

Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Masterarbeit

**Standards-based Modeling and
Generation of Platform-specific
Function-as-a-Service Deployment
Packages**

Tolunay Yüksel

Course of Study:	Informatik
Examiner:	Prof. Dr. Dr. h.c. Frank Leymann
Supervisor:	Vladimir Yussupov, M.Sc., Dr. rer. nat. Uwe Breitenbücher
Commenced:	December 1, 2021
Completed:	June 1, 2022

Abstract

Due to continuously rising popularity, cloud and serverless computing became important buzzwords, causing many companies to consider transferring their systems to cloud-native architectures. With it the novel cloud service offering Function-as-a-Service (FaaS) plays an integral role in creating serverless architectures with the help of functions used as building blocks. However, making yourself dependent on just one cloud provider can lead to vendor lock-in problems. For this reason, it is important to diversify cloud providers and make serverless applications portable. This process requires developers in-depth technical expertise across many different cloud platforms, therefore making it error-prone and very tedious. This thesis elaborated a concept, which enables developers, lacking of this specific know-how, to model provider-agnostic workflow models of FaaS functions based on BPMN, which are used to generate provider-specific deployment packages. Additionally, the prototype BPMN2FaaS was implemented based on this concept, which is able to generate FaaS functions in the programming language Python and deployment packages supported by the platforms AWS Lambda and Microsoft Azure Functions.

Kurzfassung

Aufgrund stetig steigender Popularität wurden aus den Begriffen Cloud und Serverless Computing wichtige Schlagwörter, die viele Firmen dazu veranlassen, ihre Systeme in Cloud-native Architekturen zu überführen. Dabei spielt das neuartige Cloud Service Angebot Function-as-a-Service (FaaS) eine wesentliche Rolle, wodurch serverlose Architekturen mithilfe von Funktionen als Bausteine entstehen können. Sich dabei jedoch nur von einem Cloud-Anbieter abhängig zu machen kann zu Problemen des Vendor Lock-ins führen. Aus diesem Grund ist es wichtig die Cloud-Anbieter zu diversifizieren und Portabilität für serverlose Applikationen zu gewährleisten. Dieser Prozess erfordert Entwicklern jedoch technische Expertise über viele verschiedene Cloud-Plattformen hinweg und ist somit fehleranfällig und sehr mühsam. In dieser Arbeit wurde ein Konzept erarbeitet, wodurch Entwicklern, ohne dieses spezifische Know-how, ermöglicht wird, Cloud-Anbieter agnostische Arbeitsabläufe von FaaS Funktionen basierend auf BPMN zu modellieren, welche zum Generieren von plattform-spezifischen Bereitstellungspaketen verwendet werden. Basierend auf diesem Konzept wurde zusätzlich der Prototyp BPMN2FaaS implementiert, welches FaaS Funktionen in der Programmiersprache Python und von den Plattformen AWS Lambda und Microsoft Azure Functions unterstützte Bereitstellungspaketen generiert.

Contents

1	Introduction	15
2	Background	17
2.1	Cloud Computing	17
2.2	Function-as-a-Service (FaaS)	19
2.3	BPMN	21
3	Concept	27
3.1	Step 1: Develop Functional Business Code Modules	29
3.2	Step 2: Create Generic Workflow Model	31
3.3	Step 3: Extend Model with Properties & Integrate Business Functions	34
3.4	Step 4: Add Provider-specific Service Endpoints	41
3.5	Step 5: Generate Provider-specific FaaS Function	43
3.6	Step 6: Build Deployment Packages	45
4	Implementation	47
4.1	Architecture	47
4.2	Technologies	47
4.3	Modeler	49
4.4	Generator (Core)	53
4.5	Plugins	54
5	Related Work	63
5.1	Serverless Portability	63
5.2	Graphical Workflow Modeling and Model-driven Code Generation	65
6	Conclusion and Outlook	67
6.1	Limitations	67
6.2	Future Work	67
	Bibliography	71

List of Figures

2.1	Distribution of management in SPI model compared to traditional IT [Sou10]	19
2.2	Traditional Client-Server architecture	20
2.3	Serverless architecture	21
2.4	Example Business Process Diagram	22
2.5	Core Flow Object Elements	22
2.6	BPMN Event types (from [Cam21])	23
2.7	BPMN Activity types	24
2.8	BPMN Activities with markers for execution behaviour	24
2.9	BPMN Pool with two Lanes	25
2.10	BPMN Artifacts	26
2.11	BPMN Connecting Objects	26
3.1	Method for standards-based modeling and generation of platform-specific FaaS deployment packages	28
3.2	Flowchart for handling orders in a online shop	28
3.3	Example on how to split/join workflows using Gateways	33
3.4	BPMN diagram modeling the online shop process	33
3.5	Procedure of creating provider-agnostic service calls	36
3.6	Example schema of a document store using DynamoDB (from [BS17])	36
3.7	UML diagram of azure.functions.QueueMessage class [Mic22d]	39
3.8	BPMN diagram enhanced with properties	42
3.9	Platform-specific FaaS code example (Python)	44
4.1	Architecture of BPMN2FaaS	48
4.2	Communication between Modeler and Generator	48
4.3	Creating a new diagram in BPMN2FaaS	50
4.4	Properties Panel for Start Events	52
4.5	UML class diagram of FaaSFunction.	55
4.6	UML class diagram of BPMN classes used by the plugins.	57
5.1	SEAPORT method [YBKL20]	64

List of Tables

3.1	Terminology	27
3.2	General BPMN Element Properties	34
3.3	Start Event Properties	35
3.4	Task Properties (with Data Store)	37
3.5	XOR/Exclusive Gateway Properties	37

List of Listings

3.1	Platform-agnostic FaaS code example (Python)	29
3.2	Code-smell snippet of a functional business code module (Python)	30
3.3	Correct example of a functional business code module (Python)	30
3.4	Example Amazon SQS notification event [Ama22d]	38
3.5	CloudEvent schema of an object storage event	40
3.6	Resolution of batched events into single events (Python)	41
3.7	Service call example for Azure Blob Storage (Python)	42
3.8	Example resource descriptor	43
3.9	Deployment Package Structure for AWS Lambda	45
3.10	Deployment Package Structure for Azure Functions	46
4.1	Input package structure for business code modules and dependencies	49
4.2	Meta-Model definition for Start Events using <i>moddle</i>	51
4.3	Accessing <i>moddle</i> properties and binding it with HTML elements (JavaScript)	51
4.4	XML representation of an Start Event with its properties	51
4.5	Extraction of BPMN diagram represented as XML	55
4.6	FaaS Function template for AWS (Python + Jinja)	58
4.7	FaaS Function template for Azure (Python + Jinja)	58
4.8	Template for introducing Indentation (Jinja)	58
4.9	Business Function template (Jinja)	59
4.10	for-loop template (Jinja)	59
4.11	send_message template for AWS (Jinja)	59
4.12	send_message template for Azure (Jinja)	59
4.13	Template for Timer Trigger Events in AWS (Jinja)	60
4.14	Template for End Events (Jinja)	60
4.15	Template for if-statements (Jinja)	61
4.16	Template for simulated switch-statements (Jinja)	61

Acronyms

API	Application Programming Interface.	20
ARN	Amazon Resource Name.	43
AWS	Amazon Web Services.	15
BPD	Business Process Diagram.	21
BPMN	Business Process Model and Notation.	15
CLI	Command Line Interface.	63
CNCF	Cloud Native Computing Foundation.	39
CRUD	Created/Read/Update/Delete.	35
DAG	Directed Acyclic Graph.	65
DOM	Document Object Model.	52
DSL	Domain-specific Language.	65
EC2	Elastic Compute Cloud.	66
FaaS	Function-as-a-Service.	15
FIFO	First in, first out.	34
GCF	Google Cloud Functions.	64
GUI	Graphical User Interface.	27
HTML	Hypertext Markup Language.	50
HTTP	Hypertext Transfer Protocol.	20
IaaS	Infrastructure-as-a-Service.	18
JSON	JavaScript Object Notation.	38
NIST	National Institute of Standards and Technology.	17
PaaS	Platform-as-a-Service.	18
PubSub	Publish/Subscribe.	34
REST	Representational State Transfer.	66
S3	Simple Storage Service.	20
SaaS	Software-as-a-Service.	18

SDK Software Development Kit. 15

SEAPORT SErverless Applications PORTability assessmentT. 63

SQS Simple Queue Service. 38

SVG Scalable Vector Graphic. 47

UML Unified Modeling Language. 38

URI Uniform Resource Identifier. 27

URL Uniform Resource Locator. 59

XML Extensible Markup Language. 39

1 Introduction

Nowadays cloud computing [MG11] gains more and more popularity, due to giving up responsibilities for infrastructure and scaling configuration management from developers to cloud providers, leading to its novel paradigm serverless computing also getting increasing attention. With serverless computing cloud providers take over the most responsibilities compared to other cloud offerings, aiming to help developers solely focus on developing their applications. The biggest part in developing serverless applications is Function-as-a-Service (FaaS), which allows developers to divide their monolithic applications to multiple smaller function components and use them as building blocks to design composed architectures. Cloud platforms like Amazon Web Services (AWS) Lambda or Microsoft Azure Functions offer developers to deploy their functions and bind them with numerous other service of their repertoire, e.g. storage or messaging services. This enables FaaS functions to interact with other services or get invoked by reacting to different types of events, e.g. react to uploaded files or respond to new messages.

However, creating such FaaS functions requires much technical expertise not only about the target FaaS platform, but also about every other service it is interacting with. Moreover, developers with in-depth knowledge might still be affected by the disadvantages coming from vendor lock-in [OST14]. Issues with vendor lock-in can be compromised by diversifying cloud providers and deploying on different FaaS platforms. Unfortunately, most FaaS functions are not portable to different platforms, because e.g. using provider-specific Software Development Kits (SDKs). Therefore, developers are forced to deal with gaining additional expertise across multiple clouds. This process takes additional time and costs and might lead to having a serverless application, which becomes error-prone. Today's scientist already research on solutions, trying to tone down vendor lock-in issues by developing concepts, which make cloud applications more portable. Although enabling FaaS portability is a crucial step to make whole serverless application fully portable, there is a significant research gap when it comes to portability on the level of FaaS functions.

This thesis presents a concept and its first prototype *BPMN2FaaS*, which helps developers without provider-specific knowledge across multiple cloud providers, mainly focusing on AWS Lambda or Microsoft Azure Functions, to generate FaaS deployment packages supported by the chosen target platform. First, Business Process Model and Notation (BPMN) [Whi04] with some extended properties is used to create a canonical, provider-agnostic representation of FaaS functions. This way developers have to gain less knowledge about multiple cloud providers. Afterwards, this model is used to generate a FaaS function and store it in a deployment package accordingly to the requirements of the target platform.

The structure of this work proceeds as follows: In Chapter 2 fundamental concepts of cloud computing, FaaS and BPMN are explained to provide background information needed throughout this thesis. Additionally, a set of important terminologies are introduced for this work to avoid misunderstandings. Chapter 3 presents a multi-step method realizing the concept of generating platform-specific FaaS deployment packages out of BPMN workflow models. Each step is described

in detail and accompanied by an example. Screenshots of the prototype BPMN2FaaS together with architectural and technical details about its implementation are provided in Chapter 4. Chapter 5 presents related work in the field of serverless portability and model-driven code generation. The last chapter concludes this work by giving a brief summary of the outcomes and providing an outlook for future work.

2 Background

2.1 Cloud Computing

In recent years, a new computing paradigm called *Cloud Computing* emerged. Its popularity reached such heights, which led more and more companies to adapt their systems to this new technology. Cloud Computing is a computation model, which provides on-demand availability of a pool of virtualized computer resources like storage, messaging systems, computing power and many more. Additionally, developers are aided with several tools in *managing, provisioning, monitoring, versioning* and creating highly scalable (*elastic*) and distributed applications in an automated manner. Giving up such responsibilities relieves developers and enables them to focus more on their product itself. Moreover, developers can save costs by avoiding buying new servers, licences and paying for new employees whenever they want to scale out and having to sell them afterwards when scaling back in, because the payment model of clouds usually relies on pay-as-you-go and is therefore more flexible. [AFG+10]

To this day there are many definitions of Cloud Computing existing, but the definition of the National Institute of Standards and Technology (NIST) by Peter Mell and Timothy Grance in September 2011 [MG11] is the most common one. The computation model in this definition is composed of five essential characteristics, three service models and four deployment models and goes as follows:

Definition 2.1.1 (Essential Characteristics of Cloud Computing [MG11])

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 data center). 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.*

Definition 2.1.2 (Service Models [MG11])

Software-as-a-Service (SaaS). *The capability provided to the consumer is 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. The consumer does 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.*

Platform-as-a-Service (PaaS). *The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment.*

Infrastructure-as-a-Service (IaaS). *The capability provided to the consumer is 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. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g., host firewalls).*

As a generic term the three service models defined in Definition 2.1.2 are also called SPI model [Cas18]. Figure 2.1 illustrates a comparison of the responsibility of management between the SPI model and traditional IT.

Definition 2.1.3 (Deployment Models [MG11])

Private cloud. *The cloud infrastructure is provisioned for exclusive use by a single organization comprising multiple consumers (e.g., business units). It may be owned, managed, and operated by the organization, a third party, or some combination of them, and it may exist on or off premises.*

Community cloud. *The cloud infrastructure is provisioned for exclusive use by a specific community of consumers from organizations that have shared concerns (e.g., mission, security requirements, policy, and compliance considerations). It may be owned, managed, and operated by one or more of the organizations in the community, a third party, or some combination of them, and it may exist on or off premises.*

Public cloud. *The cloud infrastructure is provisioned for open use by the general public. It may be owned, managed, and operated by a business, academic, or government organization, or some combination of them. It exists on the premises of the cloud provider.*

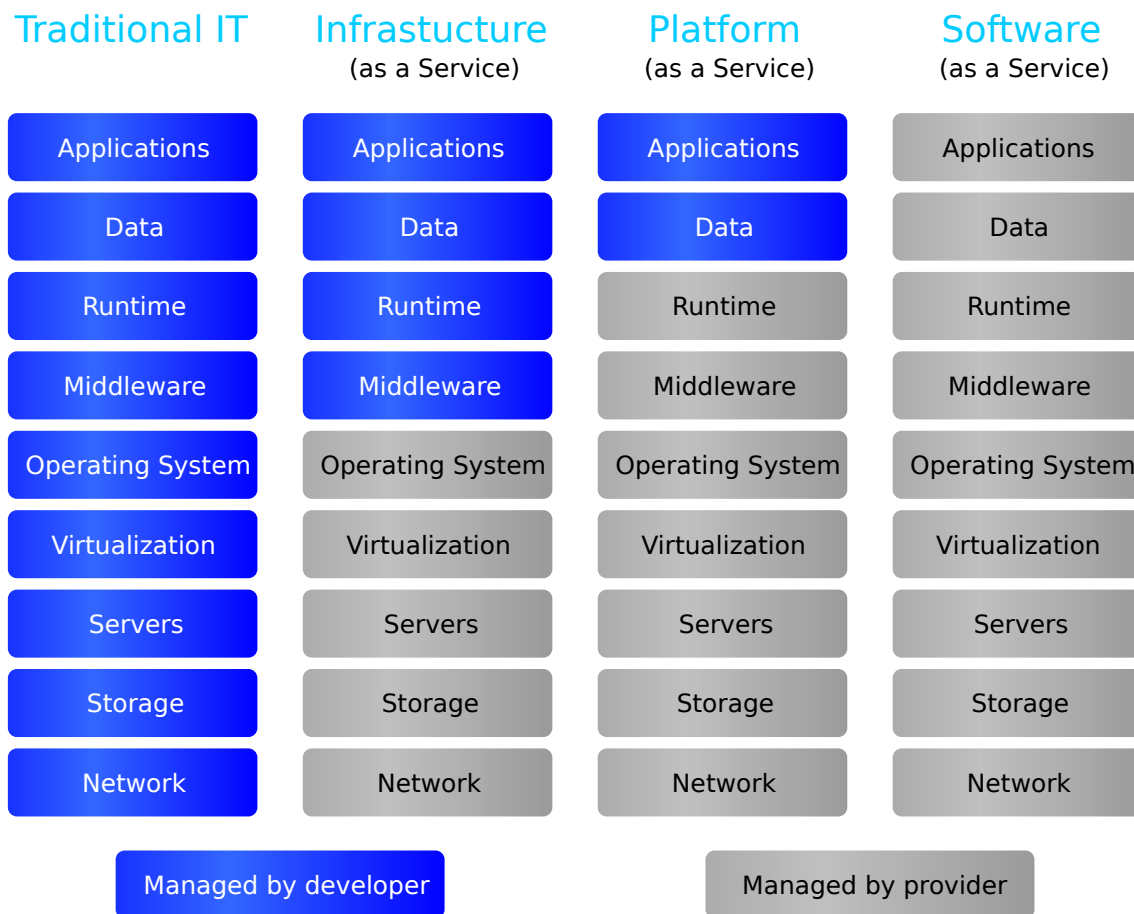


Figure 2.1: Distribution of management in SPI model compared to traditional IT [Sou10]

Hybrid cloud. *The cloud infrastructure is a composition of two or more distinct cloud infrastructures (private, community, or public) that remain unique entities, but are bound together by standardized or proprietary technology that enables data and application portability (e.g., cloud bursting for load balancing between clouds).*

2.2 Function-as-a-Service (FaaS)

Another popular and even newer computing paradigm offered by cloud providers is *serverless computing*. It allows developers to develop and deploy applications efficiently and minimizes responsibilities to manage any underlying infrastructures or platforms. With serverless computing developers are able to neglect all the overhead of monitoring, provisioning, scaling and managing the infrastructure by making the cloud service providers responsible for it. This way they can solely focus on their own business logic. [TEPN20]

As a part of serverless computing FaaS has been created to provide developers the ability to decompose and deploy traditional applications as serverless applications consisting of stateless function modules and defined triggers for executing them. Such functions are executed in response

to Hypertext Transfer Protocol (HTTP) requests or events occurring in cloud services of the FaaS platform [JCBG21]. The most popular FaaS platforms are AWS Lambda¹, Azure Functions², Google Cloud Functions³ and IBM Cloud Functions (based on Apache OpenWhisk)⁴.

