module.exports = {
    saveObject: function(message, messageId) {
        S3.putObject({
            // provider-specific logic
        }
        ...
}
```

provider. The user passes the value of the provider into this variable and automatically configures the path to the corresponding provider logic. In each of these provider directories, the required logic is implemented in generic named files with generic named methods for similar tasks across all supported providers (Listing 6.1).

Each provider directory represents a different service, that is, there is also a `serverless.yml` file for the deployment of the application on the corresponding provider in each of the provider directories. Certainly, Stigler focused on designing a provider-agnostic solution for writing the handler code in serverless applications. It is worth mentioning that this proof of concept is designed for a very simplistic architecture using an HTTP trigger as event source. Sophisticated variants of serverless applications may require more effort in terms of configuration of the deployment package files and researching the corresponding implementation patterns across the providers for additional services. The proposed solution may be integrated in automated code generation approaches which translate existing functions into modular packages of provider-specific and provider-agnostic code. The concept of code artifact generation is present in the migration model generation and can be compared to the introduced concepts in our concept for the target platform.

Demchenko et al. propose the *Intercloud Architecture Framework* (ICAF) which deals with interoperability in heterogeneous, multi-cloud scenarios [DND13]. ICAF is an ongoing research in the field of interoperability of multi-domain cloud-based applications including concepts for multi-layer cloud service models, service delivery and lifecycle management plans. In their work, four inter-related components are introduced which address several issues in cloud integration scenarios, including an *Intercloud Federation Framework* (ICFF). The proposed ICFF includes components for the translation of requests, protocols, and data formats between cloud domains for aspect, such as, *service discovery*, *attribute/namespace translation*, or *service and trust brokers*.

The work on the ICAF addresses numerous similar challenges when compared with our migration concept, e.g., the discovery and translation of incorporated services which are implemented in a proprietary manner specific to the provider.

6.2 Cloud Migration

Although cloud computing has become a commodity that has already been adopted to many business models, there are still many systems that exist in an isolated on-premise environment. On the one hand, this may have legitimate reasons, which can be attributed to legal restrictions, but on the other hand migrating these systems to the cloud can be considered a feasible action to improve their performance.

6.2.1 Legacy-to-Cloud Migration

Previous research strongly focused on migration of legacy software systems to state-of-the-art architectures and moving them to the cloud. Frey and Hasselbring proposed *CloudMIG*, a model-based approach which aims at supporting SaaS providers to semi-automatically migrate existing enterprise software systems to IaaS and PaaS-based applications with alignment to the cloud environment [FH11b]. CloudMIG is composed of six migration activities (Figure 6.2).

Extraction. First, the existing system's architecture needs to be extracted. Frey and Hasselbring propose OMG's Knowledge Discovery Meta-Model (KDM) [Obj16] for building a meta-model of the given architecture. Further, metrics concerning, for instance, usage of services are proposed to be collected via monitoring approaches like OMG's Structured Metrics Meta-Model (SMM) [Obj18] to build a meta-model for metrics of interest.

Selection. In this step, a specific cloud provider will be selected as a target platform for the migration. The corresponding model for the target platform will be an instance of the *Cloud Environment Model* (CEM) which describes common properties of cloud environments [FHS13]. In addition, the CEM includes *Cloud Environments Constraints* (CEC) which will enable the system to discover violations in the planned migration to a specific platform [FH11a; FHS13].

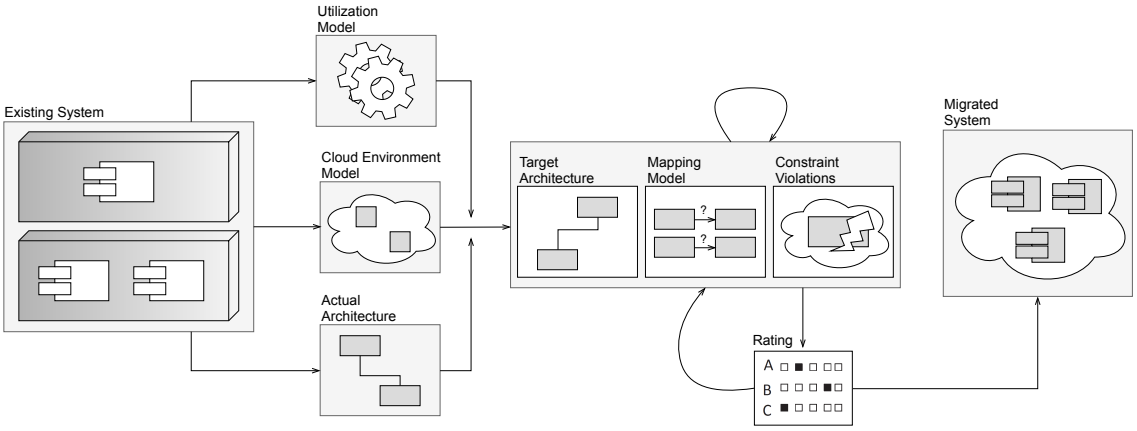


Figure 6.2: Outline of CloudMIG architecture [FH11b].

Generation. Three artifacts will be generated which will make up one of the core parts of the CloudMIG approach. First, the *Mapping Model* is generated, which defines the assignment of elements from the existing architecture to the target architecture. Further, a model for CEC violations of the planned target architecture is generated and resulting from this, the initial generation of a CSA tuple is performed. *Cloud Suitability and Alignment* (CSA) is an hierarchy model for the suitability of existing architectures to the migration to the cloud with respect to expected CEC violations. Frey and Hasselbring introduce the five levels of the CSA hierarchy: *Cloud incompatible, Cloud compatible, Cloud ready, Cloud aligned, Cloud optimized*. The classification relies on factors, such as the existence of CEC violations and their *severity* level or the ability of the software architecture for horizontally scaling. Finally, the model for the generation of the target architecture is initialized by assigning elements of the existing architecture into their counterparts of the target architecture.

Adaptation. The engineer adjusts the target architecture manually to be compliant towards case-specific requirements that could not be covered within the generation activity.

Evaluation. The resulting system's architecture should be evaluated before entering production phase. Frey and Hasselbring propose to simulate the model using CloudSim [CRB11]. Future work is planned to contribute a transformation from CEM to CloudSim's simulation model.

Transformation. The CloudMIG approach intends to generate considerable parts of the target architecture. The actual transformation is not realized, yet, but will be tackled in future research.

Certainly, CloudMIG is a promising approach for many migration scenarios. However, this approach also makes clear that the migration of applications to novel environments is a challenging task which is not yet possible to be fully-automated without integrating manual work of the engineer. Frey and Hasselbring introduce abstract concepts which are related to our concept, such as CEM to describe common properties of the environments, and the concept of a *suitability hierarchy*, as they incorporate it with the CSA hierarchy. Many steps of CloudMIG suffer from manual, provider-dependent tasks, such as, the manual creation of CEM instances for a corresponding provider. Furthermore, although this approach enables developers to port legacy systems to cloud environments, it is limited to IaaS and PaaS-based applications.

6.2.2 Inter-cloud Migration

While numerous approaches for the migration of legacy software to the cloud have been proposed, very little research deals with migration of cloud-based applications between cloud platforms. Beslic et al. [BBSR13] proposed an approach to design a refactoring system for supporting re-engineers with the task of software migration between PaaS. The underlying concepts are related to model-driven engineering and program transformation approaches which aim to enable the engineer to migrate deployed applications from one platform, e.g. Heroku, to another (e.g. AWS Elastic Beanstalk). The migration approach consists of three steps:

Discovery. Extraction of the sources in order to gather the information on how the transformation and adaptation of the software has to be performed at an architectural level. For this step, Beslic et al. integrate MoDisco [BCJM10] with built-in KDM for source discovery. Using the gained information, the engineer can be guided to a new platform that consists of the same kind of technologies which consequently reduces the amount of manual work in the transformation phase.

Transformation. The KDM model discovery results in three abstraction layers: *(1) Program Elements* including code primitives and actions, *(2) Runtime Resources* which describes data, events, UI and platform, *(3) Abstractions* for structural properties. With these portions of information, transformation strategies can be defined for elementary operations, such as boilerplate code for storing an entity in a provider-specific storage service. Engineers define transformation rules using a DSL for language processing transformation, such as TXL [Cor06].

Migration. The application is migrated on the target platform integrating offline and online tests for validation of a correct behavior. For the case of a erroneous deployment, there is a *rollback* option to the original state on the source provider's platform.

The introduced migration approach of Beslic et al. comes closest to our concept, as described in Chapter 4. While it focuses on transformation of PaaS-based applications, the conceptual steps complement our concept with a roadmap of migration tasks from which a majority of steps can be mapped to our migration approach, e.g., discovery of relevant sources, comparison of supported technologies across cloud providers, and transformation with usage of boilerplate code.

Petcu et al. [PMPC13] propose a new set of APIs for cloud application development with respect to portability. Their proposed API is integrated into the inter-cloud framework *mOSAIC* [mOS19] in order to provide solutions for portability of cloud applications and their components. The work incorporates the concept of *cloud portability levels* [Wil09] and portability category classification [OF10] to define a set of requirements which need to be fulfilled for particular migration scenarios. The proposed approaches focus on portability of PaaS-based applications which still partly rely on traditional application architecture, however, incorporate transferable concepts for incorporated cloud services within the provider platform's ecosystems. Regarding the event-driven architecture of FaaS-based applications, these concepts can be considered as baseline work for further refinements in order to form suitable approaches for the domain of serverless computing.

6.3 Model-driven Development

Model-driven development (MDD) is the art of moving away from developing a system from scratch with raw usage of programming languages. Developers are assisted with usage of suitable, abstract modeling languages which represent technical properties of the system. These abstract models are transferred into technical artifacts in an automated fashion. Van Hoorn et al. [VFG11] propose *DynaMod*, an approach which addresses model-driven modernization of software systems. The project relies on the horseshoe model [BSWW99] for re-engineering approaches and is developed in cooperation with the CloudMIG framework [FH11b].

