

Universität Stuttgart

Architectural Principles and Decision Model for Function-as-a-Service

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik
der Universität Stuttgart zur Erlangung der Würde eines Doktors der
Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von
Vladimir Yussupov
aus Almaty, Kasachstan

Hauptberichter: Prof. Dr. Dr. h. c. Frank Leymann

Mitberichter: Prof. Dr. Kostas Magoutis

Tag der mündlichen Prüfung: 02. Februar 2024

Institut für Architektur von Anwendungssystemen

2024

ZUSAMMENFASSUNG

Cloud Computing hat die Art und Weise, wie moderne Anwendungen entwickelt und betrieben werden, revolutioniert. Anstatt die Infrastruktur vor Ort beizubehalten, beziehen viele Unternehmen bei der Entwicklung ihrer Anwendungen verschiedene Cloud-Angebote ein, um die Markteinführungszeit zu verkürzen und den erforderlichen Verwaltungsaufwand zu verringern. Dazu gehört die Nutzung traditioneller Cloud-Service-Modelle wie Infrastructure-as-a-Service (IaaS) oder Platform-as-a-Service (PaaS) sowie neuartiger Modelle wie Function-as-a-Service (FaaS), das die Entwicklung von Cloud-Anwendungen durch die Zusammenstellung feingranularer, auf FaaS-Plattformen bereitgestellter Funktionen mit einer Vielzahl von anbieterverwalteten Diensten ermöglicht, z. B. Datenpersistenz- und Messaging-Dienste. Die meisten Verwaltungsaufgaben für die Komponenten in FaaS-basierten Anwendungen liegen in der Verantwortung des gewählten Cloud-Anbieters, was jedoch zu einer stärkeren Abhängigkeit von den Produkten des Anbieters und deren Implementierungs- und Paketierungsanforderungen führt. Daher kann die Entwicklung von FaaS-basierten Anwendungen von einer stärkeren Fokussierung auf die Architektur der Anwendung anstelle von spezifischen Produkten profitieren, da diese sehr schnell veralten.

Diese Arbeit konzentriert sich auf verschiedene Aspekte des anbieterunabhängigen Designs von FaaS-basierten Anwendungen und ist motiviert durch die zunehmende Abhängigkeit der Komponenten in solchen Anwendungen von produktspezifischen Anforderungen. Um eine anbieterunabhängige Diskussion über die Anforderungen an die Bereitstellung und das Management der Komponenten zu ermöglichen, wird in dieser Arbeit eine Mustersprache eingeführt, die verschiedene Abwägungen für das Hosting von Anwendungskomponenten in der Cloud dokumentiert. Um die Klassifizierung und Auswahl von Komponenten in FaaS-basierten Anwendungen zu erleichtern, wird in dieser Arbeit ein Klassifizierungsrahmen für FaaS-Plattformen vorgestellt und ein Metamodell für den Klassifizierungs-Framework eingeführt, das diese Konzepte für andere Komponententypen in solchen Anwendungen verallgemeinert. Darüber hinaus stellt diese Arbeit einen standardbasierten Modellierungsansatz für die Spezifikation von Funktionsorchestrationen und deren Transformation in anbieterspezifische Formate sowie einen automatisierten Ansatz zur Extraktion von Funktionscode und dessen Paketierung für unterschiedliche FaaS-Plattformen, um die Wiederverwendbarkeit von Funktionen zu erleichtern.

Um diese Beiträge gemeinsam nutzen zu können, wird in dieser Arbeit auch die GRASP-Methode vorgestellt, die eine schrittweise Modellierung und Verfeinerung von FaaS-basierten Anwendungen von abstrakten Topologien bis hin zu ausführbaren Deploymentmodellen ermöglicht. Die technologische Unterstützung für den Einsatz der GRASP-Methode wird durch eine integrierte GRASP-Toolchain ermöglicht. Um die Machbarkeit der vorgestellten Konzepte zu validieren, wird die GRASP-Toolchain prototypisch implementiert und in die bestehenden Werkzeuge zur musterbasierten Modellierung und Bereitstellung von Cloud-Anwendungen integriert.

ABSTRACT

Cloud computing revolutionized the way modern applications are designed and operated. Instead of maintaining the on premise infrastructure many enterprises incorporate various cloud offerings when designing their applications to decrease time-to-market and reduce the required management efforts. This includes the use of traditional cloud service models such as IaaS or PaaS as well as novel models such as FaaS that enables engineering cloud applications by composing fine-grained functions hosted on FaaS platforms with a variety of provider-managed services, e.g., data persistence and messaging services. Most management tasks for the components in FaaS-based applications become a responsibility of the chosen cloud provider, which, however, results in a stronger dependence on provider products and their implementation and packaging requirements. Therefore, engineering of FaaS-based applications can benefit from a stronger focus on the architectural considerations instead of specific products that often appear as fast as they become obsolete.

This work focuses on different aspects of provider-agnostic design for FaaS-based applications and is inspired by the increased dependence of components in such applications on product-specific requirements. To enable reasoning on component hosting and management requirements

in a provider-agnostic manner, this work introduces a pattern language capturing various trade-offs for hosting application components in the cloud. Furthermore, to facilitate classification and selection support for components in FaaS-based applications, this work presents a classification framework for FaaS platforms and introduces a classification framework metamodel that generalizes these concepts for other component types in such applications. Additionally, this work introduces a standards-based modeling approach for specifying function orchestrations and transforming them into provider-specific formats and an automated function code extraction and wrapping approach that aims to facilitate reusing functions for different FaaS platforms.

To enable using these contributions together, this thesis also introduces the GRASP method that enables gradual modeling and refinement of FaaS-based applications from abstract topologies to executable deployment models. The technological support for using the GRASP Method is enabled by an integrated GRASP toolchain. To validate the feasibility of the introduced concepts, the GRASP toolchain is implemented prototypically and integrated with the existing tools for pattern-based modeling and deployment of cloud applications.

ACKNOWLEDGMENTS

*A journey of a thousand miles
begins with a single step.*

LAO-TZU, "TAO TE CHING"

If someone had asked me about 5 years ago if I could ever imagine embarking on this journey and spending so much time on one specific topic, I probably would have laughed. And yet, here I am, thanking all the great people who have helped me along the way: Life really is full of surprises and I am so happy to be in the right place at the right time! First and foremost, I would like to thank Prof. Dr. Dr. h.c Frank Leymann for offering me a chance to begin my thousand-miles journey and experience the world of academic research at the Institute of Architecture of Application Systems (IAAS). Working under your supervision was a great pleasure and I especially appreciate the honest feedback you were always ready to give and the freedom I was given when pursuing my research ideas. I am also very grateful to Prof. Dr. Kostas Magoutis for accepting the second advisor role and finding time to review this thesis.

During my work at IAAS, I have met so many talented researchers who constantly motivated me to evolve. First, I would like to thank Dr. Michael Hahn and Dr. Andreas Weiß for employing me as a student research assistant and sparking my interest in research, and, of course, for being great office buddies. I would like to thank Prof. Dr. Uwe Breitenbücher for being a passionate and infinitely creative researcher he is. My heartfelt gratitude goes to my co-authors from University of Pisa, Prof. Dr. Antonio Brogi and Assistant Professor Dr. Jacopo Soldani: thank you for being a large part of my research journey, I really enjoyed our productive collaboration. I want to sincerely thank Michael Wurster for being the great research buddy and teammate in RADON. Our discussions and paper collaborations made the project work unforgettable, not to mention the beers in-between! My profound gratitude goes to Ghareeb Falazi for involving me in his research on blockchains, and a separate big thank you, my friend, for carefully reviewing the first draft of this thesis. Our conversations and coffee tasting sessions will always have a special place in my heart. I am very grateful to my colleagues Dr. Lukas Harzenetter, Benjamin Weder, Dr. Karoline Wild, Christoph Krieger, Dr. Michael Falkenthal, Martin Beisel, and Felix Truger for involving me in so many productive collaborations on very diverse topics. A special thank you to all German-speaking colleagues for always motivating me to learn German! Finally, my special thanks go to Sophie Schroth-Schreiner and all the IVI-GL team for always having my back with visa-related and other organizational issues.

Last but not least, I would not be the person I am today without my family. I am forever grateful to my mother Vera and my father Nail for constant support and letting me be who I want to be; I love you both so much! Special thanks go to my sister Yevgeniya for always believing in me! My main gratitude goes to my beloved wife Anna: no words can truly express how deeply I love you and how grateful I am to have you by my side. The only reason why I was able to get to this point without quitting or becoming insane is your tremendous support. I am truly grateful to all my extended family: grandmothers, aunts, uncles, cousins, in-laws - I am blessed to have you and I love you all!

TYPOGRAPHICAL CONVENTIONS

In this thesis, the following typographical conventions are used:

Italic

Emphasizes important terms or statements.

Bold

Denotes paragraph headings and title elements.

Monospace

Used for listings and to refer to code elements within paragraphs.

SMALL CAPS

Highlights unique names, e.g., pattern names.

(Optional Icon) Information Box Title

This element signifies thematic information blocks with the type of content described in the title and/or by using an optional icon.

The following icons are used throughout this thesis: **Q** presents an informal observation and **?** outlines a research question.

CONTENTS

1	Introduction	15
1.1	Preliminary Research	18
1.2	Vision of the Work and Research Questions	36
1.3	Research Contributions	39
1.4	Scientific Publications	44
1.5	Structure of the Thesis	49
2	Fundamentals and Related Work	51
2.1	Serverless Computing & Function-as-a-Service	52
2.2	Architecting Cloud and FaaS-based Applications	55
2.3	Decision Support for Selecting Cloud Services	79
2.4	Patterns and Pattern-based Design	84
2.5	Chapter Summary and Discussion	89
3	Component Hosting and Management Patterns	91
3.1	Authoring Process and Patterns Format	92
3.2	Core Terminology	94
3.3	Overview of the Pattern Categories	96
3.4	Deployment Stack Management Patterns	98
3.5	Scaling Configuration Management Patterns	103

- 3.6 Component Hosting Patterns 108
- 3.7 Pattern Relations and Their Semantics 120
- 3.8 Chapter Summary 125

- 4 Classification and Selection of Components for FaaS-based Applications 127**
- 4.1 Research Method 129
- 4.2 A Framework for Classifying FaaS Platforms 132
- 4.3 A Generic Metamodel for Technology Classification Frameworks 142
- 4.4 Architecture for Technology Selection Support 156
- 4.5 Chapter Summary 157

- 5 Artifact-level Abstractions 159**
- 5.1 Uniform Modeling of Function Orchestrations 160
- 5.2 Serverless Parachutes for Code Abstraction 185
- 5.3 Associating Artifact Abstractions With Components in Application Models 190
- 5.4 Chapter Summary 192

- 6 Gradual Refinement of FaaS-based Applications 195**
- 6.1 The GRASP Method 196
- 6.2 The GRASP Meta-Model and Language Support 205
- 6.3 Chapter Summary 209

- 7 Integrated Architecture and Prototypical Validation 211**
- 7.1 Architecture of the GRASP Toolchain 212
- 7.2 Prototypical Implementation 213
- 7.3 The GRASP Toolchain by Example 222
- 7.4 Integration with Other Systems 231

- 8 Conclusions and Outlook 235**
- 8.1 Summary of Contributions 236
- 8.2 Research Opportunities 238

Bibliography

241

INTRODUCTION

BEFORE the advent of cloud computing [MG+11], the general opinion favored traditional ownership of resources. Currently, instead of dealing with the maintenance of on premises infrastructure and putting effort in development of non-business-critical components, many enterprises choose the convenience of cloud offerings that facilitate decreasing time-to-market via on-demand access to basically any kind and amount of resources on a pay-per-use basis. The constantly increasing adoption of the cloud is, for instance, highlighted in the annual State of the Cloud Report by Flexera [Fle22], which names multi- and hybrid-cloud solutions as the most popular choices employed by the respondent companies. Choosing among various cloud service models becomes critical since cloud enables not only lifting-and-shifting existing applications, e.g., using IaaS offerings, but also designing and building entire applications in a cloud-first [BDJ18] fashion as carefully-planned compositions of various cloud services. Having a well-planned and documented cloud strategy, therefore, enables enterprises to remain competitive in economies of speed [Hoh22a].

Traditional cloud service models such as IaaS, PaaS, and Software-as-a-Service (SaaS) [FLR+14] already provided different levels of abstraction to cloud application developers to enable reducing management efforts to a desired extent. The search for other abstraction alternatives consequently resulted in the emergence of FaaS offerings and the *serverless computing* paradigm [BCC+17]. Provider-managed FaaS platforms such as Amazon Web Services (AWS) Lambda [Ama22b] or Azure Functions [Mic22] are capable of hosting arbitrary code snippets that are automatically scaled out and in by cloud providers depending on request rates. This style of implementing and hosting business logic components enabled developing cloud applications as compositions of various provider-managed services [Clo18] – hence the ambiguous term “serverless” that intends to emphasize the minimized relevance of “servers”, i.e., traditional compute, network, and storage resources, to application developers:

“ *The winning prize for awkward naming might go to the term serverless, which describes a run-time environment that surely relies on servers.* ”

Gregor Hohpe, “*Cloud Strategy: A Decision-based Approach to Successful Cloud Migration*”, 2022

While often perceived differently by researchers and industry practitioners [LWSH19], the term serverless is nevertheless frequently associated with FaaS-based applications and refers to a novel way of designing and implementing cloud applications: Typically fine-grained and stateless functions and provider-managed services interacting by means of event-driven bindings between components [Tai+20] or using function orchestration services [GSP+18], which execute functions following the control flow specified via function orchestration models that follow the well-known concept of workflows [LR00]. Designing serverless, FaaS-based applications enables developers to avoid tedious management of the underlying infrastructure by outsourcing most efforts to the chosen cloud provider,

which, however, comes at a price since developed components become tightly-coupled with the underlying provider products and their specific implementation and packaging requirements.

The decisions a software architect needs to consider vary greatly in levels of technicality [Hoh20; Zim09]. Furthermore, while specific products often go extinct as fast as they emerge, the architectural considerations tend to remain relevant for longer times [Hoh22a] making it important to document them abstractly without getting bound to specific products, so that future changes are more straightforward and inexpensive [BCK21]. As components in FaaS-based applications are hosted on services mainly managed by cloud providers, the underlying architectural considerations need to be expressed independently of particular technologies, e.g., functions implemented for AWS Lambda and integrated with other AWS services cannot be directly reused for other cloud providers. At the same time, it is particularly advantageous to apply reuse in software engineering life cycle as early as possible, and to enable reuse not only for the code artifacts, but for entire architectures to facilitate designing systems with similar requirements [BCK21].

This thesis focuses on the topic of provider-agnostic design of FaaS-based applications motivated by their increased coupling with provider requirements. For example, application designers may benefit from provider-agnostic mechanisms for expressing hosting and management requirements for components in such applications and search for suitable provider-specific offerings that fulfill them, modeling function orchestrations independently of provider-specific languages, etc. The main goal of this thesis is, therefore, to introduce a set of abstractions targeting different application- and component-level design decisions independently of cloud providers, and to enable their subsequent translation into provider-specific representations of these decisions, e.g., concrete, executable deployment models. The abstraction mechanisms introduced in the scope of this work aim to facilitate the reuse of single artifacts as well as the designed architectures for other scenarios with similar requirements.

1.1 Preliminary Research

The vision of this thesis and its boundaries were established during the preliminary research phase focused on the topic of FaaS-based applications. This section discusses and combines the core findings from two peer-reviewed publications [YBLM19; YBLW19] to highlight and motivate the research challenges addressed in this thesis.

1.1.1 Analysis of Existing Research on Engineering Function-as-a-Service Platforms and Tools

The first preliminary research work systematically analyzes and maps existing research literature focusing on engineering FaaS platforms and tools that was published in the period from 2009 to July 2019 to identify potential research gaps. This subsection briefly discusses the study design and findings for the main research question: *What are the challenges and drivers for engineering FaaS platforms and tools?* While the original study [YBLW19] also collected statistics such as preferred venues and industrial participation, these details are omitted for brevity.

Research Design and Data Extraction

The systematic mapping study was designed and conducted following the well-established guidelines and practices for systematic literature reviews [BBF+18; DLM19; KB13; PFMM08; PVK15]. Figure 1.1 shows the steps to collect and analyze the research literature. For initial search six well-known electronic databases recommended by existing guidelines [KB13; PFMM08; PVK15] were used. As FaaS is often associated with the concept of serverless computing [LWSH19], a generic search string (*serverless OR “Function-as-a-Service” OR FaaS OR “Function as a Service”*) was used to increase the list of candidate papers. The initial sets of candidate publications for each electronic database were first

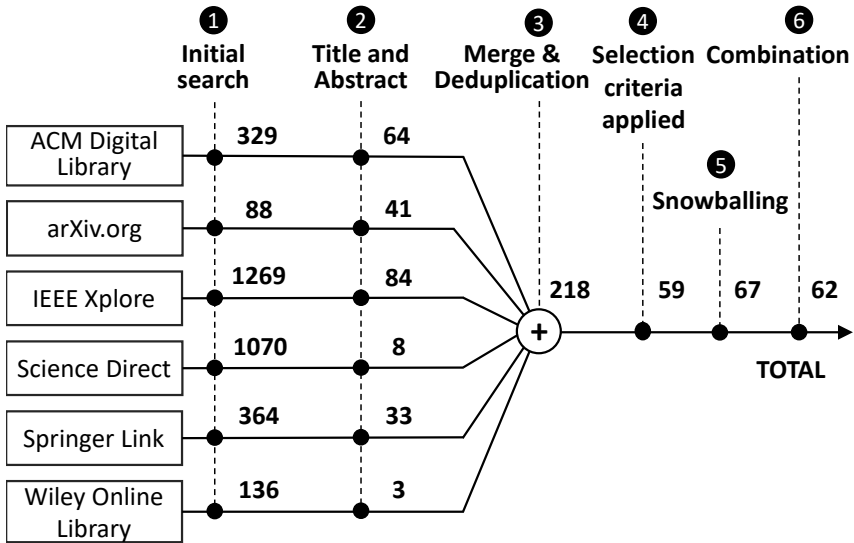


Figure 1.1: A multistep search and selection process

screened based on the titles and abstracts to exclude unrelated papers, and then merged into a single set and de-duplicated by comparing combinations of the title, author names, publication year, and venue. The resulting set was then filtered based on the defined inclusion criteria – specifically, the study included only publications that (i) introduce novel general-purpose FaaS platforms and tooling, (ii) focus on architectural solutions, methods, algorithms, and optimization techniques targeting specific aspects of FaaS platforms or tools, and (iii) publications that are written in English. Next, the forward (using Google Scholar) and backward snowballing [Woh14] were performed to include related works that cite or are cited by the included publications, respectively. Finally, all results were combined in a list of 62 related publications by removing related papers, e.g., original vs. extended paper versions. To answer the main research question regarding the challenges and drivers for engineering FaaS platforms and tools, the data were extracted and synthesized from identified publications as

follows. Firstly, an initial classification framework for categorizing the identified publications was systematically developed using the keywording technique [PFMM08]. This involved randomly choosing and analyzing 10 publications to derive an initial set of keywords that describe challenges and drivers for engineering FaaS platforms and tools. Afterwards, the keywords were clustered into the initial version of the framework, which was gradually refined by analyzing the remaining list of publications.

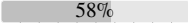


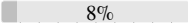
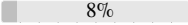
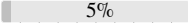
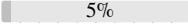

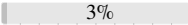
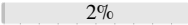
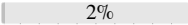
Analysis of Research Trends and the Observed Research Gap

Table 1.1 shows the identified challenges and how frequently they appear in the selected publications. The majority of identified challenges were related to function execution aspects, e.g., how to optimize function scheduling, support long-running tasks, or ensure secure function execution. Only a few works addressed such topics as deployment automation and migration of function code. In the following, the core findings that inspired the vision of this work are presented.

Challenges related to function execution. Most identified publications aimed to improve various function execution aspects in FaaS platforms as shown in Table 1.1. The largest part focused on performance optimization techniques including such problems as function scheduling and resources allocation, or function runtime enhancements, e.g., modifications on the level of Docker containers or language runtimes such as Java Virtual Machine. Improving security for executing functions, adding support for long-running tasks and function composition were among other kinds of challenges related to the function execution aspects. A more general challenge mentioned in some publications was the need to provide research-oriented FaaS platforms with publicly-available source code as it can facilitate further research.

Supporting new deployment environments. Some identified works proposed new FaaS deployment techniques, e.g., FaaS tools that enable scheduling functions in multi-cloud environments based on performance or

Table 1.1: The share of identified challenges with respect to the identified publications (one publication might address several challenges)

Category	Count	Total Share
Function execution	36 / 62	 58%
• Performance	26 / 36	
• Security	5 / 36	
• Long-running tasks support	2 / 36	
• Fault tolerance mechanisms	1 / 36	
• Function composition support	1 / 36	
• Language runtime support	1 / 36	
Deployment environments support	6 / 62	 10%
Testing & observability	5 / 62	 8%
Benchmarking	5 / 62	 8%
Costs optimization	5 / 62	 8%
Programming models	3 / 62	 5%
Research-centric platforms	3 / 62	 5%
Deployment automation	2 / 62	 3%
Migration	2 / 62	 3%
CI/CD pipelines	1 / 62	 2%
Reference architecture	1 / 62	 2%

cost preferences. Other examples aimed to support using FaaS in edge/fog computing contexts, e.g., to run functions on edge devices for data pre-processing. This group of challenges also highlights the need to represent decisions on *how an underlying FaaS platform is deployed* when designing FaaS-based applications, e.g., whether it is a self-hosted or a provider-managed platform.

Testing and observability. A more tooling-related group of challenges identified in the selected publications targeted testing, debugging, monitoring, and logging of FaaS-based applications. For example, prototypes enabling tracing function dependencies or visualizing logs for FaaS-based applications were introduced, highlighting the need in tooling supporting FaaS-based application developers.

Benchmarking and costs optimization. Another identified category of challenges is benchmarking, with related publications focusing on performance and costs of running FaaS functions. For example, one of the benchmarking approaches focuses on the so-called cold start [BCC+17] issue and factors that influence it, i.e., how to address the increased startup time for initial function calls. Additionally, certain identified publications introduced tools for optimizing costs of running FaaS-based applications.

Programming models. A few identified papers focused on new programming models for FaaS, e.g., enabling retroactive programming for FaaS-based applications such that function execution histories could be modified and replayed using event sourcing. Other examples include a serverless programming language for orchestrating function executions and a visual programming language for FaaS-based applications.

Migration, Deployment automation, and CI/CD. Deployment automation for FaaS-based applications was considered rarely. Examples included modeling and deployment of FaaS-based applications using Cloud Modeling Languages (CMLs) such as TOSCA or CAMEL [BBF+18]. Additionally, one paper investigated how Continuous Integration/Continuous Delivery (CI/CD) pipelines for such applications can be implemented. The challenge of migrating legacy functionalities to FaaS was investigated in two identified papers. The so-called *faasification* [Spi17; SD17] approaches aim to enable automated extraction of legacy code and deploying it as FaaS functions.

While the conducted study showed different potential research directions, the following research gap inspired the next steps of this work:



Determining the Research Scope

The systematic analysis of state-of-the-art research literature focusing on FaaS showed a strong community focus on designing efficient FaaS platforms, whereas the research on engineering FaaS-based ap-

plications was more scattered. In particular, the concepts and tooling focusing on provider-agnostic design of FaaS-based applications and mechanisms for transitioning from such models to chosen provider-specific deployments were underrepresented, which inspired the next research step, and the resulting vision of this thesis.

1.1.2 Analysis of Portability Challenges in FaaS Applications

FaaS-based applications can support a variety of use-cases, e.g., event-driven data pipelines, serverless Application Programming Interfaces (APIs), or advanced function orchestrations [Clo18] can all be implemented by combining FaaS functions with multiple distinct provider-specific services. However, the increased amount of provider-managed services in resulting application topologies leads to a stronger lock-in as each involved component relies on different service-specific data formats, APIs, and custom configuration Domain-specific Languages (DSLs) [OST14]. FaaS-related components introduce additional provider-specific dependencies, e.g., event-driven bindings between functions and components or different function orchestration modeling languages. To further investigate which architectural considerations are relevant in the context of the identified research gap, in the second preliminary research study, four common FaaS use cases were implemented for a baseline cloud provider and manually migrated to another two cloud providers, which enabled identifying relevant technical vendor lock-in issues and documenting possible solutions aiming to facilitate adapting to changes of provider-specific requirements.

The use cases were selected based on the analysis of common application scenarios encountered in academic and gray literature [BCC+17; Clo18; FIMS17; HFG+18] and designed to comprise heterogeneous components types, e.g., different storage types. Moreover, the use cases were implemented without optimizing them for portability and interoperability. AWS was chosen as the baseline provider, whereas Microsoft Azure and IBM Cloud were chosen as the other two providers.

Use Cases Overview

The first use case shown in Figure 1.2 is a common FaaS-based application for generating image thumbnails. Here, clients can synchronously call the *Persist Image* function over HTTP to upload an image in the *Images* object storage bucket. After the image is stored, a corresponding “Image Stored” event is emitted to trigger the *Generate Thumbnail* function, which creates a thumbnail and stores it in the *Thumbnails* bucket. Since a thumbnail must be generated for every uploaded image, the event connector [MMP00] represents the mandatory trigger with *at least once* delivery guarantee.

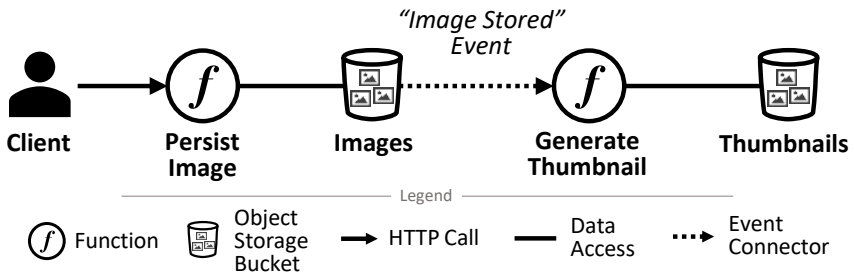


Figure 1.2: Use Case 1: Thumbnail Generation

API composition and aggregation [BCC+17] is another common FaaS-based application use case. Figure 1.3 shows a simple *To-do List API* exposed using an API Gateway [Ric18] component, which provides Create-Retrieve-Update-Delete (CRUD) functionalities for managing a list of to-be-done tasks persisted in a NoSQL database. In this use case, clients interact with the REST API via HTTP calls to respective endpoints, e.g., a POST request with the item information sent to the */items* resource to create a to-do item. The API Gateway component is responsible for forwarding client requests to the corresponding function that implements the respective functionality, e.g., the *Create Item* function for creating to-do items and storing them in the NoSQL database.

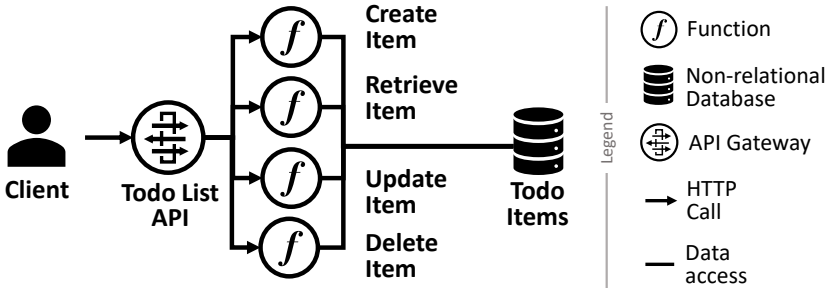


Figure 1.3: Use Case 2: To-do List API

The third use case depicted in Figure 1.4 is an event-driven application motivated by Internet of Things (IoT) scenarios in which data from multiple sensors have to be aggregated for further processing using message queues and streaming platforms. Here, different event sources, e.g., sensors, publish data to the *Incoming Events Topic* using the Event API as shown in Figure 1.4. Similar to the To-do List API use case, an API Gate-

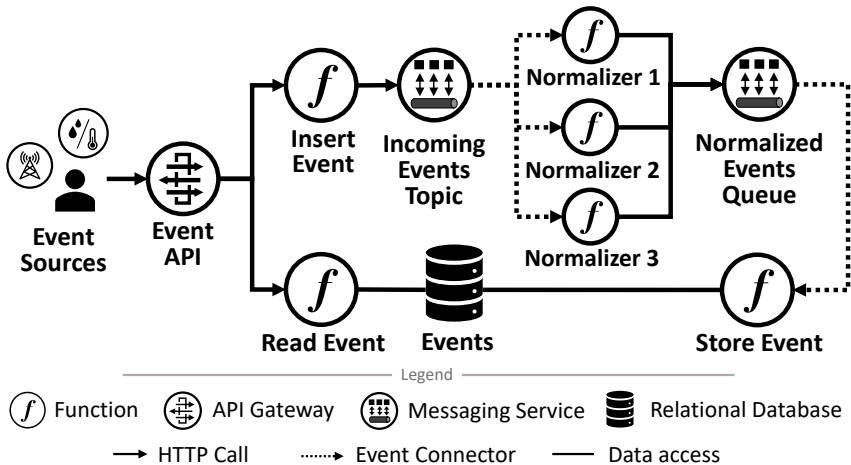


Figure 1.4: Use Case 3: Events Processing

way component is responsible for forwarding distinct HTTP-based API requests to the corresponding functions, i.e., *Insert Event* and *Read Event*. Published events are normalized using the corresponding format-specific *normalizer function* and sent via the message queue to the *Store Event* function that persists them in a Relational Database Management System (RDBMS). The *Store Event* function is triggered whenever a published event is stored in the message queue with *at least once* delivery guarantee. This use case employs both publish-subscribe and point-to-point channels to showcase more provider-specific messaging services.

FaaS platforms often do not support long-running tasks [HFG+18] due to limited function execution times, which can hinder implementation of more complex use cases. Employing the workflow technology[LR00] is one potential solution: large functions can be split into smaller pieces of logic and composed as function orchestrations [GSP+18] that can be enacted using FaaS standalone orchestration services such as AWS Step Functions or Microsoft Azure Durable Functions. Thus, as the fourth implemented use

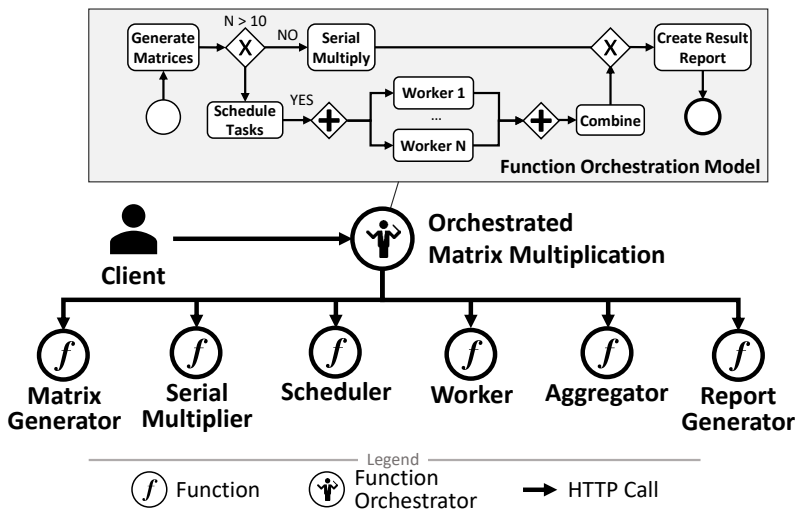


Figure 1.5: Use Case 4: Function Orchestration

case is a function orchestration. Figure 1.5 shows an orchestrated matrix multiplication: a function orchestrator service enacts six FaaS functions following the control flow defined in the orchestration model. The function orchestration model in Figure 1.5 is shown in BPMN [OMG11], whereas in practice it can be defined using general-purpose programming languages such Python, or provider-specific DSLs such as Amazon States Language (ASL) [GSP+18].

Use Cases: Implementation and Migration

Each use case application presented in Section 1.1.2 was first implemented for AWS and then manually migrated to Microsoft Azure and IBM Cloud. In the following, the implementation and migration details for each cloud provider are described together with the summary of changes required for successful migration of the baseline implementation. The main goal of this study was to identify *general categories of problems* that could be encountered when migrating FaaS-based applications instead of focusing on specific limitations of certain offerings.

Baseline implementation for AWS. Table 1.2 provides an overview of the use case implementation details for AWS. In all four cases, a different programming language was used to implement functions for AWS Lambda, the FaaS offering from Amazon. Provider-specific Software Development

Table 1.2: Baseline implementation details.

<i>Use Case</i>	<i>Prog. Lang.</i>	<i>AWS Services</i>
1: Thumbnail Generation	Java	Lambda, S3
2: To-do List API	Go	Lambda, DynamoDB, API Gateway
3: Event Processing	JavaScript	Lambda, SNS, SQS, Aurora, API Gateway
4: Function Orchestration	C#	Lambda, Step Functions

Kits (SDKs) were utilized when possible, e.g., a Java library for working with AWS S3 events or the SDK AWS provides for developing functions in Go. The overview of used AWS services is also shown in Table 1.2, e.g., AWS Step Functions service was used to model and enact function orchestrations. Service types and the corresponding provider-specific offerings are further discussed after the summary of migration efforts.

Table 1.3: Summary of modifications required for migrating the use cases to target cloud providers.

<i>Modifications</i>	<i>Microsoft Azure</i>	<i>IBM Cloud</i>
Implementation language	Use Case 2 ¹	None
Configurations	Use Cases 1-4	Use Cases 1-4
Function code	Use Cases 1-4	Use Cases 1-4
Orchestration model	Use Case 4	Use Case 4
Architectural changes	None	Use Case 3 ²

¹ All code had to be re-implemented in C# due to unsupported programming language

² Point-to-point channel had to be replaced with a publish-subscribe channel due to a lack of compatible service offering

Migration to Microsoft Azure and IBM Cloud. In each use case, certain modifications in function and/or configuration code were needed to successfully migrate to one or both target providers. Table 1.3 summarizes the changes and affected use cases – while certain modifications were encountered rarely, others were needed in all cases. For example, function code and configurations had to be modified in all use cases for all target cloud providers. Likewise, function orchestration models also required to be changed for each of the target providers to enable deploying them to respective function orchestrators. In rare cases, change of programming language or architectural modifications were needed, e.g., due to missing support or lack of compatible offerings.

When migrating FaaS-based applications it is also necessary to identify compatible service alternatives for each component type in the respective application topology. For example, as shown in Table 1.3, a point-to-point

Table 1.4: Mappings chosen for provider-specific service types.