Monitoring and handling those events can also be done with IaaS or PaaS by implementing (i) a monitoring component to detect events, (ii) a routing component to put events into the right queues for enabling asynchronous processing and (iii) a handler process to execute the business code on incoming events. However, this procedure can lead to potential downsides like

- the necessity of deploying whole software stacks (using IaaS),
- implementing functionality for monitoring events,
- the need to have the monitoring component to run all the time and
- handling varying workload and therefore scalability manually.

This drawbacks become even more painful, when a full stack needs to be deployed for only a limited number of lines of code for handling the event. Additionally, hosting and managing the three components drives the complexity and costs into unnecessary heights for a comparably small task. This is exactly where the cost model of the FaaS technology comes into play, which offers developers to save money, because the price is determined by the combination of *pay-per-use* and the duration of the execution time.

Figure 2.2 shows an architecture of a traditional client-server application. Front-end, back-end and database can be either managed manually or with the help of a cloud using the three service models defined in Definition 2.1.2 on page 18. Depending on the chosen service models, developers can save resources for managerial tasks. Moreover, scaling out each of those 3 components increases the management immensely.

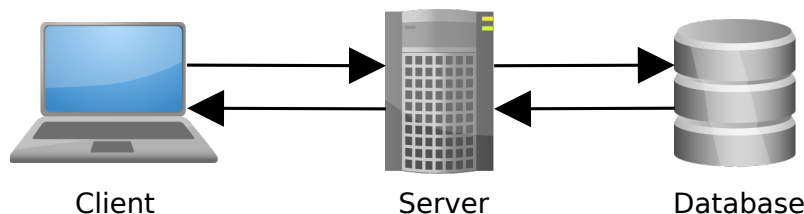


Figure 2.2: Traditional Client-Server architecture

In contrast to the traditional designing of a client-server architecture, Figure 2.3 demonstrates an exemplary serverless architecture deployed in AWS. A bucket managed by Amazon Simple Storage Service (S3) hosts the front-end code returned to the client. The API Gateway together with the deployed Lambda functions represent the back-end from Figure 2.2 in a decomposed and stateless way. With Amazon API Gateway it is possible to create fully managed Application Programming Interfaces (APIs), which can invoke the respective Lambda function depending on the called path. During their runtime Lambda functions can invoke further functions and interact with other cloud

¹<https://aws.amazon.com/lambda/>

²<https://azure.microsoft.com/services/functions/>

³<https://cloud.google.com/>

⁴<https://cloud.ibm.com/functions/>

services like the key-value database DynamoDB. In this architecture every component is managed by the cloud. Developers only need to set the architecture up once. After that the amount of management becomes minimal and they can solely focus on the development of their business code deployed in AWS Lambda.

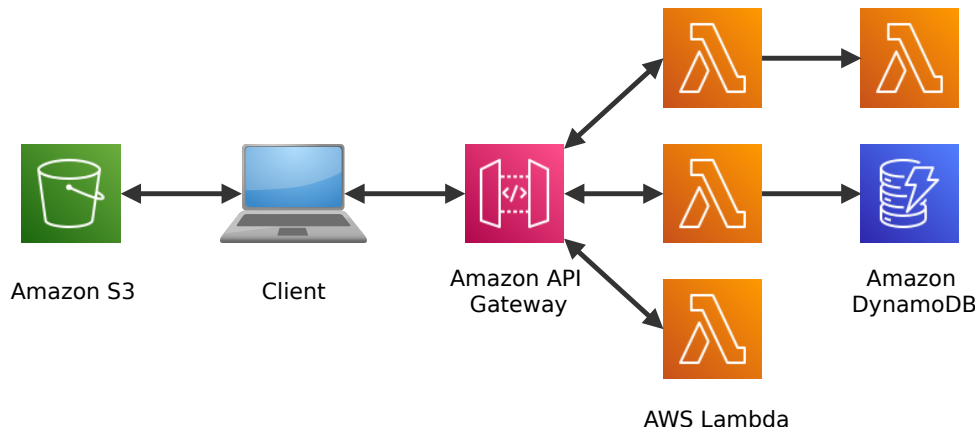


Figure 2.3: Serverless architecture

Fundamentally different than application hosting with IaaS or Platform-as-a-Service (PaaS) clouds, with serverless computing, applications decomposed by multiple FaaS functions can be deployed as code modules [LRC+18], also called *deployment packages*. The deployment model varies for each target platform and deployment automation tool used like AWS CloudFormation, Terraform⁵ or Serverless Framework⁶ [YBKL20]. Such models can be classified either as imperative, where a set of operations defines how the deployment process has to be executed, or as descriptive, where the targeted state of each component is defined.

2.3 BPMN

The BPMN specification was released to the public by IBM-employee Stephen A. White in May 2004. It quickly became the leading standard for creating Business Process Diagrams (BPDs). BPDs, often also called BPMN diagrams, use a flowcharting technique to create a graphical representation of business processes and workflow models. A set of graphical elements enable easy development and readability for users from any fields. [Whi04]

Figure 2.4 illustrates an example BPD, which models business processes invoked whenever customers order trips. The diagram is divided into 3 **Sub-processes** to demonstrate which **Tasks** are assigned to travelers, travel agencies and airlines and how those 3 instances interact with each other.

The following subsections demonstrate the various types of BPMN elements based on the definition described in the official article of BPMN in [Whi04]. The subsections are structured accordingly to the categorization of those elements.

⁵<https://www.terraform.io/>

⁶<https://www.serverless.com/>

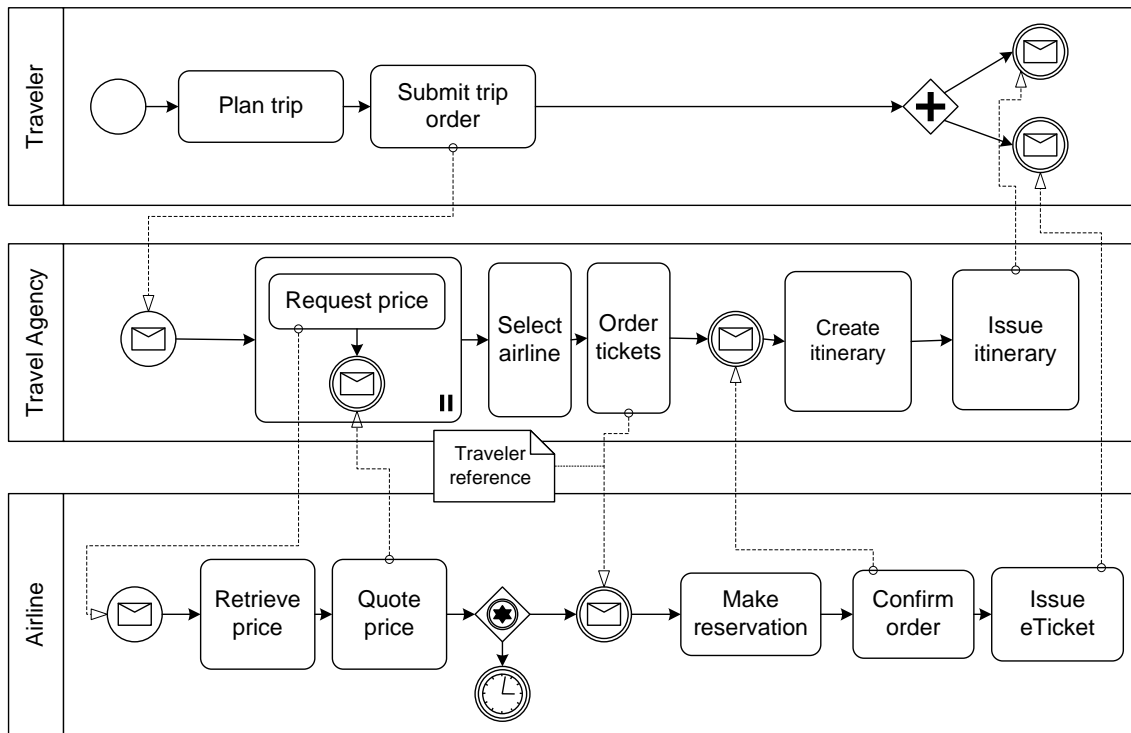


Figure 2.4: Example Business Process Diagram

2.3.1 Flow Objects

Events

Events model happenings, which can occur during a business process. The circles shown in Figure 2.5a represent the three classes of Events: **Start**, **Intermediate** and **End Events**. While **Start Events** always initiate processes, **End Events** indicate the completion of a process. **Intermediate Events** happen between **Start** and **End Events**. They can be either catching or throwing events. Moreover, they can be marked as interrupting or non-interrupting, declaring whether **Tasks** throwing this event will be interrupted or not during processing. To further specify what type of event is used, additional symbols can be used, which are listed in Figure 2.6. For example, **Timer Events** are used to indicate, that this Event is either occurring periodically or at a predefined time.

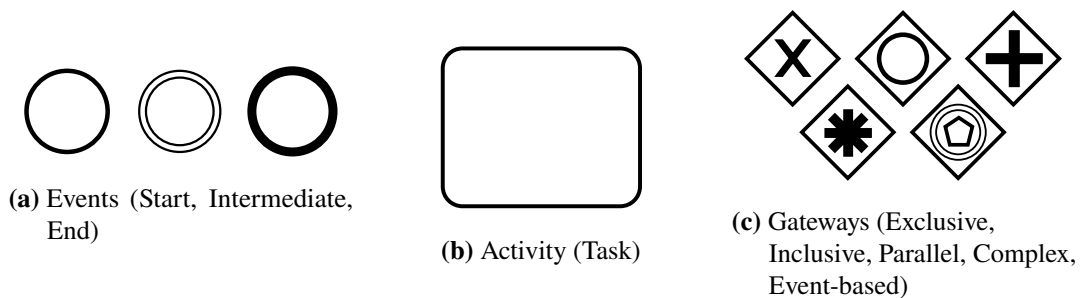


Figure 2.5: Core Flow Object Elements






























































Type	Start			Intermediate				End
	Normal	Event Sub process	Event Sub process non-interrupt	Catch	Boundary	Boundary non-interrupt	Throw	
None								
Message								
Timer								
Conditional								
Link								
Signal								
Error								
Escalation								
Termination								
Compensation								
Cancel								
Multiple								
Multiple Parallel								

Figure 2.6: BPMN Event types (from [Cam21])

Activities

Rounded-corner rectangles are used to describe activities, which represent the actions performed during a business process. Figure 2.5b on page 22 shows a default single unit of work, which is also called a **Task**. For modeling purposes BPMN also supports **Sub-processes**. They allow users to nest their processes. **Sub-processes** can either be collapsed or hidden. In their collapsed state the BPMN elements inside a **Sub-process** can be seen, whereas hidden they can only be distinguished from **Tasks** by a small plus sign in the bottom center of the shape. Same as Events, Activities also have types to further specify what kind of work is done by them. All available Activity types are illustrated in Figure 2.7. In addition to that it is also possible to define the execution behaviour of Activities. Figure 2.8 shows three markers, which can exist in combination with Activity types. Activities can be marked as loops, which suggests that the work done by this activity repeats multiple times according to some condition or even unbounded. Three vertical or horizontal bars represent activities, that can be decomposed into multiple instances, which either run in parallel or sequentially.

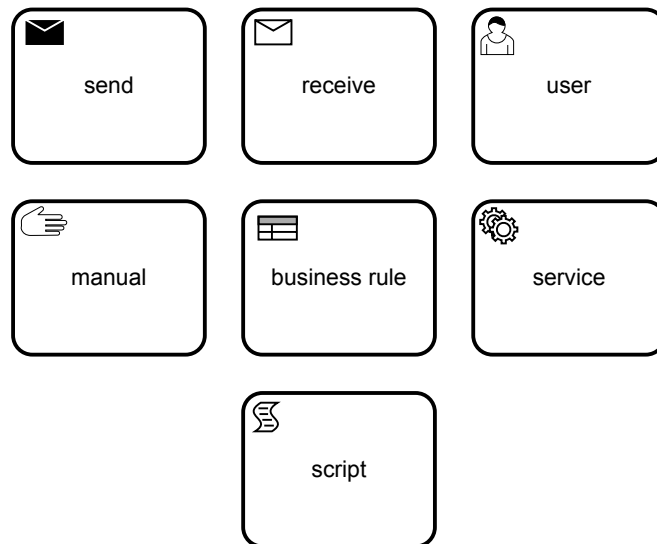


Figure 2.7: BPMN Activity types

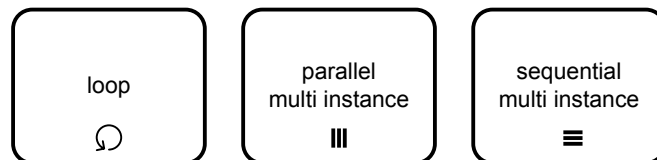


Figure 2.8: BPMN Activities with markers for execution behaviour

Gateways

Gateways are diamond shaped and used to split or join the workflow. Figure 2.5c shows the different types of **Gateways**, which indicate the types of behaviour on how paths are split or joined. Depending on that, the workflow follows the direction of either one, all or a subset of the paths.

2.3.2 Swimlanes

BPMN supports the concept of modeling multiple workflows and separating activities into visually by using Swimlanes. For that **Pools** are used as graphical containers to divide a set of activities from other **Pools**. They often represent a Participant running the process inside of the **Pool**. Figure 2.9 illustrates a **Pool** with two **Lanes**, which represent sub-processes executed by the Participant of its **Pool**.

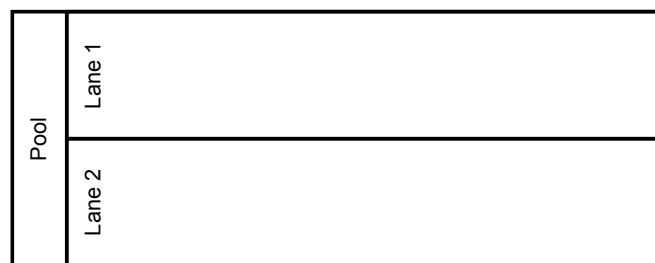


Figure 2.9: BPMN Pool with two Lanes

2.3.3 Artifacts

Artifacts are used to extend BPMN and provide additional context. The basic structure modeled in the business process is not affected by the use of Artifacts. Therefore an arbitrary number of Artifacts can be introduced to provide more details about the execution of the process. To refine the concept of providing additional details, modelers are allowed to create custom types of Artifacts. But the specification of BPMN only pre-defines the following three types of artifacts:

Data

Figure 2.10a shows a **Data Object** and a **Data Store**. Such Data Artifacts can be used to model data sources or storage, for example files or databases, to represent the required inputs or produced outputs of activities.

Group

Groups are rounded corner rectangles like **Tasks**, but drawn with a dashed line. They are used to group **Tasks** similarly like **Pools**, but without influencing execution semantics by separating the workflow from another. Its purpose is solely for visual documentation or analysis purposes.

Annotation

Annotations are text blocks, which are displayed in BPMN diagrams to provide readers additional information about BPMN elements.

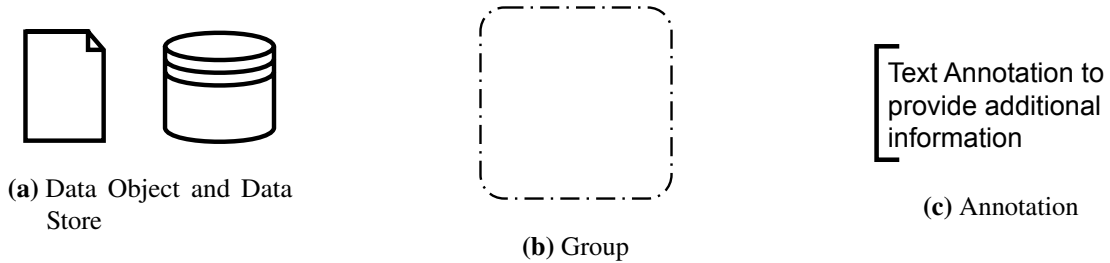


Figure 2.10: BPMN Artifacts

2.3.4 Connecting Objects

Sequence Flow

Sequence Flows are solid line arrows as illustrated in Figure 2.11a and used to connect activities. Their direction represents the execution order of activities.

Message Flow

With the ability of separating multiple workflows by using **Pool**, there is also the need to model the concept of messaging between **Tasks** of different Participants. For this purpose **Message Flows** are used, which are dashed line arrows starting with a small circle and ending with open arrowhead as demonstrated in Figure 2.11b. The direction represents the flow of the message between the sending and receiving **Task**.

Association

Figure 2.11c shows that **Associations** are dotted line arrows with a line arrowhead. **Associations** are used to connect Artifacts with flow object. In combination with Activities, associated Artifacts can be used to represent inputs and outputs.