DynaMod comprises three phases: (1) *Extraction*, (2) *Transformation*, and (3) *Generation* of implementation artifacts and tests for the modernized system. In the first phase, the analysis and extraction of the system architecture breaks down to two work packages. The system runs through a static analysis of the source code in order to gain knowledge about the structural composition of architectural entities and relations. In this scenario, the information is extracted by using code parsing generators, such as *ANTLR* [Par19]. For the second part, a dynamic analysis yields quantitative runtime behavior information, such as workload characteristics and execution frequencies of code fragments using the monitoring tool *Kieker* [VWH12]. In the transformation phase, the extracted models are translated towards the target architecture using model-based transformation techniques based on meta-model specifications like OMG's *KDM* [Obj16]. The following phase of code

generation is implemented based on a generative architecture of the projects cooperation partners[1]. In general, the usage of templates for generation of implementation artifacts, such as SOA wrappers, may be feasible for the code generation phase. Van Hoorn et al. point out that while there is a difference between software migration and modernization, the latter provides the option to change the systems capabilities and features in a beneficial way compared with the strict approach of migration, which aims to move the existing system to a new architecture as-is.

DynaMod has several similarities to our migration concept, such as, incorporation of implementation artifacts, static code analysis, and the transformation of an existing application's architecture into one that provides similar functionality on the target platform it is migrated to. On the other hand, several features which are present in DynaMod lead to very high complexity which can be attributed to the usage of dynamic code analysis and highly complex meta-model specifications. While Van Hoorn et al. introduced a powerful system to re-engineering tasks, there are still many parts of manual work in the extraction part which can quickly become cumbersome when dealing with large event-driven architectures.

6.4 Cloud Computing Ontology

The rapid evolution of the cloud computing paradigm has lead to a situation of confusion about the actual classification of offered cloud services. Youseff et al. [YBD08] propose a dissection of the cloud into five main layers with their inter-relations to each other which lead to a detailed ontology for the cloud. While the first, *Cloud Application Layer*, and the second layer, *Cloud Software Environment Layer*, are more or less equivalent to SaaS and PaaS, respectively, the third layer gives a more detailed insight to different types of cloud services. The *Cloud Software Infrastructure Layer* provides fundamental resources to other higher-level layers, which in turn can be used to construct new cloud software environments or cloud applications [YBD08]. Youseff et al. categorize the cloud services in this layer into the following three component types:

Computational Resources. Mostly the provision of VMs as computational resources. This component can be considered equivalent to IaaS.

Data Storage. This service is also known as *DaaS* and facilitates cloud applications to scale beyond their limited servers. Provided that requirements to data storage offerings are very diverse and depending on the application context, it has been proven that it is not to provide one solution to satisfy all needs [Bre00]. Therefore, storage offerings are divided into different types which satisfy specific requirements, such as *relational databases*, *key-value storage*, *blob storage* and more [FLR14]. While there is no definite solution for every use-case, each of these offerings provide an suitable storage solution of specific application requirements, such as *strict* or *eventual consistency* [Vog09].

Communication. Communication as a Service (CaaS) describes services around network security, dynamic provisioning, guaranteed message delay, communication encryption, and network monitoring [YBD08]. Components of this type comprise communication services which enable monitoring and loosely coupled networks of microservice architectures.

[1] https://www.bmiag.de/startseite/

The two bottom layers of the *cloud ontology* are formed by the *software kernel* and *hardware and firmware* layers. While the majority of software kernel offerings are prominent in grid computing applications [FK97; TL95], hardware and firmware offerings are used to provide the customer remote bootstrap environments for low-level tasks which are tightly coupled to the architectural properties of the underlying hardware [AUW08]. With this ontology, Youseff et al. contribute to a better understanding of the inter-relations between different cloud components which aims to support engineers in the re-composition of current systems from other cloud components or existing components from other systems to fully benefit from the advantages of cloud computing offerings.

Al Masud proposes an extended and granular classification of the cloud computing taxonomy [Al 12]. This detailed approach of creating a cloud ontology extends the layers of Youseff et al. [YBD08] by adding more fine grained levels in between the existing structure and concretize the existing layers with corresponding *XaaS* (Everything as a Service) categories. This classification of cloud service layers and offerings allows to identify domains of particular cloud services and to form a foundation to a more standardized implementation of those services for concerns like seamless interoperability and portability between different service offerings.

Ilyas created a portal which presents an overview of categorized services from major public cloud providers, such as AWS, MS Azure, Google Cloud Platform, IBM Cloud, Oracle Cloud and Alibaba Cloud [FII19]. This comparison provides feature level mappings between the provider-specific services and serves as reference manual for re-engineers that seek for alternate services, particularly, for cloud application migration scenarios. The project is based on a public repository [2] and continuously gets updated by an active community of contributors.

The described concepts in the field of cloud service taxonomies and ontologies for the cloud computing paradigm can be considered as baselines for further refinements in different domains, e.g., serverless computing. Although the defined classification approaches are on a very high level, the cloud ontology concepts in this section complement our migration concept in terms of creating and maintaining knowledge bases for service categories and other resources which are relevant in the field of serverless applications.

6.5 Deployment Modeling

Where once was a gap between *deployment* and *operations*, there is an interconnection in modern software development which is called *DevOps* [HM11]. Technically, manual deployment of non-trivial services is complex and hard to repeat, hence, key concepts in the automation of software delivery have evolved. One widely accepted concept are *declarative deployment models* [AW11] which is applied in the majority of today's configuration and deployment technologies.

Breitenbücher et al. analyze common modeling approaches and define pattern primitives in their recent work on modeling concepts [EBF17]. The investigated pattern primitives are used to form two modeling patterns for the automated deployment of applications: *Declarative modeling* and *Imperative modeling*. Breitenbücher et al. further analyze imperative and declarative deployment modeling approaches for their conceptual strengths and weaknesses in the context of IoT application deployment [BKLW17]. In particular, the research discovers limitations of declarative deployment modeling approaches, such as, required manual work for concerns like role management which

[2] https://github.com/ilyas-it83/CloudComparer

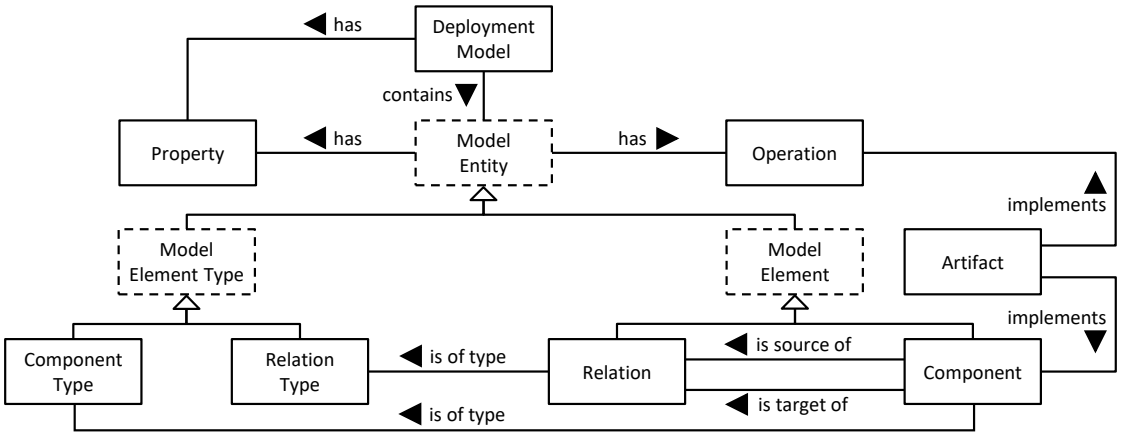


Figure 6.3: The Essential Deployment Metamodel [WBF19]

cannot be automated using declarative deployment systems. While imperative deployment modeling solves this issue, it suffers from transparency and comprehensibility for developers. Breitenbücher et al. show that the major drawbacks of each approach is solved by the other one [BKLW17] and propose the idea of a hybrid solution for IoT deployment modeling.

Wurster et al. [WBF19] present a systematic review of declarative deployment technologies and introduce the *Essential Deployment Metamodel* (EDMM) by extracting the essential parts that are supported by all these technologies. The EDMM is based on a collection of different categories of deployment technologies:

General purpose technologies - which support single-, hybrid-, and multi-cloud deployments as well as different kinds of cloud services (XaaS). This category includes technologies like Puppet, Chef, Ansible and Terraform.

Provider-specific technologies - which support single-cloud deployments for the corresponding provider's cloud offering. Examples for this category of deployment technologies are: AWS CloudFormation and Azure Resource Manager (ARM).

Platform-specific - support multiple cloud providers, however, are restricted to the underlying platform, e.g., Kubernetes, CFEngine, and Docker Compose.

EDMM is based on the various declarative deployment technologies and consists of derived entities from the analyzed technologies domain-specific languages to describe the desired state of an application (Figure 6.3). For brevity, we only provide a quick outline of the conceptual meta-model and how the introduced entities relate to each other. On the highest layer, we have a *Deployment Model* which describes declaratively the desired target state of an application including all necessary model entities. This target state is a completed deployment of all *Components* (high-level software components), *Component Types* (detailed descriptions of the used technology for each component, e.g., Tomcat server or Java application), their *Relations* (and *Relation Types*) with inclusion of specified *Properties* and *Artifacts*, as described in the deployment model. The presented EDMM technology can be used for mapping purposes in migration approaches between deployment technologies. The authors present a semantic mapping between the EDMM and deployment technologies of the above mentioned categories, such as, Chef, Terraform, Cloud Formation, Azure Resource Manager, and Kubernetes.

In conclusion, the proposed concepts can be considered a big step in the direction of unified approaches for automated deployment modeling and furthermore potentially improve migration-related concepts with respect to mapping of supported deployment technologies. Typically, cloud-native applications are modeled declaratively and deployed in an automated fashion. Wurster et al's EDMM has similarities to our concept of generic application models, however, turns out to be too complex for our use-case. The concept of mapping a generic model to commonly used formats is a desirable feature which is also considered in the step of migration model generation in our concept.