<i>Component</i>	<i>Amazon Web Services</i>	<i>Microsoft Azure</i>	<i>IBM Cloud</i>
Function	Lambda	Functions	Cloud Functions
Object storage	S3	Blob Storage	Object Storage
NoSQL database	DynamoDB	Table Storage	Cloudant
API Gateway	API Gateway	API Management	Openwhisk API
Relational database	Amazon Aurora	Database for MySQL	Compose for MySQL
Messaging: publish- subscribe	SNS	Service Bus	Event Streams
Messaging: point-to-point	SQS	Service Bus	Event Streams
Function orchestrator	Step Functions	Durable Functions	Composer

channel in the Use Case 3 had to be changed when migrating to IBM Cloud as no compatible provider-managed service offerings were available at the time. Table 1.4 shows the cloud service mappings identified and used for components in the use cases. Most function code and configuration changes were required in the context of switching from service to service due to differences in SDKs, APIs, event binding mechanisms, etc.

Lock-in Issues in FaaS-based Applications

To migrate the FaaS-based use cases, the most frequently encountered lock-in issues were related to distinct application components, e.g., tackling discrepancies between feature sets of specific service offerings. However, more global lock-in issues were also present, e.g., changes to application architecture or employed toolchains might be needed too. Most discussed lock-in issues are commonly encountered in the context of cloud [Hoh19;

OST14; SPSP14]. However, FaaS-based applications typically are even more coupled since the out-of-the-box integration between FaaS platforms and other provider offerings such as storage, messaging, or logging services promote designing applications as compositions of multiple offerings mainly managed by cloud providers. Thereby, the lock-in categories encountered for these FaaS-based use cases are discussed next in the direction from local, component-level to global, application-level.

Lock-in: Implementation Requirements. Components in FaaS-based applications are often built following provider- and service-specific implementation requirements, for example:

- *Code implementation and packaging requirements:* functions might need to implement provider-specific interfaces, e.g., Java functions for AWS Lambda in Use Case 1 (see Figure 1.2). Further, functions are packaged as defined by providers, e.g., packaging for Azure Functions is different from AWS Lambda. Function orchestrations are also modeled as required by function orchestrators, e.g., orchestration from Use Case 4 (see Figure 1.5) had to be modeled using three different languages.
- *Format and data type dependencies:* events emitted by provider services such as object storage have different format and are passed to functions differently, which requires adjusting the event handling logic in function implementations.
- *Library dependencies:* to-be-migrated code might require adjustments whenever functions rely on provider-specific libraries for working with events or provider APIs.

One way to relax the coupling is to extract the business logic as libraries, hence, separating it from provider-specific code responsible for service interactions or handling inputs and outputs. Usage of provider-agnostic event formats, e.g., CloudEvents specification [Clo22b], is another mechanism that can help to abstract away the event processing.

Lock-in: Configuration Requirements. Components and bindings in FaaS-based applications must often be configured following provider- and service-specific requirements. One example of a locked-in configuration encountered for all use cases from Section 1.1.2 is how event bindings are configured. For instance, AWS Lambda enables configuring triggers via Graphical User Interface (GUI) or in AWS-specific deployment models, whereas Azure supports configuring bindings for certain languages directly in the source code using annotations. More general provider-specific configuration examples include security-related settings or naming rules, e.g., AWS S3 bucket must have globally-unique names, whereas Azure Object Storage does not impose such requirement.

Lock-in: Features and Limitations of the Underlying Service Offering. Components in FaaS-based applications are often locked into feature sets of the respective service offerings chosen to host them, e.g., out-of-the-box integration with other provider services. Moreover, as discussed previously, components are often implemented adhering to specific requirements some of which are dictated by the limitations of the chosen offering, e.g., function execution time limits imposed by the underlying FaaS platform.

For instance, the event-driven function call via the point-to-point channel in Use Case 3 (see Figure 1.4) can be implemented using the existing binding between AWS Lambda and SQS. However, when migrating this implementation to IBM, no such feature was available for corresponding service alternatives (IBM MQ and IBM Cloud Functions). Likewise, the programming language used for the baseline implementation of the Use Case 2 (see Figure 1.3) was not supported by the corresponding service alternative from Microsoft Azure.

More general examples include the reliance on GUIs: while function orchestrations from Use Case 4 (see Figure 1.5) can be visualized in AWS Step Functions, IBM Composer does not provide such feature. Strong dependence on specific service features can complicate or even prevent

finding suitable alternatives from other providers and, hence, requiring non-straightforward solutions for migrating the component, e.g., re-engineering or changing certain architectural decisions might be needed.

Lock-in: Architectural Style. Inability to deviate from the chosen architectural style can also affect the portability of FaaS-based applications. For example, when a messaging system enabling point-to-point communication is not available in provider-managed flavor, one of the possible solutions is to choose a service mainly managed by cloud consumers from the list of available offerings of the target cloud provider.

A similar issue was encountered when migrating Use Case 3 (see Figure 1.4): a point-to-point channel was substituted with a publish-subscribe channel eventually, but the target service was still mainly managed by the cloud provider. Another option would have been to manually deploy and integrate an open source messaging system. Likewise, open source FaaS platforms can be hosted manually using less provider-managed services such as managed Kubernetes clusters, which essentially enables implementing more portable FaaS-based applications. However, setting up FaaS platforms and integrating them with event emitting components would require more effort from application developers, shifting towards more user-managed deployment architecture. Therefore, the decision to remain more provider-managed or relax the management constraints can also influence how portable the application is.

Lock-in: Tooling. Tools employed for development, deployment, or observability of FaaS-based applications can also be a potential lock-in issue. One example is deployment automation tooling – multiple providers offer technologies supporting only provider-specific services, e.g., AWS Cloud Formation or Azure Resource Manager enable automating the deployment for the respective public clouds. While multi-provider solutions such as Ansible, Terraform, or Serverless framework enable modeling deployments for different target infrastructures, the resulting models are still provider-specific due to configuration of provider services. Similar issues could

be encountered for provider-specific monitoring and testing solutions or custom CI/CD pipelines that might need to be adjusted or fully replaced due to lack of support for new target environments.

Similar to other cases of product lock-in [Hoh19], usage of standards could help to add more layers of abstraction, e.g., vendor-agnostic cloud modeling languages such as TOSCA [Lip12]. Such standards-based deployment models will still eventually have provider-specific parts, however, they enable modularizing deployment logic using custom type systems which could enable transitioning between different levels of abstraction.

A Summary of Observed Architectural Considerations

The problems encountered when migrating the use cases presented in Section 1.1.2 show that even small-scale FaaS-based applications have high numbers of provider-specific dependencies. Thus, introducing a generic approach that facilitates transitioning between general, provider-independent architectural considerations for FaaS-based applications and product-specific choices can be advantageous not only for reasoning on to-be-implemented applications, but also for scenarios when existing applications need to be migrated to another provider. Despite the constantly changing landscape of provider offerings, the observed architectural considerations remain topical and affect both the level of the entire application and single components, as summarized informally in the following.



Modeling of FaaS-based Applications

FaaS-based applications enable various architectural styles. Their application models comprise fine-grained components mainly hosted on single-purpose, provider-managed services and directly resemble deployment architectures.

FaaS-based applications can serve as enablers for various architectural styles. For example, Use Cases 1 and 3 are based on Event-driven Architectures (EDAs), whereas Use Case 2 realizes a simple Microservice-based Architecture (MSA). Use Case 4 possesses characteristics of an orchestration-driven Service-oriented Architecture (SOA) [RF20]. Therefore, understanding which modeling techniques can represent general architectural considerations for such heterogeneous component topologies and how to enable transitioning to provider-specific decisions can facilitate reasoning about FaaS-based applications.



Component Management Trade-offs

Efforts needed to manage components in FaaS-based applications depend on hosting options. Selecting more consumer-managed options affects the “serverlessness” of application models.

Previously, the presented FaaS-based use cases were intentionally not called *serverless*. Existing study [LWSH19] shows that this term is perceived differently – from exclusively focusing on FaaS to a more inclusive

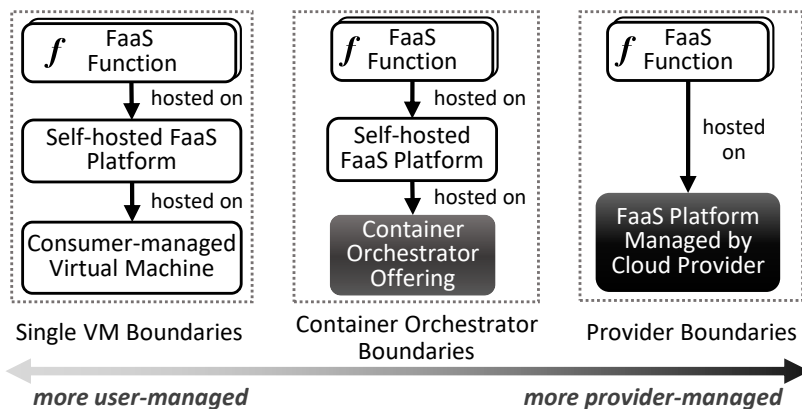


Figure 1.6: Management trade-off examples for FaaS functions hosting.

notion of applications composed using provider-managed services. FaaS platforms indeed reduce management efforts since functions can be provided as code snippets and auto-scaled by the platform, but the way a FaaS platform is hosted plays an important role too. The examples in Figure 1.6 show different management trade-offs for FaaS-hosted functions, e.g., hosting functions on provider-managed FaaS offerings such as AWS Lambda requires less effort than using self-hosted FaaS platforms on user-managed Virtual Machines (VMs). Expressing such decisions for components in FaaS-based applications can help to transition from provider-agnostic models to specific services that fulfill desired management requirements.



Classification and Selection of Components

Selection of hosting options for FaaS functions and other components in the application relies on various managerial and technical decisions that need to be combined and assessed with regard to the capabilities of available service offerings.

Most migration issues encountered for the use cases were caused by the baseline implementation decisions, e.g., chosen programming language, reliance on service-specific features, etc. When aggregated together such decisions can influence which service offerings can be considered suitable. Therefore, understanding how to classify FaaS-specific services and supporting the selection of suitable service alternatives can facilitate the decision-making process and transitioning from provider-agnostic architectural considerations to provider-specific details.



Provider-specific FaaS Artifacts

Artifact-specific lock-in issues were encountered in all migrated FaaS-based application use cases, i.e., function code and function orchestration models always required re-engineering efforts.

Although the way functions and their orchestrations have to be implemented depends on the chosen cloud provider, the intended outcome of executing such functions or orchestrations is often alike, e.g., a specific data transformation task such as reducing an image to a thumbnail, or an identical sequence of function executions that can be seen as a generic workflow [LR00]. While provider-specific dependencies can be separated using better code modularization, techniques enabling reuse of general-purpose code, e.g., automated code extraction and wrapping, can simplify transitioning between providers and reasoning about such code artifacts and orchestration models in provider-agnostic manner. Therefore, such artifact-level abstraction techniques can complement application-level abstractions to postpone making provider-specific decisions.

1.2 Vision of the Work and Research Questions

The research trends and architectural considerations observed during the preliminary research phase described in Section 1.1 showed the lack of abstraction mechanisms that could help to postpone provider-specific decisions when designing and implementing FaaS-based applications. The vision of this work is, therefore, to provide application designers with a set of abstraction and refinement mechanisms that enable model-driven specification of FaaS-based application models and FaaS-specific artifacts they encompass independently of providers and their translation into executable, provider-specific deployment models. Figure 1.7 shows the high-level vision in which FaaS-based applications can be *gradually refined* from an abstractly specified component topology to an executable, provider-specific deployment model. This vision and its building blocks pose several research questions that are discussed in the following.

Firstly, since FaaS-based application models are often tailored for a chosen provider-specific infrastructure (see Section 1.1.2), to realize the vision of this work, techniques for *provider-agnostic modeling* of FaaS-based applications are needed. This includes specification of components connectivity

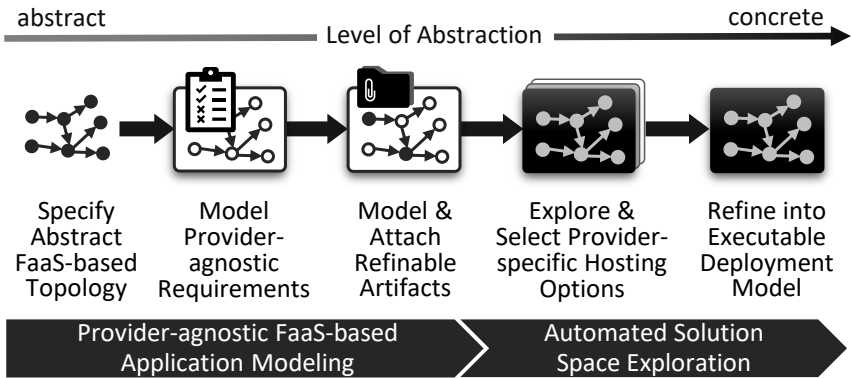


Figure 1.7: Vision: gradual refinement of FaaS-based applications.

information as well as component-specific requirements for FaaS-based applications independently of providers. Furthermore, modeling of decisions on component hosting and management trade-offs, e.g., whether a provider is responsible for autoscaling a component, needs to be supported too. In addition, to benefit from such abstract application models, refinement mechanisms that enable transitioning to concrete, provider-specific variants are needed. Therefore, the first research question focuses on provider-agnostic modeling of FaaS-based applications and mechanisms that enable transitioning between such abstract decision specifications to concrete, provider-specific deployment models as shown in Figure 1.7:

? Research Question 1: Provider-agnostic modeling

How to express application- and component-level decisions for FaaS-based applications in a provider-agnostic manner and how to enable refining them into concrete provider-specific models?

When the decisions are abstractly expressed, a task that must precede the refinement into executable deployment model is the selection of suitable hosting targets for the components present in the model, which requires

mechanisms for classifying component-specific services and supporting the selection process. Despite the focus on FaaS-specific decisions, such mechanisms have to be generic enough to support other component types present in FaaS-based applications. Thus, the second research question focuses on the topic of classification of components and generic ways of describing component type-specific decision criteria, and using those specifications to enable searching for component hosting options that fulfill desired requirements:



Research Question 2: Component Selection Support

How to classify components in FaaS-based applications and facilitate the selection of services based on given decision criteria?

Furthermore, the actual artifacts that implement FaaS-specific components, i.e., function code and function orchestration models, can benefit from abstracting provider-specific details as this could enable *transitioning from abstract to concrete decisions on the artifacts level*. For example, function orchestration models defined in a technology-agnostic way could be reused for other providers. Likewise, code-level abstractions could facilitate reuse of existing business logic, also on the level of abstract FaaS-based application models. Hence, the third research question focuses on the abstractions for FaaS-specific artifacts including function code and function orchestration models:



Research Question 3: Artifact-level abstractions

Which techniques can help developers to abstract away provider details from function orchestration models and function code?

Finally, a *toolchain supporting the described concepts* is needed to enable using them in combination, hence facilitating the envisioned translation of abstract decisions for FaaS-based applications into provider-specific

deployment models. This includes abstract decision modeling and their refinement into concrete deployment models that can be enacted by compatible deployment automation technologies.

? Research Question 4: Tooling Support

What tooling support is required to enable the envisioned gradual refinement of FaaS-based applications?

This thesis primarily focuses on cloud-first application design and development and while the proposed research contributions can be expanded or adapted towards migration of existing applications [ASL13], no discussion is further provided on how the concepts introduced in this work can be employed for migration scenarios.

1.3 Research Contributions

After presenting the vision of this work and the research questions in focus, this section provides an overview of the research contributions that address them. Figure 1.8 shows an overview of five research contributions that enable the envisioned gradual refinement of FaaS-based applications. Contribution 1 introduces a pattern language that captures various trade-offs for hosting and managing application components. Next, Contribution 2 describes how components in FaaS-based applications can be classified and how to support selecting services based on given decision criteria. Contribution 3 introduces abstraction techniques for function orchestration modeling and on the level of function code. Contribution 4 presents a model-driven method for gradual refinement of FaaS-based applications that combines the previously-introduced contributions using a generic meta-model for FaaS-based applications. Contribution 5 enables employing the

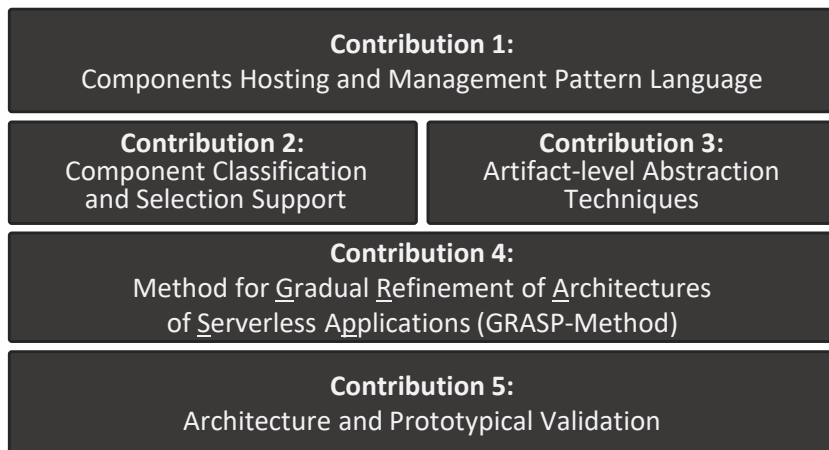


Figure 1.8: Overview of research contributions

aforementioned research contributions by means of a method-supporting toolchain and presents the architecture of the toolchain and the prototypical validation of the introduced concepts.

1.3.1 Component Hosting and Management Pattern Language

A part of Research Question 1 described in Section 1.2 is related to capturing component hosting and management trade-offs in a provider-agnostic manner, e.g., whether a FaaS function is hosted on a provider-managed platform or using on-premises infrastructure. These decisions are influenced not only by the underlying cloud service model such as IaaS or PaaS, but also by specific features the corresponding services provide. For example, provider-managed FaaS platforms often support deploying not only the code snippets, but also container images as deployment artifacts blurring the boundaries between container-centric services and FaaS platforms. These variants differ in flexibility since container images enable running custom dependencies with the function code. Additionally, the ways scaling

configuration for the component is managed differ too: provider-managed FaaS platforms auto-scale functions whereas for self-hosted platforms the underlying boundaries are defined by developers, e.g., a VM quantity or the size of the Kubernetes cluster. This problem is addressed by relying on the concept of patterns and Pattern Languages (PLs) [AIS77] as the well-established way to document and interconnect proven solutions to problems recurring in specific contexts.

Contribution 1: Component Hosting & Management PL. This contribution analyzes the existing trade-offs for hosting and managing application components and captures the common solutions in a form of the Component Hosting and Management Pattern Language that interconnects different management trade-offs and can be used to find suitable hosting options for deploying application components. The pattern language comprises nine patterns aligned with respect to two management dimensions: (i) deployment stack and (ii) scaling configuration management, forming a spectrum of hosting options ranging from provider- to user-managed variants. Application components can then be linked with the patterns to describe hosting and management decisions in a provider-agnostic manner.

1.3.2 Component Classification and Selection Support

While the pattern language introduced in Contribution 1 can help to link abstract decisions with provider-specific offerings, identifying suitable hosting options based on decisions specified for component types such as FaaS or Object Storage services requires classifying components and supporting selection of suitable services (Research Question 2). While there exist tools for comparison of specific component types, e.g., PaaS [Kol19], similar classification mechanisms are needed for FaaS-specific services and other component types constituting FaaS-based applications. Moreover, support for searching suitable hosting options is needed for the envisioned

refinement of abstract topologies into provider-specific deployment models. Thus, the next contribution focuses on classification and selection support for FaaS-specific services.

Contribution 2: Component Classification & Selection Support.

This contribution introduces a classification framework for FaaS platforms derived using a systematic analysis of existing research and gray literature. The resulting framework is then validated by applying it to ten existing FaaS platforms (including commercial and open source options) with the resulting platform data available publicly. The introduced classification framework metamodel and selection mechanisms also enable reusing these concepts for other component types in FaaS-based applications.

1.3.3 Artifact-level Abstraction Techniques

FaaS-specific artifacts depend strongly on platform- or service-specific requirements and can benefit from additional abstraction mechanisms, which are necessary to address Research Question 3. Following the idea of the gradual refinement of FaaS-based applications described in Section 1.2, artifact-level abstractions belong to the level of single components, e.g., function orchestration models deployed to compatible function orchestrator services, and can facilitate artifacts reuse. Hence, the next contribution introduces abstraction mechanisms for function orchestration modeling and conceptualizes how such artifact-level abstractions can be employed on the level of abstract application models.

Contribution 3: Artifact-level Abstraction Techniques. This contribution introduces a set of abstractions for FaaS-specific artifacts. This includes a concept for modeling function orchestrations using Business Process Model and Notation (BPMN) [OMG11] and transforming them into provider-specific formats. Furthermore, this

contribution presents how abstract function orchestration models and code extraction and packaging techniques can be employed in abstract application models to enable their refinement for the chosen provider-specific deployment models.

1.3.4 GRASP Method

To further address Research Question 1 from Section 1.2 and enable using all previous contributions together, a way to represent provider-agnostic decisions for FaaS-based applications and refine them into suitable provider-specific deployment model variants is needed. To address this challenge, the next contribution introduces a method inspired by the concept of *gradient of abstraction* by Floridi [Flo08] that enables exploring a single system using a set of interconnected Level of Abstractions (LoAs) that describe this system from different perspectives represented by varying sets of variables. For example, the wine topic can be discussed from different perspectives: “wine tasting LoA” would be different from “wine cellaring LoA” as they are described by different sets of variables, e.g., “acidity” and “decanter time” for the former and the latter, respectively. The general idea of the GRASP Method is, thus, to link the abstract decision modeling for FaaS-based applications with concrete deployment stacks using a set of different levels of abstraction for FaaS-based applications introduced in previous contributions that are combined using a provider-agnostic FaaS-based application metamodel. By transitioning between different abstractions for FaaS-based applications, application developers can gradually refine initial architectural considerations and interactively explore the solution space.

Contribution 4: GRASP Method. This contribution introduces the Gradual Refinement of Architectures of Serverless Applications (GRASP) method that enables using Contributions 1-3 together via a provider-agnostic FaaS-based application metamodel to sup-

port the envisioned refinement of abstract decisions into concrete deployment models: Patterns introduced in Contribution 1 are used to automatically transition from abstract hosting decisions to compatible provider-specific deployment stacks, which can then further be pruned using concepts from Contribution 2. Finally, the artifact-level abstractions presented in Contribution 3 enable code reuse for identified concrete models.

1.3.5 Architecture, Implementation, and Validation

To address Research Question 4 and show the practical feasibility of the research contributions, an architecture and a prototypical implementation of the toolchain enabling the vision of gradual refinement for FaaS-based applications is presented.

Contribution 5: Architecture, Implementation, and Validation.

This contribution presents a toolchain architecture and a prototype that enables the GRASP method and concepts introduced in all other contributions of this thesis. The introduced toolchain supports graphical, pattern-based modeling of FaaS-based applications, search for service offerings based on the specified requirements, and interactive refinement into provider-specific deployment models.

1.4 Scientific Publications

The research efforts that underpin the contributions of this thesis have been published in various peer-reviewed journals and conference proceedings. These publications are listed below in reverse chronological order.

1. V. Yussupov, U. Breitenbücher, A. Brogi, L. Harzenetter, F. Leymann, and J. Soldani. “Serverless or Serverful? A Pattern-based Approach for Exploring Hosting Alternatives”. In: *Proceedings of the 16th Symposium and Summer School on Service-Oriented Computing (SummerSOC 2022)*. Springer, Oct. 2022, pp. 45–67
2. V. Yussupov, J. Soldani, U. Breitenbücher, and F. Leymann. “Standards-based modeling and deployment of serverless function orchestrations using BPMN and TOSCA”. in: *Software: Practice and Experience* 52.6 (June 2022), pp. 1454–1495
3. V. Yussupov, J. Soldani, U. Breitenbücher, A. Brogi, and F. Leymann. “From Serverful to Serverless: A Spectrum of Patterns for Hosting Application Components”. In: *Proceedings of the 11th International Conference on Cloud Computing and Services Science (CLOSER 2021)*. SciTePress, May 2021, pp. 268–279
4. V. Yussupov, J. Soldani, U. Breitenbücher, A. Brogi, and F. Leymann. “FaaSSten your decisions: A classification framework and technology review of Function-as-a-Service platforms”. In: *Journal of Systems and Software* 175 (May 2021). ISSN: 0164-1212
5. V. Yussupov, U. Breitenbücher, C. Krieger, F. Leymann, J. Soldani, and M. Wurster. “Pattern-based Modelling, Integration, and Deployment of Microservice Architectures”. In: *Proceedings of the 24th International Enterprise Distributed Object Computing Conference (EDOC 2020)*. IEEE, Oct. 2020, pp. 40–50
6. V. Yussupov, U. Breitenbücher, A. Kaplan, and F. Leymann. “SEA-PORT: 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

7. V. Yussupov, U. Breitenbücher, F. Leymann, and M. Wurster. “A Systematic Mapping Study on Engineering Function-as-a-Service Platforms and Tools”. In: *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2019)*. Ed. by ACM. ACM, Dec. 2019, pp. 229–240
8. V. Yussupov, U. Breitenbücher, F. Leymann, and C. Müller. “Facing the Unplanned Migration of Serverless Applications: A Study on Portability Problems, Solutions, and Dead Ends”. In: *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2019)*. Ed. by ACM. ACM, Dec. 2019, pp. 273–283
9. V. Yussupov, U. Breitenbücher, M. Hahn, and F. Leymann. “Serverless Parachutes: Preparing Chosen Functionalities for Exceptional Workloads”. In: *Proceedings of the 23rd International Enterprise Distributed Object Computing Conference (EDOC 2019)*. IEEE Computer Society, Oct. 2019, pp. 226–235
10. V. Yussupov, G. Falazi, M. Falkenthal, and F. Leymann. “Protecting Deployment Models in Collaborative Cloud Application Development”. In: *International Journal On Advances in Security 12.1&2* (June 2019), pp. 79–94. ISSN: 1942-2636
11. V. Yussupov, M. Falkenthal, O. Kopp, F. Leymann, and M. Zimmermann. “Secure Collaborative Development of Cloud Application Deployment Models”. In: *Proceedings of The Twelfth International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2018)*. Xpert Publishing Services, Sept. 2018, pp. 48–57

Furthermore, the co-authored peer-reviewed publications that contributed to the concepts in this thesis are listed below in order of relevance:

1. M. Wurster, U. Breitenbücher, K. Képes, F. Leymann, and V. Yussupov. “Modeling and Automated Deployment of Serverless Applications using TOSCA”. in: *Proceedings of the IEEE 11th International Conference on Service-Oriented Computing and Applications (SOCA 2018)*. IEEE Computer Society, Nov. 2018, pp. 73–80
2. M. Wurster, U. Breitenbücher, L. Harzenetter, F. Leymann, J. Soldani, and V. Yussupov. “TOSCA Light: Bridging the Gap between the TOSCA Specification and Production-ready Deployment Technologies”. In: *Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER 2020)*. SciTePress, May 2020, pp. 216–226
3. M. Wurster, U. Breitenbücher, A. Brogi, G. Falazi, L. Harzenetter, F. Leymann, J. Soldani, and V. Yussupov. “The EDMM Modeling and Transformation System”. In: *Service-Oriented Computing – ICSOC 2019 Workshops*. Springer, Dec. 2019
4. T. F. Düllmann, A. van Hoorn, V. Yussupov, P. Jakovits, and M. Adhikari. “CTT: Load Test Automation for TOSCA-Based Cloud Applications”. In: *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*. Association for Computing Machinery, July 2022, pp. 89–96
5. M. Hahn, U. Breitenbücher, F. Leymann, M. Wurster, and V. Yussupov. “Modeling Data Transformations in Data-Aware Service Choreographies”. In: *Proceedings of the IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC 2018)*. IEEE Computer Society, Oct. 2018, pp. 28–34
6. M. Hahn, U. Breitenbücher, F. Leymann, and V. Yussupov. “Transparent Execution of Data Transformations in Data-Aware Service Choreographies”. In: *On the Move to Meaningful Internet Systems. OTM 2018 Conferences (CoopIS 2018)*. Vol. 11230. Lecture Notes in Computer Science. Springer International Publishing AG, Oct. 2018, pp. 117–137

7. M. Zimmermann, U. Breitenbücher, L. Harzenetter, F. Leymann, and V. Yussupov. “Self-Contained Service Deployment Packages”. In: *Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER 2020)*. SciTePress, May 2020, pp. 371–381
8. C. Krieger, U. Breitenbücher, M. Falkenthal, F. Leymann, V. Yussupov, and U. Zdun. “Monitoring Behavioral Compliance with Architectural Patterns Based on Complex Event Processing”. In: *Proceedings of the 8th European Conference on Service-Oriented and Cloud Computing (ESOCC 2020)*. Springer International Publishing, Mar. 2020, pp. 125–140
9. G. Falazi, U. Breitenbücher, F. Daniel, F. Lamparelli, F. Leymann, and V. Yussupov. “Smart Contract Invocation Protocol (SCIP): A Protocol for the Uniform Integration of Heterogeneous Blockchain Smart Contracts”. In: *Proceedings of the 32nd Conference on Advanced Information Systems Engineering (CAiSE 2020)*. Vol. 12127. Lecture Notes in Computer Science. Springer International Publishing, June 2020, pp. 134–149
10. G. Falazi, M. Hahn, U. Breitenbücher, F. Leymann, and V. Yussupov. “Process-Based Composition of Permissioned and Permissionless Blockchain Smart Contracts”. In: *Proceedings of the 2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC 2019)*. IEEE, Oct. 2019, pp. 77–87
11. G. Falazi, U. Breitenbücher, M. Falkenthal, L. Harzenetter, F. Leymann, and V. Yussupov. “Blockchain-based Collaborative Development of Application Deployment Models”. In: *On the Move to Meaningful Internet Systems: OTM 2018 Conferences (CoopIS 2018)*. Springer International Publishing, Oct. 2018, pp. 40–60

Finally, the research efforts in the context of the EU Horizon 2020 project RADON [CAH+20] and the participation in the *Dagstuhl Seminar on Serverless Computing* as a part of the software engineering group also

contributed to the results of this work. RADON deliverables are publicly available on the official website (see <https://radon-h2020.eu>). Proceedings of the seminar on serverless computing are available via Dagstuhl Research Online Publication Center:

- C. Abad, I. T. Foster, N. Herbst, and A. Iosup. “Serverless Computing (Dagstuhl Seminar 21201)”. In: *Dagstuhl Reports* 11.4 (2021). Ed. by C. Abad, I. T. Foster, N. Herbst, and A. Iosup, pp. 34–93. ISSN: 2192-5283

1.5 Structure of the Thesis

Prior to presenting the contributions, Chapter 2 elaborates on the relevant background concepts and positions the existing related publications with respect to the research contributions presented in this thesis. The remainder is structured following the order of contributions presented in Section 1.3:

Chapter 3 (Contribution 1): introduces a pattern language for hosting and management application components that enables expressing management trade-offs for components in a provider-agnostic manner.

Chapter 4 (Contribution 2): presents the classification framework and selection support system for FaaS platforms and generalizes the employed approach to other component kinds that could constitute FaaS-based applications.

Chapter 5 (Contribution 3): focuses on two kinds of artifact-level abstractions, namely (i) abstract modeling and transformation of function orchestration models and (ii) function code-level abstraction technique inspired by the idea of faasification.

Chapter 6 (Contribution 4): continues addressing the topic of provider-agnostic modeling by presenting the GRASP Method for exploring provider-specific deployment alternatives for abstractly-described FaaS-based applications.

Chapter 7 (Contribution 5): introduces the architecture and prototypical implementation of a toolchain enabling the vision of gradual refinement of FaaS-based applications.

Finally, Chapter 8 concludes this thesis by summarizing the findings, discussing the advantages and limitations of proposed concepts, and outlining the potential future research opportunities.

CHAPTER 

FUNDAMENTALS AND RELATED WORK

ESSENTIAL fundamentals are presented in this chapter together with an analysis of the relevant state-of-the-art research. Section 2.1 discusses the concepts of cloud and serverless computing, and FaaS-based applications. Next, Section 2.2 focuses on relevant aspects of engineering applications for the cloud: in particular, it discusses (i) the general-purpose and FaaS-centric application modeling approaches, (ii) elaborates on the topic of function orchestration, (iii) analyzes relevant abstraction mechanisms, and (iv) discusses patterns and pattern-based design approaches. Finally, Section 2.3 analyzes existing approaches focusing on classification and decision support for selecting cloud services, discussing research focusing on FaaS and other component types. FaaS research was systematically analyzed by repeating the initial search phase from Section 1.1.1 in September 2022. Relevant publications in more general domains were collected and analyzed using Google Scholar search until reaching theoretical saturation [SRF16].

2.1 Serverless Computing & Function-as-a-Service

The ever-increasing reliance on cloud computing [MG11] can be attributed to the considerable advantages it brings to application developers including rapid elasticity [HKR13] and continuous deployment on a global scale [IB18]. Compared to on-premises environments, cloud computing significantly facilitates the management of infrastructure resources, thus reducing costs and decreasing time-to-market. Furthermore, various abstraction layers represented by different cloud service models, such as IaaS, PaaS, or SaaS, enable flexibly balancing infrastructure management efforts and the preferred characteristics of the development and operations processes, e.g., building applications using container orchestration engines or deploying them to more provider-managed PaaS offerings [Kav14].

Serverless computing is a novel paradigm [BCC+17] that continues the trend of shifting the infrastructure management responsibilities from cloud consumers to cloud providers. While the term “serverless” can be encountered in other contexts [FIMS17], e.g., peer-to-peer systems [YKK03] or RFID protocols [TSL08], in the cloud computing context, this term often refers to an architectural style focusing on applications developed as compositions of components hosted on provider-managed service offerings. The varying perception of serverless computing and the concepts it encompasses is highlighted in existing research [LWSH19]: while typically the term is associated exclusively with FaaS, other kinds of cloud services that do not require extensive management efforts are often deemed relevant too. The white paper on serverless computing by Cloud Native Computing Foundation (CNCF) [Clo18] also distinguishes two main serverless variants: Backend-as-a-Service (BaaS) and FaaS.