Figure 2.11: BPMN Connecting Objects

3 Concept

This chapter elaborates how platform-specific FaaS deployment packages can be generated using workflow models specified using the well-known standard for modeling business processes, namely BPMN [Whi04]. Firstly, a method is presented, which offers an overview of the concept. The following sections dive into detail for each step defined in the method. To avoid misunderstandings, Table 3.1 introduces the following terminologies used throughout this thesis:

Term	Meaning
FaaS function	FaaS function (platform-agnostic or -specific) as described in Section 2.2
Business code function	Code provided by the user, which only contains functions
Service call	Invocation of a method, which interacts with cloud services
Task/Operation	BPMN element bound to either a Business Function or a Service Call

Table 3.1: Terminology

As the title of this thesis implies, the method depicted in Figure 3.1 consists on 2 main phases: (i) Standards-based modeling by creating provider-agnostic code and BPMN Workflow models and (ii) generation of provider-specific FaaS code contained inside of deployment packages.

First, users provide an input package containing their custom business code modules, leaving out any kind of provider-specific service calls (e.g. from boto3 or Azure SDK). Further information on how to write functional business code is detailed in the next Section 3.1. Additionally, this input package contains files listing all dependencies needed to implement the business code modules.

In Step 2 and 3 a graphical BPMN Modeling Tool is used to provide user an intuitive way to create a generic BPMN-based workflow model. A subset of BPMN elements represent various control flow elements and function calls, which are extended by properties in Step 3. Additionally, business code functions and service calls are bound to BPMN **Task** elements. The resulting BPMN model together with its properties and the business code modules, provided by developers in Step 1, form a canonical and provider-agnostic model of the desired FaaS function.

Depending on the chosen target platform and its services, additional provider-specific properties like connection strings or Uniform Resource Identifiers (URIs) might be required. To strictly separate these properties from the provider-agnostic model, the BPMN Modeling Tool needs to save them in a *Resource Descriptor* and present them in the Graphical User Interface (GUI) as provider-specific properties.

A Deployment Package Builder component collects the input package, BPMN workflow model and Resource Description File in Step 5 to transform them into a provider-specific FaaS function. Mappers map provider-agnostic service-calls into provider-specific ones depending on the chosen target platform and service type.

Finally, in Step 6 the deployment package is generated. The structure of the package can be build according to the target FaaS platform of cloud providers or third-party deployment automation tools like *Serverless Framework* or *Terraform*.

Many changes and additions are still made in the landscape of Cloud Computing. Therefore it is important to choose a suitable architecture for the Generator to be easily extensible and maintainable for the future.

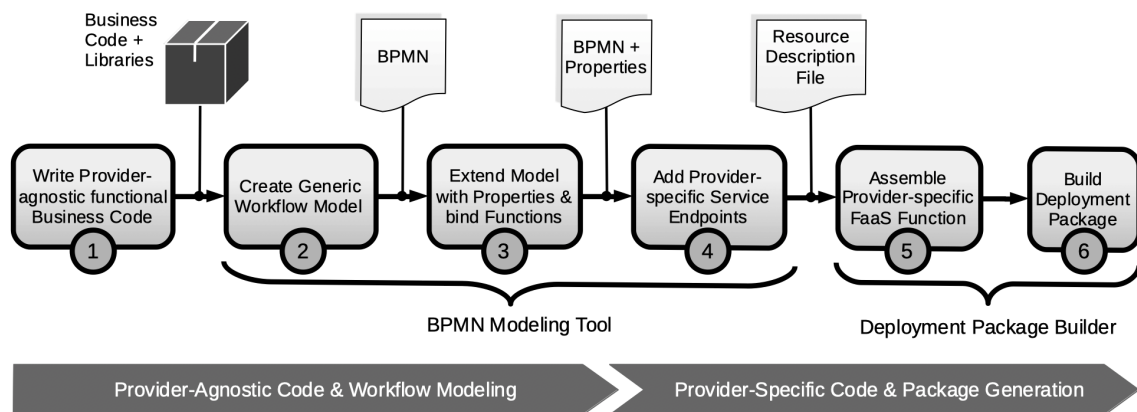


Figure 3.1: Method for standards-based modeling and generation of platform-specific FaaS deployment packages

Throughout this chapter the flowchart illustrated in Figure 3.2 is used as an example to explain each step of the method. It describes a process for handling orders in an online shop. This process receives events from a queue containing orders from customers. First, items are extracted from the message body of the incoming event, to process them separately. In the next step an inventory is retrieved from an object storage to check whether the received items are available. Depending on the current state of the inventory, either a positive or negative response is generated and sent to respective queues for further processing.

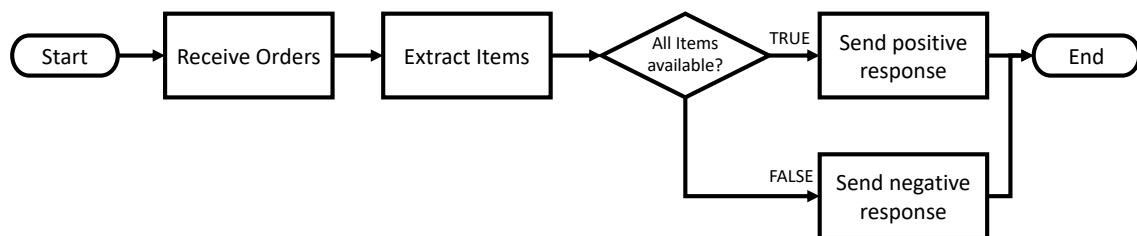


Figure 3.2: Flowchart for handling orders in a online shop

3.1 Step 1: Develop Functional Business Code Modules

To generate a FaaS function it is necessary to know, how developers want to process incoming events. Therefore developers need to provide their own custom code, written in a programming language, which is supported by the target platform of their choice. Writing FaaS function code takes in-depth expertise about the FaaS platform and the services it is interacting with. Additionally, the cloud computing landscape is heterogeneous, which means that developers need to adjust their knowledge every time, they want to deploy their functions on different platforms to avoid lock-ins by the cloud providers.

This section demonstrates developers, how to write and structure their custom *functional business code*. First, developers need to strictly differentiate between *business code* and *service calls*. Business Code is used to purely implement general purpose logic, which is completely free from any cloud related code. It excludes any kind of provider-agnostic and -specific service calls.

The purpose of structuring business code as regular functions is to modularize the code and embed these functions inside the FaaS handler. This can be enabled by importing the provided business code module and calling the functions inside the FaaS function as shown in Listing 3.1.

Listing 3.1 Platform-agnostic FaaS code example (Python)

```

1 import businessCodeModule
2
3 def handler(event, context):
4     extract_items = businessCodeModule.extract_items(event['message_body'])
5
6     get_inventory = []
7     for item in extract_items:
8         get_inventory.append(get_object('inventory', 'items/' + item + '.txt'))
9
10    check_inventory = businessCodeModule.check_inventory(get_inventory)
11
12    if check_inventory:
13        send_positive_response = send_message('deliveryQueue', event['message_body'])
14    else:
15        send_negative_response = send_message('abortQueue', event['message_body'])

```

In the following subsections a correct example of functional business code is described for the workflow depicted in Figure 3.2, as well as a *code-smell* [Fow99] explaining how it should not be done.

3.1.1 Code-Smell

Listing 3.2 illustrates a code smell, which is syntactically and semantically correct. Additionally it would even run as desired, but it is structured wrongly in regard to the following three reasons:

1. Business code and service calls are mixed up together, instead they must be separated
2. Service calls are platform-specific, instead they must be platform-agnostic

3 Concept

3. Service calls are not allowed to be contained in business code modules at all, instead they should be defined in the BPMN Modeling Tool via GUI

Listing 3.2 Code-smell snippet of a functional business code module (Python)

```
1 import boto3
2
3 def extract_items(order):
4     return order.split(',')
5
6 def check_inventory(items):
7     client = boto3.client('s3')
8
9     for item in items:
10        inv = client.get_object(
11            Bucket='inventory',
12            Key='items/' + item + '.txt',
13        )
14
15        # if inv empty:
16            return false
17
18    return true
19
20 ...
```

3.1.2 Correct Example

Due to the three problems listed in the previous Section 3.1.1, the lines 1,7 and 10-13 in Listing 3.2 need to be omitted and the functions structured accordingly to the BPMN workflow model in Figure 3.2, leading to the Listing 3.3 below, which is containing purely general purpose business logic and is therefore completely provider-agnostic:

Listing 3.3 Correct example of a functional business code module (Python)

```
1 def extract_items(order):
2     return order.split(',')
3
4 def check_inventory(inventories):
5     for inv in inventories:
6         # if inv empty:
7             return false
8
9     return true
```

3.1.3 Libraries

FaaS allows developers to implement a variety of functionalities. Therefore special libraries, also called *dependencies*, might be needed, which are not part of the chosen runtimes standard library collection. Such libraries need to be added in order to provide the FaaS function a way to access them. This can be done during deployment, but the process is handled differently by each cloud provider:

(i) One possibility to add packages for the FaaS function is to add them *pre-deployment*, where developers need to manually install them in a separate folder inside their project (e.g. *node-modules* in Node.js) and deploy them together with their FaaS function as additional source code. This process of deploying additional packages is one of the ways supported by AWS Lambda [Ama22c].

(ii) In contrast to AWS, cloud providers like Azure require files like *package.json* for Node.js or *requirements.txt* for Python, where developers can list the dependencies they need together with their versions. Afterwards, the Functions App in Azure (or any equivalent service of different cloud providers) is responsible to install them by taking the list of dependencies and install them automatically during deployment by using the respective package manager, e.g. *npm*¹ for Node.js or *pip*² for Python [Mic22a].

To obtain a provider-agnostic procedure of adding libraries to the FaaS function option (ii) is chosen, where lists of dependencies are provided by the developers together with their functional business code modules, because in this step developers are not aware of the target platform yet. The target platform is chosen using the BPMN Modeling Tool at the end of step 4 of the method depicted in Figure 3.1. Therefore it makes more sense to install the libraries at a later stage of time (in step 6 or automatically during deployment managed by the cloud provider) as done in the second option. Inside the code both standard as well as external libraries or additional modules can be imported, where ever they are needed.

3.2 Step 2: Create Generic Workflow Model

A provider-agnostic model is needed to have a canonical representation of the FaaS functions workflow, which then can be transformed into FaaS functions of the chosen platform. One of the main questions to be answered in this thesis is on how to model FaaS code using BPMN. This question needs to be answered by decomposing it into the following finer-grained questions:

- Which BPMN elements can be used to model a FaaS function?
- How can incoming events, triggered by provider-specific services, be modeled in a provider-agnostic way to help developers lacking provider-specific knowledge across multiple clouds?
- How can provider-specific service calls be modeled in a provider-agnostic way?
- How can business code functions, provided in Step 1, be integrated into the model?

¹<https://www.npmjs.com>

²<https://pip.pypa.io>

This section elaborates the first question on how to represent the workflow of a FaaS function using BPMN. The remaining questions are solved by additional properties and therefore covered in Section 3.3. First, a FaaS function needs to be analyzed to determine what needs to be modeled in BPMN diagrams. Looking into Listing 3.1 on page 29 shows that handler functions consist of parameters and a body. The body of provider-agnostic handlers are only allowed to hold business functions and service calls, which can be combined with if- or switch-statements and loops. Additionally, every FaaS function has a file name. To summarize, the goal is to find suitable BPMN elements for Step 2 of the method depicted in Figure 3.1 on page 28, which are able to represent different event types, business functions, service calls, branches, loops and ultimately the name of the file containing everything.

Start Events

In BPMN every workflow starts with a **Start Event**. It seems naturally to use **Start Event** to describe incoming events from services. Both the default **Start Event** as well as the **Message Start Event** can be used here equally, because incoming events can also be interpreted as incoming messages from a event queue. All other event types listed in Figure 2.6 on page 23 are not applicable except for timer events, which suit perfectly for timer triggers.

(Service) Tasks & Data Stores

Business functions and service calls are the actual work, which has to be done during execution of a FaaS function. Therefore, both have to be modeled using BPMN **Tasks**. While a default **Task** is used for business functions, it is reasonable to represent service calls with **Service Tasks**. This way it is ensured, that business functions and service calls can be clearly distinguished in a graphical way. For both types of **Tasks** a loop marker indicates, that this **Task** has to be embedded inside a loop when it is transformed to code. Additionally, **Data Store** elements can be used in combination with **Service Tasks** to indicate an interaction with a storage or database.

Gateways

Gateways are the only class of elements in BPMN, which allow users to split their workflow. **XOR/Exclusive Gateways** are well suited for modeling branches. Whether it is an if- or a switch-statement can be distinguished by the amount of outgoing connectors, where a **Gateway** with 2 outgoing connectors represents an if-statement and **Gateways** with more than two **Gateways** a switch-statement. Every use of a splitting **Gateway** must come with a joining **Gateway**. This rule of designing branches in workflows is required to determine their scope. In programming languages like Java or C# this is indicated syntactically by a pair of braces, whereas in Python by indentation. Figure 3.3 demonstrates an example on how to design a correct branching of the workflow. One might come up with the idea to also introduce **Parallel Gateways** to model concurrently running threads. But the problem is, that concurrency is not part of the FaaS paradigm in the level of code. Serverless applications are only able to process workload in parallel by enabling concurrency settings in the FaaS platform, which allow invocations of handlers in parallel to a maximum number of instances defined in the settings. This can increase the throughput, but also raise the price significantly. Therefore **Parallel Gateways** should not be used in the context of FaaS modeling.

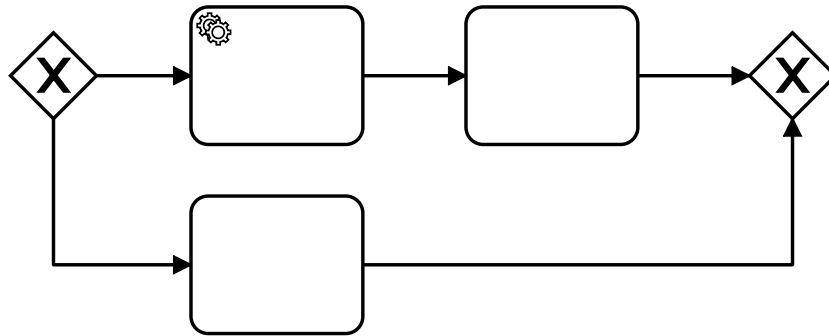


Figure 3.3: Example on how to split/join workflows using Gateways

End Events

In case of asynchronous invocations, it is optional to let the function return something, but for synchronous invocations a requester waits for an response, which makes returning mandatory. Such scenarios are modeled by using BPMN **End Events**.

Pools & Lanes

FaaS code is always contained inside of a file. This way it can be separated from other FaaS functions. BPMN workflows are used to model FaaS code and since they can be placed inside of a **Pool** of **Lanes**, it makes sense to represent files with **Lanes**. This enables users to strictly separate multiple workflows from each other and therefore allows modeling multiple FaaS modules in one diagram.

Using the introduced set of BPMN elements and their representations, the flowchart illustrated in Figure 3.2 on page 28 can now be modelled as a BPMN workflow diagram. Figure 3.4 demonstrates an example, where a **Start Event** is used to represent the receiving of incoming orders. **Tasks**, which needs to interact with other cloud services are marked as **Service Tasks**, while the remaining ones are modelled by default BPMN **Tasks**. A **Data Store** is introduced to further refine what category of service is used to retrieve the inventories. Additionally its **Service Task** is marked with a loop marker, which indicates multiple executions for each item. Two **Gateways** are used to split and join the workflow between them to send messages according to the state of the inventory.

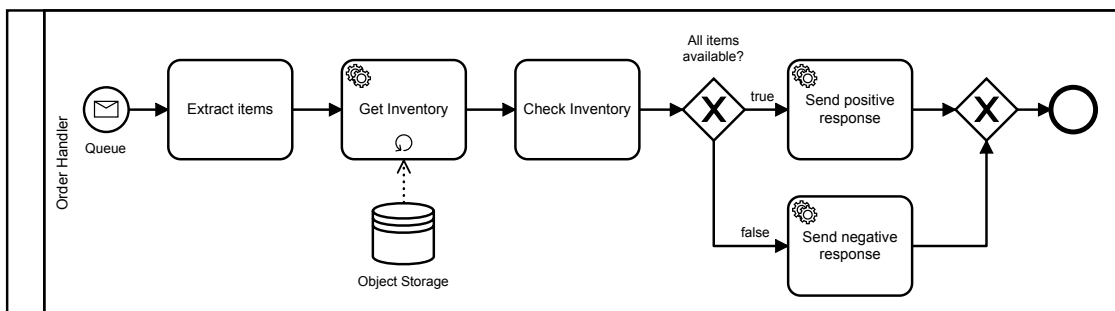


Figure 3.4: BPMN diagram modeling the online shop process

3.3 Step 3: Extend Model with Properties & Integrate Business Functions

Step 3 of the method in Figure 3.1 on page 28 requires users to add properties to their BPMN diagram created in Step 2. The goal is to model the details of their targeted FaaS function in a provider-agnostic way, so that also developers without much knowledge about the target platform can circumvent vendor lock-in and generate FaaS functions tailored for the platform of their choice.

3.3.1 Provider-agnostic Properties

Table 3.2 shows the properties, which are used by all types of BPMN elements. Every element is uniquely identified by its property `id`. This property is automatically generated on creation of the element. Additionally every element can have a name set by the user. If a name is present, it is also displayed graphically in the diagram. For documentation purposes the `documentation` property is introduced to allow users to describe their elements and add additional information. In the following the properties for all BPMN elements, chosen in Section 3.2, are introduced.