7 Conclusion and Outlook

In this work, we develop a framework which comprises interconnected concepts on supporting developers in moving existing serverless applications to another platform. Over the course of our research, we captured various features of the serverless computing paradigm, such as, developers and software architects can benefit from the fact that most modern FaaS platforms embrace the concept of infrastructure as code. More specifically, these infrastructure configurations are most often materialized in form of declarative deployment models, which can be utilized to discover architectural components and their inter-relations in an efficient fashion. In comparison, the discovery of traditional software architecture is potentially complex and therefore requires an extensive amount of manual verification. On the other hand, while we can benefit from the event-driven architecture of serverless applications, different providers typically rely on proprietary programming models, ecosystems, and formats to model the hosted applications' architecture. Translating provider-specific models to a generic format has proven to be a feasible approach for the comparison of different platforms. In order to facilitate these translations, it is typically a good idea to maintain knowledge about the service ecosystems of the various platforms. Since different providers typically design their services in a non-standardized manner, the task of aligning the providers' set of services is non-trivial. Thus, the concept of community-driven knowledge bases represent a potential solution to overcome this issue. Ultimately, we come to the conclusion that a black-box approach is infeasible for porting FaaS-based applications across providers. This can be attributed to the fact that the high level of abstraction leads to an even higher demand on human reasoning. For instance, it is typically feasible to generate migration packages which consult developers in the reengineering tasks, e.g., for exploiting the target platform's special features and consequently even improve the application's efficiency, as described in Section 5.3.

We found that there are several ways in which developers can be supported in the task of porting existing FaaS applications across providers. We presented them in a conceptualized framework and developed a proof-of-concept to showcase the added value of the introduced concepts. The concepts presented in this work are designed in an extensible fashion which allows to refine the framework with future enhancements. This is especially important in the context of serverless because vendors push updates to their platforms in a very fast pace. Our prototype provides a proof-of-concept, however, there are still aspects in the serverless migration context, which can be covered in future work. In terms of code generation, additional extensions might be added to provide the concept of an interactive code analysis engine, as described in Section 4.6.2, which analyses functions written in multiple programming languages. In this work, we mainly focus on porting applications between platforms, which means, the application should ideally be deployed on the target platform as-is. Alternatively, an interesting research question is how to consult developers in improving the existing applications, either when porting them to another platform, or when considering to stay on the original platform and conforming to new requirements, such as, SLAs, by reengineering the application to improve its performance. Furthermore, existing serverless applications may be refactored in order to make them partly provider-agnostic, as described in [Sti18] which may result in an even higher portability for future migration scenarios and thus reducing the impact of the vendor lock-in when moving to another platform. There are phases of our migration concept which

rely on the existence of knowledge bases which can be extended in future work. The framework may be extended to support different providers, include knowledge about migration paradigms, such as, alternative evaluation scores, or extensions to the mapping repositories. Essentially, the discovery phase in this work focuses on the static analysis of applications, however, when evaluating the suitability of target platforms, the inclusion of dynamic analysis approaches, e.g., workload statistic monitoring on function invocations may complement the suitability analysis for potential target platforms, especially, for performance-related reasons for moving applications to another platform.

Serverless computing slowly reaches the *plateau of productivity* [LF03], which means, serverless approaches will most likely become mainstream in the near future [Gar19b]. Current technological trends, such as, AI, are moving towards PaaS-based models [Ven19]. It is conceivable that the next step in the direction of FaaS is not too far in the future [Gar19a] which consequently underpins the importance of supporting developers in migration-related tasks within the serverless ecosystem. Providing migration support, however, should not be treated as equal with removing the vendor lock-in but rather to take over considerable parts of the migration and provide decision support for the remaining steps which require manual verification. This way the concerns from utilizing serverless offerings can be reduced to a minimum and consequently developers are able to not fear vendor lock-in, but rather embrace the service offerings of the providers in order to get the best possible value out of their usage.

Bibliography

References

[Al 12] S. M. R. Al Masud. “An Extended and Granular Classification of Cloud’s Taxonomy and Services”. In: *International Journal of Soft Computing and Engineering (IJSCE)* 2.2 (2012), pp. 278–286 (cit. on p. 83).

[AUW08] J. Appavoo, V. Uhlig, A. Waterland. “Project Kittyhawk: building a global-scale computer: Blue Gene/P as a generic computing platform”. In: *ACM SIGOPS Operating Systems Review* 42.1 (2008), pp. 77–84 (cit. on p. 83).

[AW11] P. Anderson, G. Wickler. “Automated planning for configuration changes”. In: *Proceedings of the 2011 LISA Conference. Usenix Association*. 2011 (cit. on p. 83).

[AZ18] A. Aske, X. Zhao. “Supporting multi-provider serverless computing on the edge”. In: *Proceedings of the 47th International Conference on Parallel Processing Companion*. ACM. 2018, p. 20 (cit. on p. 20).

[BBSR13] A. Beslic, R. Bendraou, J. Sopenal, J.-Y. Rigolet. “Towards a solution avoiding Vendor Lock-in to enable Migration Between Cloud Platforms.” In: *MDHPCL@ MoDELS*. Citeseer. 2013, pp. 5–14 (cit. on pp. 31, 33, 63, 80).

[BCC17] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, et al. “Serverless computing: Current trends and open problems”. In: *Research Advances in Cloud Computing*. Springer, 2017, pp. 1–20 (cit. on pp. 15, 22, 23, 29, 31, 46).

[BCJM10] H. Bruneliere, J. Cabot, F. Jouault, F. Madiot. “MoDisco: a generic and extensible framework for model driven reverse engineering”. In: *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM. 2010, pp. 173–174 (cit. on p. 80).

[BDE14] C. Baudoin, E. Dekel, M. Edwards. “Interoperability and Portability for Cloud Computing: A Guide”. In: *Cloud Standards Customer Council* (2014), pp. 1–8 (cit. on p. 33).

[BKLW17] U. Breitenbücher, K. Képes, F. Leymann, M. Wurster. “Declarative vs. Imperative: How to Model the Automated Deployment of IoT Applications?” In: *Proceedings of the 11th Advanced Summer School on Service Oriented Computing*. IBM Research Division, Nov. 2017, pp. 18–27 (cit. on pp. 83, 84).

[BLS11] T. Binz, F. Leymann, D. Schumm. “CMotion: A Framework for Migration of Applications into and between Clouds”. In: *2011 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE. 2011, pp. 1–4 (cit. on p. 15).

[Bre00] E. A. Brewer. “Towards robust distributed systems”. In: *PODC*. Vol. 7. 2000 (cit. on p. 82).

[Bre16] U. Breitenbücher. “Eine musterbasierte Methode zur Automatisierung des Anwendungsmanagements”. In: (2016) (cit. on p. 63).

[BSWW99] J. Bergey, D. Smith, N. Weiderman, S. Woods. *Options analysis for reengineering (OAR): Issues and conceptual approach*. Tech. rep. Carnegie-Mellon University, Pittsburgh PA Software Engineering Institute, 1999 (cit. on p. 81).

[CAL94] D. Cohn, L. Atlas, R. Ladner. “Improving generalization with active learning”. In: *Machine learning* 15.2 (1994), pp. 201–221 (cit. on p. 57).

[Cor06] J. R. Cordy. “The TXL source transformation language”. In: *Science of Computer Programming* 61.3 (2006), pp. 190–210 (cit. on p. 81).

[CRB11] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. De Rose, R. Buyya. “CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms”. In: *Software: Practice and experience* 41.1 (2011), pp. 23–50 (cit. on p. 80).

[DND13] Y. Demchenko, C. Ngo, C. De Laat, J. A. Garcia-Espin, S. Figuerola, J. Rodriguez, L. M. Contreras, G. Landi, N. Ciulli. “Intercloud architecture framework for heterogeneous cloud based infrastructure services provisioning on-demand”. In: *2013 27th International Conference on Advanced Information Networking and Applications Workshops*. IEEE. 2013, pp. 777–784 (cit. on p. 78).

[DOSP13] A. Desai, R. Oza, P. Sharma, B. Patel. “Hypervisor: A survey on concepts and taxonomy”. In: *International Journal of Innovative Technology and Exploring Engineering* 2.3 (2013), pp. 222–225 (cit. on p. 21).

[EBF17] C. Endres, U. Breitenbücher, M. Falkenthal, O. Kopp, F. Leymann, J. Wettinger. “Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications”. In: *Proceedings of the 9th International Conference on Pervasive Patterns and Applications (PATTERNS)*. Xpert Publishing Services, Feb. 2017, pp. 22–27. ISBN: 978-1-61208-534-0 (cit. on p. 83).

[Ern08] A. M. Ernst. “Enterprise architecture management patterns”. In: *Proceedings of the 15th Conference on Pattern Languages of Programs*. ACM. 2008, p. 7 (cit. on p. 17).

[FH11a] S. Frey, W. Hasselbring. “An extensible architecture for detecting violations of a cloud environment’s constraints during legacy software system migration”. In: *2011 15th European Conference on Software Maintenance and Reengineering*. IEEE. 2011, pp. 269–278 (cit. on p. 79).

[FH11b] S. Frey, W. Hasselbring. “The cloudmig approach: Model-based migration of software systems to cloud-optimized applications”. In: *International Journal on Advances in Software* 4.3 and 4 (2011), pp. 342–353 (cit. on pp. 79, 81).

[FHS13] S. Frey, W. Hasselbring, B. Schnoor. “Automatic conformance checking for migrating software systems to cloud infrastructures and platforms”. In: *Journal of Software: Evolution and Process* 25.10 (2013), pp. 1089–1115 (cit. on p. 79).

[FK89] G. C. Fox, J. Koller. “Code generation by a generalized neural network: general principles and elementary examples”. In: *Journal of Parallel and Distributed Computing* 6.2 (1989), pp. 388–410 (cit. on p. 60).

[FK97] I. Foster, C. Kesselman. “Globus: A metacomputing infrastructure toolkit”. In: *The International Journal of Supercomputer Applications and High Performance Computing* 11.2 (1997), pp. 115–128 (cit. on p. 83).