The term BaaS typically refers to application backends that are engineered as compositions of provider-managed APIs and do not rely on servers in a traditional sense. In contrast, FaaS is a cloud service model that enables developers to host arbitrary and typically short-lived code snippets that can be executed in an event-driven manner. Depending on the request rate,

functions are automatically scaled by providers including scaling to zero instances when no execution is necessary. The programming model behind FaaS shifts infrastructure management tasks to cloud providers, which simplifies managing business logic components in FaaS-based applications and improves their utilization [AC17; BCC+17].

Due to the limited execution time of FaaS-hosted functions [AC17], the application state is typically externalized to other kinds of provider-managed services [BGS20] such as AWS S3 [Ama22b] or Azure CosmosDB [Mic22], which also exhibit similar characteristics as FaaS, e.g., AWS S3 abstracts away the hosting- and management-related details regarding user-created buckets giving an impression of missing servers. Therefore, FaaS-based applications typically comprise other kinds of components such as storage, messaging, logging, or monitoring, which are often implemented using provider-managed offerings. While proprietary, provider-managed platforms such as AWS Lambda [Ama22b], Microsoft Azure Functions [Mic22], or IBM Cloud Functions [IBM21] reduce management efforts for application developers, open source FaaS platforms such as Apache Openwhisk [Apa22b] or OpenFaaS [Ope22] offer more customization options, e.g., they can be installed on container orchestration engines such as Kubernetes [The22c].

Serverless, FaaS-based architectures grow in popularity due to reduced hosting and management efforts as well as high scalability [ESE+21]. Existing research analyzing characteristics of publicly-available FaaS-based applications shows that this style does not prioritize any particular application type: while there are common FaaS-based scenarios such as serverless APIs or stream processing, FaaS-based applications are also used in other scenarios including complex data analysis and latency-critical tasks [ESV+21]. The increasing numbers of research published on the topic [YBLW19] is a good example of how FaaS is constantly being employed in various novel domains and contexts, e.g., building chatbots [LMM18; YCCI16], data processing [AIVP18; GMC19; SVSG18], scientific computing [SMM18], training and serving machine learning models [FKDH18; IMS18], edge and

fog computing contexts [BFG17; ILM+20], implementation of web components [AG17], processing Operating System (OS) commands [KY17], disaster management [ARI+18] and many others.

FaaS-based applications can exhibit event-driven behavior and also comprise orchestrated components since FaaS platforms can be employed in different operational styles [EGE+19]: (i) *function execution* that represents general function invocation based on the incoming events or direct calls, and (ii) *workflow execution* that represents orchestration of FaaS functions using workflow models [LR00] that specify the desired execution order using common control flow constructs such as sequence or parallel branching [AHKB03; RTVM06]. Such function orchestration models can then be enacted using external function orchestrators [GSP+18]. Multiple cloud providers offer orchestration services, e.g., AWS Step Functions, Azure Durable Functions, or IBM Composer. Additionally, there exist open source function orchestration engines that can be combined with installable FaaS platforms, e.g., Apache Openwhisk Composer can execute function compositions for Apache Openwhisk [Apa22c]. This kind of function orchestration technologies typically support only custom modeling formats that are not compatible with other orchestration engines.

Despite the aforementioned benefits, FaaS-based applications possess some well-known limitations [HFG+18]. For instance, the topic of stateful functions attracts research interest [BSS+22] since FaaS platforms often restrict the function execution time, which requires externalizing application state to various storage offerings, e.g., object storage or RDBMS. The so-called cold start problem is another well-known issue of FaaS: the time required to instantiate functions for processing initial requests is significantly longer than for subsequent requests that are processed by already available function instances [MEHW18]. The lack of standards and the increased dependency on provider-specific services are also among relevant issues [CIMS19; Hoh22b]. Further, despite the cost reduction [VGO+17], it was also shown that for certain scenarios the pay-per-use pricing model of FaaS incurs higher costs than when using traditional cloud service models such as IaaS [Eiv17].

Serverless and FaaS-centric research is blooming, with multiple topics being researched and a strong focus on the performance engineering aspects [EIA+18]. One of the recent studies [WCL+21] reinforces the insights obtained in the preliminary research phase: the challenges related to application design, deployment, and environment configuration as well as the low-code development constitute a substantial portion of questions asked by practitioners interested in FaaS-based applications. This thesis focuses on abstractions for engineering FaaS-based applications for existing offerings rather than on engineering custom FaaS platforms. Thus, the literature analysis presented next focuses on general-purpose and FaaS-specific cloud application modeling, model and code abstraction mechanisms, and service classification and selection.

2.2 Architecting Cloud and FaaS-based Applications

Whether it is migration of legacy applications [ABLS13] or cloud-native development [Ada17; LBWW17], engineering for the cloud requires deep understanding of the mechanisms behind various cloud service models and desirable properties of cloud applications: Lifting and shifting legacy software or green-field implementation of IaaS-hosted monoliths could often nullify the advantages of cloud [LFWW16]. Fehling et al. [FLR+14] identify the following IDEAL properties of cloud-native applications: (i) Isolation of state, (ii) Distribution, (iii) Elasticity, (iv) Automated Management, and (v) Loose coupling. Indeed, the distributed nature of cloud computing [BCK21] is well-aligned with SOA as it promotes better decoupling of components and loose coupling [Var10]. The rising popularity of MSAs [New15; Ric18] in the cloud reinforces this observation as MSA can be seen as a descendant of SOA [Zim17].

Works from industry and academia [CLFG15; Var10] highlight that cloud applications need to be designed for (i) fault tolerance since component instances are volatile, and (ii) scalability since cloud infrastructure can be easily scaled on-demand, which also favors parallelism. Due to volatility

of cloud resources, a special emphasis is put on deployment automation, which promotes the transparency of infrastructure changes [CLFG15] and enables CI/CD of cloud applications [HF10]. Further, since component instances might easily fail, the organized collection of logging and monitoring data becomes mandatory [CLFG15]. In addition to pre-cloud application security guidelines, all cloud layers must also be secured including protection of data in transit and at rest, management of credentials and access permissions, and timely updates of application dependencies [Var10].

A large variety of as-a-service offerings [Sch09] enables flexibly combining different infrastructure resources and functionalities to implement cloud applications, e.g., monitoring and load balancing components can be implemented using dedicated provider offerings. Typically, provider-specific services impose various restrictions [CLFG15] such as the use of custom technologies and formats. The issue of locking into product-specific requirements and characteristics is well-known [Gre97] and appears in many contexts [Hoh19; OST14]. Heterogeneity of cloud offerings and their distinct requirements becomes an additional obstacle for engineering cloud applications [HK10]. While designing and implementing for product-specific requirements is not often perceived as an issue [CLFG15], provider-agnostic reasoning can simplify making decisions for implementing new and migrating existing applications. Considering how rapidly provider offerings evolve, the substitution of services can become a necessity, e.g., due to the change of technology, cost optimization, unreliable Quality of Service (QoS), or legal requirements [Pet11].

In this context, a common recommendation is to employ well-established standards [Lip12; SRC13]. The complexity of cloud infrastructures and focus on their automated management for implementing continuous delivery pipelines [HF10] inspired numerous cloud modeling languages [BBF+18] as an alternative to more traditional architectural languages and techniques. In the following subsections, general and FaaS-centric modeling concepts are discussed together with the analysis of state-of-the-art abstraction mechanisms for engineering FaaS-based applications.

2.2.1 General Modeling Approaches

Models are vital for abstracting the reality to reason on the required purpose and there exist various modeling approaches such as closed-form mathematical representations, conceptual metaphors, or linguistic formalisms that facilitate describing different phenomena [RWLN89]. Since “all models are wrong but some are useful” [Box79], the appropriateness of a model and its abstraction level for a given task is of utmost importance [RWLN89]. Model Driven Engineering (MDE) is a paradigm that employs models as first-class citizens in software engineering processes [Ken02]. MDE relies on the concept of Model Driven Architecture (MDA), which separates the specification of system functionalities from their implementations for specific target platforms [OMG14]. In MDA, system functionalities are described using so-called Platform Independent Models (PIMs), which capture the structure and functionalities without technical details, whereas Platform Specific Models (PSMs) represent platform-specific implementation details. Various scenarios become possible using different kinds of mappings between such models, e.g., refinement of designs (PIM to PIM), refinement to different targets (PIM to PSMs), or mining abstractions (PSM to PIM) [Ken02]. MDE is widely-used in the context of cloud applications with multiple languages and modeling approaches aiming to facilitate various aspects such as model verification or code generation.

When it comes to architecture models, Bass et al. [BCK21] distinguish three major categories of structures, namely (i) component and connector, (ii) module, and (iii) allocation structures. *Component and connector structures* focus on interaction among components: typically, this is represented using Directed Acyclic Graphs (DAGs)-based models in which nodes represent different types of components at runtime and edges represent various connector types among them, e.g., synchronous or event-driven calls [MMP00]. These structures are crucial for assessing runtime characteristics, such as performance, availability, or security. *Module structures* capture which modules, e.g., classes and layers, constitute the system and how they are related. These structures are crucial for reasoning on

functional responsibility assignments and inter-module dependencies. *Allocation structures* capture how software parts are mapped, e.g., to test and execution environments [BCK21].

Software architectures can be specified using Architecture Description Languages (ADLs) – many domain-specific and general-purpose languages have been proposed over the past decades [MT00], e.g., Aesop [GAO94], Darwin [MDEK95], or ArchiMate [LPJ10]. Medvidovic and Tailor [MT00] describe the following core building blocks, which any ADL must enable specifying explicitly: (i) *components*, (ii) *connectors*, and (iii) *architectural configurations*. Modeling of interfaces for components that constitute an architecture must also be supported since without this information the architectural model becomes “a mere boxes and lines diagram” [MT00]. Another important criterion highlighted by Medvidovic and Tailor is that the *usability and usefulness of an ADL is rendered by maturity of the accompanying toolset*.

Architecture interchange languages such as ACME [GMW10] aim to enable mapping among architectural specifications in different ADLs: Such interchange between ADLs is achieved in ACME by capturing the “least common denominator semantics” of existing ADLs, which includes (i) components, (ii) connectors, (iii) points of communication with components (ports) and connectors (roles), (iv) so-called representations for hierarchical, lower-level descriptions of components or connectors, and (v) so-called rep-maps for associating internal system representations with external interface, e.g., associate internal and external ports [GMW10]. Any other information in ACME is modeled as property lists with no language-specific means to understand their semantics [MT00].

While general architectural approaches can be employed for modeling cloud applications, often the underlying languages are too coarse-grained and lack technical details. Perera and Perera [PP18] introduce a rule-based framework for the automated creation of high-level microservice and FaaS-based architecture models. System requirements, e.g., intended APIs and data storage, read/write privileges and expected client applications, are

collected as inputs to generate high-level architecture models fulfilling those requirements. The generated component-connector views, however, lack specifics and require extra effort when used in the context of modeling FaaS-based applications, e.g., extending component types and connector semantics, modeling of deployment artifacts, and specification of QoS requirements. A more comprehensive method by Lehrig et al. [LHB18] enables efficient creation of architecture models based on reusable architectural templates that formally capture best practices concerning, e.g., architectural styles and patterns. Architecture models can be iteratively refined using suitable templates and used for analyzing the QoS requirements for resulting architectures.

Unlike more abstract, general-purpose architecture languages, CMLs can be viewed as DSLs that aim to facilitate specifying applications in heterogeneous cloud infrastructures for different scenarios, e.g., cloud-native development or model verification [BBF+18]. Existing research [BBF+18] shows that CMLs often differ in scope, which includes (i) their intended purpose such as modeling architectures or generation of component implementations, and (ii) supported target cloud environments, e.g., specific cloud providers and service models. Underlying language characteristics often differ too, e.g., support for extensibility via ontological typing [Küh06], realization as an internal or external DSL [Fow10], or support for visual modeling [Moo09]. Moreover, a substantial part of available CMLs focus on the specification of deployment architectures [BBF+18] – typically as topologies of underlying components and their relations together with deployment configuration details for one or more cloud infrastructures. Additional modeling concerns such as QoS requirements are often specified using properties, policies (if such constructs are supported by the underlying language), or annotations.

Various cloud-centric approaches focus on specific subsets of offerings, e.g., IaaS or PaaS, which makes extensibility a crucial characteristic of the underlying language. For instance, Liu et al. [LLM11] introduce Cloud Orchestration Policy Engine (COPE), a framework for modeling VM-based cloud resources orchestration as a constraint optimization problem, which

can then be resolved into orchestration commands using the COPE solver. Di Cosmo et al. [DMZZ14] introduce Aeolus, a VM-centric component model for cloud environments that enables representing components with inter-component dependencies and conflicts, and non-functional requirements such as load limits. Abraham et al. [ÁCJ+16] present Zephyrus2, a tool for solving deployment optimization problems for VM-based applications that employ the Aeolus component model. Zhou et al. [ZHS+18] introduce CloudsStorm, a framework facilitating the deployment and management of VM-based cloud applications. A YAML-based DSL is used for modeling deployments and a management engine is used to enact the deployment and enable the management of deployed applications, e.g., auto-scaling or failure recovery. Gesvindr et al. [GGB20] present a model-driven approach and tooling for performance evaluation of PaaS-based architectures, which are modeled as collections of one or more components, one or more cloud resources, and an optional collection of entities representing data models. Resulting models are validated and, if valid, are used for the generation and compilation of a functional prototype. This prototype is then deployed with automatically generated sample data for benchmarking the created PaaS-based architecture. While potentially applicable, such approaches require an extra effort for supporting FaaS-specific modeling on the language and tooling levels.

Several CMLs-based approaches focus on modeling for multi-cloud settings with the underlying languages typically providing extensibility mechanisms and corresponding toolchains for the enactment and management of deployments. Leymann et al. [LFM+11] introduce the MOCCA method (MOVe to Clouds for Composite Applications) and a toolchain to facilitate moving applications to cloud infrastructures. In MOCCA, an architectural model of a given composite application is created and enriched with additional information such as deployment parameters and implementation artifact details to enable automated derivation of cloud distribution models, i.e., placement of components in multi-cloud environments. The MOCCA method relies on a metamodel that is based on labeled typed graphs, which enables representing various levels of abstractions,

e.g., coarse-grained architecture models similar to ACME or more concrete models with labels specifying deployment-related details as in CMLs. The introduced tooling enables graphical modeling of various models and enacting the deployment using the Cafe tool [MUL09]. The Methodology for Architecture and Deployment of Cloud Application Topologies (MAD-CAT) [INS+14] follows similar ideas: using multiple levels of abstraction, a cloud application model can be refined from coarse-grained component-connector models to deployment models. In MADCAT, coarse-grained architecture models are first refined into architectural units that represent technology-agnostic functional concerns, e.g., storage or computation. These architectural units can then be refined using decision trees into technical units, which represent techniques or patterns that can implement architectural units. Finally, the technical units are refined into deployment units representing actual, concrete deployments for those units.

Andrikopoulos et al. [AGLW14] introduce a variation of graph-based modeling relying on type graphs with inheritance. This approach enables flexibly exploring viable application deployment alternatives on different concrete cloud offerings relying on inheritance relations between abstract and technology-specific nodes, e.g., Ubuntu and Debian Linux inherit from an abstract Operating System node. Ferry et al. [FCS+18] present CloudMF, a framework for multi-cloud deployment of applications that relies on the CloudML language, which enables modeling multi-cloud application topologies following the MDE principles by differentiating between provider-independent and provider-specific models. Furthermore, the framework is capable of deriving deployment and adaptation plans based on provided CloudML models to enable the reconfiguration of applications at runtime.

Cloud Application Modelling and Execution Language (CAMEL) [AKR+19] is a CML and ecosystem of tools for management of multi-cloud applications at design time and runtime, which comprises several DSLs to cover different management aspects, e.g., modeling of requirements, metrics, scalability, and deployment modeling based on the metamodel of CloudML. Models created in these different DSLs are used for generating

suitable deployment plans for modeled applications, as well as plans for their reconfiguration and adaptation at runtime. Additionally, there exist special-purpose approaches focusing on particular quality attributes, e.g., Bocci et al. [BGF+22] introduce a declarative, security-centric approach for modeling cloud applications to enable analyzing whether components leak sensitive data and partitioning the application to avoid such leaks.

Standardized by OASIS, Topology and Orchestration Specification for Cloud Applications (TOSCA) [OAS20] is a CML promoting provider-agnostic modeling of cloud applications [Lip12], which supports ontological typing and is easily extensible for custom purposes. Application are modeled in TOSCA as so-called Service Templates: Figure 2.1 shows the structural elements that comprise them. A Topology Template is a DAG in which nodes represent components of some type and edges de-

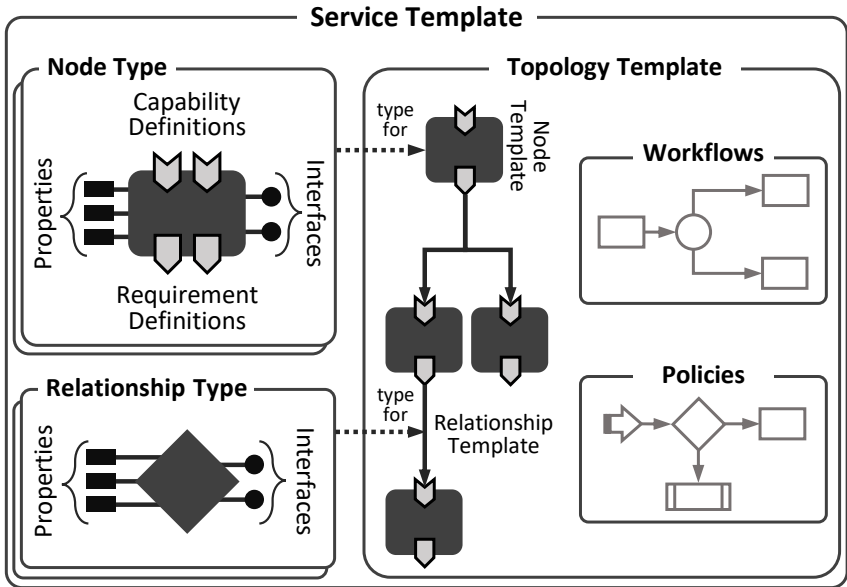


Figure 2.1: Core TOSCA Modeling Constructs [OAS20].

scribe how they are related, e.g., hosting and connectivity relationships. In TOSCA terms, the types of components and relationships are defined as reusable Node Type and Relationship Type definitions that comprise various type-specific properties, respectively. As multiple components of the same type can be present in the model, in Topology Templates these types are instantiated as Node and Relationship Templates that represent instances of components and their interactions.

To connect various nodes in TOSCA they need to have matching Capabilities and Requirements, e.g., a hosting requirement of an “Apache Tomcat” can be matched with the capability of an “Ubuntu Linux” node. Both Node and Relationship Types can specify interfaces that group lifecycle operations such as “create” or “configure” that can be enacted by a TOSCA-compliant deployment automation engine – actual implementations of these operations, e.g., in Ansible or Terraform, can be provided as TOSCA Implementation Artifacts. A deployed application is an instance of the modeled topology, which is created by running an ordered set of deployment steps – a Workflow in TOSCA terms. Such workflows can be derived automatically by the deployment automation engine or specified explicitly. In addition, non-functional and QoS requirements for a modeled application can be defined using TOSCA Policies, which are also specified as reusable Policy Types and can be instantiated as Policy Templates [OAS20].

Multiple TOSCA-based approaches and toolchains enable representing cloud applications as TOSCA Service Templates and deploying them to target infrastructures using compliant deployment orchestration engines. OpenTOSCA [BBH+13] is a software ecosystem for modeling, deploying, and managing cloud applications with TOSCA, which supports Extensible Markup Language (XML) and YAML Ain’t Markup Language (YAML) versions of the standard. OpenTOSCA includes Eclipse Winery [KBBL13], a graphical modeling environment for representing application topologies graphically. Produced application models can be deployed using OpenTOSCA Container, a deployment orchestration engine, which supports the automatic generation of imperative deployment models. EU Hori-

zon 2020 project SODALITE [KQR+22] is a TOSCA-based toolchain for modeling and deployment of applications to heterogeneous infrastructures including cloud, edge, and High Performance Computing (HPC) environments. Bogo et al. [BSNB20] and Brogi et al. [BRS18] introduce distinct TOSCA-based solutions for deployment of multi-service applications with the focus on containerized components and without supporting deployment of component orchestrations. Tsagkaropoulos et al. [TVC+21] propose a TOSCA-based approach for modeling edge and fog deployments, which also incorporates constructs for modeling FaaS functions without support for modeling function orchestrations. DesLauriers et al. [DKA+21] present MiCADO, a TOSCA-based multi-cloud orchestration framework that focuses on portable modeling of cloud applications and portable deployments to IaaS offerings.

In addition, there exist a variety of general-purpose deployment modeling approaches suitable for modeling FaaS-based applications. As deployment models can be defined both imperatively and declaratively [EBF+17], different kinds of technologies and underlying languages fit here. Imperative deployment models describe required deployment control flow explicitly, e.g., using workflow modeling languages [LR00] such as BPMN [OMG11]. Declarative models enable the programming style “where you say what you want done, but not how to do it” [AU22] and are commonly used for deployment modeling [EBF+17], e.g., modeling languages employed by technologies like Terraform [Has22] or Kubernetes [The22c] hide the imperative “hows” of the deployment process. The term Infrastructure-as-Code (IaC) [Mor20] is frequently used in the context of declarative deployment models: Major cloud providers offer provider-specific deployment automation services, e.g., AWS CloudFormation [Ama22b] enables modeling deployments for AWS resources using a custom DSL, Azure Resource Manager [Mic22] relies on a custom DSL for modeling Azure deployments, and Google Cloud Deployment Manager [Goo22] has a DSL for modeling deployments in Google Cloud. General-purpose deployment automation and infrastructure configuration technologies provide

custom DSLs too: Terraform [Has22] relies on Hashicorp Configuration Language (HCL) and Ansible [Red22] uses a YAML-based language for specifying Ansible playbooks.

Efforts similar to architectural interchange languages exist for CMLs: Generalized Topology Language (GENTL) [ARSL14] is an example topology modeling interchange language and tooling that aim to enable mappings between various topology-centric languages via a generic metamodel. GENTL enables specifying various additional information in the topology using typed annotations. Moreover, with the intention to capture the core semantics of deployment modeling languages, Wurster et al. [WBF+19] introduce the Essential Deployment Metamodel (EDMM) by analyzing most popular deployment automation technologies and underlying languages. EDMM captures essential modeling constructs present in analyzed technologies. Additionally, a toolchain that enables EDMM-compliant modeling in TOSCA and subsequent transformation of the resulting models into technology-specific formats, e.g., Kubernetes or Terraform, is presented [WBB+19; WBH+20b].

Based on the presented analysis of various modeling approaches and the observations from Section 1.1, several characteristics of a preferred CML for modeling FaaS-based applications can be highlighted. Firstly, to accommodate for heterogeneous component types in multi-cloud settings, the CML should support multiple cloud service models and cloud providers. Furthermore, to simplify incorporating novel provider-managed services, a CML needs to be easily extensible, which makes languages with ontological typing more practical [BBF+18]. In addition, the language should provide modeling constructs for the specification of various QoS requirements. Finally, to support the use of code-level abstractions, the language should enable explicit modeling of deployment artifacts such as packaged functions or function orchestration models. TOSCA, CAMEL, and generic metamodels such as MOCCA or EDMM possess these characteristics, with TOSCA, CAMEL, or EDMM being more suitable candidates for the stated task due to the actively-maintained toolchains.

2.2.2 Approaches for Modeling FaaS-based Applications

The landscape of FaaS- and serverless-oriented tooling is rapidly evolving, with multiple modeling approaches listed on CNCF Serverless Landscape [Clo22a] focusing on deployment modeling. For example, AWS SAM [Ama22e] is a framework for building serverless applications from AWS in which the application structure and configurations are described in a cleaner and more concise way than using AWS CloudFormation, their general-purpose deployment modeling offering. Another example is Serverless Framework [Ser22b] is an open source framework for modeling deployments of FaaS-based applications for different FaaS platforms. This framework relies on a DSL for specifying the structure and configuration of FaaS-based applications, but due to significant differences among FaaS platforms, the modeling constructs and the resulting models are platform-specific. Often, FaaS platforms provide some mechanisms to facilitate function deployment, e.g., Apache Openwhisk has a DSL for describing function deployments and OpenFaaS provides a GUI for deploying functions. However, being developed for specific (and often single) purpose, the industrial tooling and underlying models are often tailored for particular infrastructure and lack desirable CML characteristics such as ontological typing or means to model QoS requirements.

Multiple FaaS-centric research approaches rely on directed graphs, encoding different semantics into nodes and edges depending on the purpose of the model. Winzinger and Wirtz [WW19] present a graph-based model in which nodes represent functions/resources and edges describe different interaction semantics, e.g., direct calls between functions, storage read and write operations. Various kinds of annotations can be used to represent additional semantics such as synchronicity of calls or access rights. Lin and Khazaei [LK21] use DAG-based models for optimizing performance and costs of FaaS-based applications. Elgamal et al. [ESNA18] introduce concepts for optimizing costs of running FaaS-based applications. Applications are modeled as DAGs in which nodes represent functions and edges represent dependencies among them – these models are used

to derive cost graphs that represent various placement options for those functions (including cases when some functions are fused together) on cloud and edge resources. Other kinds of modeling approaches include, for example, a Petri nets-based model [TCBR21] of FaaS-based applications to reason about capacity and cost planning for the deployment of FaaS functions. The model is intended to help exploring various design and deployment scenarios by experimenting with different configurations of a FaaS-based applications such as different costs, execution times, or kinds of resources. Further, Moczurad and Malawski [MM18] present a visual-textual modeling approach for FaaS-based applications using the Luna, a functional, statically typed programming language that combines a textual representation of source code with a graphical view in the form of dataflow graphs. The approach introduces the extensions to the Luna toolchain to enable modeling, configuring, and invoking FaaS functions for AWS Lambda to provide a more intuitive development experience.

Many research works focus on modeling deployment architectures. In approach by Wurster et al. [WBK+18], FaaS-based applications are modeled as directed typed graphs with edges encoding dependencies among components of the application using TOSCA. For example, hosting relationships and event bindings can be implemented and automatically configured by compliant deployment orchestration engines using this approach. Kritikos et al. [KSMM19] propose extensions to CAMEL, which enable modeling and deployment of FaaS-based components including functions, event bindings, and event sources. Samea et al. [SAA+19] present a Unified Modeling Language (UML) Profile for Multi-Cloud Service Configuration (UMLPMSC) in event-driven serverless applications. This profile enables modeling functions and resources for different providers and uses events as first-class citizens in the model; however, it does not consider modeling function orchestrations and code artifacts. Sokolowski et al. [SWS21] introduce a modeling language and a deployment automation engine to enable continuous coordinated deployment of serverless applications maintained by distributed teams. This approach enables different teams to model their deployments separately, which are then continuously coor-

minated by the deployment engine that verifies and satisfies inter-stack dependencies. The authors discuss that their approach can be used in synergy with EDMM-based deployment models [WBF+19] that could provide a global application view. Pelle et al. [PPSC21] present concepts and a framework for automated and adaptive deployment of functions to edge or cloud offerings of AWS. FaaS-based applications are modeled as directed graphs with nodes representing functions and storage components, whereas function calls and storage read/write operations are modeled as edges. Developers provide application structure, deployable function packages, and QoS requirement specification as an input, and the framework optimally places functions to cloud or edge offerings from AWS.

Risco et al. [RMNB21] introduce OSCAR, a platform enabling the execution of FaaS-based data processing applications (functions are provided as Docker containers) on different infrastructures, e.g., functions can be executed on OpenFaaS and AWS Lambda. Applications are modeled using the so-called Functions Definition Language (FDL), a declarative YAML-based DSL that describes the deployment of containerized functions, storage resources, and links between functions and input/output storage resources such as object storage buckets. Ferry et al. [FDS22] present Serverless4IoT, a deployment modeling language for hybrid IoT (FaaS-based) applications and tooling that enables executing such models. Tankov et al. [TVGB21] introduce a deployment modeling framework for FaaS functions developed in Kotlin that enables specifying infrastructure in-code and automatically deploying specified resources on AWS and Azure infrastructures. The modeled infrastructure requirements are first translated into a so-called provider-agnostic schema, which is then used for the generation of a technology-specific deployment model that can be automatically enacted by the target deployment automation technology.

EU Horizon 2020 project RADON [CAH+20; DGD+21] proposes a TOSCA-based framework that aims to facilitate development and operations for FaaS-based applications. The modeling profile produced over the course of RADON was shown to be applicable in different contexts,

e.g., performance engineering [GHZ+20; ZGTC21a], data pipeline modeling [DJS+20; DJS+22], autoscaling configuration [CLT20], and continuous testing [DHY+22].

The analysis of FaaS-specific modeling approaches, underlying languages, and respective toolchains is aligned with the insights obtained from analyzing general-purpose modeling approaches: CMLs often provide means for realizing different abstraction levels and means to transition between them, e.g., by using type systems to distinguish between PIMs and PSMs. Among standard-based approaches, TOSCA is encountered most frequently – in particular, the modeling approach employed in the EU Horizon 2020 project RADON demonstrates that TOSCA enables flexible FaaS-centric modeling of various quality attribute requirements including performance, testability, and deployability.

2.2.3 Modeling Function Orchestrations

The concept of service compositions [Pel03] is crucial in the cloud domain due to the distribution, loose coupling, and finer-granularity of components that constitute cloud applications. Different approaches can be employed for modeling service compositions including service *orchestrations* and *choreographies*. Service orchestration models represent the perspective of a central coordinator [HBLW17] responsible for orchestrating services following the specified control flow to achieve a desired business goal. In the domain of Business Process Management (BPM), the terms *workflow* or *process* are often used to describe service orchestrations [LR00]. Business processes spanning typically heterogeneous, non-integrated services can be specified as workflow models using languages such as BPMN [OMG11] or Business Process Execution Language (BPEL) [OAS07] and automatically executed by compatible workflow management systems. In contrast, service choreography models represent a general view on conversations happening between multiple participants without centralized coordination, e.g., BPEL4Chor [KLW11] is an example choreography language.

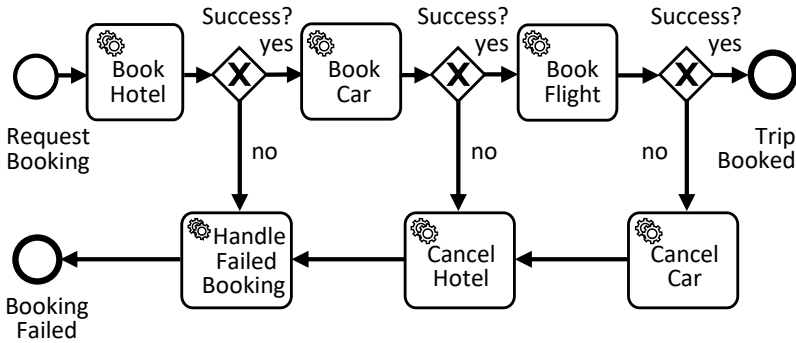


Figure 2.2: A travel booking process modeled in BPMN [Rue20].

BPMN is a well-established standard for modeling workflows [CT12; OMG11], which provides a visual notation supporting various control flow patterns [AHKB03; RTVM06] to represent complex service orchestrations that can be enacted by BPMN engines such as Camunda [Cam22]. Workflows in BPMN are modeled by connecting *Activities* with *Sequence Flows* to express the order of execution. Activities represent units of work and can be compound (*Sub-Processes*) or atomic (*Tasks*). Tasks are modeled as rounded rectangles with labels for task names. Sub-Processes are modeled (i) as regular tasks with a “+” marker (collapsed) or (ii) as explicit inner control flow similar to regular processes (expanded). *Events* express occurrences of some facts at process execution, e.g., *start events* instantiate processes (single-border circles), *end events* express the end of process instances (thick-border circles), and *intermediate events* occur in-between (double-border circles). *Gateways* express divergence and convergence of sequence flows: they are modeled using diamond shapes with internal markers describing routing behavior, e.g., “+” represents parallel whereas “X” denotes exclusive gateways to express parallel and conditional execution, respectively [COZ17]. Figure 2.2 depicts a travel booking process modeled in BPMN [Rue20]. Upon a booking request, workflow management system executes the activities for booking a hotel,

a car, and a flight. In case of a failure, the cancellation of already executed tasks takes place by following the respective conditional flows. Each task can, for example, be realized as a microservice processing corresponding booking requests. Additionally, BPMN extensions can be defined to extend language capabilities, e.g., modeling and generation of test cases from BPMN models [LL16; LL17] or modeling of blockchain-aware business processes [FHBL19]. General-purpose workflow management systems can be used for orchestrating FaaS functions, e.g., AWS Lambda functions can be orchestrated using Camunda [Rue20].

Chaining of functions via direct calls is generally not recommended, instead multiple functions can be either merged into one or composed using workflows [NT20]. In this light, several works introduce languages for modeling and execution of function compositions as a part of the FaaS platform. Baldini et al. [BCF+17] introduce the Serverless Trilemma (ST), which shows how only two out of three constraints can be satisfied when implementing function compositions using the event-driven paradigm, namely: (i) *double-billing constraint*: function calls should be billed once, (ii) *black box constraint*: function code should not be assumed available, and (iii) *substitution principle*: synchronous function calls are indistinguishable from invocation of function compositions. Further, the authors introduce an ST-safe way to implement sequential compositions in Apache Openwhisk. Jangda et al. [JPBG19] capture operational semantics of a FaaS platform in a process-calculus style: the state of a FaaS platform comprises a collection of running or idle functions, pending requests, and responses. The introduced formalisms enable reasoning, e.g., on the safeness of isolated functions. Authors also present the Serverless Composition Language (SPL) to model function compositions, and provide a prototype based on Apache Openwhisk. Giallorenzo et al. [GGL+19; GLM+20] introduce Serverless Kernel Calculus (SKC) inspired by λ - and π -calculus to enable modeling compositions of serverless functions. Gerasimov [Ger19] introduces Anzer, a DSL for the type-safe composition of FaaS functions with a prototype based on Apache Openwhisk for the functions in Go programming language.