BPMN Element
<code>id</code>
<code>name</code>
<code>documentation</code>

Table 3.2: General BPMN Element Properties

Start Events

BPMN **Start Events** need the property `trigger`, which tells the back-end what kind of service is triggering the incoming event. Therefore a list of trigger types needs to be provided to the users, from which they can choose. Those trigger types need to be stated by using a provider-agnostic terminology, due to the possible lack of knowledge about equivalent services from different cloud providers. In the scope of this thesis, the focus relies on the following set of trigger types:

- Object Storage
- Document Store
- (First in, first out (FIFO)) Queue
- Publish/Subscribe (PubSub) Channel

Whenever a user selects the queue option, an additional Boolean property `fifo` is needed, which indicates, whether the event source is a FIFO or default queue. The timer trigger type is not listed in this list, because timer triggers can simply be defined by using the BPMN timer event type. Some platforms require deployment packages to carry the information about the schedule, when timer triggers are fired. Therefore the `schedule` property is needed for **Timer Start Events**, where users have to provide a schedule in cron format. Table 3.3 summarizes all properties used by **Start Events**.

Start Event	
Default/Message	Timer
trigger	schedule
fifo	

Table 3.3: Start Event Properties

Tasks

From a programming viewpoint, both default and **Service Tasks** are substituted with function calls in code. Thus, both need to have a property, which specifies the function to be executed by the **Task**. In the context of a default **Task** the function is a business function. Therefore the name of its property is function, whereas the term `serviceCall` is used for **Service Tasks**.

Because of the fact, that both types of **Tasks** are modeling function calls, they also need properties for arguments. This property can be represented in two ways: Either using a list named `arguments` with a dynamic number of entries or defining N arguments with the properties `argument1` to `argumentN`.

For an improved user experience, a mechanism needs to be provided, which analyzes the provided functional business code and extracts its function and arguments name. Those function and argument names are returned to the front-end to provide users the ability to choose their business function from a list, instead of typing them in, which can lead to unnecessary typos. Arguments are retrieved from the business code modules to provide users the right amount of inputs and to distinguish between multiple arguments by their name. Users are able to set the arguments either by selecting available variables from a list or by typing them in by hand, since arguments can also be defined directly without variables by using for example string literals like "Hello World!".

Same as the trigger types from the trigger of **Start Event**, the offered services are stored in the service property and need to be presented with provider-agnostic terminology. This leads to the following list of options:

- (FIFO) Queue
- PubSub Channel

Using **Service Tasks**, developers are only presented a list of service calls, corresponding to the selected service. Obviously the service calls have to use provider-agnostic terminology as well. This is done by comparing the parameters of equal service calls from each provider and determining their similarities. An example of this procedure is illustrated in Figure 3.5, which describes how the provider-agnostic service call `get_object` emerges out of the comparison between `get_object` from AWS and `download_blob` from Azure. This example only compares the required parameters for those functions. Additionally, the same logic needs to be applied to determine the format of the output. In the scope of this thesis, the focus relies on the most important service calls, e.g. Created/Read/Update/Delete (CRUD) operations. However, the concepts discussed here can be extended to enable other types of calls in the future.

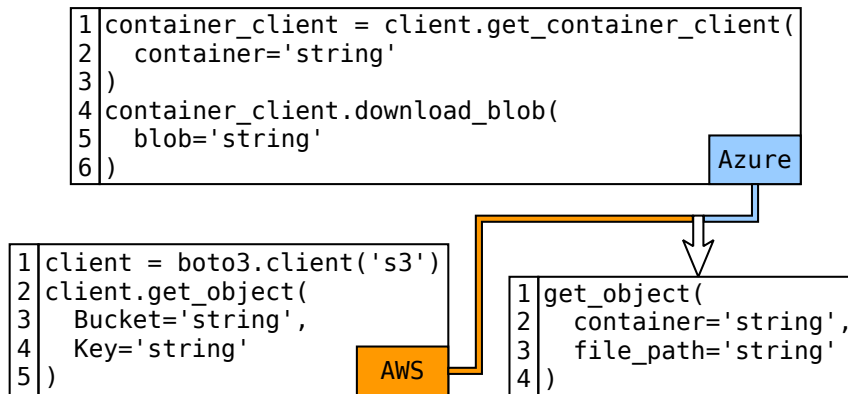


Figure 3.5: Procedure of creating provider-agnostic service calls

Again, a Boolean is stored inside an additional fifo property, whenever the queue service is selected, to declare whether it is a default or FIFO queue. In case of a connection to a **Data Store** the list of services for this **Service Task** is omitted. Instead, another list is provided when working with the **Data Store** element. Unfortunately it is not possible to represent document stores provider-agnostically. The reason lies in the differences between Amazon’s DynamoDB service and its Microsoft Azure equivalent Cosmos DB. Figure 3.6 shows the structure of key-value documents stored in DynamoDB. Items are uniquely identified by a simple primary key defined by the items partition key. A partition key can be combined with a sort key to form a composite primary key [BS17]. While partition keys are used in DynamoDB as well as in Cosmos DB to determine where items are stored, Cosmos DB does not use partition keys as primary key. Instead a id attribute is used. This leads to two different interpretations of the partition key. Since the partition key is essential to this service type, it is required to be a part of the provider-agnostic service model. But being this ambiguous, it is not possible to have a provider-agnostic representation of document stores, because it can lead to misunderstandings for users, who only know one of those two services.

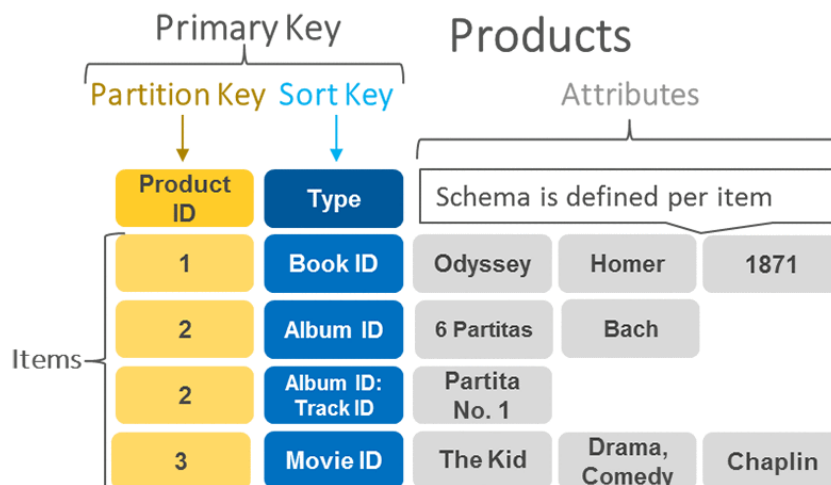


Figure 3.6: Example schema of a document store using DynamoDB (from [BS17])

If a **Task** is marked with a loop marker, user have the decision to pick whether a for- or forEach-loop should be used. True is stored in the property `loopType`, in case `forEach` is the chosen type of loop. Additionally developers have to provide a loop condition according to the selected loop mode and programming language. This loop condition is saved either in the property `for` or `forEach`, depending on the activated loop mode.

Task		Data Store
Default	Service	
	service	service
function	serviceCall	
fifo	fifo	
arguments	arguments	
loopType	loopType	
for/forEach	for/forEach	

Table 3.4: Task Properties (with Data Store)

XOR/Exclusive Gateways

Every **Exclusive Gateway**, used in the BPMN diagram, needs to be differentiated between a splitting and joining **Gateway** by their amount of incoming and outgoing connectors. Only splitting **Gateways** hold properties, which are listed in Table 3.5. Furthermore, it needs to be determined whether it represents an if- or switch-statement. This information is held by the property `mode`. The `condition` property is used to provide users the ability to define the condition under which the workflow is split. Because switch **Gateways** have multiple branches, users need to assign cases to every outgoing sequence flow attached to a **Gateway**, by using the property `case`, in order to declare the direction to go depending on the outcome of the condition. This can be omitted for one branch by declaring it as the default branch using the Boolean property `default`.

XOR/Exclusive Gateways	Sequence Flow
condition	case
mode	default

Table 3.5: XOR/Exclusive Gateway Properties

End Events

End Events are used to represent outputs of a functions. Users can define what is returned by their FaaS function by providing a literal in the same programming language used in their functional business code. This literal is stored in the property `return`.

Lanes

Lanes represent files and the only metadata needed from them are the filenames. Therefore this BPMN element only holds the property name. This property will be used to name the generated FaaS function.

3.3.2 Event Schema

The event parameter plays an integral role in event-driven programming and therefore also in the context of implementing FaaS functions. It holds all the information triggering services have about their affected object. Events are structured differently for each trigger type and cloud provider. This means, that they can be objects, JavaScript Object Notations (JSONs) or dictionaries. Additionally they can hold different properties.

Listing 3.4 shows an example event from Amazon's Simple Queue Service (SQS) service in form of a JSON with several properties representing a new message. In contrast to Amazon, Azure uses object from the `QueueMessage` class for messages from its equivalent Azure Queue Storage service. The Unified Modeling Language (UML) diagram in Figure 3.7 lists attributes and methods of objects from the `QueueMessage` class.

Listing 3.4 Example Amazon SQS notification event [Ama22d]

```
1 {
2   "Records": [
3     {
4       "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
5       "receiptHandle": "AQEBWJnKyrHigUMZj6rYigCgxLaS3SLy0a...",
6       "body": "Test message.",
7       "attributes": {
8         "ApproximateReceiveCount": "1",
9         "SentTimestamp": "1545082649183",
10        "SenderId": "AIDAIENQZJLO23YVJ4VO",
11        "ApproximateFirstReceiveTimestamp": "1545082649185"
12      },
13      "messageAttributes": {},
14      "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
15      "eventSource": "aws:sqs",
16      "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:my-queue",
17      "awsRegion": "us-east-2"
18    },
19    {
20      ...
21    }
22  ]
23 }
```

Using such platform- and service-specific events requires developers to have knowledge about the event schema. Therefore the goal of this section is to create a platform-agnostic event schema, which then can be transformed to a platform- and service-specific events schema. This concept recommends creating schemes applying to the specification of CloudEvents [Clo22a] for each service type. This way developers do not need to gain knowledge about the different service-specific

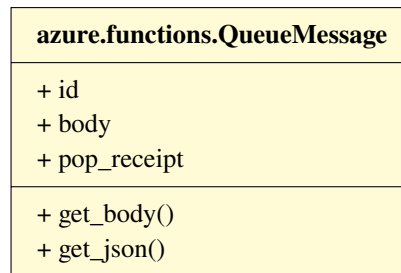


Figure 3.7: UML diagram of `azure.functions.QueueMessage` class [Mic22d]

event schemes of multiple cloud providers. Moreover, getting familiar with an event schema of one service type is already enough to understand the schemes of other service types, since they are very similar to each other.

CloudEvents

Standardized schemes like JSON or Extensible Markup Language (XML) have become an integral part of the computer science landscape. Their popularity gets many companies to change their products in a way, where they can support the use of such standards. Due to the large variety of schemes in the field of event-driven programming a standardization is needed to create a common structure between all platform- and service-specific events schemes, which are intuitive and can be quickly understood by every developer.

In May 2018 the Cloud Native Computing Foundation (CNCF) created the CloudEvents specification with the purpose of creating a common event format to aid in the portability of functions between cloud providers and the interoperability of processing of event streams. [Clo22b]

The CloudEvents specification defines the schema of all kinds of events with attributes of the three categories *required*, *optional*, and *extensions*. For those attributes only the following abstract data types are allowed and must be supported by all implementations: [Clo22b]

- Boolean
- Integer
- String
- Binary
- Absolute URI encoded as String
- URI-reference encoded as String
- Timestamps (with date and time) expression encoded as String

As the name says, required attributes have to be present in every event, while optional attributes are also predefined by the CloudEvents specification, but only used in some cases, depending on the context of the event. Those are the required attributes used in every event schema: [Clo22b]

id: String used to uniquely identify events in combination with the source attribute.

source: URI-reference of the service or event source triggering the event.

specversion: Version of the Cloudevents specification used in the event schema.

type: Describes the type of action, which triggered the event.

In addition to the mentioned required attributes, every schema will also have a data attribute, which holds the whole provider- and service-specific event in the schema it is arriving. This way users with advanced knowledge are also able to access properties of the event, which are not represented by a provider-agnostic CloudEvents attribute.

The same procedure as demonstrated by Figure 3.5 on page 36 is used to determine further attributes. Listing 3.5 shows the result, which emerged by comparing the attributes from Listing 3.4 on page 38 with the attributes provided by the QueueMessage class from Figure 3.7 on the preceding page. Due to the small amount of attributes offered by Azure additional attributes from the context object are required.

Listing 3.5 CloudEvent schema of an object storage event

```
1 {
2   "id":           "string",
3   "source":       "string",
4   "specversion":  "string",
5   "type":         "string",
6   "datacontenttype": "string",
7   "time":         "string",
8   "subject":      "string",
9   "data":         "object"
10 }
```

Supporting Event Batches

In the context of FaaS the term *batch* describes multiple events bundled to one event. This enables event source mappings to combine a predefined maximal number of events occurred in a predefined period of time, also called *batch window*, into a single invocation of the FaaS code. The main advantage of using batches is to reduce the number of invocations and therefore save resources and cost. This technique is only used by some providers and service. Listing 3.4 on page 38 shows an example, where batches are used to trigger the FaaS function with two messages from a SQS queue, which occurred in a close period of time and therefore not exceeded the batch window property of the trigger.

To avoid the need of developers to know which services provide batched events and which not, those events need to be resolved into single events inside the provider-agnostic FaaS function. Therefore the content of the handler function needs to loop through each event and encapsulating the whole content of the handler function inside that loop. This needs to be done at code generation time during step 5 of the method in Figure 3.1 on page 28. Doing this enables developers to model their FaaS function always by ignoring whether batching is used or not, no matter which cloud or service is used. A practical example is shown in Listing 3.6, where such a loop is inserted into the handler function. Inside its body the `to_cloudevents_schema` function transforms every single event into the CloudEvents schema and functional business code and service calls are executed afterwards.

Listing 3.6 Resolution of batched events into single events (Python)

```

1 # ...
2
3 def handler_function(events, context):
4     for event in events:
5         event = to_cloudevents_schema(event)
6         # functional business code and service calls
7         # ...
8
9 def to_cloudevents_schema(event):
10    # transform event into cloudevents schema and return

```

3.3.3 Online Shop Example

In Step 3 the BPMN elements of the online shop handler illustrated in Figure 3.4 have to be refined by extending them with the provider-agnostic properties introduced in Section 3.3.1. Figure 3.8 demonstrates all properties needed to model the provider-agnostic workflow of the target FaaS function. The properties of the **Start Event** define, that the the service type of event source triggering this function is a FIFO queue. Next, the function `extract_items`, provided by the user via business code module, is bound to the first **Task**. With the help of provider-agnostic event schemes based on CloudEvents, developers know, that the message body will be stored in the `message_body` attribute of the event, no matter which target platform the resulting FaaS function will be deployed on. The **Data Store** declaring to be a object storage sets also implicitly the service its **Service Task** is interacting with. According to its loop marker, the **Service Task** retrieving inventories for each item of the order is executed multiple times. Its loop condition is defined by a python literal using the output of the previous **Task** to iterate over a list of items. Using terminology, which is provider-agnostic and easy to understand helps developers finding the right service call to access inventories stored in an object storage. The splitting **Gateway** uses the Boolean returned by its previous **Task** as condition on how to split the workflow. Its mode can be implicitly determined by the number of incoming and outgoing sequence flows. The case properties of the outgoing sequence flows have to be set to `true message_body false`, depending on which direction of the branching should be followed. Specifying `event['data']` as second argument for the two messaging sending **Service Tasks**, forwards the incoming message in its provider-specific structure to allow other FaaS functions, which might be invoked by this message, to transform this message back to CloudEvents scheme if needed, implying that the other FaaS function is also generated following this concept.

3.4 Step 4: Add Provider-specific Service Endpoints

When using service calls, both Amazon's SDK *boto3* as well as the SDK from Microsoft Azure require developers to bind clients to the desired service entities. In AWS it is sufficient to create a client to bind to the service by using for example `boto3.client('s3')`. Authorization is handled in the background by roles and permissions granted to the FaaS function. Specific service entities and objects can simply referenced by their names. However, interacting with a service via clients in Azure requires developers to use *connection strings* to uniquely identify the right service entity and get the needed authorization.

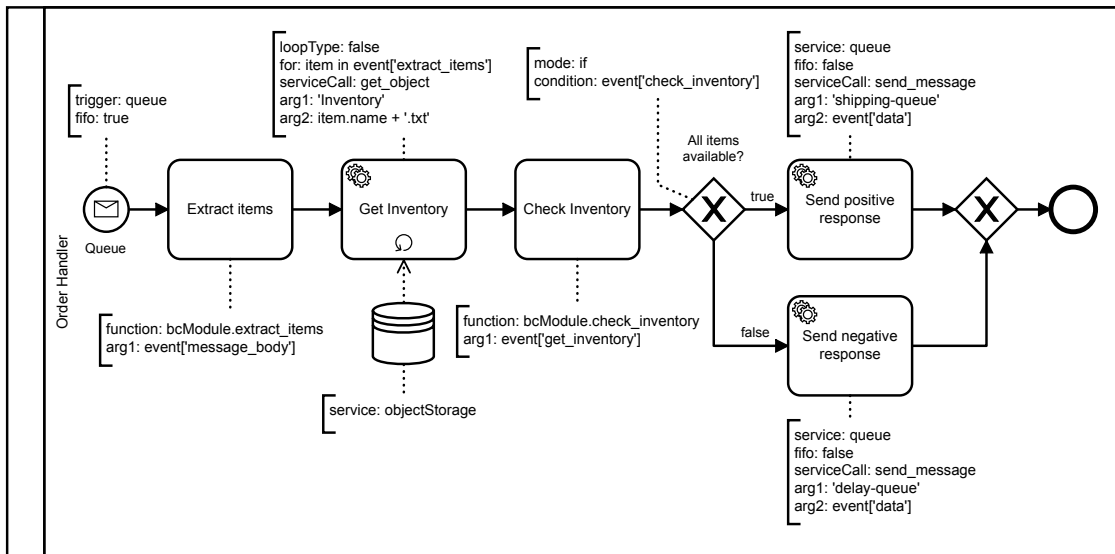


Figure 3.8: BPMN diagram enhanced with properties

Listing 3.7 demonstrates an example in which line 4 creates an client object for Azure Blob Storage service using a connection string. In line 11 another client is generated in the context of containers. This client allows line 14 to access objects stored in the container.