[FLR13] C. Fehling, F. Leymann, S. T. Ruehl, M. Rudek, S. Verclas. “Service Migration Patterns–Decision Support and Best Practices for the Migration of Existing Service-Based Applications to Cloud Environments”. In: *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications*. IEEE. 2013, pp. 9–16 (cit. on p. 18).

[FLR14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud computing patterns: fundamentals to design, build, and manage cloud applications*. Springer, 2014 (cit. on pp. 20, 21, 82).

[Fow02] M. Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002 (cit. on p. 17).

[Fow14] M. Fowler. “Richardson Maturity Model: steps toward the glory of REST. 2010”. In: *Acesso em* 10 (2014) (cit. on p. 69).

[Goo10] D. F. M. Goodman. “Code Generation: A Strategy for Neural Network Simulators”. In: *Neuroinformatics* 8.3 (Oct. 2010), pp. 183–196. ISSN: 1559-0089. DOI: 10.1007/s12021-010-9082-x (cit. on p. 60).

[Gra06] J. Gray. “A conversation with Werner Vogels”. In: *ACM Queue* 4.4 (2006), pp. 14–22 (cit. on p. 17).

[Han13] R. S. Hanmer. *Patterns for fault tolerant software*. John Wiley & Sons, 2013 (cit. on p. 19).

[HC01] G. T. Heineman, W. T. Councill. “Component-based software engineering”. In: *Putting the pieces together, addison-westley* (2001), p. 5 (cit. on p. 17).

[HM11] J. Humble, J. Molesky. “Why enterprises must adopt devops to enable continuous delivery”. In: *Cutter IT Journal* 24.8 (2011), p. 6 (cit. on p. 83).

[HW03] G. Hophe, B. Woolf. “Enterprise Integration Patterns”. In: *Designing, Building, and Deploying Messaging Solutions. Boston et. al.: Addison-Wesley* (2003) (cit. on p. 48).

[Jac01] P. Jaccard. “Distribution de la flore alpine dans le bassin des Dranses et dans quelques régions voisines”. In: *Bull Soc Vaudoise Sci Nat* 37 (1901), pp. 241–272 (cit. on p. 52).

[Kay03] D. Kaye. *Loosely coupled: the missing pieces of Web services*. RDS Strategies LLC, 2003 (cit. on p. 19).

[KBS05] D. Krafzig, K. Banke, D. Slama. *Enterprise SOA: service-oriented architecture best practices*. Prentice Hall Professional, 2005 (cit. on p. 17).

[KH07] G. Kotonya, J. Hutchinson. “A COTS-Based Approach for Evolving Legacy Systems”. In: *2007 Sixth International IEEE Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems (ICCBSS’07)*. IEEE. 2007, pp. 205–214 (cit. on p. 17).

[KR09] B. R. Kandukuri, A. Rakshit, et al. “Cloud security issues”. In: *2009 IEEE International Conference on Services Computing*. IEEE. 2009, pp. 517–520 (cit. on p. 25).

[KW14] S. Kolb, G. Wirtz. “Towards application portability in platform as a service”. In: *2014 IEEE 8th International Symposium on Service Oriented System Engineering*. IEEE. 2014, pp. 218–229 (cit. on p. 30).

[LF03] A. Linden, J. Fenn. “Understanding Gartner’s hype cycles”. In: *Strategic Analysis Report Nº R-20-1971. Gartner, Inc* (2003) (cit. on p. 88).

[LF15] J. Lewis, M. Fowler. “Microservices”. In: *http://martinfowler.com/articles/microservices.html* (2015) (cit. on pp. 15, 18).

[Mar83] J. Martin. *Managing the data base environment*. Prentice Hall PTR, 1983 (cit. on p. 67).

[MG11] P. Mell, T. Grance, et al. “The NIST definition of cloud computing”. In: (2011) (cit. on pp. 15, 19, 20).

[Mor16] K. Morris. *Infrastructure as code: managing servers in the cloud*. O’Reilly Media, Inc., 2016 (cit. on pp. 26, 28, 41).

[MR11] M. Masse, A. Rest. “Design Rulebook”. In: *O’Reilly* 3 (2011), pp. 1–61 (cit. on p. 69).

[NS14] D. Namiot, M. Sneps-Sneppe. “On micro-services architecture”. In: *International Journal of Open Information Technologies* 2.9 (2014), pp. 24–27 (cit. on p. 17).

[Nyg18] M. T. Nygard. *Release it!: design and deploy production-ready software*. Pragmatic Bookshelf, 2018 (cit. on p. 19).

[OF10] K. Oberle, M. Fisher. “ETSI CLOUD–initial standardization requirements for cloud services”. In: *International Workshop on Grid Economics and Business Models*. Springer. 2010, pp. 105–115 (cit. on p. 81).

[OST16] J. Opara-Martins, R. Sahandi, F. Tian. “Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective”. In: *Journal of Cloud Computing* 5.1 (2016), p. 4 (cit. on pp. 29, 30).

[PBSJ17] C. Pahl, A. Brogi, J. Soldani, P. Jamshidi. “Cloud container technologies: a state-of-the-art review”. In: *IEEE Transactions on Cloud Computing* (2017) (cit. on pp. 21, 27).

[Pet11] D. Petcu. “Portability and interoperability between clouds: challenges and case study”. In: *European Conference on a Service-Based Internet*. Springer. 2011, pp. 62–74 (cit. on p. 30).

[PI13] T. Perroud, R. Inversini. “Enterprise architecture patterns”. In: *Practical Solutions for Recurring ITYArchitecture Problems* (2013) (cit. on p. 17).

[PMPC13] D. Petcu, G. Macariu, S. Panica, C. Crăciun. “Portable cloud applications—from theory to practice”. In: *Future Generation Computer Systems* 29.6 (2013), pp. 1417–1430 (cit. on p. 81).

[RF18] M. Roberts, M. Fowler. “Serverless Architectures”. In: *https://martinfowler.com/articles/serverless.html* (2018) (cit. on pp. 15, 22).

[Sab11] F. Sabahi. “Virtualization-level security in cloud computing”. In: *2011 IEEE 3rd International Conference on Communication Software and Networks*. IEEE. 2011, pp. 250–254 (cit. on p. 21).

[Sch11] R. Schillinger. *Semantic service oriented architectures in research and practice*. Vol. 73. BoD–Books on Demand, 2011 (cit. on pp. 15, 17).

[SMS18] P. Seda, P. Masek, J. Sedova, M. Seda, J. Krejci, J. Hosek. “Efficient Architecture Design for Software as a Service in Cloud Environments”. In: *2018 10th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT)*. IEEE. 2018, pp. 1–6 (cit. on p. 69).

[Sti18] M. Stigler. “An Agnostic Approach”. In: *Beginning Serverless Computing*. Springer, 2018, pp. 175–195 (cit. on pp. 77, 78, 87).

[TK01] S. Tong, D. Koller. “Support vector machine active learning with applications to text classification”. In: *Journal of machine learning research* 2.Nov (2001), pp. 45–66 (cit. on p. 57).

[TL11] A. Tian, M. Lease. “Active learning to maximize accuracy vs. effort in interactive information retrieval”. In: *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*. ACM. 2011, pp. 145–154 (cit. on p. 57).

[TL95] T. Tannenbaum, M. Litzkow. “The condor distributed processing system”. In: *Dr Dobb’s Journal-Software Tools for the Professional Programmer* 20.2 (1995), pp. 40–49 (cit. on p. 83).

[TPA14] C. Teixeira, J. S. Pinto, R. Azevedo, T. Batista, A. Monteiro. “The building blocks of a PaaS”. In: *Journal of Network and Systems Management* 22.1 (2014), pp. 75–99 (cit. on p. 21).

[UHM11] D. Uckelmann, M. Harrison, F. Michahelles. “An architectural approach towards the future internet of things”. In: *Architecting the internet of things*. Springer, 2011, pp. 1–24 (cit. on p. 19).

[Ven19] K. Venkateswar. “Using Amazon SageMaker to Operationalize Machine Learning”. In: (2019) (cit. on p. 88).

[VFG11] A. Van Hoorn, S. Frey, W. Goerigk, W. Hasselbring, H. Knoche, S. Köster, H. Krause, M. Porembski, T. Stahl, M. Steinkamp, et al. “DynaMod project: Dynamic analysis for model-driven software modernization”. In: (2011) (cit. on p. 81).

[VGC15] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, S. Gil. “Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud”. In: *2015 10th Computing Colombian Conference (10CCC)*. IEEE. 2015, pp. 583–590 (cit. on pp. 15, 17, 19).

[Vog09] W. Vogels. “Eventually consistent”. In: *Communications of the ACM* 52.1 (2009), pp. 40–44 (cit. on p. 82).

[VWH12] A. Van Hoorn, J. Waller, W. Hasselbring. “Kieker: A framework for application performance monitoring and dynamic software analysis”. In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. ACM. 2012, pp. 247–248 (cit. on p. 81).

[WBF19] M. Wurster, U. Breitenbücher, M. Falkenthal, C. Krieger, F. Leymann, K. Saatkamp, J. Soldani. “The Essential Deployment Metamodel: A Systematic Review of Deployment Automation Technologies”. In: *arXiv preprint arXiv:1905.07314* (2019) (cit. on pp. 59, 63, 67, 84).

[WBFL17] M. Wurster, U. Breitenbücher, M. Falkenthal, F. Leymann. “Developing, deploying, and operating twelve-factor applications with TOSCA”. In: *Proceedings of the 19th International Conference on Information Integration and Web-based Applications & Services*. ACM. 2017, pp. 519–525 (cit. on p. 22).

[WBK18] M. Wurster, U. Breitenbücher, K. Képes, F. Leymann, V. Yussupov. “Modeling and Automated Deployment of Serverless Applications Using TOSCA”. In: *2018 IEEE 11th Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE. 2018, pp. 73–80 (cit. on p. 22).

[Wil09] J. L. Williams. “An implementors perspective on interoperable cloud APIs. Cloud interoperability roadmaps sessions”. In: *OMG Technical Meeting*. 2009, pp. 44–51 (cit. on p. 81).

[WLZ18] L. Wang, M. Li, Y. Zhang, T. Ristenpart, M. Swift. “Peeking behind the curtains of serverless platforms”. In: *2018 USENIX Annual Technical Conference (USENIX ATC’18)*. 2018, pp. 133–146 (cit. on p. 27).

[XWQ16] Z. Xiao, I. Wijegunaratne, X. Qiang. “Reflections on SOA and Microservices”. In: *2016 4th International Conference on Enterprise Systems (ES)*. IEEE. 2016, pp. 60–67 (cit. on p. 18).

[YBD08] L. Youseff, M. Butrico, D. Da Silva. “Toward a unified ontology of cloud computing”. In: *2008 Grid Computing Environments Workshop*. IEEE. 2008, pp. 1–10 (cit. on pp. 32, 82, 83).

[YN17] P. Yin, G. Neubig. “A Syntactic Neural Model for General-Purpose Code Generation”. In: *CoRR* abs/1704.01696 (2017). arXiv: 1704.01696. URL: http://arxiv.org/abs/1704.01696 (cit. on p. 60).

[ZM95] R. Zahavi, T. J. Mowbray. *The essential CORBA: systems integration using distributed objects*. Wiley, 1995 (cit. on p. 18).

Miscellaneous

[Adz19] Adzic, Gojko. *Claudia.js: Serverless JavaScript, the easy way*. 2019. URL: https://claudiajs.com/ (cit. on p. 27).

[AL19] E. Alliaume, B. Le Roux. *Cold start / Warm start with AWS Lambda*. 2019. URL: https://blog.octo.com/en/cold-start-warm-start-with-aws-lambda/ (cit. on p. 25).

[Ama19a] Amazon Web Services. *Amazon Alexa*. 2019. URL: https://developer.amazon.com/de/alexa (cit. on p. 44).

[Ama19b] Amazon Web Services. *Amazon API Gateway: Create, maintain, and secure APIs at any scale*. 2019. URL: https://aws.amazon.com/api-gateway/ (cit. on p. 45).

[Ama19c] Amazon Web Services. *Amazon CloudWatch*. 2019. URL: https://aws.amazon.com/cloudwatch/ (cit. on p. 25).

[Ama19d] Amazon Web Services. *Amazon DocumentDB: Fast, scalable, highly available MongoDB-compatible database service*. 2019. URL: https://aws.amazon.com/de/documentdb/ (cit. on p. 47).

[Ama19e] Amazon Web Services. *Amazon DynamoDB: Fast and flexible NoSQL database service for any scale*. 2019. URL: https://aws.amazon.com/dynamodb/ (cit. on p. 47).

[Ama19f] Amazon Web Services. *Amazon Kinesis: Easily collect, process, and analyze video and data streams in real time*. 2019. URL: https://aws.amazon.com/kinesis/ (cit. on p. 44).

[Ama19g] Amazon Web Services. *Amazon MSK: Amazon Managed Streaming for Apache Kafka*. 2019. URL: https://aws.amazon.com/msk/ (cit. on p. 44).