Due to the heterogeneity of FaaS platforms, function orchestrations can span different kinds of infrastructures. For example, Bocci et al. [BFFB21] identify research gaps and opportunities for FaaS orchestrations in fog environments by surveying existing works. Particularly, the authors investigate means to model and execute FaaS orchestrations in fog environments as well as available techniques and methodologies for securing such orchestrations. Multiple works focus on orchestrating FaaS functions using novel orchestration engines that often support more than one target infrastructure for running functions. Ristov et al. [RPF21a; RPF21b] introduce the Abstract Function Choreography Language (AFCL) for modeling function choreographies, which abstracts away function implementation details. The authors implement a system that can execute AFCL models with function running on AWS Lambda and IBM Cloud Functions. Arjona et al. [ALS+21] present TriggerFlow, a trigger-based event-driven architecture for function orchestration that performs state transitions using event-driven invocations. Function orchestrations are modeled as finite state machines in which states are functions, whereas transitions are represented by triggers. Jiang et al. [JLZ17] introduce DEWE, a workflow management system that enables executing tasks on different FaaS platforms, such as AWS Lambda and Google Cloud Functions, and local clusters for running large-scale scientific workflows. DEWE relies on a custom XML-based DSL for the specification of workflow models.

Smirnov et al. [SEM+21] present Apollo, a system for orchestrating functions on heterogeneous infrastructures in cloud and edge contexts. Orchestrations are modeled using a graph-based system model that comprises two distinct graphs, namely enactment graphs that model interconnections of compute (functions) and data tasks (produced/consumed data), and resource graphs that model interconnections of resources capable of running such tasks such as FaaS platforms or VMs. These two graphs are interconnected using mapping edges that indicate which tasks can run on which resources. Daw et al. [DBK20] introduce Xanadu, a just-in-time resource provisioning function orchestration engine that relies on function profiling for the mitigation of cascading cold starts. Function orchestrations are

modeled using a custom JSON-based DSL and functions are executed by Xanadu's worker components - either using container- or thread-level isolation. Burckhardt et al. [BCG+22] present concepts and tooling for enacting function orchestrations employed by Azure Durable Functions – function orchestrations are modeled using general-purpose programming languages such as Python.

John et al. [JAM+19] introduce a workflow engine for executing scientific workflows that also supports running tasks on offerings such as AWS Lambda and AWS Fargate. Malawski et al. [MGZ+17] experiment with executing scientific workflows as orchestrations of functions hosted on AWS Lambda and Google Cloud Functions using the HyperFlow workflow engine. HyperFlow relies on a custom JSON-based DSL for defining workflow models, which was shown to support FaaS function orchestrations and serverless-style containers running on AWS and Google offerings [BPB+21]. Kousouris et al. [KAC+22; KGTS22] orchestrate FaaS functions on different platforms (with the focus on Apache Openwhisk) using Node-RED, a framework for event driven applications that enables visual editing of workflows packaged as functions.

Modeling of workflows can also be supported on the level of CMLs. For example, Kritikos et al. [KZI+19] present a CAMEL-based approach for modeling and deploying workflows in multi-cloud settings. The presented approach enables deploying workflows that comprise activities implemented using SaaS offerings or as components deployed to PaaS or IaaS offerings. OpenTOSCA [BBH+13] enables modeling application deployments as workflows, which are supported in TOSCA as the imperative parts of the deployment model. Additionally, several works investigate the standards-based execution of function orchestrations. The Cloudstate project [Clo22c] focuses on standardizing the implementation of general-purpose applications that comprise stateful services, as well as reactive and data-intensive components deployed on Kubernetes ecosystem. The Serverless Workflow Specification [Ser22a] is a standard for modeling function orchestrations, which includes a DSL, language-specific tools and SDKs, and a Kubernetes-native orchestration engine.

The concepts presented in Contribution 3 differ from the aforementioned approaches since they do not depend on a novel workflow language or engine – instead an existing standard is used for the technology-agnostic modeling of function orchestrations, and transformation concepts and toolchain are provided to enable the uniform generation of different technology-specific models, e.g., ASL models executed on AWS Step Functions. The concepts in Contribution 3 can be extended to support more languages including those introduced in the aforementioned research works.

2.2.4 Abstractions for FaaS-based Applications

An abstraction can be seen as a mechanism for selectively hiding information, e.g., software systems can often be modeled on different levels of abstraction to control the amount of considered information [BEDM98]. Moreover, transitioning between such abstraction layers could enable flexible refinement of a given system [Flo08]. Cloud offerings are typically not built with interoperability in mind, meaning that an implemented application often cannot run as-is on different cloud providers, which hinders the portability of developed applications [LKBT11].

The use of standards and standard-compliant tooling is one way to abstract away the specifics of heterogeneous technologies [BC10]. Standardization of interfaces is generally considered to be a priority [Hoh20] and various standards aim to standardize interfaces. For instance, the Open Cloud Computing Interface (OCCI) is a standardized resource management interface and the Cloud Data Management Interface (CDMI) is a standard focusing on the interoperability of cloud storage services [LKBT11]. The standardization of specific deployment artifact types such as VMs or containers is another example: the Open Virtualization Format (OVF) standard [DMT15] focuses on the packaging of VMs, whereas the Open Container Initiative (OCI) specifications [The22d] aim to standardize runtime, packaging, and distribution aspects for containers.

As a cloud modeling standard example, TOSCA [Lip12; OAS20] enables hiding provider-specific details via substitution mechanisms. Additionally, TOSCA also enables incorporating multiple deployment automation technologies using the same interfaces within models that can be enacted by TOSCA-compliant deployment automation engines, e.g., a TOSCA deployment automation engine xOpera [CAH+19] relies on Ansible but also enables modeling invocations of other automation scripts using the same mechanisms. The CloudEvents specification [Clo22b] introduces a standard schema for describing heterogeneous events occurring in the cloud, which can be employed to describe FaaS-specific events using a neutral schema. The Serverless Workflow Specification [Ser22a] aims to enable executing orchestration models on different standard-compliant function orchestration engines.

In addition to standards, a common path to address interoperability and portability issues is to introduce new abstraction layers [BC10; LKBT11]. Approaches addressing cloud lock-in issues often rely on abstraction layers to enable provider-agnostic reasoning and use adapters for interacting between introduced abstractions and concrete target infrastructures [SRC13]. This can be observed in multiple existing migration approaches [BBSR13; BBKL14; BLS11; FH11] which tackle the heterogeneity of target offerings by first discovering more abstract models using legacy applications as an input, then mapping and transforming those models into suitable technology-specific models for migration.

The increased amount of provider-specific dependencies in FaaS-based applications inspires multiple approaches that focus on abstraction mechanisms. For example, the need to uniformly execute functions on different infrastructures is addressed in many approaches via a central entity that mediates the execution tasks, e.g., a novel FaaS platform or a cloud broker. Chard et al. [CBL+20] introduce FuncX, a distributed FaaS platform that enables flexibly executing functions on different target infrastructures, e.g., clouds and supercomputers. Vandebon et al. [VCL+20] introduce SLATE, a system for enabling running FaaS functions on heterogeneous infrastructure resources, including CPUs and FPGAs. Jindal et al. [JFCG21];

JGC+21] present the concept of Function Delivery Network (FDN) for running functions on heterogeneous target infrastructures and introduce Courier, a system that enables distributing calls to functions deployed on heterogeneous platforms including AWS Lambda and Google Cloud Functions. Sheshadri and Lakshmi [SL22] introduce a FaaS platform that enables executing functions in edge and cloud environments based on user-specified QoS requirements. Baarzi et al. [BKJS21] envision the concept of virtual service providers that abstract away various FaaS offerings via uniform mechanisms for deploying and invoking function code.

Elhabbash et al. [EJBE19] introduce SLO-ML, a JSON-based DSL for modeling Service Level Objectives (SLOs) that supports a large number of SLOs for various cloud application kinds. SLOs are described as maps in which keys represent components and values describe different SLOs, e.g., monthly bandwidth cost. Authors also implement a brokerage system prototype that enables cloud service selection and generates deployment models that can be enacted by Terraform using high-level provider-independent SLO descriptions in SLO-ML as inputs. Rampérez et al. [RSL+] present a brokerage-based approach, which enables mapping between high-level SLOs and low-level, metrics-based conditions for cloud applications. To enable this for multi-cloud settings, the authors introduce a set of vendor-neutral metrics for various kinds of application components, e.g., PaaS offerings, object storage, or MySQL RDBMS, as well as mappings for different service kinds across three cloud providers – AWS, Azure, and IBM. Spillner [Spi21] proposes a concept of liquid functions that can be executed on different infrastructures without requiring explicit packaging and deployment. An initial prototype, however, does not focus on how functions are deployed on heterogeneous infrastructures.

Several works focus on higher-level programming languages that abstract away platform-specific details of FaaS functions: Scheuner and Leitner [SL19] envision a code transpilation approach for the automatic generation of boilerplate code to comply with provider-specific requirements, hence enabling deploying functions implemented in a provider-agnostic manner to different FaaS platforms. Cordasco et al. [CDN+21; CDN+20]

introduce Fly, a DSL and a source-to-source compiler for enabling the generation of FaaS platform-specific functions, e.g., AWS Lambda and Azure Functions, in Java, JavaScript, and Python from functions implemented using the Fly language. Function deployment is performed via provider-specific APIs.

Kehrer et al. [KZSB21] introduce concepts and tooling for parallel cloud programming using so-called self-tuning serverless skeletons based on the idea of algorithmic skeletons [Col89] that separate user-provided business logic from non-functional execution aspects. While the presented concept of serverless skeletons is employed strictly in the context of parallel computing, it can also be seen as a way to abstract provider-specific logic for generating function packages for different platforms. Tzouros et al. [TTK21] introduce a Java-based framework providing Intermediate Representations (IR) for data analytics scenarios. Similar to classical compilers, IR serves as an abstraction layer for expressing data functions in platform-independent manner. As a result, function structures specified as a context-free grammar are used for generating portable containers that can be deployed to different FaaS platforms such as IBM Cloud Functions or OpenFaaS. Rodrigues et al. [RFS22] present QuickFaaS, a desktop tool that provides a platform-agnostic way to implement FaaS functions in Java and deploy them to multiple cloud providers, thus simplifying the reuse of FaaS code artifacts.

Sampe et al. [SGS+21] introduce Lithops, a framework that enables running distributed computations in Python on heterogeneous cloud infrastructures without requiring to modify the implemented code. The framework hides the specifics of deploying and running code on FaaS platforms, hence, providing a uniform development experience suited for parallel computations that do not require significant inter-process communication. Son et al. [SMG+22] introduce Splice, a framework for cost- and performance-aware fusion of IaaS and FaaS offerings. Annotations that specify desired code transformations, e.g., whether a function should be placed on a FaaS platform or a VM, are analyzed to automatically generate the target-compatible code.

Simplified development experience for heterogeneous FaaS platforms also inspires various approaches that aim to abstract the provider-specific requirements: Li et al. [LLTY21] introduce a framework that abstracts away the specifics of FaaS platform SDKs to provide uniform development experience, e.g., invoking functions or managing event bindings. Cordingly and Lloyd [CL22] present FaaSET, a framework aiming to abstract the differences in development and interaction with heterogeneous FaaS platforms. Chatley and Allerton [CA20] introduce Nimbus, a framework for developing FaaS-based applications in Java for the AWS services ecosystem. This approach employs annotations to specify platform-specific configurations such as event bindings and infrastructure components such as object storage buckets directly in code.

Several approaches focus on the topic of FaaSification, i.e., the automated extraction and deployment of legacy functions to various FaaS platforms. Spillner [Spi17] presents Lambada, a tool that enables extracting legacy Python code as FaaS functions. Further, Spillner et al. [SMM18] discuss different kinds of FaaSification, e.g., shallow FaaSification focuses on extracting classes and function collections, whereas deep FaaSification is related to extracting single instructions as functions. Dorodko and Spillner [DS19; SD17] present tooling for annotation-based extraction of Java code as FaaS functions for AWS Lambda. Kaplunovich [Kap19] presents ToLambda, a tool for extracting Java functions and generating JavaScript packages for AWS Lambda from them.

Klingler et al. [KTS21] further extend the concept of annotation-based FaaSification as they present various annotations that enable, e.g., warming up function instances or configuring tracing. The authors also present a prototype supporting these annotations for JavaScript code extraction and transpilation for AWS Lambda. Ristov et al. [RPWF21] introduce Dependency-Aware FaaSifier (DAF), which enables extracting JavaScript functions together with their dependencies and packaging them for AWS Lambda. Pedratscher et al. [PRF22] present M2FaaS, a framework that enables dependency-aware (code, package, and data dependencies are

considered) extraction of code blocks from Node.js applications using annotations and generating deployable packages for the FaaS platforms provided by AWS and IBM.

The analyzed research works mainly aim to enable running the same code on different target infrastructures: either by introducing an intermediary that is responsible for deploying and running functions on different infrastructures or by providing code abstraction mechanisms such as higher-level languages or annotation-based libraries that enable the generation of platform-specific function packages. The contributions presented in Contribution 3 are well-aligned with respect to existing research: The higher-level modeling abstraction and transformation approach for function orchestration models can be combined with generative approaches for function code, e.g., by Cordasco et al. [CDN+21; CDN+20] or Kehrer et al. [KZSB21]. One of the first-authored publications [YBHL19] contributing to this thesis uses the annotation-based and dependency-aware FaaSification for Java code similar to the earlier work by Dorodko and Spillner [DS19]. Additionally, none of these approaches uses the introduced abstractions on the level of application models.

2.3 Decision Support for Selecting Cloud Services

Increased vendor lock-in in FaaS-based applications [Hoh22b] complicates reasoning in a provider-agnostic manner. Further, providers typically offer similar offerings. For example, AWS Lambda and AWS Fargate enable deploying serverless-style containerized functions. The identification of suitable services and their mappings from other cloud providers is a reoccurring task that can be encountered when implementing, e.g., use case applications [PLMR19] or solutions that intend to facilitate service selection [QRD16]. This section discusses and analyzes works focusing on the classification and selection support for cloud applications including FaaS-specific research.

2.3.1 General Selection and Classification Approaches

Ontology engineering is a well-known way to classify and organize domain-specific knowledge, which is also frequently employed for classifying cloud services [AVS12]. Numerous research publications introduce taxonomies and ontologies focusing on the general classification of cloud offerings [ABMT17; DT15; GJGN13; HS10; HK11; KSHD13; MAD+11; RCL09]. Such approaches are often coarse-grained and focus on high-level business characteristics, and the lack of technical details can be credited to the significant conceptual differences between cloud service models [Kol19]. In addition, there exist approaches aiming to refine the taxonomy of cloud services, e.g., with general finer-grained offerings such as Hardware-as-a-Service and Framework-as-a-Service [KSHD13], or by focusing on specific service types such as data hosting solutions [SKLU11]. Indeed, general classifications of cloud services often focus on the least common denominator listing high-level properties, e.g., for payment, security, cloud deployment models, and licensing. Aligned with these ideas, the business view of the classification framework for FaaS platforms introduced in Contribution 2 comprises similar criteria. However, none of the aforementioned approaches considers FaaS, which makes them non-optimal for FaaS-specific decision-making.

Another classification direction targets cloud service models separately, e.g., exclusively IaaS or PaaS. An IaaS-specific classification approach was introduced by Repschlaeger et al. [RWZT12]. The authors derive a classification framework for IaaS offerings by conducting expert interviews and analyzing existing literature. The framework captures various business and technical properties organized in different dimensions, e.g., the Costs dimension comprises criteria like “price level” and “payment method”, the Scope & Performance dimension comprises criteria like “instance type” and “instance capacity”. Dukaric and Juric [DJ13] define a unified taxonomy for IaaS service offerings by analyzing existing commercial and open source technologies. The authors also introduce a layered architectural framework for IaaS that logically organizes the concepts captured by this

unified taxonomy. Prodan and Ostermann [PO09] capture various business and technical aspects for IaaS and PaaS offerings with a stronger focus on IaaS. Hilley [Hil09] presents a taxonomy covering IaaS and PaaS with various criteria organized in dimensions like Scaling, Integration, or Interoperability. Loutas et al. [LKT11] introduce a semantic interoperability framework for PaaS capturing business and technical aspects, but without focusing on automated selection support.

Multiple research works go beyond classification and also focus on automated service selection based on provided requirements. Sun et al. [SDH+14] conduct a comprehensive review and categorization of existing cloud service selection methods. The majority of methods fall into three categories: (i) Multi-Criteria Decision Making (MCDM)-, (ii) optimization-, and (iii) logic-based methods. Various MCDM-based methods can be employed when the number of decision criteria and alternatives is finite, e.g., Analytic Hierarchy Process (AHP) for hierarchically-structured decision elements, simple additive weighting, Multi-Attribute Utility Theory (MAUT) and others. Optimization-based methods can be applied to approximate the solution, e.g., dynamic programming and greedy selection algorithms can be employed when the number of alternatives is large. Additionally, first-order logic and description logic-based methods can be applied to filter irrelevant services. Since this thesis does not aim to introduce novel service selection algorithms, only selected selection support approaches are described in the following, whereas the aforementioned study can be accessed for more details on selection algorithms.

Bassiliades et al. [BSG+18; BSM+17] present an ontology-based classification of PaaS offerings and a selection support system called PaaS-support. Kolb [Kol19] introduces concepts for classification and selection of PaaS platforms, and the selection support system called PaaSfinder that prototypically implements these concepts. Di Martino et al. [DCE14] present an ontology focusing on PaaS and SaaS offerings. Magoutis et al. [MPP+15] introduce PaaSage, a social networking platform for developers and operations engineers that aims to facilitate creating deployment models and deployment automation scripts, and the reuse of artifacts shared

by the community. Additionally, PaaSage enables mining statistics and cost-benefit analyses to provide improvement suggestions. Dastjerdi et al. [DGRB15] introduce CloudPick, a framework for facilitating multi-cloud IaaS deployments by optimizing the selection of suitable services based on specified QoS criteria. Quinton et al. [QRD16] introduce concepts and tooling for the automated identification and configuration of multi-cloud environments (focusing on IaaS and PaaS) based on given functional and non-functional requirements. The approach combines ontologies and feature models to capture platform- and provider-specific knowledge in a uniform manner. Spillner et al. [SGBV20] present concepts for rule-based resource matchmaking for application deployments in the context of osmotic computing [VFD+16], i.e., containerized components opportunistically deployed in cloud, edge, and fog environments. The presented model and algorithms that return deployment plans for given component-based applications do not consider data dependencies or workflows. Other classification and selection support examples include Cloud4SOA [DBC+12] focusing on PaaS, and SeaClouds [BFI+15] and DrACO [BCS17] targeting IaaS and PaaS. As with general-purpose service classification approaches, these research works also capture some business characteristics overlapping with the criteria from the business view in Contribution 2. None of them, however, presents criteria relevant for the technical FaaS view. Similar considerations apply to other studies targeting decision support for cloud-based deployments [DPC+10; JMK+13; MR12; PZL+09; SSL12].

2.3.2 FaaS-specific Selection and Classification Approaches

Multiple existing works analyze and compare FaaS platforms, both quantitatively and qualitatively. Kritikos and Skrzypek [KS18] assess “serverless frameworks” using criteria related to application lifecycle phases including design, development, and deployment. This assessment includes qualitative analysis of open source FaaS platforms such as Fission and Kubeless that are referred to as provisioning frameworks. Additionally, authors

analyze “abstraction frameworks”, which abstract one or more platforms, e.g., Serverless framework [Ser22b] is placed into this category. Lee et al. [LSF18] evaluate the performance of commercial FaaS platforms including FaaS offerings from Amazon, Microsoft, Google, and IBM, and additionally compare them feature-wise. Lynn et al. [LRLE17] analyze commercial serverless platforms, such as AWS Lambda, Microsoft Azure Functions, and Google Cloud Functions, using a set of qualitative criteria. Mohanty et al. [MPd18] evaluate the performance of four open source FaaS platforms, namely Kubeless, OpenWhisk, Fission, and OpenFaaS, and also compare them feature-wise. Werner and Tai [WT21] present a review of four commercial FaaS platforms, namely AWS Lambda, Azure Functions, Google Cloud Functions, and IBM Cloud Functions using different quantitative and qualitative criteria related to aspects of deployment, execution, configuration, and measurement as specified in the official documentation. Wen et al. [WLC+] analyze characteristics documented in the official documentation of four FaaS platforms related to the development, deployment, and runtime aspects and quantitatively analyze the runtime using benchmarks. López et al. [GSP+18] experimentally evaluate three commercial function orchestrators, namely AWS Step Functions, IBM Composer, and Azure Durable Functions. Additionally, these orchestrators are qualitatively analyzed using a set of orchestration-related criteria. Spillner [Spi19] analyzes specification, storage, and offering of serverless applications via AWS Serverless Application Repository (SAR). As a part of a more general discussion on serverless computing, Jonas et al. [JSS+19] also provide a high-level review of AWS Lambda.

The aforementioned publications [GSP+18; JSS+19; KS18; LSF18; LRLE17; MPd18; Spi19; WLC+; WT21] as well as the other related qualitative FaaS platform reviews [BO19; GFE+20; KS19; Kum19; PKC19; Raj18; WL21] provide valuable contributions by analyzing different aspects of FaaS platforms. However, these works focus on often decoupled sets of features and classified platforms, which scatters the knowledge across different publications and hinders its reuse. Instead, Contribution 2 presented in this work (i) introduces a substantially larger set of criteria than

previously published works (see [YSB+21a] for more details), (ii) proposes a way to organize criteria (both newly-introduced and already considered in literature) in a comprehensive classification framework, and (iii) introduces selection support concepts that facilitate searching for FaaS platforms that fulfill given requirements. Additionally, the presented metamodel of the classification framework and selection support system are generalized to enable selection support for other component types in FaaS-based applications, e.g., object storage and message queues.

In addition, numerous research publications investigate the topic of benchmarking FaaS platforms. For example, Wang et al. [WLZ+18] benchmark the FaaS platforms from Amazon, Microsoft, and Google by implementing measurement functions that enable discovering hidden architectural details and gathering performance metrics. Contribution 2 presented in this work does not focus on benchmarking and instead focuses on the qualitative analysis of FaaS platforms and concepts for facilitating their selection support. Similar reasoning as above is applicable to other publications focusing on benchmarking [BA18; BG21; FGZ+18; KWB+19].

2.4 Patterns and Pattern-based Design

Proven solutions for recurring domain-specific problems are commonly captured as patterns [AIS77]. Patterns document solutions abstractly to simplify their reuse for multiple problem instances. The documentation format is typically well-structured – many existing guidelines document the best practices for writing patterns [Feh15; WF11]. Pattern languages organize domain-specific patterns as graphs: related patterns are connected using typed links that represent, e.g., refinement or combination of patterns, to enable solving interrelated problems [FBL18].

Numerous pattern languages focus on various architectural aspects and styles, e.g., object-oriented design [Gam+95], enterprise application architectures [Fow02], enterprise integration [HW04a], and microservice

architectures [Ric18]. Cloud computing patterns are covered in many works that focus on cloud-native application design [Dav19; Erl+15; FLR+14; Pah+18]. Taibi et al. [Tai+20] and Zambrano [Zam18] document design patterns for serverless, FaaS-based applications. Hong et al. [Hon+18] present patterns for enhancing the security of cloud applications using serverless computing. Among other microservice patterns, Richardson [Ric18] documents the `SERVERLESS PATTERN` which represents a FaaS-based hosting of microservices. The Component Hosting and Management pattern language presented in this thesis generalizes this notion to other kinds of components and aligns it with respect to explicit management trade-offs alongside other component hosting patterns.

Zimmermann et al. [ZLZ+20] document Interface Responsibility Patterns in which the `COMPUTATION FUNCTION` pattern can be implemented as a FaaS function, hence, enabling using this pattern as a first-class citizen in abstract application models. Fehling et al. [FLR+13] capture patterns for migrating service-based applications to the cloud including such patterns as `FORKLIFT MIGRATION`, `STATELESS COMPONENT SWAPPING`, and `DATABASE SWAPPING`. Jamshidi et al. [Jam+15; Jam+17] document cloud migration patterns and introduce a method for pattern-based migration. Morris [Mor20] presents IaC patterns focusing on many facets of deployment modeling, e.g., stack granularity, build environments, configuration, and testing. Endres et al. [EBF+17] document two approaches for specifying application deployments as patterns, namely declarative and imperative application deployment.

The aforementioned works focus on various aspects of design and deployment for cloud applications, whereas the pattern language presented in Contribution 1 captures different hosting trade-offs. Therefore, the patterns presented in this thesis are complementary to the aforementioned languages and catalogs: application developers can design applications following the design patterns and employ the Component Hosting and Management pattern language to yield the required deployment actions for application components. Further, the patterns by Morris can be employed for versioning and testing of the resulting deployment models.

Patterns also vary in their levels of abstraction [FBB+15], e.g., there exist coarse-grained patterns documenting general solutions and more finer-grained that represent their specific variations. Cloud providers documenting technology-specific patterns [Ama22a] is another example of finer-grained patterns. Hence, it is important to enable navigation through interconnected pattern languages and concrete solutions linked with chosen patterns. Falkenthal et al. [FBB+14b; FBB+15] discuss how pattern languages are linked to different concrete implementations for patterns and how the refinement from patterns to solutions can be achieved, e.g., provider-agnostic cloud computing patterns [FLR+14] can be refined into AWS-specific patterns [Ama22a]. Di Martino et al. [DCE17] present a pattern-based cloud service exploration approach based on semantic web concepts in which abstract patterns can be refined into provider-specific patterns by following semantic annotations such as “equivalent” or “subsumes”. Leymann and Barzen [LB21] introduce an approach and tooling for storing, categorizing, and navigating through interconnected pattern languages and catalogs based on the idea of cartography. Weigold et al. [WBB+20] discuss how patterns from different patterns languages can be linked using the so-called *views* to ease the exploration and discovery. These approaches could facilitate the discovery of patterns introduced in Contribution 1 and their connections to other languages, and also enable exploring concrete solutions linked with them.

Multiple research works employ patterns in the context of architecture and deployment modeling. France et al. [FCSK03] present a pattern-based refactoring approach focusing on object-oriented design patterns [Gam+95] and UML models. Patterns in this approach are associated with (i) a problem UML specification, (ii) a solution UML specification, and (iii) transformation specification that describes how the problem can be transformed into the solution. Consequently, given design models can be refined with chosen (and applicable) design patterns following the aforementioned transformation rules. Arnold et al. [AEK+07] introduce a pattern-based approach and tooling for modeling SOA deployments. In this work, patterns refer to partial deployment models capturing the best deployment prac-

tices that can comprise abstract and concrete components, which can be iteratively refined by users of the underlying platform to obtain deployable models. Fehling et al. [FLR+11; FLRS12] introduce an approach in which (i) patterns are annotated with model fragments representing concrete solutions, e.g., AWS-specific services implementing a `NoSQL DATABASE` pattern and (ii) coarse-grained component-connector models are annotated with patterns to enable manual refinement into technology-specific deployment models adhering to these patterns. Nowak et al. [NBF+12] employs a similar annotation approach for TOSCA models: (i) patterns are annotated by TOSCA Service Templates representing their concrete solutions and (ii) TOSCA Node Templates are annotated by patterns to highlight their usage in implementation of this Node Template. Fahland and Gierds [FG13] introduce an approach for generating formal, Colored Petri Nets-based models from informally-designed integration models [HW04a], which can be used, e.g., for model verification or performance analysis. Ahmad and Babar [AB14] present a pattern-based architectural adaptation method in which patterns capturing the architectural evolution over time (so-called adaptation patterns) are mined from logs and composed into a pattern language, which can then be employed at runtime to enable self-adaptive behavior of an application.

Harzenetter et al. [HBF+18; HBF+20] introduce an approach that uses patterns as first-class citizens in deployment models to represent components and their behavior. These so-called Pattern-Based Deployment Models (PBDMs) can then be refined into their deployable variants specified using the so-called Pattern Refinement Models (PRMs). The automated identification of how PBDMs can be refined happens via so-called detectors that specify which pattern-based sub-graphs can be mapped to which PRMs. Guth and Leymann [GL19] introduce an architecture rewrite approach in which a component-connector model for a designed application can be automatically refined based on a provided graph of interconnected patterns referred to as solution path [FBB+14a]. Park et al. [PKYY20] present a broker-based approach for the selection and integration of cloud services that employs patterns similar to the approach by Arnold et al. [AEK+07]:

patterns refer to partial component-connector models describing specific solutions, e.g., a video processing application. As a basis, users select provider-specific patterns, e.g., from AWS [Ama22a], that are mapped to provider-agnostic representations and can be customized and enriched with QoS requirements to enable selecting suitable cloud services using MCDM algorithms. Bibartiu et al. [BDR21] present a modeling approach that uses patterns for describing applications – it focuses on the modeling of procedures with sequence diagrams employed to specify components interaction and refinement to concrete implementations.

The aforementioned pattern-based design and refinement approaches vary significantly, yet they also share many similarities. For example, all of them use patterns to introduce higher levels of abstraction and enable reasoning about given models (coarser-grained, architectural or finer-grained, deployment-centric) in a technology- and provider-agnostic manner. This comes naturally from the very definition of patterns, which capture abstract solutions. While patterns can be used as explicit modeling constructs or assumed to be present implicitly in the model, most approaches rely on mappings between “some form” of a problem and solution definitions, e.g., both defined as sub-graphs of an application topology graph. Such approaches require a knowledge base containing the pattern-related definitions and transformation rules, which generally makes them tailored only for specific domains, e.g., object-oriented patterns. Further, the type of patterns, e.g., structural or behavioral, influences the way problems and solutions can be specified and how the mappings between them are implemented. This work focuses on different levels of abstraction for designing FaaS-based applications and how to enable transitioning between them: The patterns capturing component hosting and management decisions introduced in (Contribution 1) are used as first-class citizens in application models (Contribution 4) and the approach by Harzenetter et al. [HBF+18; HBF+20] is used as a basis to enable the refinement of coarse-grained component-connector models of FaaS-based applications into concrete, provider-specific deployment models.

2.5 Chapter Summary and Discussion

This chapter presented the necessary fundamentals and provided an analysis and comparison with relevant state-of-art-research. While there exist approaches for modeling FaaS-based applications at different levels of abstraction, i.e., from coarse-grained component-connector models to detailed provider-specific deployment models, often these approaches are tailored for specific use cases such as cost or performance modeling, and cannot be easily combined due to the heterogeneity of the languages and tooling. Further, the existing code abstraction mechanisms and service selection tools are often built as standalone tools not used in combination.

The analysis of modeling approaches highlighted the need to abstractly represent hosting decisions for components in FaaS-based applications. The resulting research effort lead to the pattern language introduced in Contribution 1, which can be used as a standalone language or in pattern-driven modeling approaches to abstractly model desired hosting decisions for FaaS functions and other component kinds in application topologies. Moreover, the analyzed research related to service classification and selection support showed significant differences among cloud service models. The use of general-purpose approaches appears to be insufficient since the technical details of specific offerings such as PaaS or FaaS are missing. Hence, to enable service selection for FaaS-based applications, (i) flexible storage and organization of qualitative and quantitative criteria for different component types and (ii) querying components using different selection algorithms must be supported in a uniform way. Therefore, the two core tasks addressed in Contribution 2 are the *classification of FaaS platforms* and the *support for uniform storage and querying* of different suitable component types.

The investigation of function orchestration modeling approaches and available code abstraction mechanisms showed the lack of abstract orchestration models as well as the need to support modeling artifact abstractions on the level of application models. Therefore, Contribution 3 introduces

(i) concepts for the uniform modeling of function orchestrations using BPMN and their transformation into provider-specific formats and (ii) discusses how to use such artifact abstractions on the level of a CML. To use the introduced concepts together, Contribution 4 presents a pattern-based method that enables expressing decisions on the level of coarse-grained architecture models and refining them into concrete deployment models with target-specific artifacts, inspired by the MOCCA [LFM+11] methodology and the concept of gradient of abstraction [Flo08].

Finally, based on the analysis of existing cloud modeling languages, the TOSCA standard was employed as a baseline for Contribution 5 due to several fitting characteristics. Firstly, it enables representing both coarse-grained and fine-grained application models and transitioning between them using TOSCA ontological typing and inheritance mechanisms. Furthermore, the applicability of TOSCA for FaaS-based applications was shown in the context of the EU Horizon 2020 project RADON: the resulting modeling approach and toolchain introduced by iterative refinements from the consortium and use case partners were successfully validated in three industrial use cases.

COMPONENT HOSTING AND MANAGEMENT PATTERNS

EXISTING cloud service models enable various options for hosting and managing application components: this chapter presents Contribution 1 that captures such trade-offs in a pattern language. The presented Component Hosting and Management pattern language is based on two peer-reviewed publications [YBB+22; YSB+21b] and aims to enable flexibly expressing decisions for hosting components in a provider-agnostic way with respect to two management dimensions, namely (i) *deployment stack* and (ii) *scaling configuration management*. The former dimension describes who (cloud provider or cloud consumer) is more responsible for managing the underlying deployment stack, e.g., deploying a Java function on a VM requires more effort than deploying it as a Java Archive (JAR) on a provider-managed FaaS platform. The latter dimension is, on the other hand, concerned with the responsibilities for managing scaling configuration, i.e., scaling rules and the infrastructure resources needed to host a component.


The main motivation for this pattern language is to document component hosting alternatives not from the perspective of cloud service models such as IaaS or PaaS, but based on explicit preferences regarding who manages deployment stack and scaling configuration – cloud providers or cloud consumers such as application developers or DevOps engineers. While IaaS or PaaS (also documented as eponymous cloud computing patterns [FLR+14]) implicitly express such preferences, specific offerings often support different options, e.g., AWS Lambda enables hosting both container images and language-specific packages such as JARs, which makes the hosting choice less obvious. This contribution, hence addresses the part of Research Question 1 from Section 1.2 that focuses on expressing decisions for component hosting in FaaS-based applications in a provider-agnostic manner. Next sections briefly discuss the patterns authoring process, documentation format, and basic terminology followed by the detailed description of the pattern language and patterns it comprises.

3.1 Authoring Process and Patterns Format

The patterns were captured by gathering and analyzing solutions reoccurring in existing commercial and open source products and technologies (product pages, technical documentation, whitepapers and research papers focusing on component hosting), and following the guidelines for authoring patterns [FEL+12; RBF+16; WF11], which involved (i) establishing the naming conventions and basic terminology, (ii) designing the pattern language and describing the pattern compositions, and (iii) creating detailed pattern documents. In addition, during the pattern language revision phase [FBBL14] the initial references to other patterns were documented, e.g., deployment modeling [EBF+17] and cloud computing [FLR+14] patterns. To keep the format consistent and simplify the data comprehensibility [Feh15; Pet95], the best practices used in research and practice were employed [AIS77; BHS07; Cop96; FLR+14; Gam+95]. Figure 3.1 shows the pattern format, which is discussed next.