Listing 3.7 Service call example for Azure Blob Storage (Python)

```

1 # ...
2
3 connection_string = 'myConnectionString'
4 client = BlobServiceClient.from_connection_string(connection_string)
5
6 def handler_function(events, context):
7     for event in events:
8         event = to_cloudevents_schema(event)
9         # ...
10        event['get_inventory'] = []
11        container_client = client.get_container_client("mycontainer")
12        for item in event['extract_items']:
13            event['get_inventory'].append(
14                container_client.download_blob(item.name+'.txt')
15            )
16        # ...
17
18 # ...

```

Connection strings are structured differently for each service type. For Storage Accounts they are composed out of DefaultEndpointsProtocol, AccountName and AccountKey, which leads to this example: [Mic22c]

DefaultEndpointsProtocol=https;AccountName=mystorage;AccountKey=<account-key>

In other services `DefaultEndpointsProtocol`, `AccountName` can be substituted by a HTTP endpoint. In AWS the counterpart for this are Amazon Resource Names (ARNs). In contrast to connection strings in Azure, ARNs are structured according to one of the following three formats: [Ama22a]

```
arn:partition:service:region:account-id:resource-id
arn:partition:service:region:account-id:resource-type/resource-id
arn:partition:service:region:account-id:resource-type:resource-id
```

Unfortunately connections strings and ARNs are fundamentally different from each other. Therefore it is not possible to model this information in a provider-agnostic way. Because of this drawback resource descriptors are introduced in step 4 of the method. A resource descriptor holds information about the target platform and all provider-specific *endpoints* needed for each **Service Task** in the BPMN diagram. Storing this data in a JSON ensures, that provider-agnostic and provider-specific properties are separated from each other. An example resource descriptor is shown in Listing 3.8.

Listing 3.8 Example resource descriptor

```
1 {
2   "provider": "azure",
3   "endpoints": {
4     "element_id1": "<connection_string1>",
5     "element_id2": "<connection_string2>"
6   }
7 }
```

3.5 Step 5: Generate Provider-specific FaaS Function

Step 5 of the method depicted in Figure 3.1 on page 28 is about transforming the provider-agnostic representation of a FaaS function into a provider-specific version of it, implemented in the same programming language as the provided business code modules. All the information about the function is divided on multiple BPMN elements. Therefore, it requires a mapping from provider-agnostic to -specific and a mechanism to put all the peaces together. As a solution this section introduces a function template, which describes the structure of a FaaS function and allows to assemble it by injecting code snippets into it, which are generated by the transformation of each BPMN element occurring in the workflow diagram.

In order to create a template, a FaaS function needs to be decomposed first. Figure 3.9 shows the provider-specific version of the function demonstrated in Listing 3.1 on page 29 and is targeted for the AWS platform. Every FaaS function is divided into the sections imports, clients, handler, batch, tasks and cloudevents. By iterating over all BPMN elements all business code modules, which need to be imported, are determined. Those imports are injected at the top of the file. The interacted services are also be determined during this iteration. For every detected service the respective clients are listed below the imports. The handler section always contains the same line of code as seen in the example. The only difference that can be made is to use the plural version of the term event whenever the function is invoked by a trigger. which uses a batched event schema. If that is the case a for-loop is introduced in the optional batch section.

3 Concept

```
import businessCodeModule                                     imports
s3_client = boto3.client('s3')                               clients
sqs_client = boto3.client('sqs')
def handler(events, context):                                handler
    for event in events:                                     batch
        event = to_cloudevents(event)                       tasks
        event['extract_items'] = businessCodeModule.extract_items(event['message_body'])
        event['get_inventory'] = []
        for item in event['extract_items']:
            event['get_inventory'].append(
                s3_client.get_object(Bucket='inventory', Key='items/' + item + '.txt')
            )
        event['check_inventory'] = businessCodeModule.check_inventory(event['get_inventory'])
        if event['check_inventory']:
            event['send_positive_response'] = sqs_client.send_message(QueueUrl='deliveryQueue',
                MessageBody=event['message_body'])
        else:
            event['send_negative_response'] = sqs_client.send_message(QueueUrl='abortQueue',
                MessageBody=event['message_body'])
def to_cloudevents(event):                                   cloudevents
    # ...
```

Figure 3.9: Platform-specific FaaS code example (Python)

The biggest portion is taken by the tasks section, which contains the whole work done by the function. After transforming (**Service**) **Tasks** and **Gateways** to code, it is injected into this section in the order given by the workflow in the BPMN diagram. Both service calls and business functions might need arguments and can return data which then might be reused again in upcoming service calls or business functions as parameters. This data needs to be stored in the outermost scope inside the handler function, where it is accessible as an input for every other function call coming afterwards. A possible solution is to extend the event parameter, since it is accessible in the whole handler function. This can be done by storing the outputs in properties labeled with the name or identifier of the **Task**.

At the end of the file the CloudEvents transformation function is inserted. The body of this function depends on the chosen platform and trigger type.

With this knowledge a template can be created, described by the syntax and semantics of template engines like *Jinja*³ or the ones offered by *Django*⁴ or *Express*⁵. The decision on choosing the right engine, depends on the selected programming language of the Generator, which also depends on the targeted programming language by the user. Optimally the Generator is written in the same language as the generated function.

3.6 Step 6: Build Deployment Packages

This sections presents the structures of deployment packages for AWS Lambda and Azure Functions, which are generated in the last step of the method and returned to the user. Depending on the chosen runtime deployment packages are structured differently. The following examples are under the assumption, that Python is the chosen programming language. Furthermore, the only additional modules needed are the one business code module introduced in Section 3.1.2 on page 30 and the *numpy*⁶ package defined in `requirements.txt`.

3.6.1 AWS Lambda

Deployment packages in AWS Lambda follow are very easy and straight forwards structure as illustrated in Listing 3.9. They are simple `.zip` files containing the FaaS function, the business code module and the package folder for additional dependencies. Dependencies are added by using `npm` by providing the path of the `requirements.txt` file and the path of the package folder as destination. [Ama22b]

Listing 3.9 Deployment Package Structure for AWS Lambda

```
my-deployment-package.zip
| - order_handler.py
| - businessCodeModule.py
| - package/
| | - numpy/
```

3.6.2 Azure Functions

Azure Functions also uses `.zip` packages for deployment, but in contrast to AWS Lambda, deployment packages in Azure Functions are not structured in the point of view of individual FaaS functions, but of a serverless project as a whole. Listing 3.10 shows a package containing two FaaS functions in separate folders together with their business code modules. Additionally the following files are contained: [Mic22b]

³<https://jinja.palletsprojects.com/>

⁴<https://docs.djangoproject.com/en/4.0/topics/templates/>

⁵<https://expressjs.com/de/guide/using-template-engines.html>

⁶<https://numpy.org/>

function.json: Holds information about the trigger and binds it to the right parameter in the right python script.

host.json: Contains settings for all function instances of the project like batch sizes, timeouts or configurations for the logging service.

local.settings.json: Is used to save configurations for local execution and connection strings. This file is not published on Azure.

requirements.txt: List of dependencies to be installed by Azure Functions during deployment.

Listing 3.10 Deployment Package Structure for Azure Functions

```
my-deployment-package.zip
| - order_handler/
| | - __init__.py
| | - function.json
| | - businessCodeModule.py
| - my_second_function/
| | - __init__.py
| | - function.json
| | - businessCodeModule2.py
| - host.json
| - local.settings.json
| - requirements.txt
```

4 Implementation

BPMN2FaaS¹ is a prototype, which implements the concept introduced in Chapter 3. In the scope of this thesis BPMN2FaaS was implemented to generate FaaS functions written in Python for the two platforms AWS Lambda and Azure Functions. First, this chapter gives an overview about the components of this prototype and their interactions with each other. Section 4.2 introduces technologies, which are used by BPMN2FaaS. The last sections of this chapter cover each component in detail.

4.1 Architecture

The architecture of BPMN2FaaS is depicted in Figure 4.1. It contains of two main components: Modeler and Generator. The Modeler enables users to upload their functional business code, create a provider-agnostic BPMN model of their FaaS function, select the target platform and add additional provider-specific properties. The architecture of the Generator is designed as a core module, which can be extended by plugins. This way the core module can be easily extended in the future to support additional platforms. The core module consists of the components Function Extractor and Plugin Manager. The task of the Function Extractor is to analyze the functional business code modules and extract information about the provided business functions. The Plugin Manager checks the existence of plugins and assigns the resources from the Modeler to the corresponding plugin. Afterwards the plugins are responsible to generate provider-specific deployment packages containing the targeted FaaS function. At the end the core module returns the deployment package to the Modeler. Figure 4.1 provides an overview on the architecture of BPMN2FaaS, while Figure 4.2 illustrates the order of interactions taking place between Modeler and Generator.

4.2 Technologies

The Modeler allows users to view and create BPMN diagrams in their browser. This feature is enabled by the library *bpmn-js*², which is build by the team of *Camunda*³. Moreover, diagrams can be exported into a XML or Scalable Vector Graphic (SVG) file. The XML export can be used later

¹<https://github.com/BPMN2FaaS>

²<https://bpmn.io/toolkit/bpmn-js>

³<https://camunda.com>

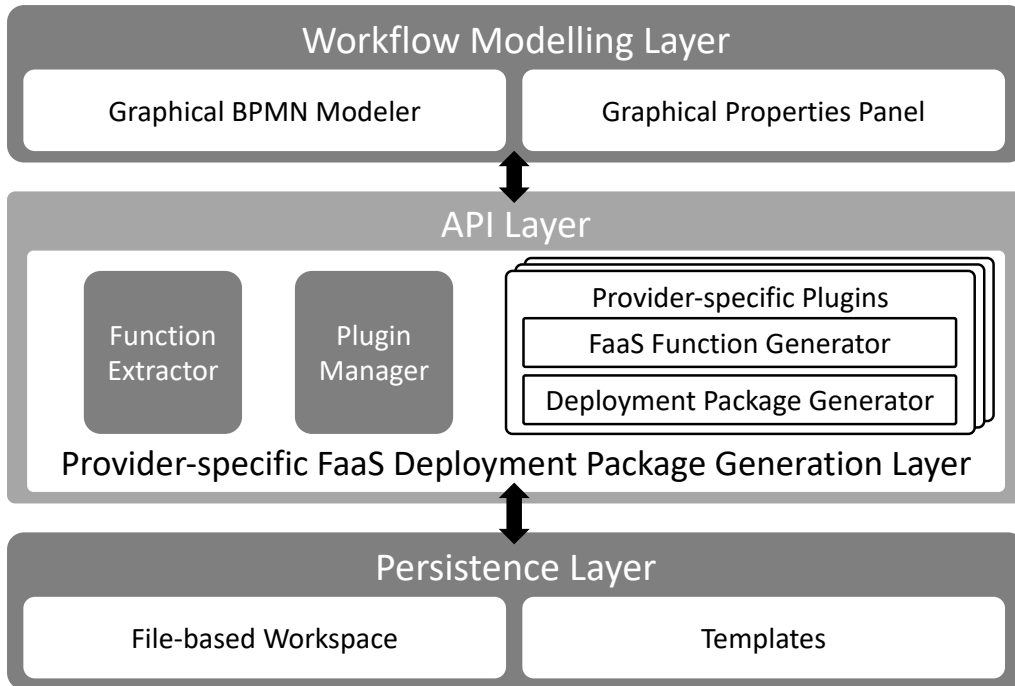


Figure 4.1: Architecture of BPMN2FaaS

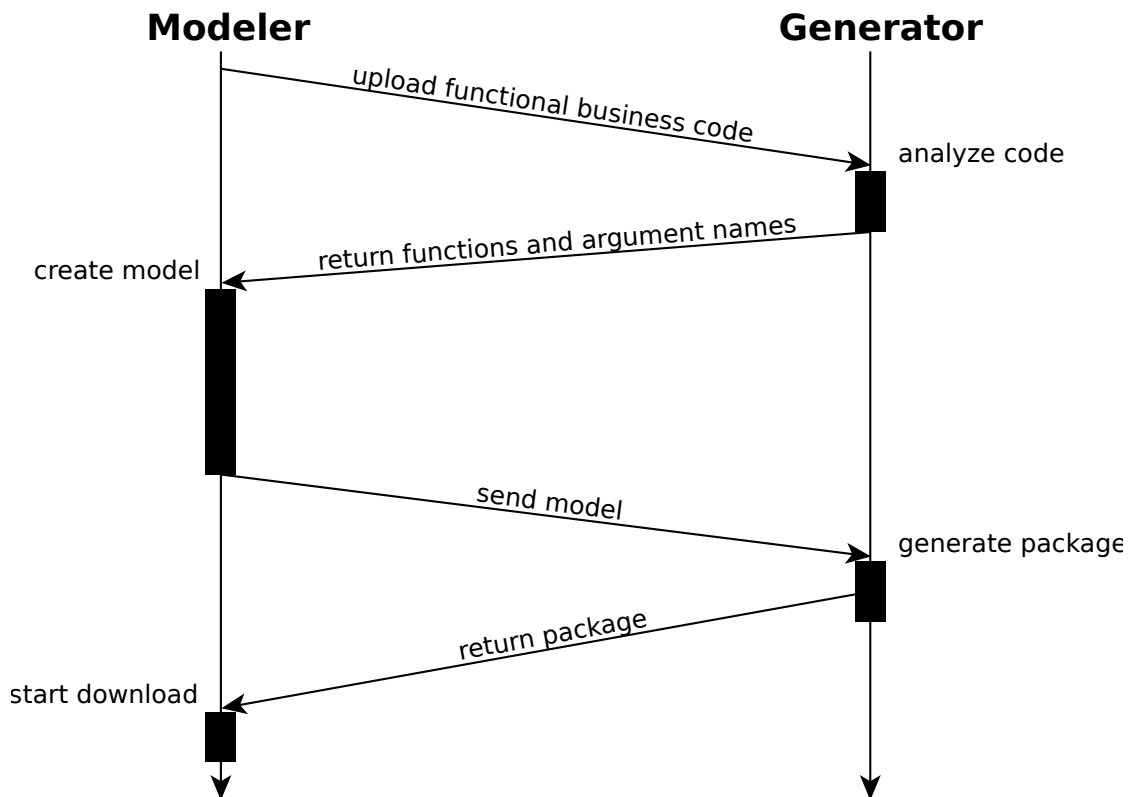


Figure 4.2: Communication between Modeler and Generator

on to import and edit the diagram saved in it. Additionally their *bpmn-js-properties-panel*⁴ library is integrated to provide users the ability to add properties to BPMN elements. The JavaScript task runner *Grunt*⁵ is used to build and run the Modeler.

The core component of the Generator uses the web framework *Flask*⁶ to communicate with the Modeler. *Jinja2*⁷ is used by the plugins as a template engine to define templates and render them into FaaS functions.

4.3 Modeler

This section provides detailed information about the implementation of the Modeler and illustrates its user interface with several screenshots. When starting up the Modeler via the command `grunt auto-build`, it automatically opens a tab in the browser with BPMN2FaaS running. First users are asked to provide a zip file containing their functional business code modules together with `requirements.txt` files listing the required dependencies via drag and drop. Listing 4.1 demonstrates the structure of such an input. Users have to store the `requirements.txt` files in separate folders. By naming those folders and the **Lanes** in the BPMN diagram equally, the Generator is able to assign each `requirements.txt` with its corresponding **Lane**. When uploading a file, the modeler checks its file type. Should it be different than zip, users are asked to repeat the upload until a zip file is detected.

Listing 4.1 Input package structure for business code modules and dependencies

```
project.zip
| - function_1/
| | - requirements.txt
| - function_2/
| | - requirements.txt
| - businessCode1.py
| - businessCode2.py
| - businessCode3.py
```

Next, users are asked whether they want to create a new BPMN diagram or open and edit an already existing one via drag and drop. Figure 4.3 shows a screenshot of BPMN2FaaS. In the center the BPMN diagram with its initial elements is illustrated. On the left side a tool palette allows users to edit their diagram and add BPMN elements to it. The properties panel is displayed on the right side. Its context is determined by the currently selected BPMN element. In the bottom left corner buttons are located, which provide users the ability to transform it into a FaaS function or export their diagram as XML or SVG.