[Ama19h] Amazon Web Services. *Amazon S3: Object storage built to store and retrieve any amount of data from anywhere*. 2019. URL: https://aws.amazon.com/s3/ (cit. on p. 46).

[Ama19i] Amazon Web Services. *AWS CloudFormation*. 2019. URL: https://aws.amazon.com/cloudformation/ (cit. on p. 26).

[Ama19j] Amazon Web Services. *AWS Config: Record and evaluate configurations of your AWS resources*. 2019. URL: https://aws.amazon.com/config/ (cit. on p. 44).

[Ama19k] Amazon Web Services. *AWS Lambda*. 2019. URL: https://aws.amazon.com/lambda/ (cit. on p. 24).

[Ama19l] Amazon Web Services. *AWS Serverless Application Model*. 2019. URL: https://aws.amazon.com/serverless/sam/ (cit. on p. 26).

[Ama19m] Amazon Web Services. *AWS Serverless Application Model (SAM)*. 2019. URL: https://github.com/awslabs/serverless-application-model (cit. on p. 45).

[Ama19n] Amazon Web Services. *Programming Model - AWS Lambda*. 2019. URL: https://docs.aws.amazon.com/lambda/latest/dg/programming-model-v2.html (cit. on p. 25).

[Ama19o] Amazon Web Services. *Serverless: Build and run applications without thinking about servers*. 2019. URL: https://aws.amazon.com/serverless/ (cit. on p. 29).

[Apa19a] Apache Software Foundation. *A step-by-step guide for deploying Apache OpenWhisk applications using Package Manifest files*. 2019. URL: https://github.com/apache/openwhisk-wskdeploy/blob/master/docs/programming_guide.md#wskdeploy-utility-by-example (cit. on p. 26).

[Apa19b] Apache Software Foundation. *Apache Cassandra*. 2019. URL: http://cassandra.apache.org/ (cit. on p. 47).

[Apa19c] Apache Software Foundation. *Apache CouchDB*. 2019. URL: http://couchdb.apache.org/ (cit. on p. 47).

[Apa19d] Apache Software Foundation. *Apache OpenWhisk*. 2019. URL: https://openwhisk.incubator.apache.org/ (cit. on p. 24).

[Apa19e] Apache Software Foundation. *Apache Spark: Lightning-fast unified analytics engine*. 2019. URL: https://spark.apache.org/ (cit. on p. 47).

[Apa19f] Apache Software Foundation. *API Gateway HTTP response*. 2019. URL: https://github.com/apache/openwhisk-wskdeploy/blob/master/docs/wskdeploy_apigateway_http.md#api-gateway-http-response (cit. on p. 45).

[BA18] F. Brazeal, C. Anderson. *Azure Functions wants to make it easy for developers to get started with serverless*. 2018. URL: https://read.acloud.guru/azure-functions-wants-to-make-it-easy-for-developers-to-get-started-with-serverless-896766af985f?gi=8aa1c1b9b58c (cit. on p. 24).

[Bit19] Bitnami Inc. *Kubeless: The Kubernetes Native Serverless Framework*. 2019. URL: https://kubeless.io/ (cit. on p. 27).

[Bro18] S. Brown. *The C4 model for software architecture*. 2018. URL: https://c4model.com/ (cit. on p. 65).

[Cha19] M. Chan. *Why you should embrace, not fear, cloud vendor lock-in*. 2019. URL: https://www.thorntech.com/2017/10/embrace-not-fear-cloud-vendor-lock-in/ (cit. on p. 76).

[Cho09] D. Chou. *Windows Azure Platform*. 2009. URL: https://www.slideshare.net/davidcchou/windows-azure-platform (cit. on p. 21).

[Clo19a] Cloud Native Computing Foundation. *CNCF Serverless Working Group*. 2019. URL: https://github.com/cncf/wg-serverless (cit. on p. 32).

[Clo19b] Cloud Native Computing Foundation. *etcd: A distributed, reliable key-value store for the most critical data of a distributed system*. 2019. URL: https://etcd.io/ (cit. on p. 47).

[Clo19c] CloudEvents.io. *CloudEvents: specification for describing event data in a common way*. 2019. URL: https://cloudevents.io/ (cit. on pp. 32, 40, 56).

[CNC18] CNCF. *CNCF-WG Serverless Whitepaper v1.0*. 2018. URL: https://github.com/cncf/wg-serverless/blob/master/whitepapers/serverless-overview/cncf_serverless_whitepaper_v1.0.pdf (cit. on pp. 21, 23, 24, 44, 45).

[DMT19] DMTF. *Open Virtualization Format*. 2019. URL: https://www.dmtf.org/standards/ovf (cit. on p. 33).

[Doc19] Docker Inc. *Docker: Enterprise Container Platform for High-Velocity Innovation*. 2019. URL: https://www.docker.com/ (cit. on p. 21).

[Eic19] J. Eickmeyer. *Embrace Lock-in in the Age of Cloud*. 2019. URL: https://scratchpad.blog/embrace-lock-in-in-the-age-of-cloud/ (cit. on pp. 31, 76).

[Ell17] A. Ellis. *OpenFaaS: Serverless Functions, Made Simple*. 2017. URL: https://www.openfaas.com/ (cit. on p. 27).

[Eva19] B. Evans. *The World's Top 5 Cloud-Computing Suppliers*. 2019. URL: https://cloudwars.co/worlds-top-5-cloud-vendors-cloud-wars/ (cit. on p. 24).

[Fas19] FasterXML LLC. *Jackson Project*. 2019. URL: https://github.com/FasterXML/jackson (cit. on p. 68).

[Fil19] F,Ilyas. *Public Cloud Comparison*. 2019. URL: http://comparecloud.in/ (cit. on pp. 40, 56, 83).

[Gar19a] Gartner Inc. *Gartner Predicts the Future of AI Technologies*. 2019. URL: https://www.gartner.com/smarterwithgartner/gartner-predicts-the-future-of-ai-technologies/ (cit. on p. 88).

[Gar19b] Gartner Inc. *Top Trends in the Gartner Hype Cycle for Emerging Technologies 2017*. 2019. URL: https://www.gartner.com/smarterwithgartner/top-10-trends-impacting-infrastructure-and-operations-for-2019/ (cit. on p. 88).

[Gol19] B. Golden. *Don't avoid cloud vendor lock-in. Embrace it*. 2019. URL: https://techbeacon.com/enterprise-it/dont-avoid-cloud-vendor-lock-embrace-it (cit. on p. 76).

[Goo19] Google. *Angular*. 2019. URL: https://angular.io/ (cit. on p. 69).

[Gra19] Grafana Labs. *Grafana: The leading open source software for time series analytics*. 2019. URL: https://grafana.com/ (cit. on p. 25).

[gun19] gun.io. *ZAPPA: Serverless Python Web Services - Powered by AWS Lambda and API Gateway*. 2019. URL: https://www.zappa.io/ (cit. on p. 27).

[Has19] HashiCorp. *Terraform: Write, Plan, and Create Infrastructure as Code*. 2019. URL: https://www.terraform.io/ (cit. on p. 27).

[IBM19a] IBM. *About the IBM Cloud Object Storage S3 API*. 2019. URL: https://cloud.ibm.com/docs/services/cloud-object-storage?topic=cloud-object-storage-compatibility-api (cit. on p. 46).

[IBM19b] IBM. *Binding IBM Cloud services to Cloud Functions entities*. 2019. URL: https://cloud.ibm.com/docs/openwhisk?topic=cloud-functions-services (cit. on p. 57).

[IBM19c] IBM. *IBM Cloud Functions*. 2019. URL: https://www.ibm.com/cloud/functions (cit. on p. 24).

[IBM19d] IBM. *IBM Cloud Functions*. 2019. URL: https://cloud.ibm.com/docs/openwhisk?topic=cloud-functions-actions#actions_params (cit. on p. 25).

[IBM19e] IBM. *IBM Cloud Log Analysis*. 2019. URL: https://cloud.ibm.com/docs/services/CloudLogAnalysis (cit. on p. 25).

[IBM19f] IBM. *IBM Cloudant*. 2019. URL: https://www.ibm.com/cloud/cloudant (cit. on p. 47).

[IBM19g] IBM. *IBM Event Streams for IBM Cloud*. 2019. URL: https://www.ibm.com/cloud/event-streams-for-cloud (cit. on p. 49).

[Kna19] Knative Inc. *Knative: Kubernetes-based platform to build, deploy, and manage modern serverless workloads*. 2019. URL: https://knative.dev/ (cit. on p. 27).

[Meu95] R. Meunier. *The Pipes and Filters Architecture, volume 1 of "Pattern Languages of Program Design", chapter 22*. 1995 (cit. on p. 42).

[Mic19a] Microsoft Azure. *Azure CosmosDB: Globally distributed, multi-model database service for any scale*. 2019. URL: https://azure.microsoft.com/services/cosmos-db/ (cit. on p. 47).

[Mic19b] Microsoft Azure. *Azure Functions*. 2019. URL: https://azure.microsoft.com/services/functions/ (cit. on p. 24).

[Mic19c] Microsoft Azure. *Azure Functions Documentation*. 2019. URL: https://docs.microsoft.com/azure/azure-functions/functions-reference (cit. on p. 25).

[Mic19d] Microsoft Azure. *Azure Functions error handling*. 2019. URL: https://docs.microsoft.com/azure/azure-functions/functions-bindings-error-pages (cit. on p. 25).

[Mic19e] Microsoft Azure. *Azure Functions HTTP triggers and bindings*. 2019. URL: https://docs.microsoft.com/en-us/azure/azure-functions/functions-bindings-http-webhook (cit. on p. 45).

[Mic19f] Microsoft Azure. *Azure Resource Manager*. 2019. URL: https://azure.microsoft.com/features/resource-manager/ (cit. on p. 26).

[Mic19g] Microsoft Azure. *Overview of Microsoft Graph*. 2019. URL: https://docs.microsoft.com/graph/overview (cit. on p. 44).

[Mic19h] Microsoft Azure. *Work with Azure Functions Core Tools*. 2019. URL: https://docs.microsoft.com/azure/azure-functions/functions-run-local (cit. on p. 26).

[Mic19i] Microsoft Azure. *Zip deployment for Azure Functions*. 2019. URL: https://github.com/MicrosoftDocs/azure-docs/blob/master/articles/azure-functions/deployment-zip-push.md (cit. on pp. 39, 70, 71).

[Min19] MinIO Inc. *MinIO: Object Storage for AI*. 2019. URL: https://min.io/ (cit. on pp. 27, 46).

[Mon19a] MongoDB Inc. *MongoDB Atlas*. 2019. URL: https://www.mongodb.com/cloud/atlas (cit. on p. 27).

[Mon19b] MongoDB Inc. *MongoDB: The databse for modern applications.* 2019. URL: https://www.mongodb.com/ (cit. on p. 47).

[mOS19] mOSAIC. *mosaic: Open source api and platform for multiple clouds.* 2019. URL: http://www.mosaic-cloud.eu/ (cit. on p. 81).

[Obj16] Object Management Group Inc. *Architecture-Driven Modernization (ADM): Knowledge Discovery Meta-Model*. 2016. URL: https://www.omg.org/spec/KDM/ (cit. on pp. 37, 79, 81).

[Obj18] Object Management Group Inc. *"Architecture-Driven Modernization (ADM): Structured Metrics Meta-Model (SMM)*. 2018. URL: http://www.omg.org/spec/SMM/ (cit. on pp. 37, 79).

[Ora16] Oracle Corporation. *Fn Project: Open Source. Container-native. Serverless platform.* 2016. URL: https://fnproject.io/ (cit. on p. 27).

[Par19] Parr, Terence. *ANTLR: ANother Tool for Language Recognition*. 2019. URL: https://www.antlr.org/ (cit. on p. 81).

[Piv19a] Pivotal Software. *RabbitMQ*. 2019. URL: https://www.rabbitmq.com/ (cit. on p. 49).

[Piv19b] Pivotal Software Inc. *Riff is for functions*. 2019. URL: https://projectriff.io/ (cit. on p. 27).

[Piv19c] Pivotal Software Inc. *spring by Pivotal*. 2019. URL: https://spring.io/ (cit. on p. 69).

[Pla17] Platform9 Inc. *Fission: Open source, Kubernetes-native Serverless Framework*. 2017. URL: https://fission.io/ (cit. on p. 27).

[Rao19] L. Rao. *PiCloud launches serverless computing platform to the public*. 2019. URL: https://techcrunch.com/2010/07/19/picloud-launches-serverless-computing-platform-to-the-public/ (cit. on p. 24).

[Red19] Redis Labs. *Redis*. 2019. URL: https://redis.io/ (cit. on p. 49).

[Ser19] Serverless Inc. *Serverless: Build apps with radically less overhead and cost*. 2019. URL: https://serverless.com/ (cit. on pp. 29, 73).

[Som19] A. Somov. *snakeyaml*. 2019. URL: https://bitbucket.org/asomov/snakeyaml/src/default/ (cit. on p. 68).

[Str19] Strumenta Consulting. *JavaParser: For processing Java code*. 2019. URL: https://javaparser.org/ (cit. on p. 72).

[The19] The Linux Foundation. *Kubernetes: Production-Grade Container Orchestration*. 2019. URL: https://kubernetes.io/ (cit. on p. 21).

[Thö16] M. Thömmes. *Uncovering the magic: How serverless platforms really work!* 2016. URL: https://medium.com/openwhisk/uncovering-the-magic-how-serverless-platforms-really-work-3cb127b05f71 (cit. on p. 25).

[Wig17] A. Wiggins. *The Twelve-Factor App*. 2017. URL: https://12factor.net/ (cit. on p. 22).

All links were last followed on October 18, 2019.

A Domain-specific Deployment Models

Listing A.1 Exemplary deployment model of a Lambda-hosted application in Serverless Framework' DSL