- Pattern Name:** a name that uniquely identifies the pattern in the scope of the language. In this language, pattern names are restricted to noun-phrase names [BHS07] to enable using them as nouns.
- Problem:** a summary of the domain- and context-specific problem that a pattern helps solving, formulated in the form of a question.
- Icon:** a graphical element shaped as a rounded rectangle, which visualizes the essence of the pattern to simplify its memorability.
- Context:** the situation in which a pattern is applicable.
- Forces:** the conflicting factors that characterize the problem.
- Solution:** the action or structure intended to solve the context-specific problem and balance the forces. This section also includes a sketch that abstractly visualizes the solution.
- Example:** a concrete example of the pattern implementation accompanied by a sketch that visualizes this particular solution instance.
- Result:** the context after the pattern is applied discussing the advantages and potential new requirements caused by applying the pattern.
- Known Uses:** a description of at least three different real-world occurrences [Cop96] of the pattern implementation.
- Related Patterns:** references to other relevant patterns documented as a part of the same or other pattern language.

Pattern Name

 **Problem :** *A summary of the problem solved with this pattern*

Context : _____

Forces : _____

Solution : _____

Example : _____

Result : _____

Known Uses : _____

Related Patterns : _____


 solution & example sketches

Figure 3.1: The pattern format and graphical layout inspired by cloud computing patterns [Feh15].

3.2 Core Terminology

The patterns are documented using a consistent set of terms specific to the domain of application deployment. Therefore, the core terminology is briefly summarized before the detailed pattern descriptions.

Application: groups software components to provide a certain business functionality. The components run on particular infrastructure, i.e., processing, storage, and network resources [Mes07], and interact with each other and external clients [LW07], e.g., via remote procedure calls or messaging [HW04b].

Software Component: a unit of code that provides needed functionalities via specified interfaces, can be deployed independently and composed by third parties [CH01; LW07; SGM02], e.g., a web server (the former) or Java e-commerce component (the latter).

Hosting Requirements and Capabilities: to be hosted, software components might require certain conditions to be fulfilled, e.g., a compatible Java Runtime Environment (JRE) is required to host a Java 11 function. These requirements can be matched with hosting capabilities of other components to create deployment stacks, e.g., a JRE can be hosted on an operating system but not on a RDBMS.

Deployment Stack: a set of software components and infrastructure resources, e.g., processing, storage, and network resources, that can host software components with matching requirements. Different deployment stacks can host the same software components, e.g., Figure 3.2 shows hosting examples for a Java 11 function using AWS offerings. Since *JRE 8* cannot host a Java 11 function, one stack option is incompatible and cannot be used. The Compatible Stack #1 is based on AWS EC2 (IaaS) – consumers can install the JRE 11 on the chosen virtual machine. Compatible Stack #2 relies on the provider-managed container engine and the JRE can be a part of the container image. Compatible Stack #3 is a predefined stack supporting Java 11 managed by AWS. Deployment stacks can comprise provider- and consumer-managed components.

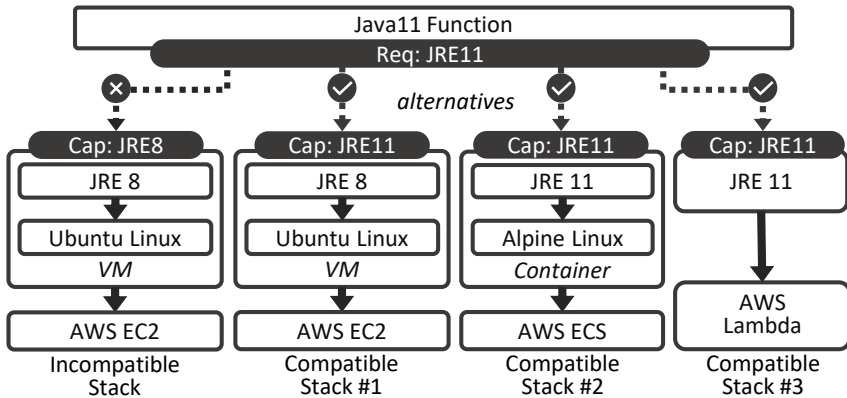


Figure 3.2: Hosting examples for a Java 11 function using AWS.

Consumer-Managed and Provider-Managed Components: consumer-managed components in the stack are managed by cloud consumers, i.e., consumers are responsible for installing, configuring, and managing its dependencies, whereas provider-managed components are mainly managed by respective cloud providers. For instance, the JRE in the Compatible Stack #1 from Figure 3.2 is installed on top of the virtual machine on AWS EC2 – the underlying VM is provider-managed, whereas the cloud consumer is responsible for installing and configuring the JRE on top of it. Conversely, the JREs in PaaS and FaaS are managed by providers since the deployment stacks are provided as predefined and already configured options and can simply be selected from the list of available stacks.

Scaling Configuration: a consumer- or provider-managed combination of horizontal/vertical scaling rules and infrastructure resources required for hosting a software component. For instance, in the Compatible Stack #1 from Figure 3.2, the virtual machine size and scaling rules are defined by cloud consumers, whereas in the Compatible Stack #3 the provider is responsible for allocating resources and scaling the functions.

3.3 Overview of the Pattern Categories

The Component Management and Hosting pattern language captures hosting trade-offs related to two management aspects shown in Figure 3.3 as two axes. Each axis represents a pattern category documenting consumer- and provider-managed solutions for (i) deployment stack and (ii) scaling configuration management aspects. Combinations of these management patterns can be further refined into component hosting patterns that represent composite solutions for both management aspects.

The *Deployment Stack Management* category comprises (1) FIXED DEPLOYMENT STACK and (2) CUSTOMIZABLE DEPLOYMENT STACK patterns, which describe deployment stack alternatives that are either more provider- or more consumer-managed, respectively. The former facilitates hosting components that require no extra modifications on stacks predefined by

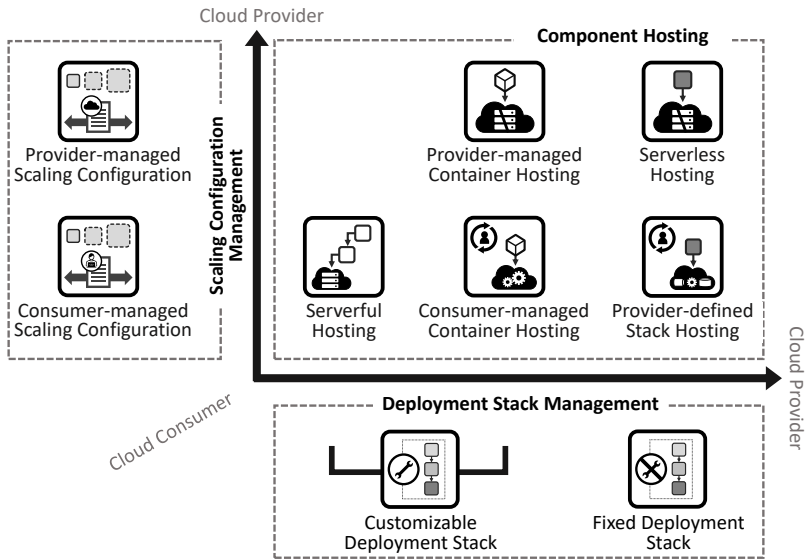


Figure 3.3: Categories of component hosting and management patterns.

providers, e.g., no installation of custom dependencies is needed. The latter represents an opposite variant for cases when components require customization of the underlying environment, e.g., installing custom software components on the VM or as a part of the defined container image.

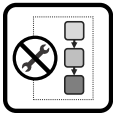
Likewise, the *Scaling Configuration Management* category comprises (3) PROVIDER-MANAGED SCALING CONFIGURATION and (4) CONSUMER-MANAGED SCALING CONFIGURATION patterns that describe two variants of scaling configuration management. When no custom scaling configuration requirements need to be fulfilled to host a component, provider-managed hosting options that implement the former pattern can be chosen, e.g., when no manual specification of a VM cluster or horizontal scaling rules is needed. The latter variant represents an alternative for hosting components with a customized scaling configuration, e.g., using IaaS offerings.

The *Component Hosting* category comprises five patterns that capture specific combinations of patterns from the previous two categories, namely (5) SERVERFUL HOSTING, (6) CONSUMER-MANAGED CONTAINER HOSTING, (7) PROVIDER-DEFINED STACK HOSTING, (8) PROVIDER-MANAGED CONTAINER HOSTING, and (9) SERVERLESS HOSTING patterns. The terms “serverful” and “serverless” in pattern names were employed to mark two extremes that correspond to stronger or weaker management responsibilities for cloud consumers, i.e., the SERVERFUL HOSTING pattern is managed substantially by consumers since both the deployment stack and scaling configuration are easier to customize. Conversely, the SERVERLESS HOSTING pattern enables mainly provider-managed deployment stacks and scaling configuration, e.g., function code deployed as-is to FaaS platforms and scaled out and in automatically by cloud providers. The hosting patterns located in-between these extremes, thus capture different combinations of the aforementioned deployment stack and scaling configuration management variants. The following subsections present each category and respective patterns in detail using the format and layout discussed in Section 3.1. Afterwards, the semantic links among the patterns are discussed and formalized to highlight how link traversal and pattern combination enable iterative exploration of component hosting variants.

3.4 Deployment Stack Management Patterns

This section introduces the `FIXED DEPLOYMENT STACK` and `CUSTOMIZABLE DEPLOYMENT STACK` patterns focusing on deployment stack customization trade-offs and is based on the previously published work [YBB+22].

3.4.1 Fixed Deployment Stack



Problem: *“How to host a software component when it requires no special underlying infrastructure or customization of the host environment it is running on?”*

Context: A software component needs to be hosted without any special customization requirements, e.g., adding, removing, or changing components in the deployment stack or configuring it in a certain way.

Forces: Cloud service models incur varying management efforts for deployment stacks, e.g., FaaS functions hosted on a provider-managed platform require less management effort than a virtual machine hosted on IaaS where consumers are responsible for installing patches. Software components often require only common dependencies, e.g., a Java function that has no customization requirements, or a standard relational database.

Solution: Host the software component on a `FIXED DEPLOYMENT STACK` for which all infrastructure, execution environment, and middleware components needed to host and execute the given component are set up, configured, and maintained by the cloud provider. For example, PaaS offerings provide predefined deployment stacks for different Java runtime versions and Database-as-a-Service (DBaaS) offerings may provide different database management system versions without requiring to manage the hosting components and dependencies. As a result, software components are hosted as-is on provider-managed offerings without extra stack configuration or customization efforts as shown in Figure 3.4a.

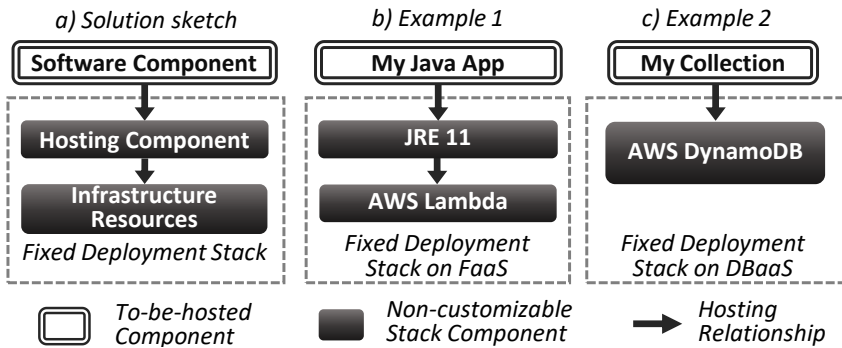


Figure 3.4: FIXED DEPLOYMENT STACK: solution sketch and examples.

Examples: Examples in Figure 3.4 show the FIXED DEPLOYMENT STACK pattern for different component types: The “My Java App” component in Figure 3.4b is hosted using a provider-defined stack on AWS Lambda that enables choosing a specific Java runtime version for a given component. Another example is shown in Figure 3.4c – a NoSQL collection is created on AWS DynamoDB, a DBaaS offering. In both cases, cloud consumers directly choose the desired stack from a list of provider-defined options instead of setting up all required hosting components and dependencies.

Result: When applied, this pattern reduces configuration overhead since the deployment stack is selected from a list of available options. However, this also results in a stronger dependence on the provider-specific service, e.g., the implementation and configuration of software components depend more on provider requirements such as service-specific libraries, formats, packaging, and configuration requirements.

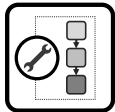
Known Uses: The FIXED DEPLOYMENT STACK pattern can be implemented using different provider-managed services. For instance, PaaS offerings, such as AWS Beanstalk [Ama22b] or Azure App Service [Mic22], or FaaS offerings, such as AWS Lambda [Ama22b] and Azure Functions [Mic22], enable hosting business logic components on provider-defined stacks. An-

other example is DBaaS offerings, such as AWS S3 [Ama22b] and IBM Cloud Databases for Redis [IBM21], that enable hosting other component types on provider-defined stacks.

Related Patterns:

- **PROVIDER-DEFINED STACK HOSTING** and **SERVERLESS HOSTING**: represent different refinements of **FIXED DEPLOYMENT STACK** pattern.
- **CUSTOMIZABLE DEPLOYMENT STACK**: this pattern can be used as an alternative for cases when stack customizations are needed.
- Patterns such as **PLATFORM-AS-A-SERVICE** [FLR+14] and **SOFTWARE-AS-A-SERVICE** [FLR+14] enable using fixed deployment stacks.

3.4.2 Customizable Deployment Stack



Problem: “*How to host a software component when it requires customization of the underlying infrastructure or the host environment it is running on?*”

Context: A software component needs to be hosted with special customization requirements, i.e., cloud consumers can customize its stack by adding, removing, or changing components or configuring it in a certain way.

Forces: Cloud service models differ with respect to how customizable the underlying deployment stacks are, e.g., additional software can be installed on a VM on IaaS offerings, whereas FaaS offerings enable hosting functions without the need to manage the underlying stacks by selecting a deployment stack from the list of available options. On the other hand, customization requirements for component hosting can emerge due to multiple reasons, e.g., legacy applications might require special dependencies such as other software components that need to run in the same operating system. The **SIDE CAR** pattern [BO16] is another customization example: an implementation of this pattern can add custom features to the component and is required to run alongside the component.

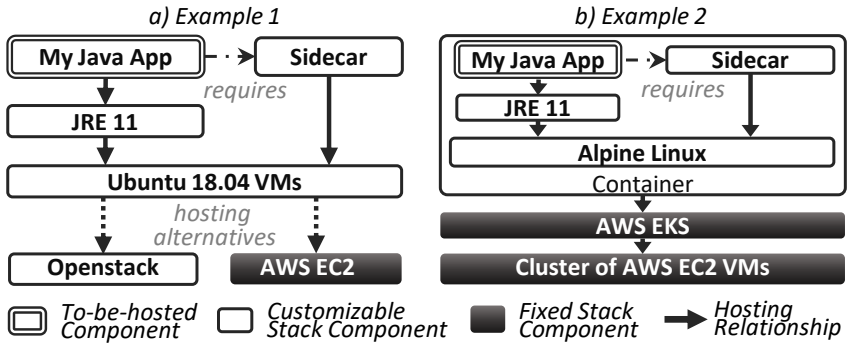


Figure 3.6: CUSTOMIZABLE DEPLOYMENT STACK: examples.

Result: When applied, this pattern enables more customization options – for the first variant also the physical infrastructure can be customized if desired. However, in the cloud context, typically virtual machines are used meaning that VM properties and software components that run on the operating system are customizable. In case when the second variant is employed, the customization is possible within the container boundaries.

Known Uses: The CUSTOMIZABLE DEPLOYMENT STACK pattern can be implemented using different services, e.g., bare metal offerings like IBM Cloud Bare Metal Servers [IBM21] and IaaS offerings such as AWS EC2 [Ama22b] or Azure IaaS [Mic22]. Another option is to employ container orchestration services such as Azure Kubernetes Service [Mic22] and AWS Fargate [Ama22b].

Related Patterns:

- SERVERFUL HOSTING, CONSUMER-MANAGED CONTAINER HOSTING, and PROVIDER-MANAGED CONTAINER HOSTING: represent different refinements of the CUSTOMIZABLE DEPLOYMENT STACK pattern.
- FIXED DEPLOYMENT STACK: when no stack customizations are needed, this pattern can be used instead.
- Patterns from other languages such as INFRASTRUCTURE-AS-A-SERVICE [FLR+14] enable customizing deployment stacks.

3.5 Scaling Configuration Management Patterns

This section introduces the PROVIDER-MANAGED SCALING CONFIGURATION and CONSUMER-MANAGED SCALING CONFIGURATION patterns focusing on scaling configuration management trade-offs and is based on the previously published work [YBB+22].

3.5.1 Provider-Managed Scaling Configuration



Problem: *“How to host a software component when it needs to be scaled horizontally but requires no special scaling configuration?”*

Context: A software component needs to be hosted and scaled without any special requirements for infrastructure resources or horizontal scaling behavior, e.g., explicit configuration of a VMs cluster or scaling rules.

Forces: Cloud offerings differ in degree of control over scaling configuration. For instance, multiple offerings require cloud consumers explicitly managing the scaling configuration by defining the size/amount of virtual machines and scaling rules, whereas other alternatives abstract away the required virtual machines and enable specifying infrastructure resources in virtual memory and CPU units. Further, multiple provider-managed services provide default autoscaling mechanisms, e.g., AWS Lambda or AWS S3 are auto-scaled by default with requiring configuring scaling rules. Customization of scaling configuration for components, on the other hand, is not always required, e.g., for hosting time-triggered functions. In addition, provider-managed offerings often do not incur extra licensing costs while providing reduced scaling configuration management, e.g., purchasing a number of licenses for a messaging middleware is not needed.

Solution: Host a software component on a deployment stack with PROVIDER-MANAGED SCALING CONFIGURATION, hence shifting the responsibility for specification of the underlying infrastructure resources and

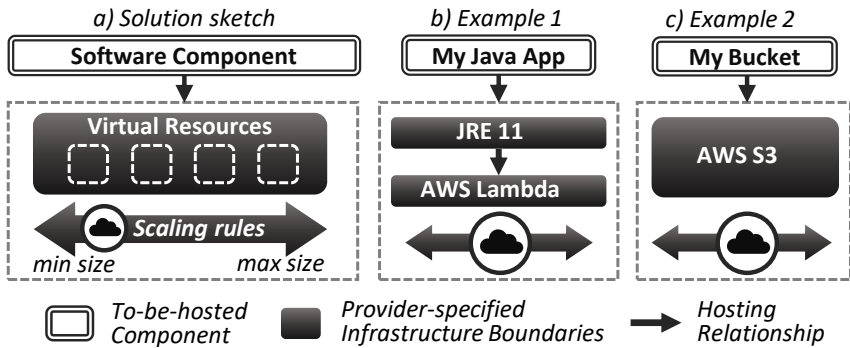


Figure 3.7: PROVIDER-MANAGED SCALING CONFIGURATION: solution sketch and examples.

scaling rules to cloud providers. Figure 3.7a depicts the solution sketch: a component is hosted on a fixed deployment stack with abstractly specified infrastructure resources and horizontal scaling rules mainly managed by providers, thus reducing management effort for cloud consumers.

Examples: Figure 3.7 depicts two example AWS-based deployment stacks enabling this pattern. For both the Java application (Figure 3.7b) and the object storage bucket (Figure 3.7c) the required infrastructure is abstractly specified in virtual resource units, whereas the horizontal scaling rules are completely provider-managed – AWS Lambda functions are automatically scaled (also to zero instances when execution completes), and AWS S3 buckets also rely on default scaling mechanisms.

Result: When applied, this pattern results in less effort for managing infrastructure resources and scaling rules, which can be preferable when no custom scaling behavior is required. Further combination with one of the deployment stack management patterns from Section 3.4 can help to find a more compatible hosting variant (see Section 3.7 for more details).

Known Uses: Various offerings support PROVIDER-MANAGED SCALING CONFIGURATION, e.g., it is supported by various FaaS offerings such as AWS Lambda [Ama22b] or Azure Functions [Mic22]. Provider-managed

container services such as Google CloudRun [Goo22] also abstract away infrastructure resources and autoscale containers based on the rate of incoming of requests. Storage services such as AWS DynamoDB [Ama22b] and Azure Blob Storage [Mic22] support this pattern since no explicit specification of the infrastructure resources and scaling rules is needed.

Related Patterns:

- **SERVERLESS HOSTING** and **PROVIDER-MANAGED CONTAINER HOSTING**: represent different refinements of the **PROVIDER-MANAGED SCALING CONFIGURATION** pattern.
- **CONSUMER-MANAGED SCALING CONFIGURATION**: can be used instead if scaling configuration needs to be more customizable.
- **FIXED DEPLOYMENT STACK**, **CUSTOMIZABLE DEPLOYMENT STACK**: can be used in combination with the **PROVIDER-MANAGED SCALING CONFIGURATION** pattern.
- Patterns such as **SOFTWARE-AS-A-SERVICE** [FLR+14] support this the **PROVIDER-MANAGED SCALING CONFIGURATION** pattern.

3.5.2 Consumer-managed Scaling Configuration



Problem: “How to host a software component when it needs to be scaled horizontally but requires a tailored scaling configuration?”

Context: A software component needs to be hosted with custom requirements for the underlying infrastructure resources and horizontal scaling behavior, i.e., explicit configuration of VMs clusters and scaling rules.

Forces: Cloud service offerings differ in how scaling configuration is managed, e.g., “serverless-style” offerings aim to abstract away the infrastructure by specifying underlying resources in virtual memory and CPU units and often require no specification of scaling rules. In contrast, many services, such as AWS EC2 or Azure EKS, are more customizable as they

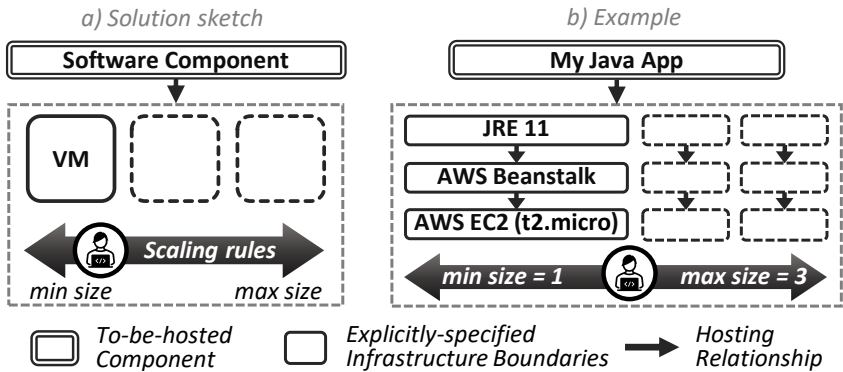


Figure 3.8: CONSUMER-MANAGED SCALING CONFIGURATION: solution sketch and example.

require cloud consumers to explicitly manage the scaling configuration by defining the size/amount of virtual machines and scaling rules. Customizable scaling configuration for components might be needed in certain cases, e.g., if licenses are available for an exact number of instances, or to enable custom dependencies in the deployment stack.

Solution: Host a software component on a deployment stack with CONSUMER-MANAGED SCALING CONFIGURATION, hence enabling cloud consumers to retain more control over the specification of the infrastructure resources and horizontal scaling rules. Figure 3.8a shows the solution sketch: a component is hosted on a deployment stack with explicitly defined infrastructure boundaries and consumer-managed horizontal scaling rules, thus enabling fulfilling customization requirements.

Examples: Figure 3.8b depicts an AWS Beanstalk [Ama22b] stack supporting the CONSUMER-MANAGED SCALING CONFIGURATION pattern. Cloud consumers choose virtual machine images for the Java application and specify the horizontal scaling rules. Consumer-managed offerings, such as bare-metal, IaaS, or container orchestration services, typically provide even more control over the scaling configuration.

Result: When applied, this pattern results in a consumer-controlled specification of infrastructure resources and horizontal scaling rules, which can be preferable when such customizations are required. Scaling configuration is performed according to the underlying service requirements, e.g., autoscaling rules for container orchestration services might differ from rules for PaaS offerings. This pattern can be combined with either the `FIXED` or the `CUSTOMIZABLE DEPLOYMENT STACK` patterns to enable flexibly refining these management decisions into suitable hosting options (see Section 3.7 for more details).

Known Uses: Various offerings support `CONSUMER-MANAGED SCALING CONFIGURATION`, e.g., bare-metal and IaaS offerings such as IBM Cloud Bare Metal Servers [IBM21], AWS EC2 [Ama22b], or Azure IaaS [Mic22]. Container orchestration offerings, such as Azure Kubernetes Service [Mic22] or AWS Elastic Kubernetes Service [Ama22b], also enable configuring the cluster size and the horizontal scaling rules for containers. In addition, certain PaaS offerings such as AWS Beanstalk support the `CONSUMER-MANAGED SCALING CONFIGURATION` pattern, e.g., by allowing cloud consumers to configure specific numbers of instances and enabling finer-grained configuration of horizontal scaling rules.

Related Patterns:

- `SERVERFUL HOSTING`, `CONSUMER-MANAGED CONTAINER HOSTING`, `PROVIDER-DEFINED STACK HOSTING`: represent different refinements of the `CONSUMER-MANAGED SCALING CONFIGURATION` pattern.
- `PROVIDER-MANAGED SCALING CONFIGURATION`: can be used instead when no custom scaling configuration is needed.
- `FIXED DEPLOYMENT STACK`, `CUSTOMIZABLE DEPLOYMENT STACK`: can be used in combination with the `CONSUMER-MANAGED SCALING CONFIGURATION` pattern.
- Many patterns from other languages such as `INFRASTRUCTURE-AS-A-SERVICE` [FLR+14] support the `CONSUMER-MANAGED SCALING CONFIGURATION` pattern.

3.6 Component Hosting Patterns

This section introduces five patterns representing different component hosting trade-offs and is based on the previously published work [YSB+21b].

3.6.1 Serverful Hosting



Problem: *How to host a software component when it requires customization of the underlying deployment stack and scaling configuration?*

Context: A software component needs to be hosted on a custom deployment stack with multiple required software dependencies such that the cloud consumer is able to customize the required infrastructure resources and scaling rules.

Forces: Cloud service models differ in the degree of control cloud consumers have over the underlying infrastructure resources and scaling rules, which also influences the customizability of the underlying deployment stack. For instance, multiple services abstract away the infrastructure as virtual memory and CPU units, and enable selecting predefined stacks that cannot be customized and have default scaling rules. However, cloud consumers may require the customization of hosting components, e.g., to enable specific dependencies in the hosting environment or have custom scaling configuration. The chosen architectural style could also influence this decision, e.g., hosting a monolith with various custom requirements vs. hosting fine-grained functions with no external dependencies.

Solution: Host a software component on a deployment stack that is primarily consumer-managed and, hence enables manually specifying the underlying infrastructure resources and scaling rules, as well as setting up, configuring, and maintaining all the required hosting components, e.g., using on-premises servers, bare metal or IaaS offerings. Figure 3.9a shows

Known Uses: Various offerings enable this pattern, e.g., IaaS offerings such as AWS EC2 [Ama22b], Azure IaaS [Mic22], or Google Compute Engine [Goo22] enable provisioning VMs to host components and dependencies, and supporting consumer-managed scaling configuration. Furthermore, bare metal offerings such as IBM Cloud Bare Metal Servers [IBM21] can be used to provision dedicated physical servers, hence enabling cloud consumers customizing deployment stack components and scaling rules for to-be-hosted software components.

Related Patterns:

- **CUSTOMIZABLE DEPLOYMENT STACK, CONSUMER-MANAGED SCALING CONFIGURATION:** can be refined into this pattern.
- Patterns such as **INFRASTRUCTURE-AS-A-SERVICE [FLR+14]** or **PRIVATE CLOUD [FLR+14]** support this pattern.
- **DECLARATIVE DEPLOYMENT MODEL, IMPERATIVE DEPLOYMENT MODEL [EBF+17]** can specify stacks implementing this pattern.

3.6.2 Consumer-managed Container Hosting



Problem: *How to host a software component when it requires customization of the hosting environment it runs on and a tailored scaling configuration?*

Context: A software component needs to be hosted on a deployment stack with customizable hosting environment such that the cloud consumer is able to define the infrastructure resources and scaling rules.

Forces: Cloud service models differ in the degree of control cloud consumers have over the underlying deployment stack and scaling configuration. While multiple services enable cloud consumers to manually define VM clusters and scaling rules, this requires technical expertise, e.g., network configuration. Provider-managed services simplify such configuration tasks, but often reduce the customizability of the deployment stack

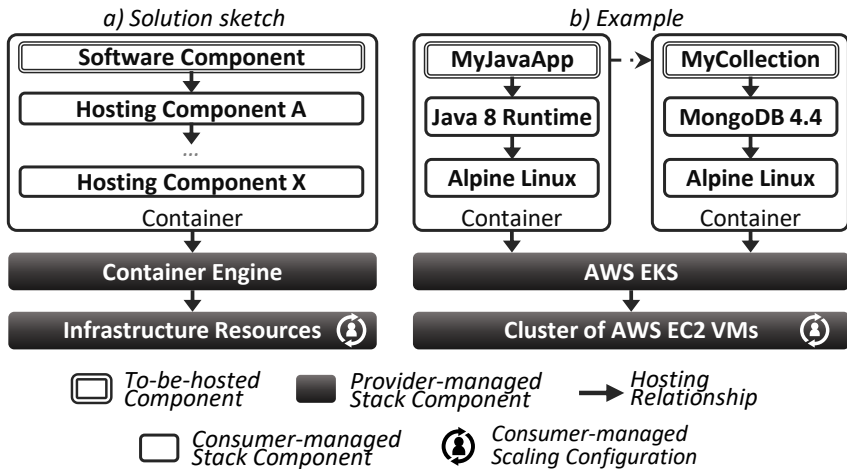


Figure 3.10: CONSUMER-MANAGED CONTAINER HOSTING: solution sketch and example.

and scaling configuration. However, managing only the hosting environment software components run on and their scaling configuration without focusing on the infrastructure resources may be needed, e.g., add custom software libraries and defining scaling rules.

Solution: Host a software component on a deployment stack implemented using container orchestration services that enable customizing the scaling configuration. Figure 3.10a shows the solution sketch: a containerized component is hosted on a provider-managed container engine that supports consumer-managed scaling configuration. The infrastructure for running containers is provisioned and managed by cloud providers, whereas cloud consumers can still customize container images that encompass components and specify the desired scaling configuration including the infrastructure resources for container engine and scaling rules.

Example: Figure 3.10b shows CONSUMER-MANAGED CONTAINER HOSTING for two components using the managed Kubernetes [The22c] service from AWS: a Java application and a NoSQL database. The hosting environment

is customizable by cloud consumers as a part of the provided container image. The scaling configuration including infrastructure resources and scaling rules is also managed by consumers.

Result: When applied, this hosting pattern results in reduced management efforts for lower infrastructure layers in the deployment stack that are provider-managed, whereas cloud consumers can still manage the scaling configuration and customize the hosting environment via container images. While the degree of control is reduced, cloud consumers still can introduce various modifications making this pattern a less demanding variant for hosting components that require customizations for the hosting environment and scaling rules. Provider-managed container orchestration offerings may vary in built-in features and integrations with other provider services. However, since deployment stacks are defined as container images, such applications are easier to reuse and port to similar environments from other cloud providers.

Known Uses: Various container orchestration services enable this pattern including IBM Cloud Kubernetes Service [IBM21], Azure Kubernetes Service [Mic22], and AWS Elastic Kubernetes Service [Ama22b]. Furthermore, some Container-as-a-Service (CaaS) offerings such as AWS Elastic Container Service [Ama22b] with the EC2-based pricing mode enable hosting containers such that consumers can configure the infrastructure resources for planned containerized components and define the scaling rules for running container instances.

Related Patterns:

- CUSTOMIZABLE DEPLOYMENT STACK, CONSUMER-MANAGED SCALING CONFIGURATION: can be refined into this pattern.
- Multiple patterns, e.g., INFRASTRUCTURE-AS-A-SERVICE, PRIVATE CLOUD [FLR+14], SOFTWARE CONTAINER [SF15], or CONTAINER MANAGER [SF17] support this pattern.
- DECLARATIVE DEPLOYMENT MODEL, IMPERATIVE DEPLOYMENT MODEL [EBF+17] can specify stacks implementing this pattern.

3.6.3 Provider-defined Stack Hosting



Problem: *How to host a software component when it requires a tailored scaling configuration but no deployment stack customization is necessary?*

Context: A software component needs to be hosted on a deployment stack without any customization requirements such that the cloud consumer is able to define the infrastructure resources and scaling rules.

Forces: Cloud service models differ in how customizable the underlying deployment stack and scaling configuration are. Many services offer more control as cloud consumers to manually define VM clusters and scaling rules, which, however, adds more management overhead and requires additional technical expertise. On the other hand, software components can often be hosted on standard platforms without extra customizations, e.g., a Java Web Application Archive (WAR) or a relational database schema. Provisioning VMs and installing dependencies is an unnecessary overhead for such cases. However, managing scaling configuration may still be needed for tailored availability requirements, e.g., to manually fine-tune scaling rules. This is also relevant for products like databases or message queues for hosting database schemas and topics with consumer-managed infrastructure resources and scaling rules.

Solution: Host a software component on a deployment stack capable to run this component without additional customizations, but also enables cloud consumers to manage the scaling configuration. Figure 3.11a shows the solution sketch: a component is hosted as-is on a suitable provider-defined stack that supports consumer-managed scaling configuration. For example, certain PaaS and DBaaS offerings enable hosting components by selecting a compatible predefined stack variant, e.g., a Java 8 application running on a stack with JRE 8 or a RDBMS of specific version, whereas the scaling configuration can still be managed by cloud consumers, e.g., by specifying the number of VMs and the scaling rules.

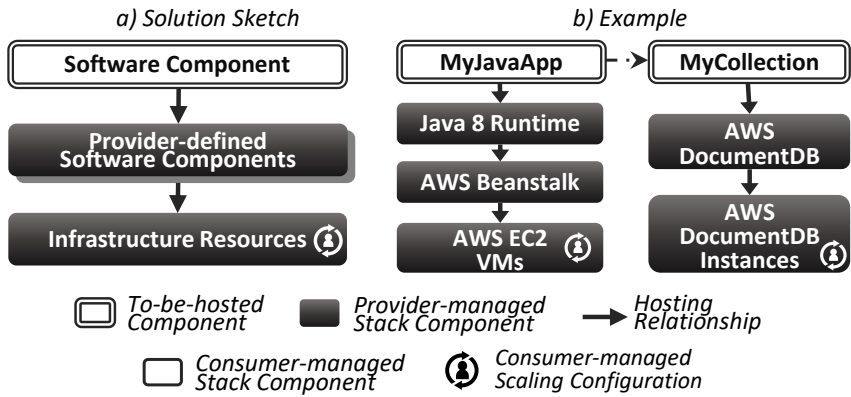


Figure 3.11: PROVIDER-DEFINED STACK HOSTING: solution sketch and example.