⁴<https://www.npmjs.com/package/bpmn-js-properties-panel>

⁵<https://gruntjs.com>

⁶<https://flask.palletsprojects.com>

⁷<https://jinja.palletsprojects.com>

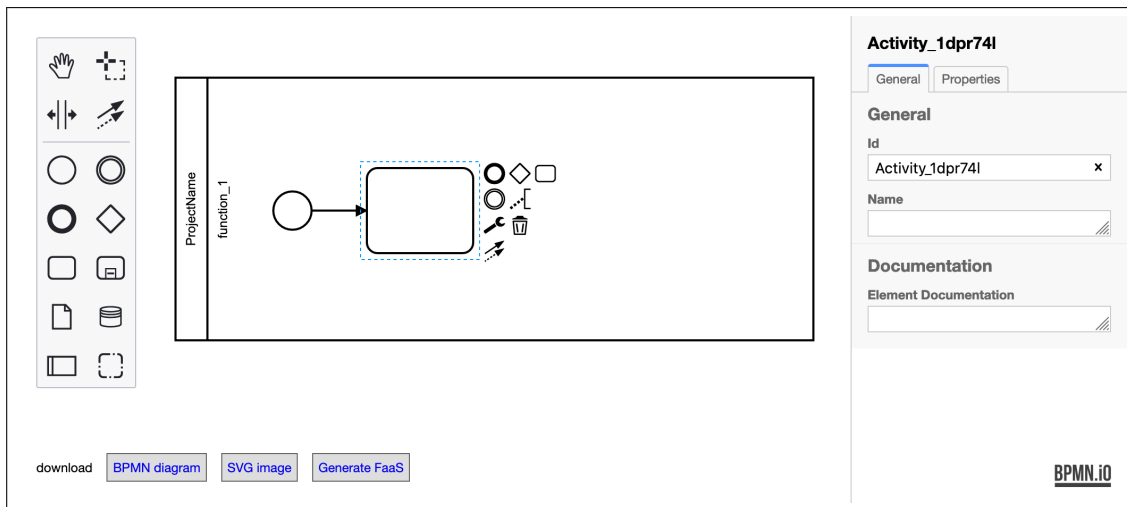


Figure 4.3: Creating a new diagram in BPMN2FaaS

4.3.1 Extending BPMN Properties Using Moddle

In *bpmn-js* all BPMN elements are represented as JSON objects, which hold attributes like position, size, name, id, BPMN element type, incoming and outgoing connectors, children elements and a `businessObject` with additional custom properties. Furthermore, those objects need to be extended with properties introduced in Section 3.3. The properties panel consist of different types of Hypertext Markup Language (HTML) elements, which allow users to interact with some of the properties. Those properties are not only needed by the Generator to generate FaaS functions, but also by the Modeler itself to check the state of the BPMN element to provide users an interactive panel by determining, which HTML elements need to be displayed.

*moddle*⁸ is a extension for *bpmn-js*, which allows developers to define meta-models for BPMN element types. Using a simple JSON structure, additional properties can be defined by declaring their name, type and whether to be stored as additional XML element or attribute when exporting the diagram. Properties defined with *moddle* can be simply accessed in `businessObject` by using their names as attributes. Furthermore, the names can be referenced when displaying HTML elements in the panel to store their data automatically in `businessObject`.

Listing 4.2 shows the definition of **Service Task** properties with *moddle* accordingly to the properties defined in the concept. In BPMN2FaaS every property defined with *moddle* is declared to be stored as XML attribute of the corresponding BPMN element.

With the help of *moddle* the defined properties can be accessed to check whether the trigger type of the **Service Task** exists and if it is of the type queue, as demonstrated in the first line of Listing 4.3. In case of a queue trigger, a checkbox is added to the properties panel to let the user decide whether it is a default queue or FIFO. In line 5 the name of the `fifo` property is provided, to store the state of the checkbox automatically in the `businessObject` of the selected BPMN element.

⁸<https://github.com/bpmn-io/moddle>

Listing 4.2 Meta-Model definition for Start Events using *moddle*

```

1 {
2   "name": "FaaSStartEvent",
3   "extends": [
4     "bpmn:StartEvent"
5   ],
6   "properties": [
7     {
8       "name": "schedule",
9       "isAttr": true,
10      "type": "String"
11    },
12    {
13      "name": "trigger",
14      "isAttr": true,
15      "type": "String"
16    },
17    {
18      "name": "fifo",
19      "isAttr": true,
20      "type": "Boolean"
21    }
22  ]
23 }

```

Listing 4.3 Accessing *moddle* properties and binding it with HTML elements (JavaScript)

```

1 if (element.businessObject.trigger && element.businessObject.trigger === TriggerTypeConstants.queue) {
2   entryFactory.checkbox(translate, {
3     id : 'fifo',
4     label : 'FIFO',
5     modelProperty : 'fifo'
6   });
7 }

```

Figure 4.4 illustrates the HTML elements used in the properties panel to edit the properties of **Service Tasks**. It consists of a grouped select element to choose the right trigger type and the checkbox created in Listing 4.3. When exporting the diagram, the set of properties with the demonstrated values are represented in XML as shown in Listing 4.4. Every property defined by *moddle* is marked with the tag `bpmn2faas`. This prefix is also set inside of the *moddle* JSON. They can be used as XML namespaces, to distinguish the context of multiple different set of properties.

Listing 4.4 XML representation of an Start Event with its properties

```

1 <bpmn2:startEvent id="StartEvent_1" bpmn2faas:trigger="queue" bpmn2faas:fifo="true">
2   <bpmn2:outgoing>Flow_0n2io5b</bpmn2:outgoing>
3 </bpmn2:startEvent>

```

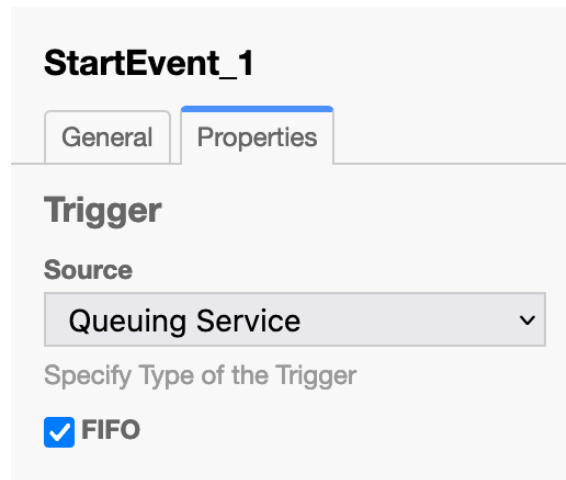


Figure 4.4: Properties Panel for Start Events

4.3.2 Adding HTML Elements to the Properties-Panel Using Factories

The factory module is part of the *bpmn-js-properties-panel* library and provides developers an interface for HTML elements like text inputs, checkboxes, select options or toggle switches, which can be added to the properties panel. Listing 4.3 shows an example on how to create a checkbox. All HTML elements created by the factory use the `modelProperty` option to store the state of the element in the specified attribute of the BPMN element. In addition to the factory provided by *bpmn-js-properties-panel*, BPMN2FaaS also uses a custom factory, which extends the original factory by enhancing HTML elements with additional features to provide a better user experience. The following presents the motivations behind each new feature and describes briefly how they are realized.

Grouped Selection

In contrast to standard select options, grouped select options allow to cluster their entries. When user have to chose an option from a list of services, grouped options can aid the selection by categorizing similar services under a common generic term. Furthermore, grouped select options help user to differentiate between two option, which have the same name, but a different context. Such a scenario could arise if users provide multiple business code modules, which might contain functions with the same name. In this case functions can be differentiated by creating groups of functions for each uploaded module.

The factory of *bpmn-js-properties-panel* uses the *domify*⁹ library, which turns HTML strings into Document Object Model (DOM) elements, to construct the HTML elements. Therefore the grouped selection is simply implemented by extending the options argument structure to define the entries inside of groups and by wrapping the options by `<optgroup label="myCategory">` for each group.

⁹<https://www.npmjs.com/package/domify>

Default Text

When creating select options, users are initially presented the first option, when the list is still collapsed, although no option is selected yet. HTML offers a `selected` attribute for options, to initially set a predetermined option. A better solution is to provide users with a visual feedback showing that no option is selected yet. For this case default texts are introduced to the select option version of the custom factory. Every time the list of options is created, an additional option is inserted, which has no value and displays a default text like "Choose Service...". With the help of the attributes `disabled` and `hidden` this option can not be selected again and will be hidden from the list, once another option has been selected. The same feature is also applied to grouped select options.

OnChange

Every time a user interacts with a HTML element for the first time, the respective property is added to the selected BPMN elements `businessObject` by adding an attribute to it. Changing the value of the property simply results in overwriting the previously created attribute. But properties like arguments depend on the chosen service call. Changing the `serviceCall` property leads to already defined arguments to become invalid. In the worst case the former service call had required more arguments than the new one. Resetting the arguments after changing the service call does not solve the problem, because some arguments might be left untouched. This can lead to service calls with too many arguments. Therefore, a mechanism is needed, which is able to detect changes of the HTML elements value and cascade deleting operation on attributes, which are dependent on the changed property. Unfortunately, this feature is not supported by the original factory. Thus, the custom factory needs to extend its elements with this feature to prevent such scenarios. This is done by inserting the `selected` attribute in each HTML element, which invokes events and executes predefined callback functions, whenever the value of the element changes.

4.4 Generator (Core)

The implementation of the core is entirely written in Python and consists of an API reachable via HTTP endpoints, the Function Extractor and Plugin Manager. The core serves as back-end of the Modeler and as intermediary between Modeler and the corresponding plugin. Its task is to extract the names and arguments of functions provided by business code modules and start the chosen plugin to generate the respective deployment package. The following sections describe those three modules in more detail.

4.4.1 HTTP Endpoints

`app.py` serves as the main module and is therefore the starting point of the core module. Using `flask` the core can be started using the command `python -m flask run --port 8001`. First, the two routes `upload-files` and `generate` are exposed for HTTP PUT methods. The route `upload-files` is invoked by the Modeler whenever users provide their input in form of a zip package containing business code modules and dependencies. Additionally, a session cookie is included in this request,

which is generated by the Modeler every time the browser opens or refreshes the Modeler. This cookie is later used by the `generate` route to associate the model with the right files. This is done by creating a workspace in a temporary folder named after the session cookie. Inside this workspace the received zip package is extracted and analyzed by the *function extractor*. The result of this analysis is sent back to the Modeler as a response.

After finishing the model and clicking on the generate button a request is sent to the `generate` path, which includes the XML representation of the provider-agnostic model and the resource descriptor in form of an JSON, storing the target platform and additional provider-specific properties. Then a generator from the `Generator` class, provided by the Plugin Manager, is instantiated with the path to the right workspace and the target platform. The generator loads the correct plugin and starts it. When the plugin is finished, a path to the generated deployment package is returned to the core component to forward the package to the Modeler.

4.4.2 Function Extractor

The goal of the Function Extractor component is to analyze the business code modules and extract all function names and their arguments. The Function Extractor receives the path of the incoming zip package as an input. This package is extracted into the same folder. The analysis starts by iterating over every business code module and storing the module names inside of a JSON. Each module declared in it consist of functions, which themselves consist of arguments. The standard library *ast* is used to create an abstract syntax tree for each module accordingly to the grammar of Python. Each node in this tree is matched with the `FunctionDef` class of *ast* to check if it is a function. For every function node the name and its arguments are extracted and stored in its respective module.

4.4.3 Plugin Manager

The Plugin Manager is implemented in *core.py*. When creating an instance, the needed plugin is imported dynamically with the help of the library *importlib*. Every plugin must be named accordingly to the pattern `bpmn2faas_{provider}_plugin_{runtime}`, where the runtime is Python in this case. Additionally, every plugin must provide a `Plugin` class inside its *main.py* and implement the function `generate`, in order to let the generator import and start every plugin homogeneously. If the plugin is not present, a `ModuleNotFoundError` exception is raised and the operation is aborted.

4.5 Plugins

In the scope of this thesis, two plugins have been implemented, generating deployment packages for the platforms AWS Lambda and Azure Functions. Every plugin has to be installed in the plugin folder of the core, in order to be imported. The goal of a plugin is to create deployment packages by mapping provider-agnostic properties to provider-specific properties and transforming them to code.

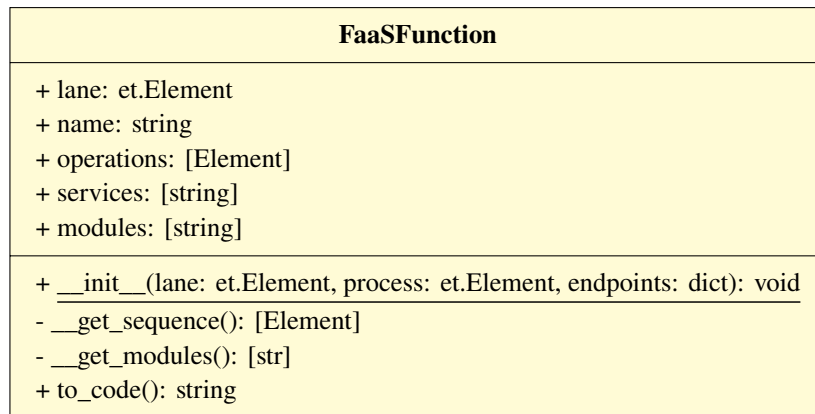


Figure 4.5: UML class diagram of FaaSFunction.

First, plugins parse the BPMN diagram, with the help of the `xml.etree.ElementTree` module, from XML to an element tree. Elements of this tree (`et.Element`) can be derived using *XPath* expressions. Next, the BPMN **Pool** needs to be extracted to process its **Lanes** and transform each one of them into FaaS functions. Listing 4.5 shows the process section of the xml representation, which holds all BPMN elements and their properties. The `laneSet` element represents the **Pool**, consisting of its **Lanes**. Each **Lane** stores a list of IDs referencing the BPMN elements of its workflow.

Listing 4.5 Extraction of BPMN diagram represented as XML

```

1 ...
2 <bpmn2:process id="Process_1" isExecutable="false">
3   <bpmn2:laneSet id="LaneSet_1qthya9">
4     <bpmn2:lane id="Lane_0lzqw52" name="function_1">
5       <bpmn2:flowNodeRef>StartEvent_1</bpmn2:flowNodeRef>
6       <bpmn2:flowNodeRef>Activity_1dpr74l</bpmn2:flowNodeRef>
7     </bpmn2:lane>
8   </bpmn2:laneSet>
9   <bpmn2:startEvent id="StartEvent_1" bpmn2faas:trigger="queue" bpmn2faas:fifo="true">
10     <bpmn2:outgoing>Flow_0n2io5b</bpmn2:outgoing>
11   </bpmn2:startEvent>
12   <bpmn2:task id="Activity_1dpr74l" bpmn2faas:function="module1:hello_world">
13     <bpmn2:incoming>Flow_0n2io5b</bpmn2:incoming>
14   </bpmn2:task>
15   <bpmn2:sequenceFlow id="Flow_0n2io5b" sourceRef="StartEvent_1" targetRef="Activity_1dpr74l" />
16 </bpmn2:process>
17 ...

```

For each **Lane** an instance of the class `FaaSFunction` is created. The class members of `FaaSFunction` are listed in Figure 4.5. Instances of this class take the XML objects `lane` and `process` as input to determine the BPMN elements of its **Lane** and order them accordingly to the workflow. `FaaSFunction` uses a *Jinja2* template, composed out of multiple smaller templates, to create the FaaS function. To render the template, this class needs to process each element and transform them into snippets, which are injected into the template.

The plugins implement classes, which represent the BPMN element types used inside of **Lanes** to model workflows. Figure 4.6 illustrates an UML class diagram, which shows the relationships between the classes. Only the common class members are listed, which are used by both plugins. The class `Element` serves as superclass for the subclasses `Task`, `ServiceTask` and `StartEvent`. It provides fundamental attributes and methods for every BPMN class by inheritance. Every BPMN class implements the function `to_code()`, which transforms the BPMN element and its properties into code snippets using their own templates.

After the generation of the FaaS function is done, the main module retrieves the list of dependencies and installs them into a folder named `package`, in case AWS Lambda has been chosen as the target platform. For Azure this procedure is dismissed, since the platform of Azure Functions itself handles dependencies during deployment automatically.

4.5.1 BPMN to Code Transformation

This section presents the templates used in both plugins and describes how they are rendered to a FaaS function by transforming BPMN elements together with their properties to code. *Jinja2* templates can be rendered by using the `render()` function with a dictionary as input, which defines the data to be injected into a template. In *Jinja2* syntax control structures like for-loops appear in `{% . . %}` statements, while Python expressions are defined in `{{ . . }}`. When rendered, variables and expressions are substituted by the template engine with values of equally named attributes defined in the input dictionary. In the following listings of this section variables and expressions will be highlighted in purple.

FaaS Function

Listings 4.6 and 4.7 illustrate the main templates used by each plugin to generate FaaS functions. While processing each BPMN element, the `FaaSFunction` class determines a list of modules used by its **Task** elements. For each module an import statement is inserted by the template engine. Similarly, **Service Task** elements are retrieved to determine their services and transform them into provider specific terminology. In AWS all clients are created by the *boto3* SDK. Additionally one client per service type is enough, because *boto3* created clients on service level. Specific service instances are specified by service call parameters. Whereas in Azure, clients are directly bound to specific service instances, leading to one client for each **Service Task** in the worst case. Additionally clients are not created directly by the *azure* SDK, but its submodules. To prevent importing the whole *azure*, the `FaaSFunction` class provides all import statements for each service type. Moreover different service calls of the same service type might require different clients. For instance, service calls retrieving a list of all containers in a blob storage require a client on storage level, whereas service calls retrieving a list of all object contained by a specific container require a client on container level. Therefore the Azure template defines imports to client classes, instead of creating them. The clients themselves are created in the body of the handler for each **Service Task**.