```
service:
  name: broadcast-function
provider:
  name: aws
  runtime: nodejs8.10
  iamRoleStatements:
    - Effect: Allow
      Action:
        - SNSPublishMessagePolicy
      Resource: "arn:aws:sns:..."
functions:
  BroadcastFunction:
    handler: src/handlers/broadcast/index.handler
    events:
      - http:
          path: /
          method: get
    environment:
      SNS_TOPIC_ARN: "arn:aws:sns:..."
resources:
  Resources:
    EmailTopic:
      Type: AWS::SNS::Topic
      Properties:
        Subscription:
          - Endpoint: cook.tim@apple.com
            Protocol: email
```

Listing A.2 Exemplary implementation template for a code pattern that retrieves an entity from an S3 bucket, modifies it and puts it to another bucket.

```
function pattern_aws_modifiedObjectStorageCopy() {
    async.waterfall([
        function download(next) {
            s3.getObject({
                    Bucket: "",
                    Key: ""
                },
                next);
            },
        function modify(response, next) {
            const data = function(response) {
                // add business logic here
                return response
            }
            next(null, data);
        },
        function upload(data, next) {
            s3.putObject({
                    Bucket: "",
                    Key: "",
                    Body: "",
                },
                next);
            }
        ], function (err) {...}
    );
}
```

Listing A.3 Example for AWS Lambda application model using a SAM template which is invoked by a HTTP request to a API Gateway and broadcasts emails over AWS SNS topics.

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Resources:
  BroadcastFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: index.handler
      Runtime: nodejs8.10
      MemorySize: 512
      Timeout: 5
      Events:
        BroadcastApi:
          Type: Api
          Properties:
            Path: /
            Method: GET
      Policies:
        - SNSPublishMessagePolicy:
            TopicName:
              Fn::GetAtt:
                - EmailTopic
                - TopicName
      Environment:
        Variables:
          SNS_TOPIC_ARN:
            Ref: EmailTopic
      CodeUri: s3://my-lambda-repository/084b55658168cf644d0923ea8c301843
  EmailTopic:
    Type: AWS::SNS::Topic
    Properties:
      Subscription:
        - Endpoint: cook.tim@apple.com
          Protocol: email
```