Example: Figure 3.11b shows PROVIDER-DEFINED STACK HOSTING for two components using AWS Beanstalk and AWS DocumentDB [Ama22b]: a Java application and a NoSQL database. The compatible deployment stack is simply selected from a list of options, i.e., the compatible JRE and a NoSQL document store. However, cloud consumers can still manage the scaling configuration by defining the desired number and flavor of VMs and specifying scaling rules.

Result: When applied, this hosting pattern results in reduced efforts for managing the deployment stack as consumers only select a predefined, compatible option for the given packaged component. However, since cloud consumers retain control over scaling configuration, the infrastructure resources (number and flavor of underlying VMs) and scaling rules can be customized. Components hosted using this pattern can often benefit from the built-in integration mechanisms targeting other provider services, but the degree of lock-in is also increased.

Known Uses: Various PaaS offerings enable this pattern, e.g., AWS Elastic Beanstalk Service [Ama22b] and Azure App Service [Mic22]. The former enables hosting components packaged for various platforms using

available and predefined deployment stacks. Infrastructure resources required to run the components must be specified by developers as AWS EC2 VMs that will host the predefined deployment stacks. Likewise, Azure App Service enables hosting software components implemented for various platforms and control the underlying infrastructure resources. Cloud service models such as DBaaS, e.g., Amazon DocumentDB [Ama22b] or Oracle Database Classic Cloud Service [Ora22], enable managing the infrastructure resources for using predefined deployment stacks providing different versions of the respective database management systems.

Related Patterns:

- **FIXED DEPLOYMENT STACK, CONSUMER-MANAGED SCALING CONFIGURATION:** can be refined into this pattern.
- Multiple patterns, e.g., **PLATFORM-AS-A-SERVICE, EXECUTION ENVIRONMENT** [FLR+14] support this pattern.
- **DECLARATIVE DEPLOYMENT MODEL, IMPERATIVE DEPLOYMENT MODEL** [EBF+17] can specify stacks implementing this pattern.

3.6.4 Provider-managed Container Hosting



Problem: *How to host a software component when it only requires customization of the hosting environment it is running on?*

Context: A software component needs to be hosted on a deployment stack with a customizable hosting environment such that cloud consumers are not required to manage the infrastructure resources and scaling rules.

Forces: Cloud service models vary in how customizable the deployment stack and scaling configuration are: while many services enable manual specification of VM clusters and scaling rules, others simplify stack management and provide default autoscaling mechanisms. Managing the

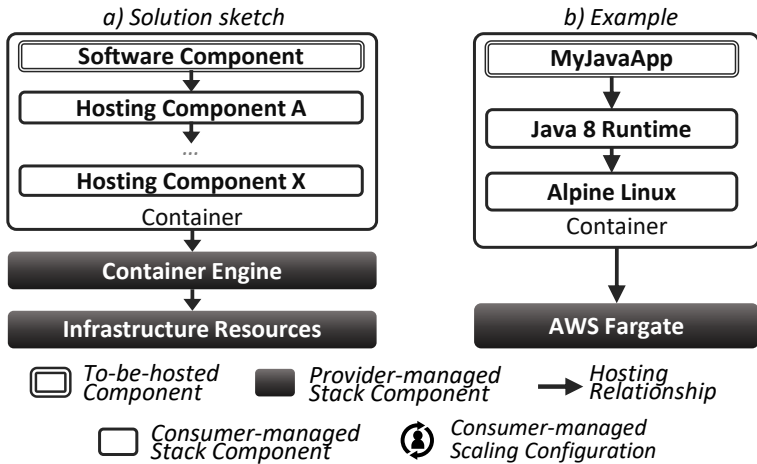


Figure 3.12: PROVIDER-MANAGED CONTAINER HOSTING: solution sketch and example.

infrastructure resources and scaling rules may be an overhead for components that need no tailored scaling rules and only require modified hosting environment, e.g., to install custom libraries.

Solution: Host a software component on a deployment stack implemented using container orchestration services that do not require managing scaling configuration by providing default autoscaling mechanisms. Figure 3.12a shows the solution sketch: a containerized component is hosted on a provider-managed container engine. Customization of the hosting environment is possible via container images; however, the scaling configuration is not required to be managed by cloud consumers. With provider-managed scaling configuration, infrastructure resources are abstracted away and providers are responsible for scaling containers in and out based on their operation mode, e.g., long-running tasks vs. short-lived tasks.

Example: Figure 3.12b shows PROVIDER-MANAGED CONTAINER HOSTING for a Java application using AWS Fargate, a serverless container-centric offering from Amazon. The hosting environment is specified via container

images, i.e., the compatible Java runtime, whereas the scaling configuration is provider-managed. The usage of serverless-style container services that scale components to zero instances for standard products such as databases may incur unnecessary overhead due to the need to re-instantiate such components for new requests. Therefore, to fully benefit from reduced management requirements, the `SERVERLESS HOSTING` can be implemented for such component types instead (see Section 3.6.5).

Result: When applied, this hosting pattern results in reduced management efforts for lower infrastructure layers in the deployment stack, and in scaling configuration that are provider-managed, whereas cloud consumers can still customize the hosting environment using container images. While provider is responsible for the majority of tasks, cloud consumers can still customize the deployment stack, which makes this pattern a customizable, serverless-style hosting option. Provider-managed container services may differ in features and available integrations with other provider services; however, deployment stacks specified as container images can often be reused for similar environments from other cloud providers.

Known Uses: Various CaaS offerings enable this pattern, e.g., AWS Fargate [Ama22b], Azure Container Instances [Mic22], or Google CloudRun [Goo22] enable hosting container images with provider-managed scaling configuration that often includes scaling containers to zero instances. Another example is Iron Worker [Iro22] that enables executing containerized background tasks in a serverless manner. Certain FaaS offerings, e.g., AWS Lambda [Ama22b], that support container images as deployment artifacts also enable this pattern.

Related Patterns:

- `CUSTOMIZABLE DEPLOYMENT STACK, PROVIDER-MANAGED SCALING CONFIGURATION`: can be refined into this pattern.
- Multiple patterns, e.g., `SOFTWARE CONTAINER` [SF15] or `CONTAINER MANAGER` [SF17] support this pattern.
- `DECLARATIVE DEPLOYMENT MODEL, IMPERATIVE DEPLOYMENT MODEL` [EBF+17] can specify stacks implementing this pattern.

3.6.5 Serverless Hosting



Problem: *How to host a software component when it requires neither customization of the hosting environment it runs on nor tailored scaling configuration?*

Context: A software component needs to be hosted on a deployment stack without any customizations of its hosting environment and no custom requirements related to infrastructure resources and scaling rules.

Forces: Cloud service models vary in the degree of control over the deployment stack and scaling configuration, e.g., many services enable manual specification of VM clusters and scaling rules, while others simplify deployment stack and scaling configuration management for cloud consumers. To host components that require no tailored scaling rules or custom deployment stack, it may be preferable to rely on predefined stacks and leverage provider-managed scaling configuration, e.g., hosting small code snippets with no custom dependencies. Tighter coupling with the underlying provider offering can help leveraging built-in integration mechanisms with other provider-specific services.

Solution: Host a software component on a deployment stack capable to run it without additional customizations and without manual scaling configuration, e.g., by leveraging default autoscaling mechanisms. Figure 3.13a shows the solution sketch: a software component is hosted on a provider-defined stack which does not require cloud consumers to manage scaling configuration. For instance, public FaaS offerings enable hosting event-driven code snippets that are autoscaled based on incoming requests. Certain database and message queue offerings also do not require managing underlying infrastructure resources and scaling rules and are provided as predefined stacks, e.g., specific database versions with all the required dependencies. Some SaaS offerings also support this pattern by enabling using components with minor configuration efforts and abstracted away deployment stacks, e.g., static web pages can be hosted using GitHub Pages if the repository is configured in a specific way.

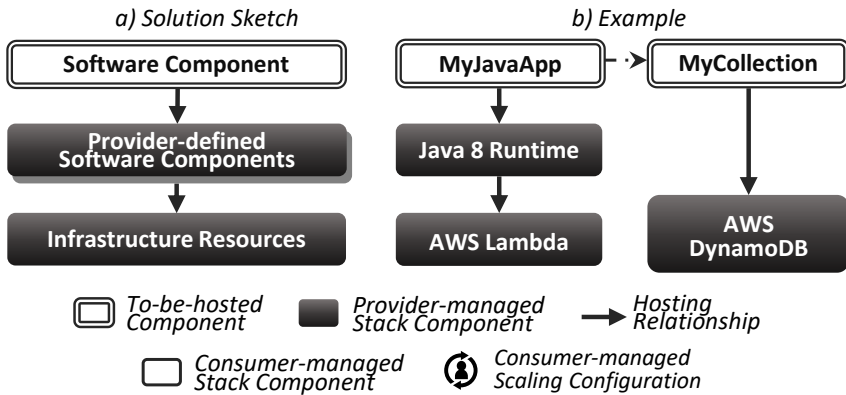


Figure 3.13: SERVERLESS HOSTING: solution sketch and example.

Example: Figure 3.13b shows SERVERLESS HOSTING for two components (a Java application and a NoSQL database) using AWS Lambda and AWS DynamoDB, serverless FaaS and DBaaS offerings from Amazon. The compatible deployment stack is chosen from the list of available options, i.e., the compatible JRE and a NoSQL document database. Moreover, cloud consumers do not need to manage the scaling configuration as both services provider default autoscaling mechanisms.

Result: When applied, this hosting pattern results in significantly reduced management efforts required from cloud consumers since cloud providers are responsible for setting up, configuring, and scaling the underlying deployment stacks. Typically, only a packaged component or even product-specific configuration (in case of certain SaaS offerings) are to be provided to enable using the component, with the underlying platform also providing default autoscaling mechanisms. This pattern is the least consumer-managed option, which is suitable for components with no custom management requirements and enables consumers to focus more on business logic and integration tasks instead. Additionally, the integration with other provider-specific services is often simpler due to built-in integration mechanisms, e.g., built-in event triggers for FaaS functions.

Known Uses: Various FaaS offerings such as AWS Lambda [Ama22b], Azure Functions [Mic22], or Google Cloud Functions [Goo22] enable this pattern. The hosted code snippets are automatically scaled based on incoming requests and can be integrated with events from other services due to built-in integration mechanisms. Other examples are object storage services such as AWS S3 [Ama22b] or IBM Object Storage [IBM21], or database offerings, e.g., Amazon Aurora Serverless [Ama22b] does not require scaling configuration management – providers use pricing models in which consumers are charged based on how computational power, storage, data transfer, etc. are utilized.

Related Patterns:

- **FIXED DEPLOYMENT STACK, PROVIDER-MANAGED SCALING CONFIGURATION:** can be refined into this pattern.
- **SERVERLESS DEPLOYMENT PATTERN [Ric18]:** is a more specialized variant of this pattern for hosting microservices on FaaS platforms.
- **DECLARATIVE DEPLOYMENT MODEL, IMPERATIVE DEPLOYMENT MODEL [EBF+17]:** can specify stacks implementing this pattern.

3.7 Pattern Relations and Their Semantics

The introduced patterns capture higher- and lower-level solutions and are interrelated with each other via typed relationships, thus forming a pattern language as shown in Figure 3.14. As pattern relationships are often described in free text [FBL18] and can be ambiguously interpreted [CFH+02], this section discusses the semantics of documented relationships among patterns both informally and formally to clearly explain the patterns interplay used for the provider-agnostic modeling of FaaS-based applications presented in Chapter 6. Since multiple pattern formalisms exist [BZ08; CFH+02; FBL18; RFL19], the semantics of transitions between patterns and their combinations using typed relationships is described by building on top of existing formal approaches.

The patterns from the *Deployment Stack Management* and *Scaling Configuration Management* categories discussed in Sections 3.4 and 3.5 capture higher-level solutions related to particular management problems, i.e., customizability of deployment stacks and manageability of scaling configuration by cloud consumers. In the presented pattern language, each of these categories captures mutually-exclusive solutions for hosting software components with respect to specific management aspect, e.g., the `FIXED DEPLOYMENT STACK` pattern can be used when no customizations are needed for the deployment stack, whereas the `CUSTOMIZABLE DEPLOYMENT STACK` is applicable in the opposite case. Therefore, to represent patterns that cannot be combined together, the relationship of type *Conflict* [Nob98] is used in the pattern language as shown in Figure 3.14.

Further, the *Deployment Stack Management* and *Scaling Configuration Management* patterns can be combined to address specific *combinations* of these management aspects forming so-called *compound patterns* [BHS07]. To represent such compound solutions, the relationships of type *Combina-*

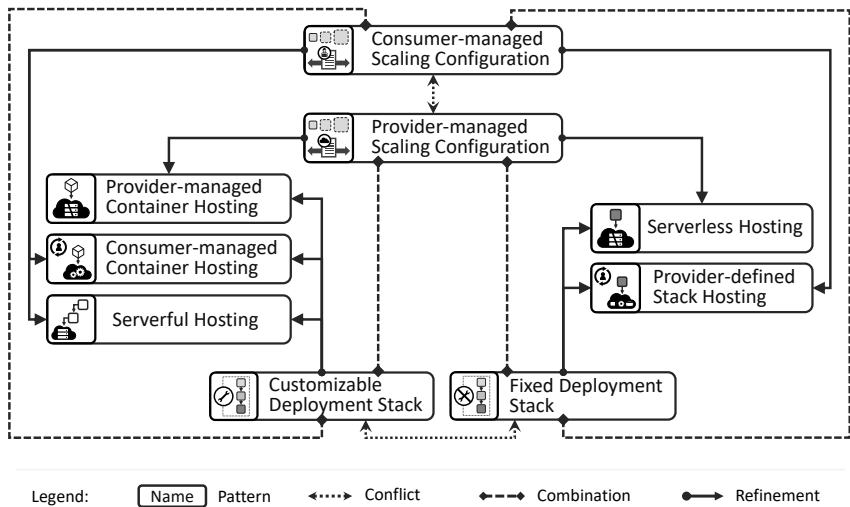


Figure 3.14: The Component Hosting and Management pattern language.

tion [Nob98] is used in the pattern language as shown in Figure 3.14. Moreover, single as well as compound patterns *can be refined* into more concrete patterns using the relationships of type *Refinement* [KP10; Nob98]: as seen in Figure 3.14 abstract patterns related to a specific management dimension can typically be refined into one of the suitable and more concrete hosting options. Finally, *Generalization* is the inverse of the *Refinement* relationship (not shown explicitly in Figure 3.14) and describes the transitioning from a more concrete to more abstract pattern [Nob98].

Following the existing definition [RFL19], a pattern can be represented as a combination of facts describing the problem, context, and solution (the so-called minimal triangle [GM05]), as well as the facts describing the resulting context that can possibly contain new problems:

Definition 3.1 (Pattern, based on [RFL19])

Let F denote the set of all facts, i.e., true or false propositions. A pattern can then be specified as a tuple $p = (\Pi, \sigma, K, R)$, where Π is the problem and σ is the solution, K is the context in which the pattern is applicable, and R is the resulting context, s.t.: (i) $K \subseteq F$, (ii) $\Pi \subseteq K \setminus \emptyset$, (iii) $R \subseteq F \setminus \{\Pi\} \wedge (\neg\Pi \in R)$, (iv) $\sigma : K \rightarrow R$. \square

For example, the CUSTOMIZABLE DEPLOYMENT STACK pattern can be applied when a deployment stack requires customizations and the problem is expressed as the fact “*stack is customizable = false*”. The solution provided by the pattern negates this fact, i.e., “*stack is customizable = true*”, and results in the new context in which the stack requires additional management effort from consumers, effectively introducing an opposite problem “*stack is provider-managed = false*”. Since not all patterns can be meaningfully combined due to conflicts between initial and resulting contexts, the mutually-exclusive patterns can be linked using the *Conflict* pattern relationship. Using semantic equivalence for facts is important, e.g., “*stack is provider-managed = false*” is equivalent to “*stack is consumer-managed = true*” and both are in conflict with “*stack is provider-managed*

= true". Thus, the Definition 3.2 describes a symmetric relationship: conflicting patterns are mutually-exclusive independent of the order they are applied in.

Definition 3.2 (Pattern Conflict)

Let $p_1 = (\Pi_1, \sigma_1, K_1, R_1)$ and $p_2 = (\Pi_2, \sigma_2, K_2, R_2)$ be two distinct patterns that resolve problems Π_1 and Π_2 using solutions σ_1 and σ_2 , respectively. The patterns are in conflict (denoted $p_1 \Rightarrow \Leftarrow p_2$) iff the following condition holds: $(\exists \rho \in R_1, \pi \in \Pi_2 \text{ s.t. } \rho \equiv \pi) \wedge (\exists \rho' \in R_2, \pi' \in \Pi_1 \text{ s.t. } \rho' \equiv \pi')$, where \equiv denotes the semantic equivalence of facts. \square

Unidirectional conflicts, i.e., if condition in Definition 3.2 is relaxed to $(\exists \rho \in R_1, \pi \in \Pi_2 \text{ s.t. } \rho \equiv \pi) \vee (\exists \rho' \in R_2, \pi' \in \Pi_1 \text{ s.t. } \rho' \equiv \pi')$, however, are rather needed for describing the correct (non-conflicting) order in pattern sequences and not discussed further due to the absence of such cases in the language. As discussed previously, patterns can also be combined using the *Combination* pattern relationship to address problems specified in each of the involved patterns. Essentially, such a combination of distinct patterns yields a new, composite pattern that resolves the underlying combination of problems and is applicable in the context when each of the combined patterns are applicable. For example, the `FIXED DEPLOYMENT STACK` and `PROVIDER-MANAGED SCALING CONFIGURATION` patterns can be combined to address the respective problems in both management aspects and, hence enable a more concrete, composite solution simplifying the management requirements for cloud consumers.

Definition 3.3 (Pattern Combination)

Let $p_1 = (\Pi_1, \sigma_1, K_1, R_1)$ and $p_2 = (\Pi_2, \sigma_2, K_2, R_2)$ be two distinct patterns that resolve problems Π_1 and Π_2 using solutions σ_1 and σ_2 , respectively. Their combination p_c exists iff $(\neg p_1 \Rightarrow \Leftarrow p_2) \vee (p_1 \Rightarrow \Leftarrow \neg p_2)$, and it can be formally specified as a tuple $p_c = (\Pi_c, \sigma_c, K_c, R_c)$, where Π_c is the composite problem and σ_c

is the combined solution resolving it, K_c is the context in which the combination is applicable, and R_c is the resulting context, s.t.: (i) $\Pi_c = \Pi_1 \cup \Pi_2$, (ii) $K_c = K_1 \cup K_2$, (iii) $R_c = R_1 \cup R_2$, and (iv) $\sigma_c : K_c \rightarrow R_c$. \square

Moreover, patterns can be refined into patterns of other granularity to address a specialization of the respective problems using a similar solution, which may also deal with additional forces [Nob98]. This behavior is described using the pattern relationship *Refinement*, which is also implicitly related to the opposite *Generalization* relationship that enables abstracting away specialized patterns into more general ones by reversing the direction of *Refinement* relationship.

Definition 3.4 (Pattern Refinement)

Let $p_1 = (\Pi_1, \sigma_1, K_1, R_1)$ be a pattern that resolves a problem Π_1 using a solution σ_1 . Then the pattern $p_2 = (\Pi_2, \sigma_2, K_2, R_2)$ is said to refine p_1 (denoted as $p_2 \xrightarrow{r} p_1$) when the following conditions hold: (i) $\Pi_1 \in \Pi_2$, (ii) $K_2 \subset K_1$, (iii) $\neg\Pi_1 \in R_2$, and (iv) $\sigma_2 : K_2 \rightarrow R_2$. \square

Finally, since patterns may potentially have several refinement and/or generalization variants, this affects the way composite patterns can be refined. For example, the combination of `FIXED DEPLOYMENT STACK` and `PROVIDER-MANAGED SCALING CONFIGURATION` patterns shown in Figure 3.14 can only be refined into the `SERVERLESS HOSTING` pattern since this is the only variant that addresses both respective problems unlike, e.g., `PROVIDER-DEFINED STACK HOSTING` pattern that only refines one of those patterns. The *Refinement of a Pattern Combination* can be specified as:

Definition 3.5 (Refinement of a Pattern Combination)

Let p_c be a combination of n distinct patterns $\{p_1, \dots, p_n\}$. Then the pattern $p_r = (\Pi_r, \sigma_r, K_r, R_r)$ is said to refine P if the following condition holds: $\forall i \in \{1, \dots, n\}: p_r \xrightarrow{r} p_i$. \square

The discussed semantics of pattern relationships enables different kinds of transitioning between abstract and concrete decisions, which is used as a building block in Contribution 4.

3.8 Chapter Summary

This chapter introduces the Component Hosting and Management pattern language, which categorizes and interconnects higher- and lower-level solutions focused on hosting application components in the context of two management aspects: deployment stack and scaling configuration management. Using the introduced patterns and relationships among them enables flexibly expressing hosting decisions independently of the target infrastructure. These patterns can be used independently or in combination with other pattern languages such as cloud computing patterns [FLR+14] to decide on FaaS-based as well as any other application deployments. Moreover, these patterns can also be used in the context of provider-agnostic modeling of FaaS-based applications, which is addressed by Contribution 4 presented in Chapter 6.

Furthermore, the introduced pattern language has multiple directions to evolve as it may cover more aspects of modeling application deployments, e.g., since there already are documented semantic links showing that hosting patterns can be modeled imperatively or declaratively [EBF+17], new patterns documenting *how* such models can be created or *which kinds of technologies* can be employed to enact the deployment can be added in the future. This could eventually lead to a more generalized pattern language focusing on deployment modeling and which subsumes the Component Hosting and Management pattern language as its part.

CLASSIFICATION AND SELECTION OF COMPONENTS FOR FAAS-BASED APPLICATIONS

RAPIDLY changing feature sets and technical constraints of FaaS platforms lead to a substantial heterogeneity of existing solutions. Hosting and integration requirements often vary across different FaaS platforms, even for similar function implementations. For example, commercial offerings such as AWS Lambda [Ama22b], Azure Functions [Mic22], or Google Cloud Functions [Goo22] provide various built-in integration mechanisms, e.g., out-of-the-box event bindings for provider-specific services, which are often lacking in open source platforms. In contrast, open source platforms, such as OpenFaaS [Ope22] or Apache OpenWhisk [Apa22b], provide more customization options as they can be hosted on-premises or using container orchestration engines such as Kubernetes [The22c] while also reducing the dependence on provider-specific services for hosting FaaS-based applications.

Finding a suitable platform for a given set of requirements, thus depends on both higher-level decisions, e.g., available licensing and platform hosting options, and also on more technical, DevOps-oriented decisions, e.g., supported language runtimes and integration of event sources, and available deployment automation support. To address this decision-making problem and provide means for interactive search of provider-specific hosting alternatives for FaaS-related components based on given provider-agnostic requirements, this chapter presents Contribution 2 that introduces a classification framework and selection support mechanisms for FaaS platforms and is based on the peer-reviewed journal publication [YSB+21a].

To provide means for a uniform classification of FaaS platforms, a multi-vocal literature review was conducted, which included a systematic review of academic literature and an analysis of documentation of ten existing commercial and open source FaaS platforms. The resulting FaaS platform classification framework encompasses selection criteria organized into two disjoint views, namely the *business* view that groups higher, management-level criteria and the *technical* view with more technical criteria relevant for developers and operators. These two separate views aim to facilitate finding FaaS platforms suiting the specified provider-agnostic decisions specified by (i) project managers to first identify FaaS platforms that comply with the project and business requirements, and (ii) development and operation specialists to select the FaaS platform supporting the required technical features. To validate this framework, a technology review of ten FaaS platforms (3 commercial and 8 open source) was conducted. For brevity, the review data are provided here only for exemplary purposes when discussing the framework (see [YSB+21a] for more details). In the following, the research method is briefly described followed by an in detail discussion of the captured FaaS-specific classification framework. Further, to enable the classification and interactive search for other component types in FaaS-based applications, this contribution generalizes the introduced concepts by presenting a generic classification framework metamodel and discussing the aspects of data organization and exploration inspired by existing work on the classification of PaaS offerings [Kol19].

4.1 Research Method

This section describes the research process employed to derive the classification framework for FaaS platforms. Figure 4.1 shows the performed sequence of five steps. Since the comparison and selection of services is a well-established research topic, e.g., classification of IaaS [RWZT12] and PaaS [Kol19] offerings, an analysis of academic literature focusing on classification and comparison of FaaS platforms was performed first. Analysis of existing research publications enabled specifying the initial classification framework that organized the community knowledge on the topic. Afterwards, the documentation of ten selected FaaS platforms was analyzed to further improve and refine the classification framework. Each step of the process is briefly discussed in the following.



Figure 4.1: The sequence of steps performed to derive the classification framework for FaaS platforms.

Step 1: Academic Literature Review. Similar to the process described in Section 1.1.1, the initial set of publications was identified by issuing the generic search query (*serverless OR “Function-as-a-Service” OR FaaS OR “Function as a Service”*) against seven electronic databases, namely ACM Digital Library, arXiv.org, Google Scholar, IEEE Xplore, Springer Link, Science Direct and Wiley Online Library. The following inclusion (✓) and exclusion (×) criteria were used to identify relevant publications:

- ✓: Publications written in English that review / evaluate / compare existing FaaS platforms or function orchestrators.
- ×: Publications that are not accessible or do not provide enough details, e.g., extended abstracts, tutorials, demonstration papers.

The initial set of publications was screened using the adaptive reading depth technique [PVK15], which resulted in the identification of five relevant publications [GSP+18; KS18; LSF18; LRLE17; MPd18]. Additionally, the snowballing technique [Woh14] was applied to increase the number of relevant publications including (i) forward snowballing using Google Scholar, to find the research works citing the selected publications, and (ii) backward snowballing to find the research works cited by the selected publications. The snowballing enabled finding six more relevant publications [BO19; GFE+20; KS19; Kum19; PKC19; Raj18], hence increasing the total amount to eleven relevant papers.

Step 2: Derive Initial Classification Framework. The keywording technique [PFMM08] was applied to derive an initial classification framework, which resulted in a set of high-level keywords representing distinct categories, e.g., “Licensing” and “Installation”, and finer-grained keywords describing dimensions and concrete criteria. For example, the “Installation” category comprised the “Type” and “Target Hosts” dimensions describing installation types and available target hosts. Since the reviewed publications rely on different terminology, de-duplication was needed to derive the set of distinct keywords. The initial version of the classification framework captured the core concepts present in state-of-the-art research; however, it lacked technical depth and had no clear organization – these issues were addressed in the next steps of the process.

Step 3: Select Relevant FaaS Platforms. To refine the initial classification framework, the documentation analysis step was performed next: Firstly, a list of candidates was populated by combining the platforms referenced in the reviewed publications with the platforms tracked by CNCF [Clo18]. This list was then pruned to keep only general-purpose platforms, i.e., not tailored only for specific use cases such as machine learning, and with actively maintained code repositories. The pruned list was sorted by popularity, based on search engine hits statistics [WBF+19], and the “bounded effort” stopping criteria [GFM19] was applied by selecting ten platforms. At the time when review was conducted, the serverless landscape from CNCF listed 19 provider-managed and 14 open source FaaS platforms. To

Table 4.1: List of selected FaaS platforms (sorted alphabetically).

Name	Documentation Sources
<i>Apache</i>	https://openwhisk.apache.org ,
<i>Openwhisk</i>	https://github.com/apache/openwhisk
<i>AWS Lambda</i>	https://docs.aws.amazon.com/lambda
<i>Fission</i>	https://docs.fission.io , https://github.com/fission/fission
<i>Fn</i>	https://fnproject.io/ , https://github.com/fnproject
<i>Google Cloud Functions</i>	https://cloud.google.com/functions/docs
<i>Knative</i>	https://knative.dev/docs , https://github.com/knative
<i>Kubeless</i>	https://kubeless.io/docs , https://github.com/kubeless
<i>Microsoft Azure Functions</i>	https://docs.microsoft.com/en-us/azure/azure-functions , https://github.com/Azure/Azure-Functions
<i>Nuclio</i>	https://nuclio.io/docs , https://github.com/nuclio/nuclio
<i>OpenFaaS</i>	https://docs.openfaas.com , https://github.com/openfaas/faas

better capture possible gaps between commercial and open source platforms, the review covered 50% (7 entries) of open source FaaS platforms, thus three commercial platforms were selected due to the bounded effort limit. Three most popular proprietary platforms were chosen by comparing Google Search hits resulting in AWS Lambda, Microsoft Azure Functions, and Google Cloud Functions being selected. For open source platforms, the number of GitHub stars was compared, resulting in OpenFaaS, Apache Openwhisk, Nuclio, Fission, Fn, Kubeless, and Knative being selected. Table 4.1 lists the resulting platforms and documentation sources.

Step 4: Analyze Platform Documentation. Only official documentation sources, e.g., official websites and code repositories, were considered for the analysis. The initial classification framework with its keywords was used as a starting point for the analysis, resulting in refinements of

mappings between high- and lower-level keywords, e.g., “Development”: (“Function Runtimes”, ...) was extended with the “Runtime Customization” criterion since multiple platforms provided extension mechanisms for enabling unsupported function runtimes. During this step, multiple additional technical criteria were added, hence requiring a better categorization and structuring of the framework in the next step.

Step 5: Refine Classification Framework. The final version of the FaaS platforms classification framework was derived by reviewing and reorganizing the mappings between captured keywords representing categories, dimensions, and concrete selection criteria, e.g., “Testing” and “Debugging” were combined into one category due to the close relation with each other. Likewise, the “Observability” category was introduced to combine criteria related to logging and monitoring. Further, certain initially captured keywords were excluded, e.g., security-related aspects like code isolation mechanisms, were removed as they required more specialized analysis beyond the scope of this research work.

4.2 A Framework for Classifying FaaS Platforms

To choose a FaaS platform different requirements must be considered: more general aspects such as licensing and code availability intertwine with technical requirements such as native support for particular event bindings. To address such separation of concerns, the presented FaaS platform classification framework is organized into two views – the *business* view intended for non-technical personnel and the *technical* view targeting software developers and operators. This section elaborates on both views and corresponding criteria using examples obtained during the technology review of the ten FaaS platforms, which was conducted using this classification framework [YSB+21a] as discussed in Section 4.1. In addition, the detailed discussion of the technology review results is omitted and can be accessed in the original publication [YSB+21a].

4.2.1 A Business View for Classifying FaaS Platforms

The *business* view organizes categories, dimensions, and criteria representing general, project-level requirements including such aspects as the platform licenses and the possibility of on-premises installation since both decisions may affect how the software developed in a project can be integrated with the platform [Lau04]. Figure 4.2 depicts the view with its dimensions, categories, and criteria, which are explained below.

Licensing. This category comprises licensing-related details: *License* specifies the actual license name, whereas *Type* reflects the corresponding license type. Figure 4.2 shows example values for names and types of licenses under which FaaS platforms are released, e.g., open source [Lau04] or proprietary options. For instance, multiple non-commercial FaaS platforms such as Apache Openwhisk [Apa22b] or Fission [The22a] are released under the permissive *Apache 2.0 License*, and the permissive *MIT License* are encountered too (OpenFaaS [Ope22]).

Installation. This category classifies FaaS platforms based on the supported hosting options including the *Type* of installation, e.g., *as-a-service* and/or *installable*, as well as on which *Target Hosts* the platform can be installed, e.g., operating systems and container orchestration engines as shown in Figure 4.2. In case a platform is only offered *as-a-service*, such as AWS Lambda [Ama22b] or Google Cloud Functions [Goo22], the *Target Hosts* remains empty. There exist platforms such as Azure Functions [Mic22] or Nuclio [Igu22] that are both installable and offered as-a-service. Open source platforms can be installed on different hosts and Kubernetes [The22c] is one of the most frequently supported targets.

Source Code. This category helps differentiating platforms based on their code *Availability*, i.e., *closed source* or *open source* platforms. Open source platforms can be distinguished based on the *Open Source Repository* and main *Programming Language* in which they are implemented, as shown in Figure 4.2. Although commercial platforms are mainly closed source, there exist opposite cases, e.g., the Azure Functions runtime is open source

4 Classification and Selection of Components for FaaS-based Applications

and can be installed on-premises. Additionally, open source platforms are typically available on GitHub – multiple existing open source FaaS platforms are implemented in Go.

Licensing	<i>License</i>	Apache 2.0, GNU GPL 3.0, MIT, ...	
	<i>Type</i>	permissive, copyleft, proprietary, ...	
Installation	<i>Type</i>	as-a-service, installable	
	<i>Target Hosts</i>	Kubernetes, Linux, MacOS, ...	
Source Code	<i>Availability</i>	open source, closed source	
	<i>Open Source Repository</i>	BitBucket, GitHub, SourceForge, ...	
	<i>Programming Language</i>	C, Go, Java, JavaScript, Python, ...	
Interface	<i>Type</i>	CLI, API, GUI	
	<i>Application Management</i>	creation, retrieval, update, deletion	
	<i>Platform Administration</i>	deployment, configuration, enactment, termination, undeployment	
Community	<i>GitHub</i>	<i>Stars</i>	number
		<i>Forks</i>	number
		<i>Issues</i>	number
		<i>Commits</i>	number
		<i>Contributors</i>	number
	<i>Stackoverflow</i>	<i>Questions</i>	number
Documentation	<i>Functions</i>	development, deployment	
	<i>Platforms</i>	usage, development, deployment, architecture	

Figure 4.2: The *business* view of the classification framework: white cells contain categories, lighter gray cells contain classification dimensions, dark gray cells list example values [YSB+21a].

Interface. This category classifies platforms based on the supported interface *Types* such as Command Line Interface (CLI), API, and/or GUI. Additionally, it includes *Application Management* and *Platform Administration* dimensions: the former describes operations a platform provides to manage applications, whereas the latter indicates whether a FaaS platform supports *deployment*, *configuration*, *enactment*, *termination* and *undeployment* operations for administering the platform itself. While many platforms provide CLIs and APIs, GUIs are often provided only by commercial platforms. Further, platform administration interfaces typically vary more among platforms in terms of the supported functionalities compared to application management interfaces.