The goal of each template is to generate FaaS functions as similar as possible to the template FaaS function provided by each platform. For those reasons in AWS the handler function is named *handler*, whereas in azure the handler is named *main* and its arguments are typed. If (FIFO) Queue

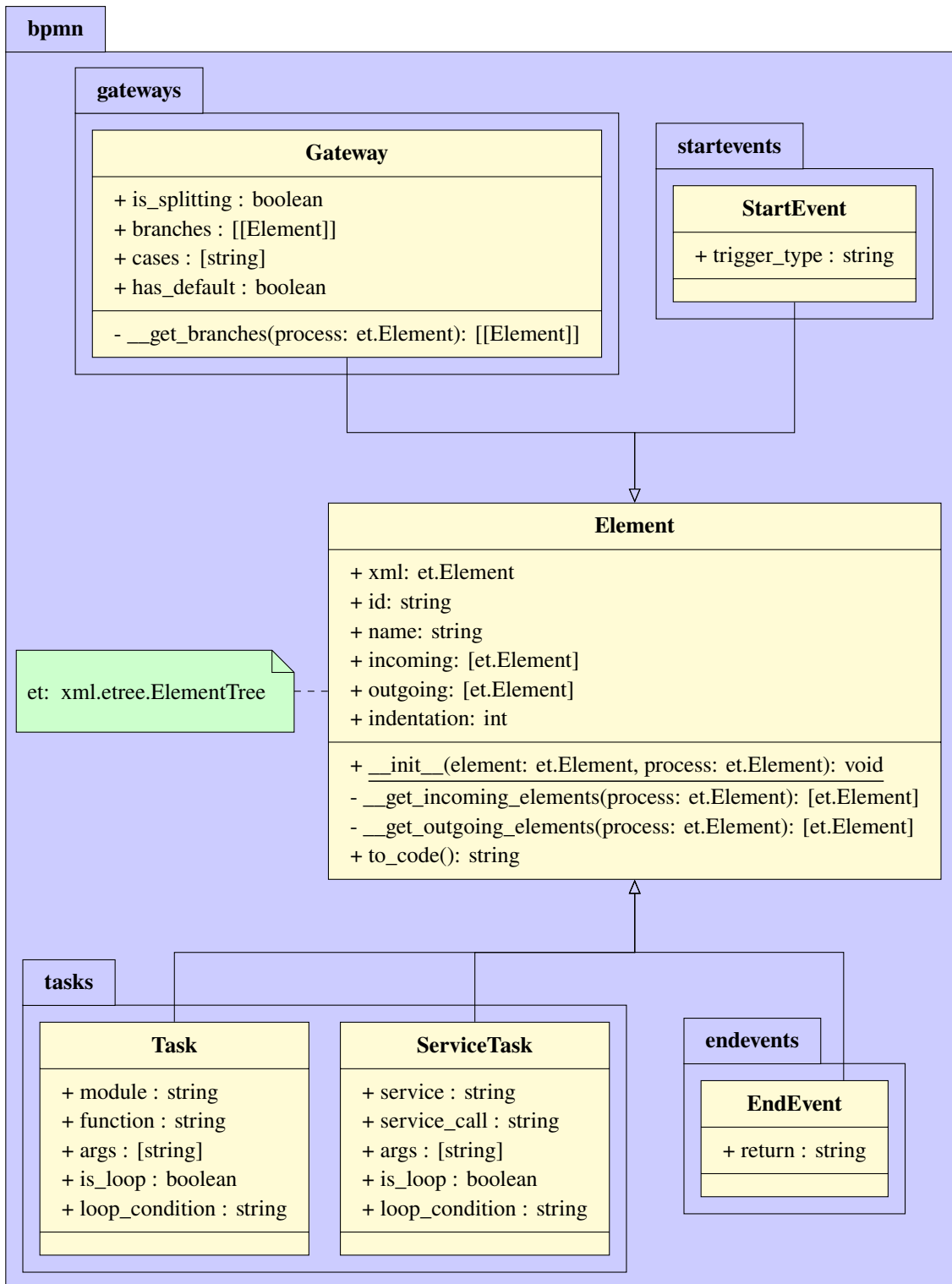


Figure 4.6: UML class diagram of BPMN classes used by the plugins.

4 Implementation

Listing 4.6 FaaS Function template for AWS (Python + Jinja)

```
1 import boto3
2
3 {% for module in data.modules %}import .{{ module[:-3] }}
4 {% endfor %}
5 {% for client in data.clients %}{{ client }}_client = boto3.client('{{ client }}')
6 {% endfor %}
7
8 def handler(event{% if data.is_batch %}s{% endif %}, context):
9     {% if data.is_batch %}for event in events:
10        {% endif %}    event = to_cloudevents_schema(event)
11        {% for op in data.operations %}{% include 'indentation.jinja' %}{{ op.to_code() }}
12        {% endfor %}
13
14 {{ data.start_event.to_code() }}
```

Listing 4.7 FaaS Function template for Azure (Python + Jinja)

```
1 import logging
2
3 {% for module in data.modules %}import .{{ module[:-3] }}
4 {% endfor %}
5 import azure.functions as func
6 {% for client in data.clients %}{{ client }}
7 {% endfor %}
8
9 def main(event: func.{{ data.input_class }}, context: func.Context):
10    event = to_cloudevents_schema(event)
11    {% for op in data.operations %}{{ op.to_code() }}
12    {% endfor %}
13
14 {{ data.start_event.to_code() }}
```

or PubSub is the chosen trigger type, in AWS the plural of the term “event” is used to indicate batched events and a for-loop is introduced to process each event individually. Afterwards in both templates the function `to_cloudevents_schema(event)` is inserted to transform the incoming event to the cloudevents schema. The service type specific definition of this function is placed at the bottom of the template, which is generated by the **Service Task**. After the transformation of the event schema all operations (business functions and service calls) are listed one after another. Depending on the use of for-loops and if-statements the individual indentation values are increased and inserted by using the template shown in Listing 4.8.

Listing 4.8 Template for introducing Indentation (Jinja)

```
1 {% for indent in range(op.indentation) %}    {% endfor %}
```

Tasks

Listing 4.9 shows the template to render business functions. It simply inserts modules, functions and arguments and stores the output in an attribute of the event named after the **Tasks** name or id if no name is provided. In case of multiple executions of the business function, a for loop is inserted by the template in Listing 4.10 and the output is added to a list of output analogously.

Listing 4.9 Business Function template (Jinja)

```

1 {% if data.is_loop %}event['{{ data.task_name }}'] = []
2 {% include 'loop.jinja' %}
3     event['{{ data.task_name }}'].append({{ data.module }}.{{ data.function }}({{ data.args }}))
4 {% else %}event['{{ data.task_name }}'] = {{ data.module }}.{{ data.function }}({{ data.args }})
5 {% endif %}

```

Listing 4.10 for-loop template (Jinja)

```

1 {% if data.is_loop %}    for {{ data.loop_condition }}: {% endif %}

```

Service Tasks

Each supported service call has its own template. Finding the corresponding template and mapping the service call from provider-agnostic to provider-specific is done implicitly by the filename of the template. Each template is named after the provider-agnostic service call and contains its provider-specific code representation. Listing 4.11 and Listing 4.12 show the templates of the provider-agnostic service call `send_message(queue_name, message)` for AWS SQS and Azure Queue Storage respectively. Due to presentation reasons, the templates demonstrated only focus on the service calls and their clients, but do not display indentations and their combination with for-loops, which are analogously the same as in Listing 4.9. On Azure's side the queue client requires a connection string, access from the resource descriptor, and the queue name. The service call itself only needs the message to be sent. Whereas in the AWS version, no arguments are needed to bind to the client, since the arguments of the service call store the target queue and message. Due to the target queue having to be specified as an Uniform Resource Locator (URL), the service call `get_queue_url` has to be executed additionally to transform the queue name into its corresponding URL.

Listing 4.11 send_message template for AWS (Jinja)

```

1 queue_url = sqs_client.get_queue_url(QueueName={{ data.args[0] }})
2 event['{{ data.task_name }}'] = sqs_client.send_message(queue_url['QueueUrl'], {{ data.args[1] }})

```

Listing 4.12 send_message template for Azure (Jinja)

```

1 queue_client = QueueClient.from_connection_string({{ data.connection_string }}, {{ data.args[0] }})
2 event['{{ data.task_name }}'] = queue_client.send_message({{ data.args[1] }})

```

Start Events

To support provider-agnostic event schemes, every generated FaaS function has to include the `to_cloudevents_schema` function, which translates every kind of object representing the incoming event with provider-specific schema to a provider-agnostic dictionary applying to the CloudEvents specification. The mapping is handle the same as for service calls: For each trigger type there is a service-specific template named after the provider-agnostic terminology of the service type. Listing 4.13 shows the `to_cloudevents_schema` function defined for timer triggers in AWS. In addition to the required attributes mentioned in Section 3.3.2, timer trigger events also have the attributes `datacontenttype`, specifying the type of the provider-specific event stored in data, and `time` storing a timestamp.

Listing 4.13 Template for Timer Trigger Events in AWS (Jinja)

```
1 def to_cloudevents_schema(event):
2     return {'id':          event['id'],
3            'source':      event['source'],
4            'specversion': '1.0',
5            'type':        'com.amazonaws.s3.'+event['detail-type'],
6            'datacontenttype': 'dict',
7            'time':        event['time'],
8            'data':        event}
```

End Events

The template shown in Listing 4.14 demonstrates how **End Events** are represented as code. Due to **Gateways** splitting the workflow, a FaaS function could have multiple return statements placed in different branches. Therefore, the indentation template needs to be included to shift the return statement dynamically to the right, depending on the nesting provided by the **Gateways**.

Listing 4.14 Template for End Events (Jinja)

```
1 {% include 'indentation.jinja' %} return {{ data.return }}
```

Gateways

Depending on the number of incoming and outgoing connections, first, **Gateways** have to be classified either as splitting or joining **Gateways**. Next, all elements for each branch have to be determined by traversing in all outgoing directions until the corresponding joining **Gateway** has been reached. The corresponding joining **Gateway** is found by increasing a counter each time a splitting **Gateway** has been visited and decreasing it for each joining **Gateway**. The corresponding joining **Gateway** is found if counter is zero.

Also depending on the number of outgoing connections is the branch type. If a splitting **Gateway** has two outgoing sequence flows it represents an if-statement, in case there are more than two, it represents a switch-statement. Listing 4.15 shows the template used to render if-statements. Splitting **Gateways** are responsible to determine all **(Service) Tasks** for each branch, which need

to be embedded into the body of their if-statement, and transform them into code. Similarly, Listing 4.16 demonstrates the template used to simulate switch-statements, since Python does not support switch-statements. While **Gateways**, which represent if-statement, only have the cases true or false, switch-statements have different cases for each branch. Additionally, there might be a default branch, representing all other cases not fulfilling the other ones. For all non default branches the template generates an `elif`-block, with the exception for the first if-block, comparing the condition with the corresponding case. In case on of the outgoing sequence flows has been marked as default branch, the template adds an additional `else`-block.

Listing 4.15 Template for if-statements (Jinja)

```

1 if {{ data.condition }}:
2     {% for op in data.operations[0] %}{% include 'indentation.jinja' %}{{ op.to_code() }}
3     {% endfor %}
4 else:
5     {% for op in data.operations[1] %}{% include 'indentation.jinja' %}{{ op.to_code() }}
6     {% endfor %}

```

Listing 4.16 Template for simulated switch-statements (Jinja)

```

1 {% for case in data.cases %}{{ 'if' if loop.first else 'elif' }} {{ data.condition }} == {{ case }}:
2     {% for op in data.operations[loop.index0] %}{% include 'indentation.jinja' %}{{ op.to_code() }}
3     {% endfor %}
4 {% endfor %}{% if data.has_default %}else:
5     {% for op in data.operations[-1] %}{% include 'indentation.jinja' %}{{ op.to_code() }}
6     {% endfor %}{% endif %}

```

5 Related Work

This chapter presents existing projects, which provide solutions to similar problems as faced by this thesis and points out their similarities and differences. The sections of this chapter cluster those projects by differentiating them between projects tackling challenges on portability in the context serverless computing and projects working on model-driven code generation based on workflow models.

5.1 Serverless Portability

One way to overcome the disadvantages of vendor lock-ins is portability of serverless applications. The projects described in this section developed concepts, in which serverless applications can be modeled in a provider-agnostic way. Transforming such a model to representations of different target platforms helps developers to deploy their applications on clouds of different providers and thus becoming independent of just one single provider.

Serverless Framework [Ser22] and *Terraform* [Ter22] are open source deployment automation tools. Via Command Line Interface (CLI) developers are able to build auto-scaling serverless applications and deploy them across multiple cloud providers. Same as in this work, both use declarative deployment modeling to generate provider-specific deployment packages. Their models also have in common, that they use as many provider-agnostic properties as possible, but also facing the problems of the heterogeneous landscape of cloud computing, leading to the need of additional provider-specific properties, similar to the resource descriptor introduced in Section 3.4, thus making their models also not fully provider-agnostic. Additionally, both tools also use a plugin architecture, where one plugin for each cloud provider is responsible provider-specific tasks. The main difference to this work is, that both tools operate on deployment packages at the level of serverless applications as a whole, whereas this work focuses on deployment packages of FaaS functions, which is only a subset of possible components in serverless architectures.

Serverless Applications PORTability assessment (SEAPORT) is a method developed by Yussupov et al. [YBKL20], which confronts issues of serverless portability by automatically assessing the portability of a given serverless application by analyzing its deployment model. In contrast to this work, SEAPORT does not only support provider-specific deployment models as presented in Section 3.6, but also third-party model from deployment automation tools like the already mentioned Serverless Framework and Terraform. The concept of assessing whether and to which extend a given serverless application is portable to a specified target platform is divided into the three steps illustrated in Figure 5.1. In the first step a user provides his deployment package, which also containing code artifacts for FaaS components. Same as in this thesis, SEAPORT only works with declarative models. In Step 2 the technology-specific deployment model is transformed into canonical model. By using canonical models, SEAPORT's concept and the concept introduced in

Chapter 3 benefit both from reduced number of translations from source to target platform. The last step performs the portability assessment based on the canonical model and the specified target platform by splitting the task into evaluating the portability of each component defined in the deployment model (Step 3a) and analyzing the code artifact for potential pitfalls (Step 3b). In Step 3a SEAPORT classifies the service type of each component provider-agnostically by creating categories for related service type with common properties shared between different cloud providers. This is done for invoked services as well as for service producing events to trigger other components. The same procedure was applied during this work to find matching properties to create provider-agnostic event schemes and service calls, as described by Figure 3.5. But in contrast to SEAPORT, where a similarity score determines the best fitting service alternative of the target cloud provider among multiple choices, in this work each service type has exactly one alternative. Step 3b analyzes code artifacts to check whether incompatible service calls are involved. Similar issues were also discovered during this work, where the problem with AWS DynamoDB and Azure’s Cosmos DB explained in Section 3.3.1 is just one example of many. Same as Serverless Framework and Terraform, SEAPORT’s assessment is on the level of serverless applications, whereas this thesis operates on the finer-grained level of FaaS functions components.

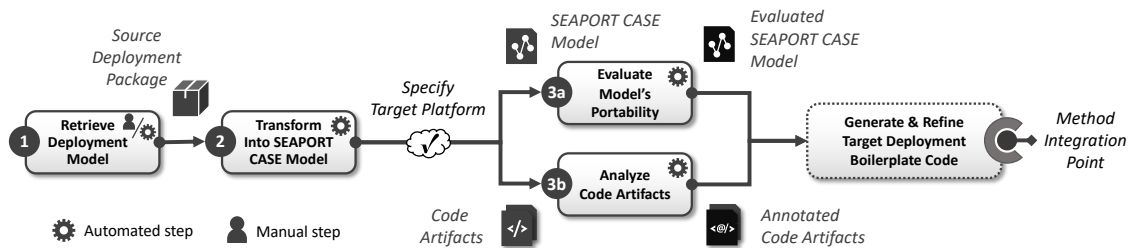


Figure 5.1: SEAPORT method [YBKL20]

In contrast to BPMN2FaaS following the concept of BPMN-based modeling of FaaS deployment packages, *BPMN2FO* [YSBL22] follows the concept of BPMN-based modeling and deployment of serverless function orchestrations. Also the motivations are similar: BPMN2FO faces the challenges of heterogeneous function orchestration tools by creating a technology-agnostic modeling approach for serverless function orchestrations. Shifting the finer-grained view from single FaaS functions to orchestrations of multiple functions leads to **Tasks** modeling a whole serverless function instead of just one operation of it.

In [HMW22] Hartauer et al. made an empirical investigation about the differences of FaaS functions concerning their implementation, packaging and deployment based on a experiment. First three hello world FaaS functions with HTTP triggers were implemented for the platforms AWS Lambda, Azure Functions and Google Cloud Functions (GCF) and compared based on (i) the handler implementation, (ii) logging capabilities and (iii) access to other services of the respective cloud provider. Each one of the handlers defines their parameters differently in terms of order and data. These differences are also considered in the templates demonstrated in Section 4.5.1, where provider-specific templates are used to render the handler function accordingly and transform provider-specific event schemes into provider-agnostic event schemes using CloudEvents format. Both, AWS Lambda and GCF use the default logging functions of each programming language, for example `console.log()` in JavaScript, whereas Azure Functions uses `context.log()` provided

by its second parameter. The concept introduced in this work does not distinguish between the different logging techniques. Logging statements included in business code modules makes them invalid, since they are required to be implemented completely in a provider-agnostic manner. The biggest difference between FaaS functions of different platforms lies in their service calls. Each platform provides their service calls via custom APIs and SDKs. In this thesis this problem is faced by introducing custom provider-agnostic service calls for developers lacking provider-specific expertise and mapping them to their provider-specific representations using templates.