B Skywalker - User interface design and API specification

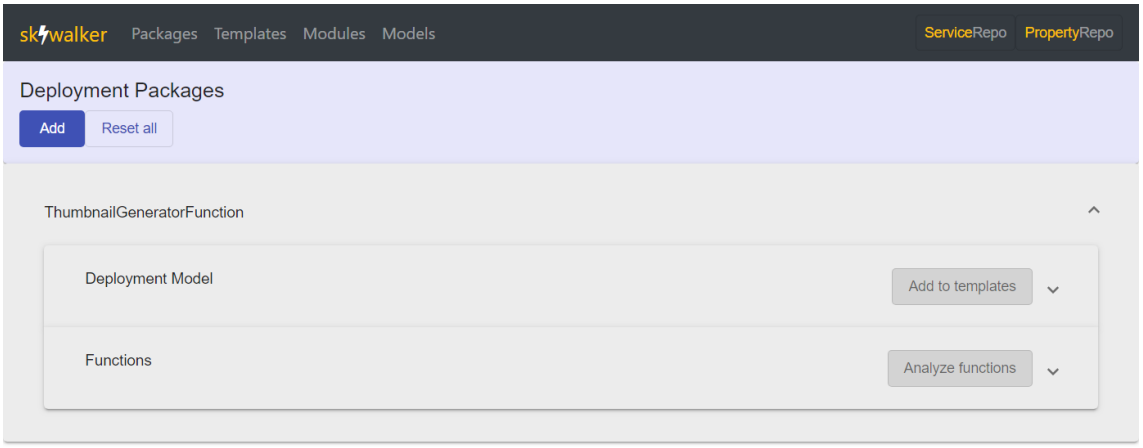


Figure B.1: View of the deployment package component of Skywalker Client

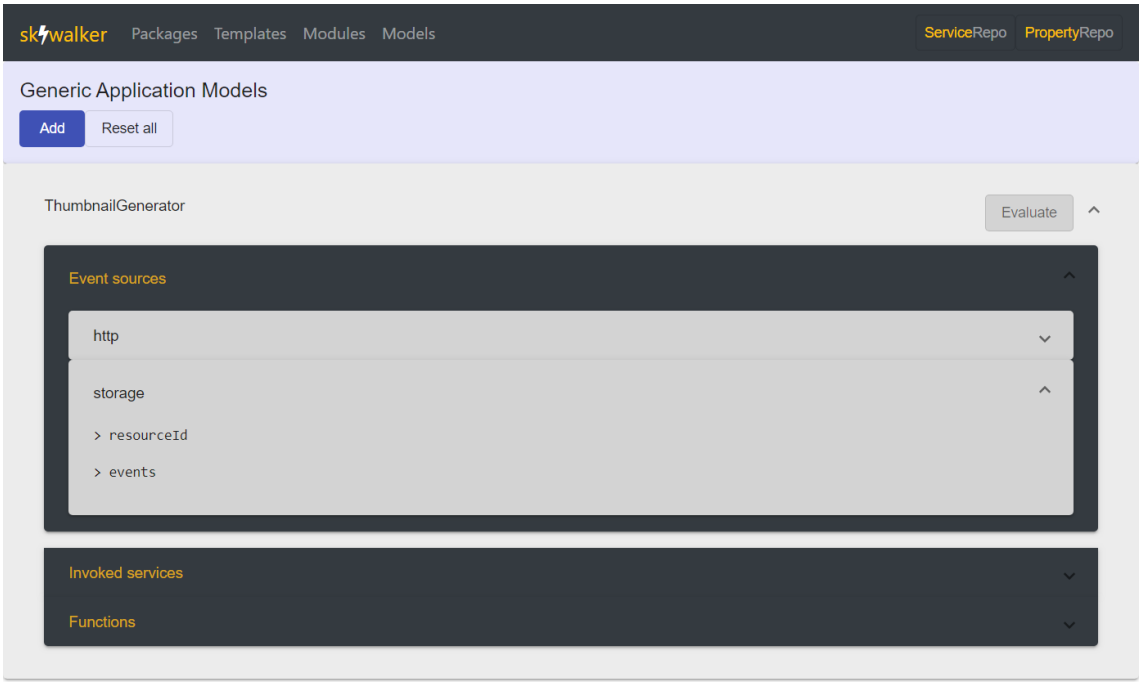


Figure B.2: View of the application model component of Skywalker Client

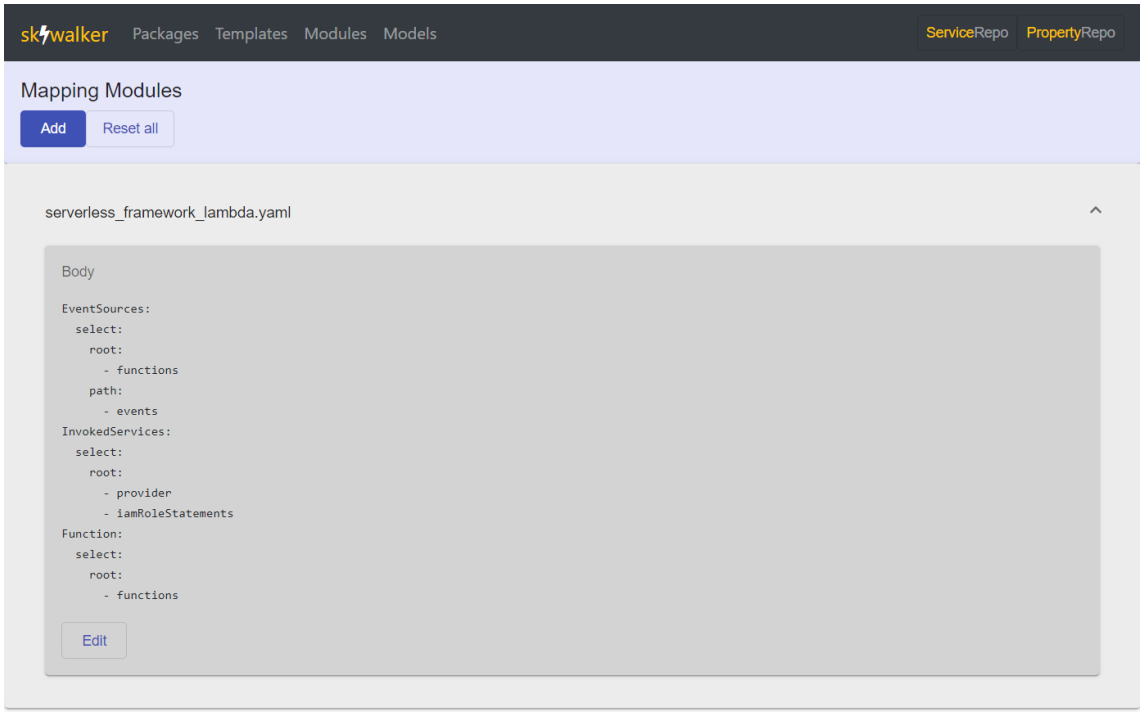


Figure B.3: View of the mapping module component of Skywalker Client

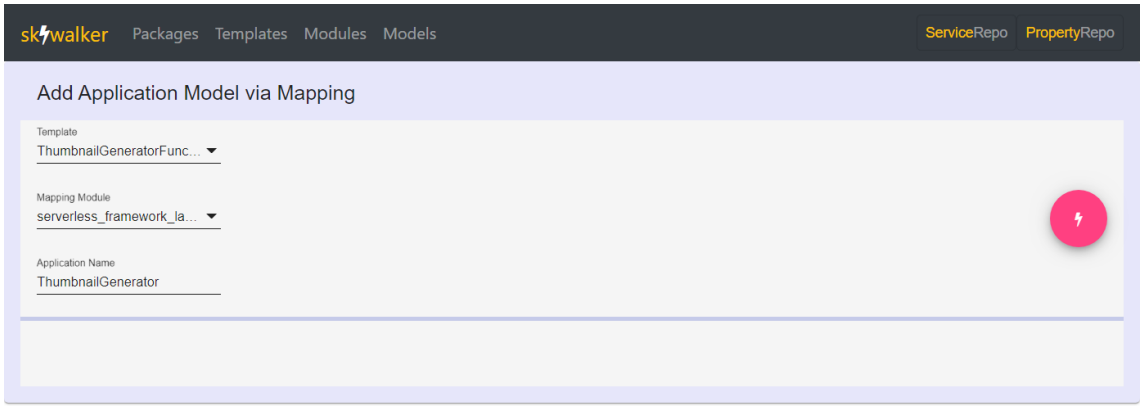


Figure B.4: View of the application model generation component of Skywalker Client

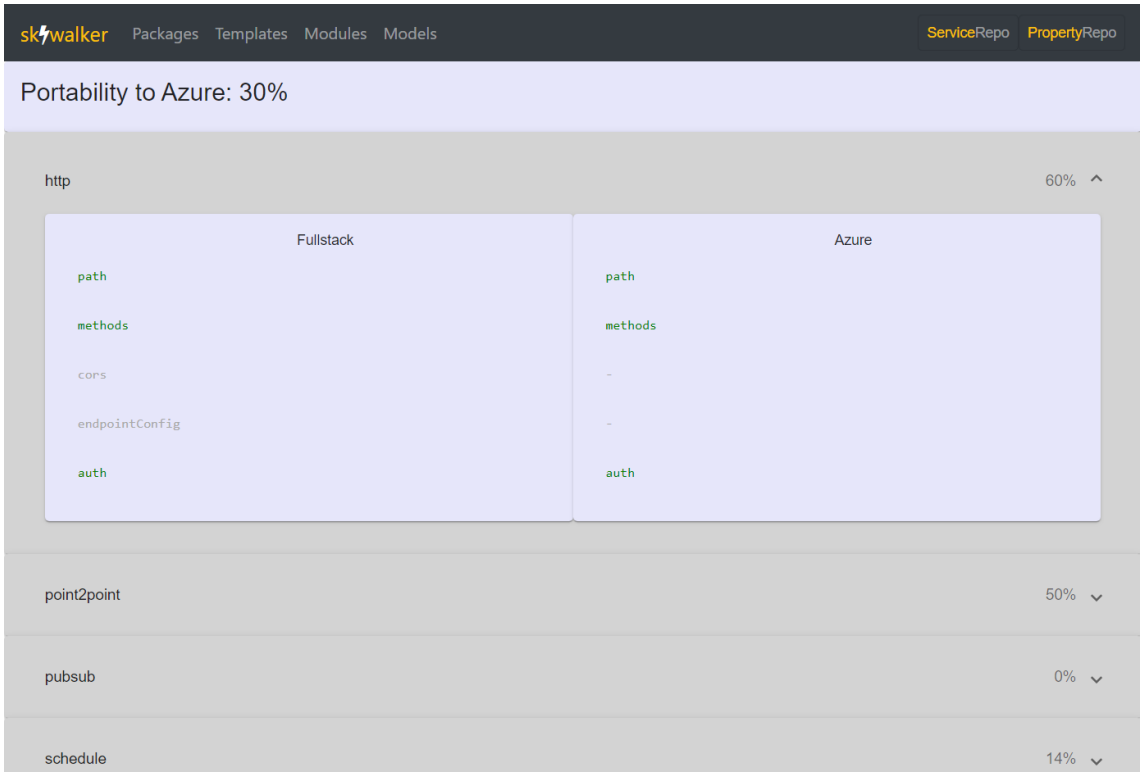


Figure B.5: View of the portability evaluation component of Skywalker Client for a complex application

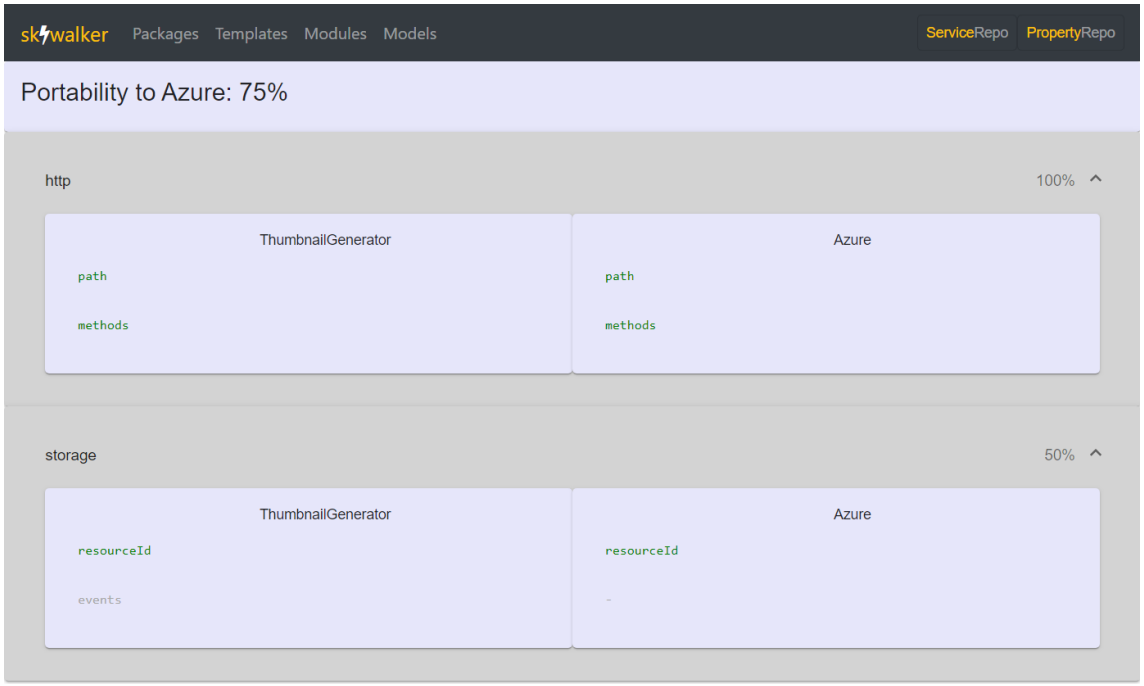


Figure B.6: View of the portability evaluation component of Skywalker Client for a simple storage-trigger application