Community. Platforms can also be characterized based on the development community, e.g., size and popularity of open source repositories [GSN+19]. Thus, this category includes quantitative information such as the number of *Stars*, *Forks*, *Issues*, *Commits* and *Contributors* obtained from the *GitHub* repository of a platform. Another dimension keeps track of platform-related *Questions* on *Stackoverflow* as an additional indicator of the interest in platforms. The amount of questions on StackOverflow is generally significantly larger for commercial FaaS platforms due to the maturity of the corresponding product. e.g., AWS Lambda questions are significantly more frequent than questions about the Fission [The22a] platform. Open source platforms, on the other hand, differ significantly in popularity and community characteristics: platforms such as OpenFaaS [Ope22], Apache Openwhisk [Apa22b], and Knative [The22b] demonstrate stronger public interest compared to other platforms.

Documentation. This category classifies platforms based on the kinds of documentation available: *Functions* comprises function *development* and *deployment* documentation. Additionally, the *Platform* comprises criteria related to the documentation of platform *usage*, *development*, *deployment* and *architecture*. While the documentation on deployment and platform usage is generally provided by most platforms, not all open source platforms actually provide architecture and development documentation.

4.2.2 A Technical View for Classifying FaaS Platforms

Technical, lower-level classification criteria constitute the *technical* view of the FaaS platforms classification framework. Using this view, software development and operations specialists can identify suitable platforms based technical requirements, e.g., support for a specific function runtime. Figure 4.3 and Figure 4.4 present the view with its categories and dimensions, which are discussed in detail below.

Development. This category organizes criteria related to different aspects of functions development as shown in Figure 4.3. This includes dimensions that cover supported *Function Runtimes*, e.g., Java 11 runtime, and whether the platform provides means for *Runtime Customization*, e.g., using customized container images. Java, Node.js, and Python runtimes are among the most popular function runtimes supported in both commercial and open source FaaS platforms. Additionally, runtime customization is often achieved by supporting Docker images. Further, a platform can offer plugins for *IDEs and Text Editors* such as IntelliJ IDEA or Visual Studio Code to enable syntax highlighting or automated code packaging, and hence, facilitate implementing functions. Likewise, language-specific *Client Libraries* can be provided to enable programmatic access to APIs of the platform. Typically, IDE and text editor plugins are provided by commercial platforms. While language-specific client libraries are often also provided for open source platforms, the extensiveness of SDKs and the number of supported languages is often higher for commercial platforms. Finally, the *Quotas* dimension indicates if the *Deployment Package Size* or *Execution Time* are restricted by the platform. Such quotas often exist in commercial platforms, whereas open source platforms often enable configuring limits, e.g., to control the resource consumption.

Version Management. The next category shown in Figure 4.3 covers the *Version Management* mechanisms offered by the platform, i.e., managing single *Function versions* or *Application versions* grouping multiple functions and, possibly, the related components that interact with functions. While versions can be encoded *implicitly*, e.g., a semantic version

value added as part of a function name or its namespace, the support for *dedicated mechanisms* is a more organized way to manage versions. Open source platforms more frequently support only implicit versioning opposing to commercial platforms that often provide dedicated versioning mechanisms. Additionally, versioning of FaaS-based applications in open source platforms is mainly limited to grouping functions since open source platforms are standalone products without built-in integration with other provider-specific services, e.g., databases or message queues.

Event Sources. This category groups criteria related to the support for different kinds of event sources that can trigger functions as shown in Figure 4.3. As discussed in Section 1.1.2, exposing a FaaS function as an *Endpoint* using API Gateways is a common use case for FaaS-based applications. Several additional criteria are also relevant for this event source type, namely the support for (i) *Synchronous Call* or *Asynchronous Call* of functions made via specific protocols such as Hypertext Transfer Protocol (HTTP), (ii) *Endpoint Customization*, e.g., to customize the name of the endpoint, and (iii) *TLS Support* for the secure triggering of functions using HTTPS. The support for these aspects may differ among platforms, e.g., synchronous, HTTP-based function calls are supported more often than asynchronous function calls, while FaaS platforms generally support the customization of endpoints and calls via HTTPS.

The *Data Store* dimension covers event sources that represent higher-level storage types [MTB18]. *File Level* covers object stores like AWS S3 [Ama22b], whereas *Database Mode* is concerned with relational and non-relational databases like Azure CosmosDB [Mic22]. Commercial platforms typically document support for different provider-specific data stores, whereas open source platforms often provide less built-in integration mechanisms with different storage types as they are mainly standalone products and have no direct relation to any provider-specific service ecosystems.

Next, the *Scheduler* dimension covers platform support for scheduled function invocation, which is often implemented as cron jobs. The *Message Queue* and *Stream Processing Platform* dimensions cover the platform

support for triggering function using messaging and streaming services such as AWS Simple Notification Service (SNS) [Ama22b] or Apache Kafka [Apa22a]. Support for scheduled function invocation and messaging-related even sources is documented by multiple platforms. Services such as AWS Alexa [Ama22b] or IBM Watson [IBM21] in certain cases can also serve as event sources – such special cases are covered by the *Special-*

Development	<i>Function Runtime</i>	Go, Java, JavaScript, Docker, ...	
	<i>Runtime Customization</i>	supported, not supported	
	<i>IDEs & Text Editors</i>	IntelliJ IDEA, Eclipse, VSCode, ...	
	<i>Client Libraries</i>	Go, Java, JavaScript, Python, ...	
	<i>Quotas</i>	<i>Deployment Package Size</i>	present, not present
		<i>Execution Time</i>	present, not present
Version Management	<i>Application Versions</i>	dedicated mechanisms, implicit versioning	
	<i>Function Versions</i>	dedicated mechanisms, implicit versioning	
Event Sources	<i>Endpoint</i>	<i>Synchronous Call</i>	HTTP, gRPC, ...
		<i>Asynchronous Call</i>	HTTP, gRPC, ...
		<i>Endpoint Customization</i>	supported, not supported
		<i>TLS Support</i>	supported, not supported
		<i>Data Store</i>	<i>File Level</i>
		<i>Database Mode</i>	Azure Cosmos
	<i>Scheduler</i>	supported, not supported	
	<i>Message Queue</i>	AWS SQS, RabbitMQ, ...	
	<i>Stream Processing Platform</i>	AWS Kinesis, Apache Kafka, ...	
	<i>Special-purpose Service</i>	AWS Alexa, GitHub, IBM Watson, ...	
<i>Event Source Integration</i>	plugins development, messaging-based integration		

Figure 4.3: The *technical* view of the classification framework: white cells contain categories, lighter gray cells contain dimensions, dark gray cells list example values [YSB+21a].

purpose Service dimension. Finally, the *Event Source Integration* dimension covers supported ways to enable triggering functions using custom event sources, e.g., by developing plugins or using webhooks.

Function Orchestration. This category shown in Figure 4.4 groups criteria related to the support for function orchestration. In particular, it covers platform support for modeling the interactions of multiple functions as workflows [LR00] that can be executed by compatible function orchestrators, e.g., functions hosted on AWS Lambda can be composed into complex orchestrations executed using the AWS Step Functions [Ama22b] offering. Such function orchestrators are typically standalone products offered as-a-service, e.g., AWS Step Functions, or complementary parts of open source platforms, e.g., Apache Openwhisk Composer [Apa22c].

Several dimensions are included as a baseline for the classification of such function orchestrations. Since the function orchestration modeling approaches vary for different function orchestrators [GSP+18], the *Workflow Definition* dimension indicates both whether function orchestrations are possible and which modeling approach is supported, e.g., using custom DSL such as ASL [Ama22b] or general-purpose programming language such as Python. Next dimension covers the availability of documentation for supported *Control Flow Constructs*, e.g., how to model parallel and sequential function invocations, or how to enable error handling. Similar to FaaS platforms, function orchestrators can impose certain *Quotas*, i.e., constraints on the *Execution Time* or the *Task Input and Output Size*.

Testing and Debugging. The next category shown in Figure 4.4 focuses on the testing and debugging aspects. For instance, platforms may provide mechanisms to simplify *Functional* and *Non-functional* testing, e.g., by offering platform-specific libraries or dedicated CLI commands. Likewise, mechanisms that facilitate *Local and Remote* debugging can also be provided by the platform – for both dimensions this could include platform-native and third-party tooling. Commercial and open source

4 Classification and Selection of Components for FaaS-based Applications

FaaS platforms often provide some mechanisms that facilitate functional testing and local debugging of functions with commercial platforms often offering more sophisticated solutions.

Observability. This category groups criteria related to the observability aspects for FaaS-based applications as shown in Figure 4.4. This includes the *Logging* and *Monitoring* dimensions that indicate supported *platform-native* and *third-party* tools for logging and monitoring, respectively. The documented means of integrating existing logging and monitoring tools is

Function Orchestration	<i>Workflow Definition</i>	standard language, custom DSL,	
	<i>Control Flow Constructs</i>	documented, not documented	
	<i>Quotas</i>	<i>Execution Time</i>	present, not present
		<i>Task Input and Output Size</i>	present, not present
Testing and Debugging	<i>Testing</i>	<i>Functional</i>	platform-native tooling, 3 rd party tooling
		<i>Non-functional</i>	platform-native tooling, 3 rd party tooling
	<i>Debugging</i>	<i>Local</i>	platform-native tooling, 3 rd party tooling
		<i>Remote</i>	platform-native tooling, 3 rd party tooling
Observability	<i>Logging</i>	platform-native tooling, 3 rd party tooling	
	<i>Monitoring</i>	platform-native tooling, 3 rd party tooling	
	<i>Tooling Integration</i>	push-based, pull-based, plugin development, ...	
Application Delivery	<i>Deployment Automation</i>	platform-native tooling, 3 rd party tooling	
	<i>CI/CD Pipelining</i>	supported, not supported	
Code Reuse	<i>Function Marketplace</i>	official marketplace, 3 rd party marketplaces	
	<i>Code Sample Repository</i>	present, not present	
Access Management	<i>Authentication</i>	built-in, external, ...	
	<i>Access Control</i>	functions, resources, ...	

Figure 4.4: The *technical* view of the classification framework (continued): white cells contain categories, lighter gray cells contain dimensions, dark gray cells list example values [YSB+21a].

covered by the *Tooling Integration* dimension, e.g., the platform supports sending events and logs to an external monitoring component (push-based integration). As in previous categories, commercial platforms often support sophisticated provider-specific observability solutions such as AWS CloudWatch [Ama22b], whereas open source platforms mainly rely on the integration with third-party tools.

Application Delivery. This category shown in Figure 4.4 groups criteria related to the delivery of FaaS-based applications. Firstly, the *Deployment Automation* dimension covers the documented support for provider-specific and third-party deployment automation tools, such as AWS Cloud Formation [Ama22b] and Serverless Framework [Ser22b]. FaaS platforms tend to support deployment automation tools that rely on declarative deployment models [EBF+17]. In addition, the *CI/CD Pipelining* dimension covers the documented mechanisms for pipelining the DevOps processes: while commercial platforms often provide native support for CI/CD tooling, open source platforms rarely document dedicated integration mechanisms.

Code Reuse. The criteria related to code reuse aspects, such as available repositories of existing applications and code examples, are covered by this category as shown in Figure 4.4. Firstly, the *Function Marketplaces* dimension indicates whether application marketplaces that enable reusing existing FaaS-based applications are available. Likewise, the *Code Sample Repositories* dimension reflects the code example repositories maintained by cloud providers or open source communities. In general, the availability of function marketplaces such as AWS SAR [Ama22c] is more limited than code example repositories that are provided by most platforms.

Access Management. Finally, the supported access management mechanisms are covered by this category shown in Figure 4.4. This includes support for native or third-party *Authentication* mechanisms as well as support for *Access Control* mechanisms that enable defining access rules for functions, e.g., to prevent the function accessing certain data stores. Similar to other categories, commercial platforms often provide native

mechanisms for authentication and access control while open source platforms often rely on the features of the underlying hosting environment, e.g., mechanisms provided by the underlying container orchestration engine.

4.3 A Generic Metamodel for Technology Classification Frameworks

While the introduced classification framework focuses on FaaS platforms, multiple technologies can be categorized using similar criteria, e.g., licensing of storage components or event-driven invocation of container-based services. To enable using the same technology classification mechanisms as discussed previously for other technology types, this section presents a generic metamodel for technology classification frameworks. The meta-

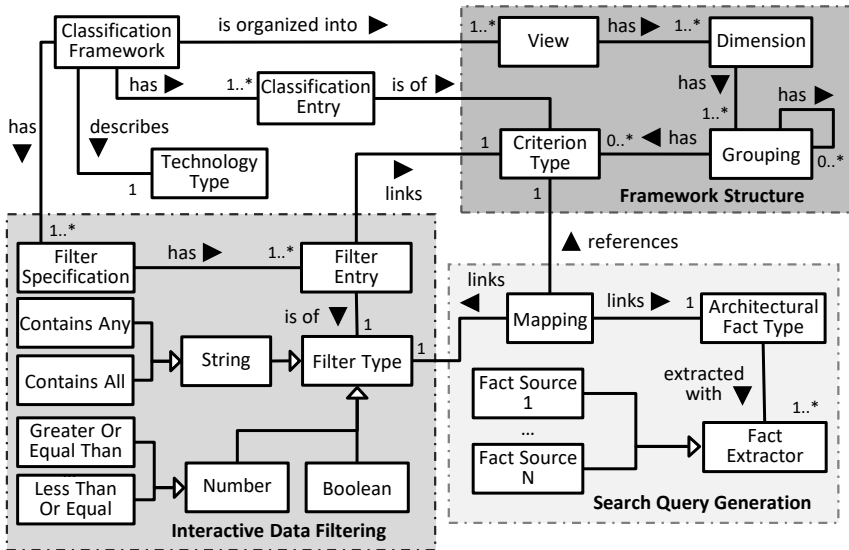


Figure 4.5: A metamodel of the technology classification framework.

model shown in Figure 4.5 focuses on supporting three major aspects, namely (i) organization of technology classification data, (ii) interactive search of technologies, and (iii) generation of search queries from external *architectural fact sources* such as application deployment models.

To formally describe the classification framework metamodel, let Σ denote the set of all ASCII characters, then the set of all possible strings over Σ is denoted using the Kleene closure, i.e., Σ^* when the empty string is included and Σ^+ otherwise. Let $DE \subseteq \Sigma^+ \times \Sigma^+ \times (\Sigma^+ \cup \{\perp\})$ be a set of all data elements, then $de = (ID, DataType, Value) \in DE$ is a uniquely identifiable data element. The *DataType* describes the allowed type of values such as string, integer, or unspecified in case $Value = \perp$. Additionally, to refer to specific tuple elements, a projection operator π_j is employed: Let X_1, \dots, X_n be arbitrary sets and $X = X_1 \times \dots \times X_n$ be their Cartesian product representing the set of all n -tuples with i -th component, $1 \leq i \leq n$, being an element $x_i \in X_i$. The projection operator π_j , $1 \leq j \leq n$, is then the mapping $\pi_j : X_1 \times \dots \times X_n \rightarrow X_j$, $(x_1, \dots, x_n) \mapsto x_j$ that maps a tuple to its j -th element. In the following, the metamodel shown in Figure 4.5 is formally described starting from its central entity, the Technology Classification Framework:

Definition 4.1 (Classification Framework)

Let the set of all technology classification frameworks be denoted as Φ and the set of all technology types be denoted as Θ . A classification framework $\phi_\theta \in \Phi$ describing technologies of type $\theta \in \Theta$ is a tuple: $\phi_\theta = (KT_\phi, K_\phi, \Gamma_\phi, \Delta_\phi, Y_\phi, FT_\phi, F_\phi, FS_\phi, AT_\phi, A_\phi, AS_\phi, E_\phi, M_\phi, data_{KT}, type_K, nested_\Gamma, data_{AT}, type_A)$ □

The metamodel entities constituting the tuple are:

- KT_ϕ is the set of all *Criterion Types* (Definition 4.2),
- K_ϕ is the set of all *Criterion Entries* (Definition 4.3),
- Γ_ϕ is the set of all *Criterion Groupings* (Definition 4.4),
- Δ_ϕ is the set of all *Framework Dimensions* (Definition 4.5),

- Y_ϕ is the set of all *Framework Views* (Definition 4.5),
- FT_ϕ is the set of all *Filter Types* (Definition 4.6),
- F_ϕ is the set of all *Filter Entries* (Definition 4.7),
- FS_ϕ is the set of all *Filter Entries* (Definition 4.7),
- AT_ϕ is the set of all *Architectural Fact Types* (Definition 4.8),
- A_ϕ is the set of all *Architectural Facts* (Definition 4.8),
- AS_ϕ is the set of all *Architectural Fact Extractors* (Definition 4.9),
- E_ϕ is the set of all partial functions that map data in fact sources to architectural facts (Definition 4.9),
- M_ϕ is the set of all *Criterion Mappings* (Definition 4.10).

Moreover, the tuple comprises the following maps for associating:

- $data_{KT}$: criterion types with allowed data elements (Definition 4.2),
- $type_K$: criterion entries with criterion types (Definition 4.3),
- $nested_\Gamma$: criterion groupings with nested groupings (Definition 4.4),
- $data_{AT}$: fact types with allowed data elements (Definition 4.8),
- $type_A$: facts with fact types (Definition 4.8).

In the next subsections, the metamodel entities are discussed in detail following the same order listed above.

4.3.1 Structure of Classification Frameworks

Classification frameworks allow describing technologies using clearly-structured, typed classification criteria. This subsection discusses the structuring of classification frameworks and describes how frameworks can be visualized. A *Classification Criterion Type* represents a distinct characteristic of one or more technology types and defines how the corresponding *technology classification criterion entries* are structured, e.g., an integer value or a list of string values. Criterion types specify how the underlying technology classification data are structured using primitive

types and compound data structures. For example, the criteria discussed in Section 4.2 can be atomic boolean and integer values, or lists of strings, e.g., listing supported installation targets.

Definition 4.2 (Classification Criterion Type)

Let $KT_\phi \subseteq \Sigma^+$ be the set of technology classification criteria types, then $\kappa\tau \in KT_\phi$ identifies a distinct classification criterion type. Further, the map $data_{KT}: KT_\phi \rightarrow \wp(DE)$ associates with each criterion type $\kappa\tau$ a set of allowed data elements $data_{KT}(\kappa\tau)$ describing the classification criterion type data. \square

As classification frameworks describe one or more technologies of some type θ , multiple *Criterion Entries* of the same criterion type can exist. For example, distinct entries about the supported event triggers can exist for different FaaS platforms. Each criterion entry has a unique identifier that enables selectively accessing it and a *Value* $\subseteq \wp(DE)$ that describes the respective technology characteristic, and is constructed according to the corresponding criterion type $\kappa\tau \in KT_\phi$ with the *Value* $= \perp$ representing null values. Furthermore, each criterion entry has a *TechnologyName* representing concrete technology, e.g., a classification criterion entry (“license.type”, “AWS Lambda”, “proprietary”) describes the license type for the AWS Lambda FaaS platform.

Definition 4.3 (Classification Criterion Entry)

Let $K_\phi \subseteq \Sigma^+ \times \Sigma^+ \times (\wp(DE) \cup \{\perp\})$ be the set of technology classification criteria entries, then $\kappa = (ID, TechnologyName, Value) \in K_\phi$ represents a distinct classification criterion entry. Additionally, the map $type_K: K_\phi \rightarrow KT_\phi$ associates with each criterion entry κ its type $type_K(\kappa)$. \square

As shown in Figure 4.5, criterion types are organized using the so-called *Groupings*, which can also be nested to enable isolating criterion types into finer-grained subgroups. The concepts of groupings and their nesting relation are defined as follows:

Definition 4.4 (Criterion Groupings)

Let $\Gamma_\phi \subseteq \Sigma^+ \times \wp(KT_\phi) \setminus \emptyset$ be the set of criterion groupings and $\gamma = (\text{ID}, \text{CriteriaTypes}) \in \Gamma_\phi$ be a distinct grouping that comprises a unique identifier used for ordering and selective access, and a set of zero or more *CriteriaTypes* $\subseteq KT_\phi$ referenced in this grouping. To represent the nesting relation, let $<$ be a strict partial order relation over Γ_ϕ s.t. $\forall \gamma_x \in \Gamma_\phi: \{\gamma_y \in \Gamma_\phi: \gamma_x < \gamma_y\}$. To access the nested groupings, let $nested_\Gamma: (\Gamma_\phi, <) \rightarrow \wp(\Gamma_\phi)$ be the map that associates groupings with their nested groupings. \square

The ordered groupings of criteria are organized using the so-called *Dimensions* that comprise the root-level groupings, which can have zero or more nested groupings. Finally, the dimensions of a classification framework constitute one or more non-empty *Views*, such as managerial or DevOps view discussed in Section 4.2. These framework elements are shown in Figure 4.5 and can be defined as follows:

Definition 4.5 (Framework Dimensions and Views)

Let $\Delta_\phi \subseteq \Sigma^+ \times \wp(\Gamma_\phi)$ be the set of framework dimensions and $\delta = (\text{ID}, \text{Groupings}) \in \Delta_\phi$ be a distinct dimension that has a unique identifier for selective access and a set of one or more groupings. Then, the set of framework views can be defined as $\Upsilon_\phi \subseteq \Sigma^+ \times \wp(\Delta_\phi)$, then $\nu = (\text{ID}, \text{Dimensions}) \in \Upsilon_\phi$ represents a distinct framework view referencing one or more *Dimensions*. \square

The described structures can be used (i) for visualizing the framework itself, e.g., to present criteria similar to Section 4.2 with example values, and (ii) for exploring the actual criteria values for different technologies.

Algorithm 4.1 Generate a visual representation of a framework

Input: Framework views Y_ϕ

Output: Printable view representations R

```

1: function GENERATEFRAMEWORKREPRESENTATION( $Y_\phi$ )
2:    $R \leftarrow \emptyset$  // Initialize the set of printable view representations
3:   for all  $v \in Y_\phi$  do
4:      $r \leftarrow \text{ViewRepresentation}()$  // Initialize view representation
5:     for all  $\delta \in \pi_2(v)$  do
6:       GeneratePrintableRepresentations( $\delta, r$ )
7:     end for
8:      $R \leftarrow R \cup \{r\}$ 
9:   end for
10:  return  $R$ 
11: end function
12: function GENERATEPRINTABLEREPRESENTATIONS( $\delta, r$ )
13:  for all  $\gamma \in \pi_2(\delta)$  do
14:    // Generate a representation of nested groupings and their criteria
15:     $r.\text{add}(\text{createGroupingRepresentation}(\pi_1(\delta), \gamma));$ 
16:    GeneratePrintableRepresentations(nested $_\Gamma(\gamma), r$ );
17:  end for
18: end function

```

Algorithm 4.1 outlines the general flow: for each view, all dimensions are traversed, and *printable representations*, e.g., HyperText Markup Language (HTML) or Markdown snippets, are generated for groupings and their criteria – in case the framework itself is visualized then example values for each criterion can be printed. Hence, the *createGroupingRepresentation* function in Algorithm 4.1 is shown only abstractly, without detailing how nested criteria groups are intended to be visualized, e.g., using HTML elements. The next subsection discusses the aspects of interactive technology data exploration, i.e., how to enable searching for technologies that fulfill some combination of criteria-specific requirements.

4.3.2 Technology Data Filtering

To enable flexible specification of which framework-specific requirements should be available for technology search formulation, the metamodel shown in Figure 4.5 employs framework-specific *Filter Configurations* that group together one or more *Filter Entries* of specific *Filter Type* that is compatible with to-be-filtered Criteria Types.

Definition 4.6 (Filter Type)

Let $C = K_\phi \times \wp(DE) \rightarrow \{True, False\}$ be the set of all partial Boolean functions associated with comparing criteria entries. Then, $FT_\phi \subseteq \Sigma^+ \times C$ be the set of all filter types for classification criteria. Then $ft = (ID, comparator) \in FT_\phi$ is a distinct filter type with a unique identifier, which is compatible with some data type such as string or integer. A partial Boolean function $comparator \in C$ represents some comparison of data values. \square

Filter types represent different ways of deciding on the inclusion of classification entries for a given user query, e.g., a set of technology classification entries can be filtered based on one or more required license values. For instance, if technologies licensed under the “Apache2.0” license are needed, a filter type (“*string.equality*”, $comparator_{streq}$) can be used where $comparator_{streq}$ represents the string equality comparison, i.e., $\forall x \in K_\phi, \forall s \in \wp(DE): (comparator_{streq}(x, s) = True \Leftrightarrow \pi_3(x) = s)$. Thus, the set of filtered criteria entries K_f will be constructed as $K_f = \{x \in K_\phi: comparator_{streq}(x, “Apache2.0”) = True\}$. When several licenses are allowed, e.g., {“Apache2.0”, “MIT”}, the comparison can be made based on disjunction or conjunction of value comparisons in the list, i.e., *Contains Any* and *Contains All* filter types in Figure 4.5.

While filter types only represent possible kinds of comparisons for technology classification data, the actual specification of how specific criteria types can be filtered is provided as *Filter Entries* that specify how specific

criteria types are preferred to be compared, e.g., searching for a technology with the smallest number of contributors vs. the technologies with large communities of contributors. A set of specified filter entries for criteria types constitute the *Filter Specification* of a classification framework, which enables generating visual filter representations for interactive exploration of the technology review data.

Definition 4.7 (Filter Entry and Filter Specification)

Let $F_\phi \subseteq \Sigma^+ \times FT_\phi \times KT_\phi$ be the set of filter entries of a classification framework, then $f = (ID, FilterType, CriterionType) \in F_\phi$ represents a distinct association of a filter type to a criterion type. Then, a filter specification $FS_\phi \subseteq F_\phi$ represents a specific combination of filter entries that can be used to filter the criteria entries in the classification framework. □

The filtering-related entities in the metamodel shown in Figure 4.5 enable flexible configuration of filters for data exploration: by defining filter specifications, the corresponding visual filter representations, e.g., HTML form elements, for each contained filter entry can be generated as shown in the simplified Algorithm 4.2. The filter representations generation (shown with the `GenerateFilterRepresentation()` method in Algorithm 4.2) is technology-specific, e.g., different templating engines can be used to generate the underlying HTML representations, and enable formulating user queries against desired criteria types, e.g., by entering values in generated HTML form elements representing respective filter entries. Filter specifications also enable restricting which data are to be filtered, i.e., only a subset of criteria types in the framework can be made filterable by users. Using the generated visual representation of the filter specification, interactive search for suitable technology options can be performed by combining the available filter entries. Essentially, each employed filter entry is a sub-query made with the comparator for the underlying filter type, e.g., a sub-query made using the $comparator_{streq}(x, "Apache2.0") = True$ searches the

Algorithm 4.2 Generate a visual representation of a filter specification

Input: Framework filter specification FS_ϕ

Output: Printable filter representations FP

```
1: function GENERATEFILTERREPRESENTATIONS( $FS_\phi$ )
2:    $FP \leftarrow \emptyset$  // Initialize the set of printable filter representations
3:   for all  $f \in FS_\phi$  do
   // Generate a type-specific filter representation
4:      $fp \leftarrow \text{GenerateFilterRepresentation}(f)$ 
5:      $FP \leftarrow FP \cup \{fp\}$ 
6:   end for
7:   return  $FP$ 
8: end function
```

technologies licensed under “Apache2.0”. Thus, a *user-defined combination of sub-queries* for the chosen filter entries yields the actual query Q for searching technologies that fulfill the given requirements.

There exist different ways to combine the results for sub-queries and rank the identified technology alternatives. For example, strict or relaxed matching approach [Kol19] can be used in which the inclusion of a technology as a suitable alternative is defined as the conjunction of all sub-queries specified via the filter entries in strict mode and if no results are returned some criteria (and the corresponding filter entries) can be considered optional to find partially suitable technologies. Additionally, multi-criteria decision making algorithms can help to rank the technologies fulfilling the specified requirements, e.g., Simple Additive Weighting (SAW) [SDH+14] can be used to calculate the score of an alternative as $A_i = \sum w_j x_{ij}$ with w_j being the weight of the criterion j and x_{ij} being the score of the A_i w.r.t. j . As a simple example of its application, all criteria weights can be considered equal to one and scores equal to one only if the filter condition is fulfilled,

which would correspond to the relaxed matching approach [Kol19] since the identified technology alternatives will be ranked based on the number of sub-queries they fulfill.

4.3.3 Generation of Technology Search Queries Based on Extracted Architectural Facts

The classification criteria for FaaS platforms presented in Section 4.2 represent various kinds of facts for FaaS-specific application components that may influence specific architectural decisions. Following the ontology of architectural design decisions by Kruchten [Kru04], such architectural facts can contribute to making (i) *existence decisions*, e.g., the presence of a specific kind of event sources such as timers or databases that trigger a function, (ii) *property decisions* such as support for HTTPS for function calls, and (iii) *executive decisions* such as programming language requirements in which components have to be implemented. While being related to particular components, e.g., a specific FaaS function, such architectural facts may, however, be distributed across several components on the level of architecture models.

For example, consider the technology-agnostic model fragment shown in Figure 4.6 in which a FaaS function can be triggered by an object storage or a message queue and can store results in a NoSQL database. The properties related to the FaaS function, and, hence the underlying FaaS platform, are contained in different entities of the model, e.g., the required event sources support is represented via the “Triggers” connectors, the required function runtime is encoded in the component type “Java Function”, and the availability of a Java SDK is an optional property of a FaaS platform that could be used to rank equally-suitable alternatives. These component-specific *architectural facts* contained in application model fragments can be used to automatically generate search queries for finding suitable component alternatives when designing FaaS-based applications.

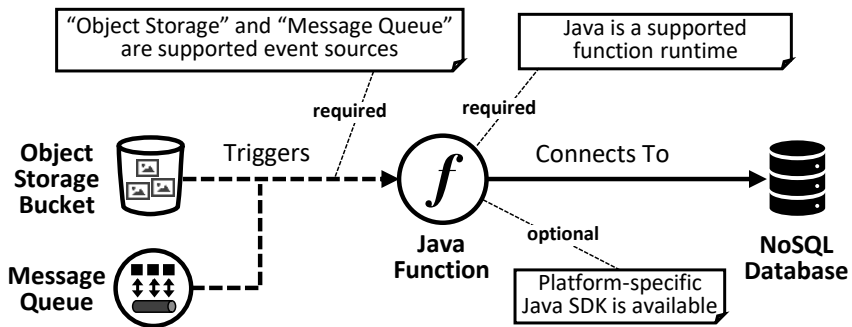


Figure 4.6: Example architectural facts in an application model.

In this context, architectural facts can be seen as distinct typed values that reflect, e.g., the presence of a required event trigger or function runtime for specific component of the application.

Definition 4.8 (Typed Architectural Facts)

Let $AT_\phi \subseteq \Sigma^+$ be the set of architectural fact types associated with a classification framework, then $\alpha\tau \in AT_\phi$ represents a distinct architectural fact type. The map $data_{AT} : AT_\phi \rightarrow \wp(DE)$ associates with each fact type $\alpha\tau$ a set of allowed data elements $data_{AT}(\alpha\tau)$ describing the underlying fact type data. Further, let $A_\phi \subseteq \Sigma^+ \times \Sigma^+ \times \wp(DE)$ be the set of all architectural facts and $\alpha = (ID, Component, Value) \in A_\phi$ be a distinct fact for some application component with a value of the corresponding data type such as string or array of strings. The map $type_A : A_\phi \rightarrow AT_\phi$ associates with each fact α its type $type_A(\alpha)$. □

However, given some *fact source* such as a deployment model fragment and a specific component for which the search query should be generated, the related architectural facts need first to be (i) extracted and (ii) mapped to the respective framework criteria. As application models can be specified in different languages, e.g., deployment models specified in TOSCA or

Terraform, architectural facts contained in those model formats require different *fact extractors*. While source code or running application instances can also be considered as fact sources, only deployment models as example fact sources are assumed in the context of this chapter.

Definition 4.9 (Architectural Fact Extractors)

Let $AS_\phi \subseteq \Sigma^+ \times \Sigma^+ \times \wp(DE)$ be the set of all fact sources and $\alpha\sigma = (ID, SourceType, Value) \in AS_\phi$ represents a distinct fact source of some type that comprises structured data about an application relevant for searching a specific technology, e.g., properties of components and their relationships. Let E_ϕ be the set of all partial functions that map data in fact sources to architectural facts, then a distinct partial function $e \in E_\phi: AS_\phi \rightarrow \wp(A_\phi)$ is called a fact extractor capable of extracting facts for application components from a given fact source. □

Further, *mappings* between extracted architectural facts and classification criteria are needed to enable generating search queries from different fact sources such as application model fragments. Typed architectural facts are extracted from different fact sources, e.g., deployment or architectural model fragments that comprise some desired parts of the planned application. Figure 4.7 shows a simplified example of two mappings for architectural facts discussed previously (see Figure 4.6) related to some FaaS function “F_1”, namely (1) support for Java runtime and (2) existence of an object storage trigger, to the corresponding criteria types in the FaaS classification framework (values of the facts are simplified for brevity).

To enable using this extracted information for searching suitable FaaS platforms via related classification criteria entries, the facts have to be mapped to the underlying criteria types and filter types. For instance, if the respective framework criterion type “dev.runtime” is of type `String[]`, the correct query generated from that fact would be checking the “Java” string containment in all classification criteria entries of type “dev.runtime” using the corresponding *comparator_{contains}* that can be defined as follows:

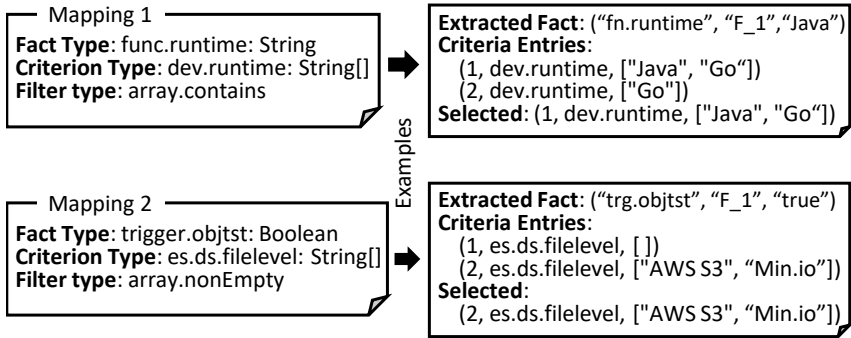


Figure 4.7: Example mappings and their use for querying criteria entries.