Another project sharing the same motivations as this work, in terms of providing developers with limited knowledge about FaaS an easier way to develop serverless applications across different FaaS platforms, is *FLY* [CDN+20]. *FLY* is a programming environment for scientific computing applications on FaaS platforms, consisting of its own Domain-specific Language (DSL), compiler and deployment tool. Scientific computing involves solving computing-intensive problems, which requires distributed computational power of paradigms like serverless computing. However, typical scientific applications do not require general purpose services, e.g. storage. With the help of the programming language *FLY* developers are able to implement *FLY* functions with provider-agnostic syntax. On one hand, using a DSL tailor-made for scientific computing minimizes the additional knowledge required by experts on this field to create applications suitable for FaaS platforms, but on the other hand it introduces severe limitations by not supporting other general purpose cloud services, thus only covering a very small subset of all possible FaaS functions, which can be implemented. However, the concept presented in this thesis limits the amount of supported services only if service alternatives of different cloud providers do not have enough matching properties to be modeled provider-agnostically.

5.2 Graphical Workflow Modeling and Model-driven Code Generation

According to [LWSH19] there is a very limited number of FaaS development tools other than CLIs and SDKs offered by most cloud providers. Due to this reason, this section presents related works in the field of model-driven code generation based on workflow models.

PHYSICS [KAC+22], being the most relatable one to this work, is a framework, which enables developers an easier way to create workflow models and FaaS functions for target platforms supporting OpenWhisk. Instead of using BPMN to model FaaS workflows, *PHYSICS* uses *Node-RED* [Ope22], a low-code programming tool for event-driven applications developed by IBM. *PHYSICS* modified *Node-RED* allowing their users to develop FaaS functions tailor-made for OpenWhisk. Using *Node-RED*, *PHYSICS* is limited to only being able to generate FaaS functions written in JavaScript, whereas *BPMN2FaaS* is extendable to support multiple programming languages, but also additional FaaS platforms in the future. Another difference between *PHYSICS* and *BPMN2FaaS* is that custom business functions can be implemented directly in *Node-RED* using a rich text editor. Furthermore, *PHYSICS* offers predefined code patterns, which can be used as building blocks when defining the workflow model. It is also possible to save functions as macros, which allows reuse of frequently used functions. This way developers have to implement less custom business code.

Pegasus [DVJ+15] is a workflow management system for scientific applications. It allows to create abstract workflow models and map them to highly scalable and distributed computing infrastructures like clusters, grids [FK03] or clouds. In contrast to using BPMN, *Pegasus* uses Directed Acyclic

Graphs (DAGs) to model workflows. Both, third-party graphical workflow composition tools as well as custom CLIs and APIs provided by Pegasus can be used to generate DAGs. By only focusing on scientific computing, same as FLY introduced in the previous section, applications managed by Pegasus do not support interactions with other general purpose cloud services. Therefore, Pegasus is only able to deploy and execute IaaS offerings as Amazon Elastic Compute Cloud (EC2).

ProcessEngine.io [5Mi19] is an open-source workflow engine for BPMN-based business processes provided by the team of 5Minds. It uses the same BPMN editor and properties panel. In contrast to using default **Task** elements, ProcessEngine.io uses **Script Tasks** to bind business code. **Service tasks** are used to execute either Representational State Transfer (REST) services via endpoints or programming language-independent external task workers stored in a task storage.

6 Conclusion and Outlook

This chapter concludes this work, which presented a multi-step concept for standards-based modeling and generation of FaaS deployment packages. During the Elaboration of this concept, the focus mostly laid on AWS and Microsoft Azure. Nonetheless, the concept is extendable for additional cloud providers offering FaaS platforms and applicable for all programming languages supported by various cloud platforms. The main contributions of this work are (i) a partially provider-agnostic workflow model, helping developers with limited knowledge in the field of serverless computing across different cloud providers and (ii) the transformation of the workflow model into a provider-specific FaaS deployment package. The workflow model consists of a graphical workflow diagram based on the widely known modeling standard BPMN, which is enhanced by provider-agnostic properties and business functions, and a provider-specific resource descriptor specifying service endpoints. This concept was realized by implementing the prototype BPMN2FaaS, which provides developers the ability to design workflow models in a BPMN editor and generate deployment packages for AWS Lambda and Azure Functions supporting the programming language Python. In the following, this chapter elaborates on limitations of the concept and its implementation faced during this work. At the end future work is presented to discuss further improvements on this work.

6.1 Limitations

As already discussed in Section 3.3.1, due to the heterogeneous service offerings in the landscape of cloud computing, ambiguity and insufficient amount of common properties between service alternatives of different cloud providers make it impossible to model all service types provider-agnostically. Supporting additional cloud platforms might aggravate the process of creating provider-agnostic service types. Moreover, provider-specific event schemes might not deliver sufficient attributes to create provider-agnostic schemes according to the specification of CloudEvents. This problem can be reduced by introducing provider-specific properties, which always comes with the caveat of requiring additional knowledge from developers.

6.2 Future Work

BPMN2FaaS just being a prototype already implies its unfinished state. Therefore, this section presents additional features, which can be added in the future to further optimize BPMN2FaaS and its concept by improving the user experience before and during the development of FaaS functions.

6.2.1 Documentation & Tutorials

Everything developers can think of working with, e.g. frameworks, libraries and even programming languages have their own documentations and tutorials providing detailed information on their use. Introducing provider-agnostic event schemes and service calls developers do not need to gain provider-specific knowledge for each service type alternative anymore. Nonetheless, developers still need to learn about provider-agnostic representations, although the goal is to use widely known terminologies, which need to be easy to understand. Therefore, BPMN2FaaS requires detailed information about event schemes, service calls, BPMN elements using to model FaaS functions. Developers should reach such documentations through websites, e.g. GitHub repositories or some kind of “docs/wiki” pages, but also interactively directly in the editor, e.g. via question mark icons. Additionally, tutorials can help developers to implement correct business code modules and describe patterns and anti-patterns for BPMN modeling as recommended in [RPH08].

6.2.2 New Project Templates

When creating a new serverless application using the Serverless Framework, developers can choose from a variety of project templates as starting point for their development, categorized in target platform and trigger type. This procedure could also refine the process of creating new workflows with BPMN2FaaS. If developers choose to creating a new diagram, instead of opening an already existing one, they could be asked what kind of FaaS function they to create in terms of trigger type. According to the choice, the initial BPMN diagram could already come with the corresponding predefined **Start Event**.

6.2.3 Predefined Tasks

Using Node-RED, PHYSICS is able to provide predefined code patterns, which can be embedded into the workflow diagram. This mechanism improves low-code capabilities by allowing developers to save time on implementing reoccurring business functions. Therefore, it is a good idea to add this feature to BPMN2FaaS by offering additional functions when setting the `function` property of default **Task** elements. Although logging is not a service call, but still handled provider-specifically as stated in [HMW22], this problem can be solved by also offering a provider-agnostic logging function for default **Tasks** and mapping it to its provider-specific representation during code generation.

6.2.4 Rich Editor for Direct Implementations

A much more convenient way of implementing business code modules is using rich editors with syntax highlighting embedded in the BPMN editor. This way business functions can also be embedded in the same file as the XML representation of the BPMN diagram. ProcessEngine.io, presented in Section 5.2, also uses this technique. This enables developers to adapt their code easier and be more aware about which exact business function is executed by each **Task**.

6.2.5 Additional Support for Services Types, Runtimes and Cloud Providers

Due to the limited scope of this thesis, the amount of supported service types, programming languages and cloud providers by BPMN2FaaS is restricted. Adding service triggers and service calls allows developers to create FaaS functions in a bigger spectrum. Portability can be further increased by generating deployment packages suitable for more target cloud providers. Supporting additional programming languages allows developers to be more flexible and use libraries, which might not be supported by other runtimes.

6.2.6 Additional BPMN Queue Artifact

Intuitively developers would attempt to use **Data Store** elements to model **Service Tasks** interacting with services like storage, relational or NoSQL databases. Unfortunately the official specification of BPMN in [Whi04] does not list a element to model channels representing messaging services like FIFO or PubSub queues. However, the specification allows extending BPMN with context-based artifact. Therefore, it is a good idea to offer developers a visual representation of channels.

6.2.7 Validation

Validation of BPMN diagram and its properties is very crucial. So far, no validation techniques were implemented in BPMN2FaaS, other than enabling package generation only if the target platform has been set and all **Service Tasks** state their connection strings (only for Azure Functions). Generating deployment packages only works under the assumption, that all components are specified correctly. Therefore, developers need visual feedback, which checks the completeness and correctness of the properties, before enabling the generation of deployment packages.

Bibliography

- [5Mi19] 5Minds. 2019. URL: <https://www.process-engine.io/docs/getting-started/> (cit. on p. 66).
- [AFG+10] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, M. Zaharia. “A View of Cloud Computing”. In: *Commun. ACM* 53.4 (Apr. 2010), pp. 50–58. ISSN: 0001-0782. DOI: 10.1145/1721654.1721672. URL: <https://doi.org/10.1145/1721654.1721672> (cit. on p. 17).
- [Ama22a] Amazon Web Services. *Amazon Resource Names (ARNs)*. 2022. URL: <https://docs.aws.amazon.com/general/latest/gr/aws-arns-and-namespaces.html> (cit. on p. 43).
- [Ama22b] Amazon Web Services. *Deploy Python Lambda functions with .zip file archives*. 2022. URL: <https://docs.aws.amazon.com/lambda/latest/dg/python-package.html> (cit. on p. 45).
- [Ama22c] Amazon Web Services. *Updating a function with additional dependencies*. 2022. URL: <https://docs.aws.amazon.com/lambda/latest/dg/nodejs-package.html#nodejs-package-dependencies> (cit. on p. 31).
- [Ama22d] Amazon Web Services. *Using Lambda with Amazon SQS*. 2022. URL: <https://docs.aws.amazon.com/lambda/latest/dg/with-sqs.html> (cit. on p. 38).
- [BS17] G. Balasubramanian, S. Shriver. *Choosing the Right DynamoDB Partition Key*. Feb. 2017. URL: <https://aws.amazon.com/de/blogs/database/choosing-the-right-dynamodb-partition-key/> (cit. on p. 36).
- [Cam21] Camunda. *BPMN 2.0 symbols - a complete guide with examples*. Oct. 2021. URL: <https://camunda.com/bpmn/reference/> (cit. on p. 23).
- [Cas18] K. Casey. *SPI model*. Oct. 2018. URL: <https://www.techtarget.com/searchcloudcomputing/definition/SPI-model> (cit. on p. 18).
- [CDN+20] G. Cordasco, M. D’Auria, A. Negro, V. Scarano, C. Spagnuolo. “FLY: A Domain-Specific Language for Scientific Computing on FaaS”. In: *Euro-Par 2019: Parallel Processing Workshops*. Ed. by U. Schwardmann, C. Boehme, D. B. Heras, V. Cardellini, E. Jeannot, A. Salis, C. Schifanella, R. R. Manumachu, D. Schwamborn, L. Ricci, O. Sangyoon, T. Gruber, L. Antonelli, S. L. Scott. Cham: Springer International Publishing, 2020, pp. 531–544. ISBN: 978-3-030-48340-1 (cit. on p. 65).
- [Clo22a] Cloud Native Computing Foundation (CNCF). *CloudEvents*. 2022. URL: <https://cloudevents.io/> (cit. on p. 38).
- [Clo22b] Cloud Native Computing Foundation (CNCF). *CloudEvents specification*. 2022. URL: <https://github.com/cloudevents/spec> (cit. on p. 39).

- [DVJ+15] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, K. Wenger. “Pegasus, a workflow management system for science automation”. In: *Future Generation Computer Systems* 46 (2015), pp. 17–35. ISSN: 0167-739X. DOI: <https://doi.org/10.1016/j.future.2014.10.008>. URL: <https://www.sciencedirect.com/science/article/pii/S0167739X14002015> (cit. on p. 65).
- [FK03] I. Foster, C. Kesselman. *The Grid 2: Blueprint for a new computing infrastructure*. Elsevier, 2003 (cit. on p. 65).
- [Fow99] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999. ISBN: 9788131734667. URL: <https://books.google.de/books?id=pqBXBlqwBAC> (cit. on p. 29).
- [HMW22] R. Hartauer, J. Manner, G. Wirtz. “Cloud Function Lifecycle Considerations for Portability in Function as a Service”. In: *Proceedings of the 12th International Conference on Cloud Computing and Services Science*. 2022, pp. 133–140 (cit. on pp. 64, 68).
- [JCBG21] A. Jindal, M. Chadha, S. Benedict, M. Gerndt. “Estimating the capacities of function-as-a-service functions”. In: *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion*. 2021, pp. 1–8 (cit. on p. 20).
- [KAC+22] G. Kousiouris, S. Ambroziak, D. Costantino, S. Tsarsitalidis, E. Boutas, A. Mamelli, T. Stamati. *Combining Node-RED and Openwhisk for Pattern-based Development and Execution of Complex FaaS Workflows*. 2022. DOI: [10.48550/ARXIV.2202.09683](https://doi.org/10.48550/ARXIV.2202.09683). URL: <https://arxiv.org/abs/2202.09683> (cit. on p. 65).
- [LRC+18] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, S. Pallickara. “Serverless Computing: An Investigation of Factors Influencing Microservice Performance”. In: *2018 IEEE International Conference on Cloud Engineering (IC2E)*. 2018, pp. 159–169. DOI: [10.1109/IC2E.2018.00039](https://doi.org/10.1109/IC2E.2018.00039) (cit. on p. 21).
- [LWSH19] P. Leitner, E. Wittern, J. Spillner, W. Hummer. “A mixed-method empirical study of Function-as-a-Service software development in industrial practice”. In: *Journal of Systems and Software* 149 (2019), pp. 340–359. ISSN: 0164-1212. DOI: <https://doi.org/10.1016/j.jss.2018.12.013>. URL: <https://www.sciencedirect.com/science/article/pii/S0164121218302735> (cit. on p. 65).
- [MG11] P. Mell, T. Grance. “The NIST definition of cloud computing”. In: *Recommendations of the National Institute of Standards and Technology* (Sept. 2011). DOI: [10.6028/nist.sp.800-145](https://doi.org/10.6028/nist.sp.800-145) (cit. on pp. 15, 17, 18).
- [Mic22a] Microsoft Azure. *Azure Functions Python developer guide*. 2022. URL: <https://docs.microsoft.com/en-us/azure/azure-functions/functions-reference-python#folder-structure> (cit. on p. 31).
- [Mic22b] Microsoft Azure. *Azure Functions Python developer guide*. 2022. URL: <https://docs.microsoft.com/azure/azure-functions/functions-reference-python> (cit. on p. 45).
- [Mic22c] Microsoft Azure. *Configure a connection string for an Azure storage account*. 2022. URL: <https://docs.microsoft.com/en-us/azure/storage/common/storage-configure-connection-string#configure-a-connection-string-for-an-azure-storage-account> (cit. on p. 42).

- [Mic22d] Microsoft Azure. *QueueMessage Class*. 2022. URL: <https://docs.microsoft.com/en-us/python/api/azure-functions/azure.functions.queuemessage?view=azure-python> (cit. on p. 39).
- [Ope22] OpenJS Foundation. 2022. URL: <https://nodered.org/docs/> (cit. on p. 65).
- [OST14] J. Opara-Martins, R. Sahandi, F. Tian. “Critical review of vendor lock-in and its impact on adoption of cloud computing”. In: *International Conference on Information Society (i-Society 2014)*. IEEE. 2014, pp. 92–97 (cit. on p. 15).
- [RPH08] T. Rozman, G. Polancic, R. V. Horvat. “Analysis of most common process modeling mistakes in BPMN process models”. In: *Eur SPI’2007 (2008)* (cit. on p. 68).
- [Ser22] Serverless. 2022. URL: <https://www.serverless.com/framework/docs> (cit. on p. 63).
- [Sou10] D. Soumow. *Windows Azure Platform*. May 2010. URL: <https://www.slideshare.net/soumow/windows-azure-platform-4321694> (cit. on p. 19).
- [TEPN20] D. Taibi, N. El Ioini, C. Pahl, J. R. S. Niederkofler. “Serverless cloud computing (function-as-a-service) patterns: A multivocal literature review”. In: *Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER’20)*. 2020 (cit. on p. 19).
- [Ter22] Terraform. 2022. URL: <https://www.terraform.io/language> (cit. on p. 63).
- [Whi04] S. A. White. “Introduction to BPMN”. In: *Ibm Cooperation 2.0 (2004)* (cit. on pp. 15, 21, 27, 69).
- [YBKL20] V. Yussupov, U. Breitenbücher, A. Kaplan, F. Leymann. “SEAPORT: Assessing the Portability of Serverless Applications”. In: *Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER 2020)*. SciTePress, May 2020, pp. 456–467. ISBN: 978-989-758-424-4. DOI: [10.5220/0009574104560467](https://doi.org/10.5220/0009574104560467) (cit. on pp. 21, 63, 64).
- [YSBL22] V. Yussupov, J. Soldani, U. Breitenbücher, F. Leymann. “Standards-based modeling and deployment of serverless function orchestrations using BPMN and TOSCA”. In: *Software: Practice and Experience* 52.6 (2022), pp. 1454–1495. DOI: <https://doi.org/10.1002/spe.3073>. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.3073>. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.3073> (cit. on p. 64).

All links were last followed on May 30, 2022.

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