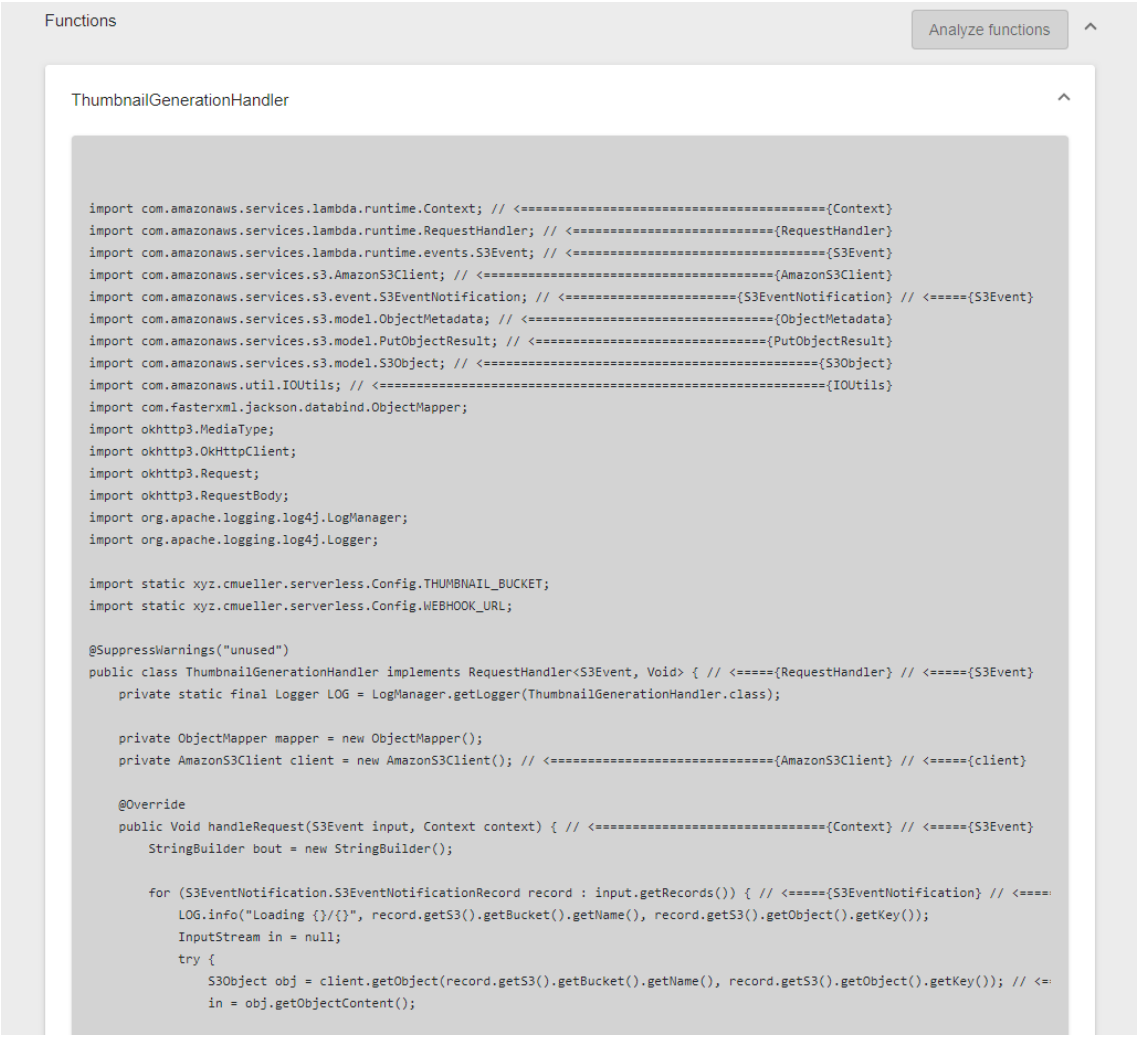


Figure B.7: View of an annotated function in the deployment package component of Skywalker Client

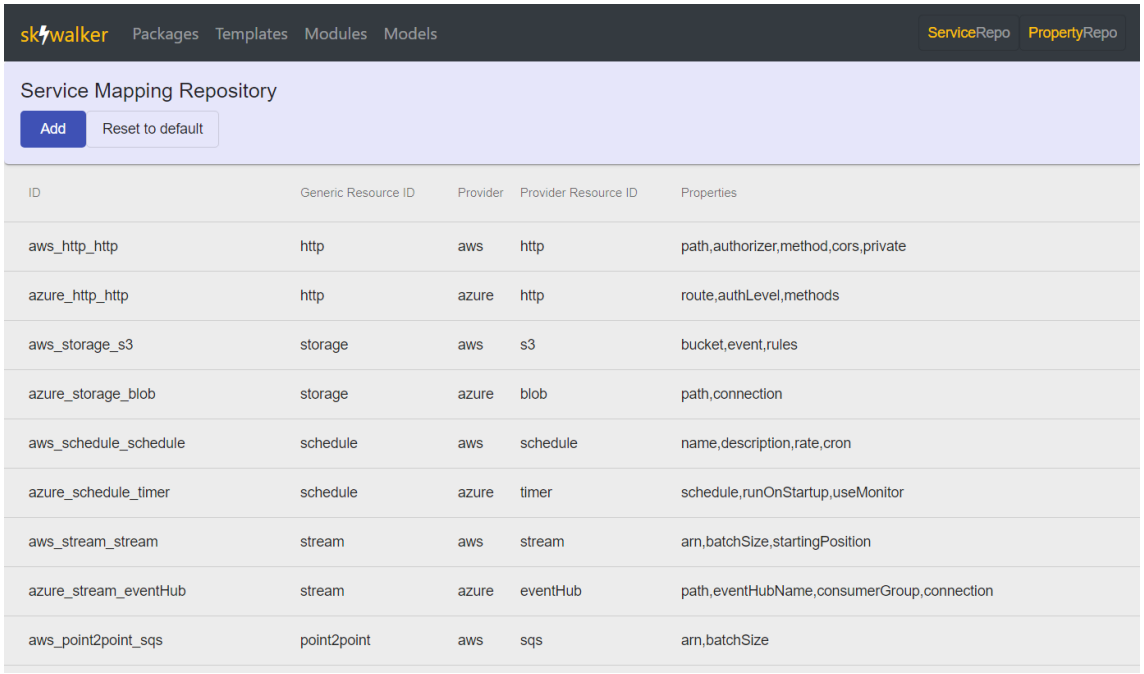


Figure B.8: View of the service mapping repository component of Skywalker Client

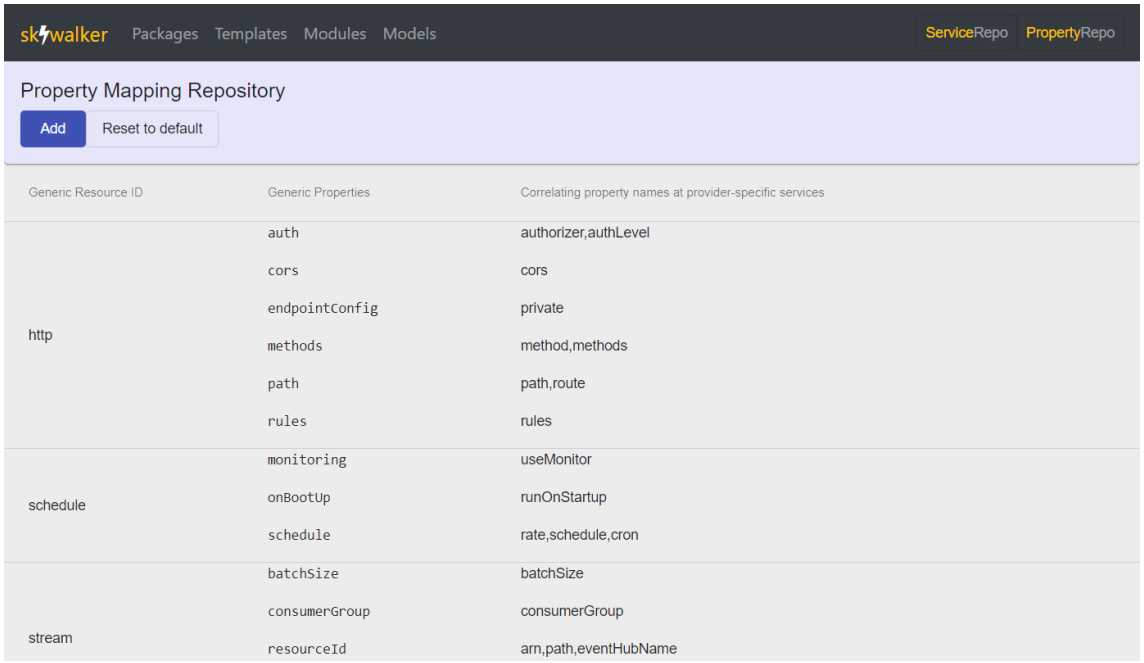


Figure B.9: View of the service property mapping repository component of Skywalker Client

Path	HTTP Method	Description
	Service Mapping Repository	
/services	GET	Returns the service mappings table.
/services/{serviceForm}	PUT	Adds a new entry in the service mappings table.
/services	DELETE	Resets the service map pings table to the default state
	Service Property Mapping Repository	
/properties	GET	Returns the service property mappings table.
/properties/{propertyForm}	PUT	Adds a new entry in the service property mappings table.
/properties	DELETE	Resets the service property mappings table to the default state

Table B.1: Overview of Skywalker's API specification for CRUD operations on the provider service knowledge bases.

Path	HTTP Method	Description
	Deployment Models	
/templates	GET	Returns a list of deployment models that the repository contains.
/templates/{template}	PUT	Adds a new deployment model to the repository or updates an existing model.
/templates/{id}	DELETE	Removes one or more deployment models from the repository.
	Mapping Modules	
/mappings	GET	Returns a list of mapping modules that the repository contains.
/mappings/{module}	PUT	Adds a new mapping module to the repository or updates an existing model.
/mappings/{id}	DELETE	Removes one or more mapping modules from the repository.
	Application Models	
/apps	GET	Returns a list of application models that the repository contains.
/apps/{config}	PUT	Adds an application model to the repository by generating it from the passed configuration or updates an existing model.
/apps/evaluation/{bundle}	POST	Evaluates the coverage score for an application model and a target platform wrt. services and their similarity. Gets a bundle passed which comprises application model id and target platform.
/apps/translation/{bundle}	POST	Translates the existing deployment model into a boilerplate template for deployment on the target platform.
/apps/{id}	DELETE	Removes one or more application models from the repository.
	Deployment Packages	
/packages	GET	Returns a list of deployment packages that the repository contains.
/packages/{package}	PUT	Adds or updates a deployment package to the repository.
/packages/crawl/{extractionData}	POST	Crawls deployment package directly from the deployed application's platform.
/packages/{functions}	POST	Passes the deployment package's functions to the Code Analysis Engine and updates it with the annotated functions in the repository.
/packages/{id}	DELETE	Removes one or more application models from the repository.

Table B.2: Overview of Skywalker Seed's service API specification.

Declaration

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

---

place, date, signature