$\forall x \in K_\phi, \forall s \in \Sigma^+ : comparator_{contains}(x, s) = True \Leftrightarrow s \in \pi_3(x)$. Therefore, mappings must also prescribe the suitable comparison, e.g., value containment in an array, hence enabling to select only criteria entries compatible with the values of architectural facts. This concept can be defined as follows:

Definition 4.10 (Classification Criterion Mapping)

Let $M_\phi \subset \Sigma^+ \times KT_\phi \times AT_\phi \times FT_\phi$ be the set of all technology classification criteria mappings, then $\mu = (ID, CriterionType, FactType, FilterType) \in M_\phi$ represents a distinct classification criterion mapping that links a specific criterion type with an architectural fact type and prescribes how to compare criteria entries of this type based on the specific filter type. □

Technology search queries can then be generated based on the available mappings that describe how extracted architectural facts are related to criteria types and which filter types should be used, e.g., the comparators for checking the string equality or containment discussed previously. Therefore, for each extracted architectural fact relevant for searching some technology for an application component such as FaaS function, all the

Algorithm 4.3 Generate a technology search query**Input:** Fact source $\alpha\sigma \in AS_\phi$, component name N , mappings M_ϕ **Output:** Technology search query Q

```

1: procedure GENERATEQUERY( $\alpha\sigma, N, M_\phi$ )
2:    $Q \leftarrow \emptyset$  // Initialize the set of sub-queries
3:    $EF \leftarrow e(\alpha\sigma)$  // Extract facts from a fact source
4:   for all  $\alpha \in EF$  do
5:     if  $N = \pi_2(\alpha) \wedge \exists \mu \in M_\phi: \pi_3(\mu) = type_A(\alpha)$  then
6:        $q \leftarrow generatePredicate(\alpha, \mu)$  // Generate sub-query
7:        $Q \leftarrow Q \wedge \{q\}$ 
8:     end if
9:   end for
10:  return  $Q$ 
11: end procedure

```

sub-queries need to be combined. In this work, the strict matching approach [Kol19] is considered to be used for combining the sub-queries, i.e., the conjunction of all sub-queries.

The general flow for generating a search query as a combination of sub-queries for each extracted architectural fact is outlined in Algorithm 4.3: using the suitable mapping, the generic method $generatePredicate(\alpha, \mu)$ produces an implementation-specific sub-query representation, i.e., depending on how the data can be queried from the tooling that supports this metamodel. In the shown algorithm it is assumed that only a single mapping for the same architectural fact type exists. For example, the result of the query generation can be an API query string that comprises sub-queries for all extracted facts for which a mapping exists, e.g., <https://my.api?funcRuntime=Java&objectStorageTrigger=true>. Another example is related to database queries, e.g., an SQL query to return a list of suitable alternatives for a set of extracted facts can be constructed and returned. Since interactive filtering is intended to be an iterative,

GUI-driven exploration of available framework data, use of the generated queries in this context can be seen rather as a complementary part, e.g., the generated query can be used as a starting point for interactive filtering to further refine the technology search. This can be achieved by using the generated API query as a starting point for initializing the filter entries of the generated visual representation of a filter specification, e.g., HTML form elements, and then iteratively refining the search by incorporating additional, not yet utilized filter entries.

4.4 Architecture for Technology Selection Support

To support the use of the presented generic technology classification framework metamodel and the concepts it encompasses, the architecture shown in Figure 4.8 is introduced in the following (the details on its prototypical implementation are further discussed in Section 7.2.2). Firstly, to enable exploring the technology data graphically, the architecture provides a *Classification Framework Explorer* capable of representing available classification frameworks for different technologies and managing the classification framework data, e.g., adding new technologies of specific types and specifying filter configurations or mappings that are then stored in a *Classification Framework Repository*.

To process the technology data exploration requests made via the *Technology Selection Support API*, e.g., using the generated filter specifications, are processed by the *Technology Selection Support Backend* that comprises several subcomponents as depicted in Figure 4.8. The *Framework Representations Builder* subcomponent is responsible for creating the framework representations based on specified criteria groupings, dimensions, and views as well as generating filter representations based on available filter specifications. To retrieve and store the frameworks-related data such as criteria types, mappings, filter specifications, and actual technology classification entries, the *Framework Content Manger* subcomponent is used. Finally, to enable generating search queries based on referenced

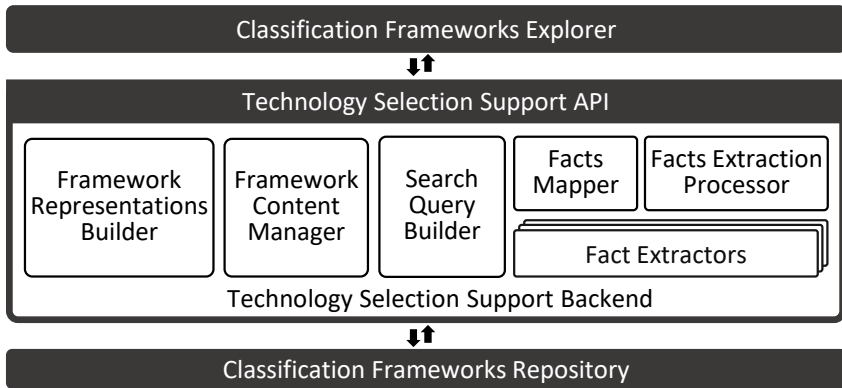


Figure 4.8: Architecture for technology selection support using the generic technology classification framework metamodel.

fact sources, the *Search Query Builder* subcomponent is used, which is responsible for triggering the facts extraction via the *Facts Extraction Processor* subcomponent and corresponding *Fact Extractors* compatible with specific architectural fact types, e.g., capable of processing TOSCA Service Template and extracting facts for specific Node Templates in them. After the extraction is completed, the *Facts Mapper* subcomponent is used to provide the corresponding mapping-related details such as how the criteria data are to be filtered, i.e., how the extracted facts are mapped to the classification criteria types and filter types.

4.5 Chapter Summary

This chapter presented the classification framework for FaaS platforms aiming to facilitate the selection of FaaS platforms based on a variety of different classification criteria. To enable using the underlying framework structure for the classification of other technology types in FaaS-based applications, a generic technology classification framework metamodel was

introduced. Finally, this chapter also presents a conceptual architecture for a selection support system that enables using this metamodel. The concepts presented in this chapter can be used independently of other contributions as a way to explore different technology types such as FaaS platforms interactively. The sets of extracted facts may vary depending on fact source types, e.g., deployment models, coarse-grained application models, source code, or running application instances. Since not all classification criteria can be easily mapped to extracted facts, e.g., documentation-related criteria or support for specific testing and monitoring tooling can be missing in deployment models, the search query generation from a referenced fact source can be used as an initial step and combined with the interactive exploration using generated filter representations. Additionally, one way to enhance the coverage of criteria when generating initial search queries is to combine the information from different fact sources, e.g., deployment models combined with architectural decision records.

ARTIFACT-LEVEL ABSTRACTIONS

THE technical requirements for implementing provider-specific function code and orchestration models often need to be taken into account. For example, function orchestration engines and underlying modeling languages vary significantly, e.g., in supported fault handling and control flow modeling constructs, which makes such technology-specific models non-reusable for other providers. Likewise, the function code developed for specific platforms is often not directly reusable, e.g., functions rely on provider-specific libraries, event schemas, and packaging formats. Based on two peer-reviewed, first-authored publications [YBHL19; YSBL22], this chapter presents Contribution 3 that focuses on artifact-level abstractions for FaaS-based applications and their usage in application models. This includes (i) a method and tooling for provider-agnostic modeling of function orchestrations and their transformation into proprietary formats, and (ii) a FaaSification-based approach for specializing function code for different platforms. Additionally, this chapter discusses how such code abstraction mechanisms can be represented in application models.

5.1 Uniform Modeling of Function Orchestrations

To tackle the differences in provider-specific modeling approaches, this section presents a model-driven method that relies on BPMN 2.0 [OMG11] to enable modeling function orchestrations independently of function orchestration engines and transforming resulting abstract models into technology-specific formats. BPMN is chosen because (i) it is a proven workflow modeling standard specifying a machine-readable language generic enough to express function orchestration modeling semantics [Rue20], and (ii) it defines a semantically-transparent [Moo09] visual notation that can simplify readability of produced models to non-technical staff. The presented concepts are based on the peer-reviewed journal publication [YSBL22].

5.1.1 Overview

The proposed method for uniform modeling of function orchestrations relies on three major parts, namely (i) analysis of the function orchestration domain to identify frequently occurring modeling elements, (ii) design of a BPMN profile to enable modeling function orchestrations independently of function orchestrators, and (iii) design and implementation of a transformation framework that enables generating proprietary function

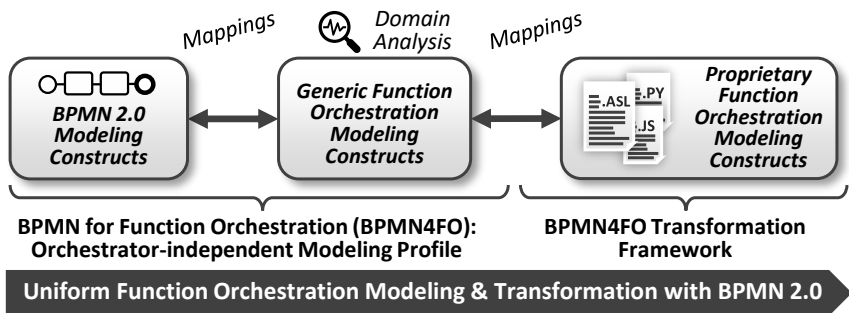


Figure 5.1: Uniform function orchestration modeling and transformation.

orchestration models. Figure 5.1 depicts the overall idea: firstly, Generic Function Orchestration Modeling Constructs (GFOMCs) frequently used in function orchestration models are captured as a list of generic control flow modeling elements. Additionally, the mappings (i) between GFOMCs and BPMN 2.0 modeling constructs are identified to enable the uniform modeling of function orchestrations, whereas the mappings (ii) between GFOMCs and orchestrator-specific formats are captured to enable the transformation of produced uniform models into target formats.

Although the introduced method and tooling can be used for different transformation directions, e.g., mining provider-agnostic representations from provider-specific function orchestration models, only the refinement direction is covered in the following, i.e., the transformation from provider-agnostic to provider-specific function orchestration models. Note, that since BPMN has no notion of a profile [OMG11] and BPMN extensions are in fact used to enrich the capabilities of the language, the term “profile” is used in this chapter analogous to other MDA profiles for BPMN, e.g., Lübke and Lessen introduce a BPMN profile for model-driven testing of service-based processes [LL17].

5.1.2 Generic Function Orchestration Modeling

The baseline list of generic constructs for modeling function orchestrations was derived by analyzing existing pattern catalogs focusing on general workflow modeling [RTVM06] and FaaS function compositions [Tai+20] that document control flow patterns independently of specific technologies such as general-purpose workflow management systems and function orchestrators. Additionally, since traditional workflow modeling languages often enable advanced features not common to function orchestrators such as transaction and compensation scopes [OAS07], the list of GFOMCs was refined using an analysis of prominent function orchestration engines to extract the core modeling capabilities relevant for function orchestration.

Name	Description
<i>Task</i>	expresses an invocation of a FaaS-hosted function.
<i>Sequence</i>	represents an ordered list of FaaS functions that are executed sequentially by a function orchestration engine.
<i>Conditional Branching</i>	splits orchestration into several branches and exactly one is chosen based on the branching condition.
<i>Parallel Branching</i>	diverges the flow into multiple concurrently-executed branches all of which have to be completed before proceeding to next construct in the flow.
<i>Fan-out</i>	invokes a new function instance for processing each data element in a given array-like structure. All invoked functions must be completed before proceeding to the next construct in the flow.
<i>Looping</i>	represents iterative execution, e.g., of one function or a sequence of functions, while an associated looping condition is true similar to the while loop in general-purpose programming languages.
<i>Delay</i>	pauses the execution of the orchestration for a specified time interval.
<i>Sub-Workflow</i>	represents a sub-orchestration executed as a part of an enclosing parent orchestration. For the parent orchestration this construct is similar to a regular Task.
<i>Error Handling</i>	represents fault handling tasks that must be invoked in the occurrence of errors occurred during the execution.

Table 5.1: Name and description of each considered GFOMC.

Particularly, the languages of three major function orchestrators were analyzed to identify a suitable subset of GFOMCs: AWS Step Functions [Ama22b] and Azure Durable Functions [Mic22] were included as function orchestration offerings from currently the two largest cloud providers based on market revenues [Mli21]. Additionally, an installable Apache Openwhisk Composer [Apa22c] was included as it can be used on

premises with Apache Openwhisk and also as a service with IBM Cloud Functions [IBM21]. Table 5.1 describes the considered generic modeling constructs including basic workflow elements representing atomic task calls or sequential execution, and more specific ones such as Fan-out.

Formally, a function orchestration metamodel supporting the identified GFOMCs can be seen as a specialization of the Process Model Graph (PM-Graph) model by Leymann and Roller [LR00]. Prior to introducing the BPMN for Function Orchestration (BPMN4FO) profile, the necessary concepts are discussed based on the PM-Graph formalism: Let the set of all function orchestration models be denoted as \mathfrak{F} . Then, $f \in \mathfrak{F}$ as a specific function orchestration model is described as a specialization of the PM-Graph metamodel [LR00] called *FO-Graph*. Let \mathcal{T}_A be the set of all uniquely identifiable orchestration element types, such as Task, Parallel Gate, Exclusive Gate, or Fan-Out. Then, all *typed orchestration elements* in a FO-Graph f are contained in the node set N , whereas the edge set E describes all possible partial orders of orchestration elements in f . FO-Graphs are associated with various data, e.g., inputs and outputs for orchestration elements [LR00].

Let V denote the set of all data for a function orchestration model f with $v \in V$ being a data element of specific structure that also has a name for selective access. Based on the Data Element definition in PM-Graphs [LR00], data elements in FO-Graphs enable flexibly describing data in function orchestration models as primitive or compound objects built with various atomic structures:

Definition 5.1 (Data Element)

Let M be a set of names and S be a set of structures. Then,

1. $m \in M \wedge s \in S \Rightarrow \langle m; s \rangle \in V$, i.e., a valid data element is a pair consisting of a name and an already defined structure.
2. $\zeta_1, \dots, \zeta_k \in S (k \in \mathbb{N})$ where ζ_1, \dots, ζ_k represent atomic structures, e.g., INTEGER or STRING.
3. $d \in V \Rightarrow d(k) \in S (k \in \mathbb{N})$ where $d(k)$ denotes an array of

k elements over d ; that means an array of an already defined data element is a new valid structure.

4. $\bar{V} \subseteq V \Rightarrow \times \bar{V} \in S$ where $\times \bar{V}$ denotes a tuple over V , i.e., a tuple over an already defined data elements is a new valid structure, and $|V| < \infty$. □

To complement the array or tuple constructors for data elements, the definition can also be extended with additional data structures such as sets [LR00]. Having discussed data elements in function orchestrations, the notion of *domains* prescribes how valid data elements are created based on the Domain definition in PM-Graphs [LR00]:

Definition 5.2 (Domain of a Data Element)

Each data element $v \in V$ has an associated domain $\text{DOM}(v)$ that represents all of its corresponding well-formed values:

1. $\text{DOM}(\langle m; \zeta_i \rangle) := \text{domain}(\zeta_i)$ meaning that the domain of an atomic element is the set of all valid values of the associated atomic structure, e.g., $\text{domain}(\text{FLOAT})$ are all floating point numbers.

2. $\text{DOM}(\langle m'; \langle m; s \rangle (k) \rangle) := \text{DOM}(\langle m; s \rangle)(k)$ meaning that the domain of a data element structured as a k -element array is the set of all arrays with k values from the domain of the data element being the base for the array.

3. $\text{DOM}(\langle m'; \times \langle m_1; s_1 \rangle, \dots, \langle m_r; s_r \rangle \rangle) := \prod_{1 \leq i \leq r} \text{DOM}(\langle m_i; s_i \rangle)$ meaning that the domain of a tuple-structured data element is the Cartesian product of the domain of the components of the tuple constructor. □

Function orchestration models and their orchestration elements representing units of work rely on input and output data. Moreover, the order in which orchestration elements are executed in a function orchestration instance is determined by a set of conditions C , e.g., business rules or predicates, that are formulated in the context of the data consumed and produced by orchestration elements. Please, note that only input data are

relevant for evaluating such conditions since the output of a condition is always a truth value. To facilitate the reasoning about the input and output data, both are conveniently grouped in so-called *data containers* which represent collections of data elements as discussed in the following.

Let \mathcal{H}_f be the set of all function orchestration models and orchestration elements they encompass, i.e., $\mathcal{H}_f = \mathfrak{F} \cup N$. Similar to PM-Graphs [LR00], data containers in FO-Graphs are valuable abstractions for discussing orchestration elements and control flow connectors and can be seen as data operators for function orchestration instances:

Definition 5.3 (Data Container)

The map i_f assigns to each orchestration element, function orchestration model, and condition its input container: $i_f: \mathcal{H}_f \cup C \rightarrow \wp(V)$, meaning that: $\forall h \in \mathcal{H}_f \cup C: i_f(h) \subseteq V$ with $|i_f(h)| < \infty$.

Likewise, the map o_f assigns to each orchestration element and process model its output container: $o_f: \mathcal{H}_f \rightarrow \wp(V)$. This means $\forall h \in \mathcal{H}_f: o_f(h) \subseteq V$ with $|o_f(h)| < \infty$. □

Note, that the modeling of data flows in FO-Graphs is omitted in this work due to the capabilities of the analyzed orchestrator-specific modeling languages in which data flows are not represented using dedicated constructs. This is similar to BPEL [OAS07], in which data flows cannot be modeled explicitly using dedicated constructs, but can be derived via variables shared between activities [KKL08]. While FO-Graphs represent the control flow-related semantics, the definitions can be extended to support explicit data flows via data objects and data connectors as in PM-Graphs.

Data containers, however, become relevant in the context of transforming generic orchestration models into proprietary formats: depending on the underlying orchestrator-specific language data containers might need to be represented explicitly. For example, in function orchestrators that rely on general-purpose programming languages, the output of orchestration elements such as Tasks (see Table 5.1) needs to be collected, e.g., in a local

variable, to be passed to the subsequent orchestration element. Hence, orchestration elements in a function orchestration model can be perceived as abstract operators defining their inputs and outputs. Further, implementations of an orchestration element, e.g., Java code that implements a Task, enable producing instances of output data using consumed instances of input data. Based on the Activity definition in PM-Graphs [LR00], orchestration elements are defined as in the following.

Definition 5.4 (Orchestration Element)

Let $\mathcal{T}_A \subseteq \Sigma^+$ be the set of all orchestration element types, then $t \in \mathcal{T}_A$ identifies a distinct type of orchestration element. Let N be the set of all orchestration elements of a function orchestration model $\mathfrak{f} \in \mathfrak{F}$ and each member in \mathfrak{F} is addressed by its name A . An orchestration element is then described as an operator $A: i_{\mathfrak{f}}(A) \rightarrow o_{\mathfrak{f}}(A)$. In addition, let $\tau_{\mathfrak{f}}: N \rightarrow \mathcal{T}_A$ be the map that associates with each orchestration element A its type $\tau_{\mathfrak{f}}(A)$, and let $isGate_{\mathfrak{f}}: N \rightarrow \{true, false\}$ be the map that identifies whether an orchestration element is derived from type “Gate”, e.g., “Parallel Gate”. \square

Orchestration elements represent specific units of work, e.g., an element of type Task (see Table 5.1), and their *implementations* perform certain work by consuming an instance of the associated input data container and producing the corresponding output container instance. User-provided implementations of the same functionality in different programming languages exemplifies the difference between orchestration elements and their implementations. Another example is fork and join elements responsible for divergence and convergence of control flow, which are represented by the Conditional Branching GFOMC as shown in Table 5.1 – these orchestration elements are instead built-in capabilities of function orchestrators. Based on the Activity Implementation definition in PM-Graphs [LR00], orchestration element implementations are defined as:

Definition 5.5 (Orchestration Element Implementation)

Let \mathcal{E} be the set of all possible implementations of all orchestration elements; that means a member of \mathcal{E} can be a function or a function orchestration model. The map $\Psi: N \rightarrow \mathcal{E}$ associates with each orchestration element A its implementation $\Psi(A)$. An orchestration element implementation itself is perceived as a map:

$$\Psi(A): \prod_{v \in i_f(A)} \text{DOM}(v) \rightarrow \prod_{v \in o_f(A)} \text{DOM}(v). \quad \square$$

While orchestration elements and their implementations represent actionable units of work, the order of their execution is defined using *conditions* and *control flow sequences* that connect different orchestration elements. Orchestration elements can have *exit conditions* that signify the semantic completion of performed work; this concept is defined based on the definition of Exit Conditions in PM-Graphs [LR00]:

Definition 5.6 (Exit Condition)

The map $\varepsilon: N \rightarrow C$ assigns to each function orchestration element a predicate called exit condition. The exit condition $\varepsilon(A)$ of orchestration element A has its input container $i_f(\varepsilon(A)) \subseteq V$ such that an exit condition is considered as a Boolean function:

$$\varepsilon(A): \prod_{v \in i_f(\varepsilon(A))} \text{DOM}(v) \rightarrow \{0, 1\}. \quad \square$$

In Table 5.1, only the LOOPING construct requires an explicit exit condition indicating completion of an iteration, i.e., when an orchestration element implementation $\Psi(A)$ completes, its exit condition $\varepsilon(A)$ is evaluated with respect to the instance of its associated input container $i_f(\varepsilon(A))$. Hence, the LOOPING orchestration element is considered to be completed iff $\varepsilon(A)(i_f(\varepsilon(A))) = 1$. In the context of model-to-model transformation, such conditions need to be transformed into target formats too. To accommodate for such format discrepancies, a uniform conditions format is introduced as a part of the BPMN4FO Transformation Framework described in Section 5.1.5.

Orchestration elements in FO-Graphs are interconnected using control flow connectors, i.e., for an orchestration element A there could be one or more directed edge connecting it with successor elements A_1, \dots, A_x and the pairs $(A, A_1), \dots, (A, A_x)$ represent potential paths to the next orchestration element. The actual flow, however, is defined based on *transition conditions*, i.e., predicates that determine the actual transitioning among potential paths $(A, A_1), \dots, (A, A_x)$. Therefore, control flow connectors in conjunction with transition conditions define whether the corresponding transition from orchestration elements A to B should happen based on the corresponding instance input data, which can be defined based on the Control Flow Connector definition in PM-Graphs [LR00].

Definition 5.7 (Control Flow Connector)

Let set $E \subseteq N \times N \times C$ contain all control flow connectors of a function orchestration model $\mathfrak{f} \in \mathfrak{F}$. For a control flow connector $(A, B, p) \in E$, the predicate $p \in C$ is a transition condition represented as a Boolean function in its input container $i_{\mathfrak{f}}(p) \subseteq V$: $p: \prod_{v \in i_{\mathfrak{f}}(p)} \text{DOM}(v) \rightarrow \{0, 1\}$. □

In FO-Graphs, the transition condition needs to be explicitly modeled for the Conditional Branching GFOMC described in Table 5.1. Therefore, the transition to the next orchestration element for the diverged control flow is decided based on the input data container instance and the control is passed to the branch for which $p(i_{\mathfrak{f}}(p)) = 1$. Similar to PM-Graphs [LR00], the FO-Graph representing a generic function orchestration can be defined based on the Definitions 5.1 to 5.7 as follows:

Definition 5.8 (Function Orchestration Graph)

A tuple $F = (V, i_{\mathfrak{f}}, o_{\mathfrak{f}}, N, \tau_{\mathfrak{f}}, isGate_{\mathfrak{f}}, \Psi, E, C, \varepsilon)$ is called a function orchestration model graph, or FO-Graph for short, representing a function orchestration model $\mathfrak{f} \in \mathfrak{F}$. □

To enable model-to-model transformations, this contribution focuses on defining control graphs that specify how the execution of functions should be ordered, whereas the data flow modeling with data connectors and execution semantics with concepts such as join conditions [LR00] are omitted in the definition of FO-Graphs. These aspects, however, can be further added based on the definition of PM-Graphs [LR00].

5.1.3 BPMN4FO Abstract Syntax and Modeling Requirements

The BPMN4FO profile is a realization of the FO-Graph and GFOMCs it encompasses using BPMN 2.0 to enable modeling function orchestrations with a well-known standard instead of orchestrator-specific modeling languages. Similar to existing domain-specific BPMN profiles [BSBE14; LL17; WS07], the metamodel for BPMN4FO is defined using UML. Figure 5.2 depicts the simplified metamodel in which the added properties are not shown and only the major BPMN elements used for this extension are colored in gray: BPMN PROCESS represents a distinct function orchestration model that encompasses Orchestration Elements (see Definition 5.4) interconnected by means of Control Flow Connectors (see Definition 5.7). The former are derived from BPMN FLOW ELEMENT by defining a subset of allowed flow constructs, whereas the latter inherit from BPMN SEQUENCE FLOWS. As discussed previously, CONDITION EXPRESSIONS can be specified for Conditional Branching and Looping GFOMCs to represent transition and exit conditions, respectively. For convenience, in BPMN4FO, the transition condition for conditional branching is specified on the level of the incoming EXCLUSIVE GATEWAY in the property branches to group together conditions for each outgoing control flow connector – to uniform condition format is presented in Section 5.1.5. Additionally, to represent Looping and Fan-out GFOMCs, the LOOPING CHARACTERISTICS need to be specified for desired kind of ACTIVITIES as corresponding markers. While data containers are not explicitly represented in BPMN4FO, Orchestration

Elements include optional `inputSchema` and `outputSchema` properties that can specify input and output data container formats and used for validating the compatibility of orchestration elements at transformation time.

Unlike BPMN 2.0 that also defines operational semantics for its elements [CT12], BPMN4FO only aims to enable model-to-model transformations and not the execution using BPMN engines. Based on the existing classification framework for BPMN profiles [BE14], the BPMN4FO profile is a specialization of BPMN, i.e., a subset of BPMN elements is used with some constraints and additional properties. BPMN4FO focuses on both describing the domain of function orchestration modeling and enabling the execution of produced models using model-to-model transformation.

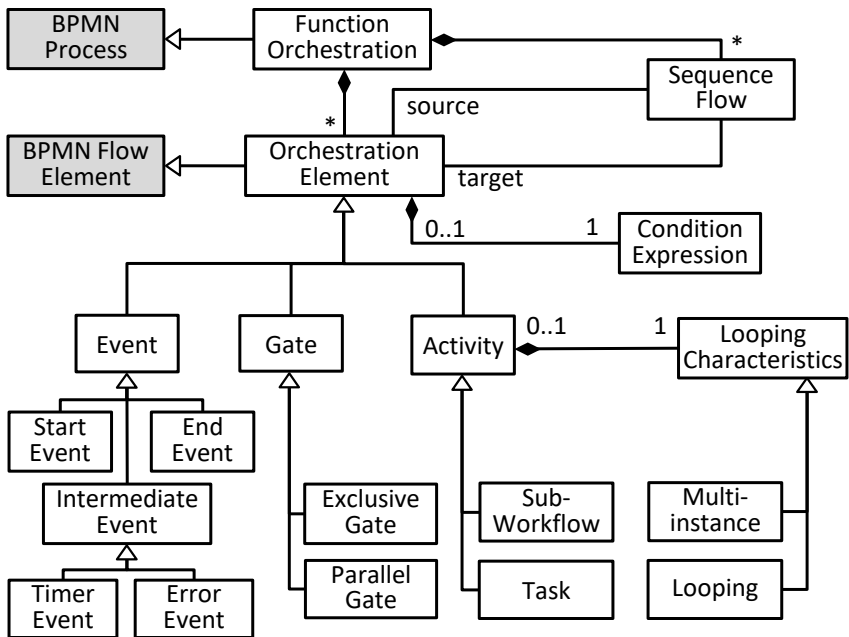


Figure 5.2: A simplified metamodel of the BPMN4FO profile.

Further, BPMN4FO does not introduce changes to the concrete syntax of BPMN, i.e., standard BPMN elements were specialized for this domain without introducing new notations as will be shown in Section 5.1.4.

To improve comprehensibility of BPMN4FO models, several restrictions concerning different elements were introduced. Firstly, to avoid arbitrary use of control flow connectors when connecting function orchestration elements, the set E in a function orchestration model $f \in \mathfrak{F}$ must be *unified* [LR00] meaning that two function orchestration elements can be connected with at most one control flow connector. While the standard BPMN allows modeling uncontrolled flow [OMG11], i.e., multiple incoming SEQUENCE FLOWS for ACTIVITIES, the resulting uncontrolled flow of tokens may result in instantiation of multiple separate Activity instances, which, in turn hinders the transformation semantics. BPMN also allows modeling parallel branching implicitly via multiple outgoing SEQUENCE FLOWS for ACTIVITIES [OMG11], which could hinder its readability. To avoid these ambiguities when transforming BPMN4FO models to target formats, the cardinality of SEQUENCE FLOWS for all orchestration elements except GATES is restricted to exactly one.

AS GATES in BPMN4FO (see Figure 5.2) represent divergence and convergence of control flow, effectively, only the orchestration elements that (i) diverge the flow into more than one branch (called Forks in PM-Graphs [LR00]) and (ii) converge several branches into one (called Joins in PM-Graphs [LR00]) can have a cardinality of sequence flows more than one. Aside from the fact that a valid function orchestration model must have a converging Gate for each corresponding diverging Gate (see [EB14], for example), the number of control flow connectors is restricted in BPMN4FO. Similar to the definition in Section 4.3, let the projection operator π_j , $1 \leq j \leq n$ be the mapping $\pi_j : X_1 \times \dots \times X_n \rightarrow X_j$, $(x_1, \dots, x_n) \mapsto x_j$ that maps a tuple to its j -th element. These restrictions are formulated based on the PM-Graph definitions [LR00] as follows:

Definition 5.9 (Restrictions on Control Flow Connectors)

Let $E^{\leftarrow}(A) = \{e \in E : \pi_2(e) = A\}$ and $E^{\rightarrow}(A) = \{e \in E : \pi_1(e) = A\}$ denote sets of control flow connectors incoming to and outgoing from the orchestration element A , respectively. The following restrictions are imposed for the fork and join elements:

- an orchestration element A_1 is a valid forking element iff $isGate_{\uparrow}(A_1) = true \wedge |E^{\leftarrow}(A_1)| = 1 \wedge |E^{\rightarrow}(A_1)| > 1$,
- an orchestration element A_2 is a valid joining element iff $isGate_{\uparrow}(A_2) = true \wedge |E^{\leftarrow}(A_2)| > 1 \wedge |E^{\rightarrow}(A_2)| = 1$, and
- $\tau_{\uparrow}(A_1) = \tau_{\uparrow}(A_2)$.

For any other orchestration element A , these restrictions are imposed:
 $isGate_{\uparrow}(A) = false \wedge 0 \leq |E^{\leftarrow}(A)| \leq 1 \wedge 0 \leq |E^{\rightarrow}(A)| \leq 1$
 $\wedge |E^{\leftarrow}(A)| + |E^{\rightarrow}(A)| \geq 1. \quad \square$

To generate orchestrator-specific models unambiguously, restrictions are also imposed on the quantity of start and end function orchestration elements. The former is modeled using the BPMN START EVENT and the latter is modeled by BPMN END EVENT (see Figure 5.2). Based on PM-Graphs [LR00], this restriction is described as follows:

Definition 5.10 (Restrictions on Start and End of Orchestration)

Let $A_{start} = \{A : |E^{\leftarrow}(A)| = 0 \wedge |E^{\rightarrow}(A)| > 0\}$ and $A_{end} = \{A : |E^{\leftarrow}(A)| = 1 \wedge |E^{\rightarrow}(A)| = 0\}$ denote the sets comprising start and end orchestration elements. The function orchestration model is considered valid only if $|A_{start}| = 1 \wedge |A_{end}| = 1. \quad \square$

An exception is made for fault handling in which END EVENTS can optionally mark the termination of fault handling branches, therefore elements that close such paths are ignored. Although loops can be modeled by combining BPMN TASKS and BPMN CONDITIONAL GATEWAYS, in BPMN4FO, modeling of loops is restricted to the usage of BPMN LOOP MARKERS

meaning that the actual iteration is either modeled as a Sub-Workflow or is a part of the underlying orchestration element implementation. In PM-Graph terms, this property is called *acyclicity* [LR00], i.e., no path in the function orchestration can contain a cycle. For brevity, this definition is omitted since more details on acyclicity can be found in the original PM-Graph definition [LR00].

5.1.4 BPMN4FO Concrete Syntax

Having discussed the abstract syntax of BPMN4FO and underlying generic function orchestration modeling concepts, this section presents the concrete syntax of the profile by example. Firstly, as mentioned previously, the BPMN PROCESS is employed as a container construct representing the function orchestration itself. The BPMN START EVENT marks the beginning of a function orchestration and the BPMN END EVENT denotes its end. Other aspects of function orchestrations are modeled as follows:

Modeling Tasks. BPMN TASK represents an invocation of a FaaS function. Since functions in orchestration models must have a name, a mandatory *name* attribute is used in a BPMN TASK. As discussed previously, exactly one incoming and one outgoing BPMN SEQUENCE FLOW is expected for each modeled TASK.

Modeling Sequential Control Flow. BPMN SEQUENCE FLOWS enable connecting various elements such as Tasks together. Hence, the sequential invocation of FaaS functions is modeled by combining BPMN TASKS using BPMN SEQUENCE FLOWS (see Figure 5.3). In such sequences, the outputs of preceding functions become inputs for the next functions.

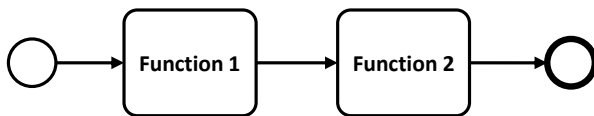


Figure 5.3: Example Sequence modeled in BPMN.

Modeling Conditional Branching. Conditional flows in function orchestrations are modeled using BPMN EXCLUSIVE GATEWAYS which enable executing only one branch based on associated conditions (Figure 5.4). Please, note that to specify conditions in a technology-agnostic fashion, a uniform syntax for modeling conditions is needed – in Section 5.1.5 such uniform syntax for expressing conditions is defined as a part of the transformation framework supporting this method.

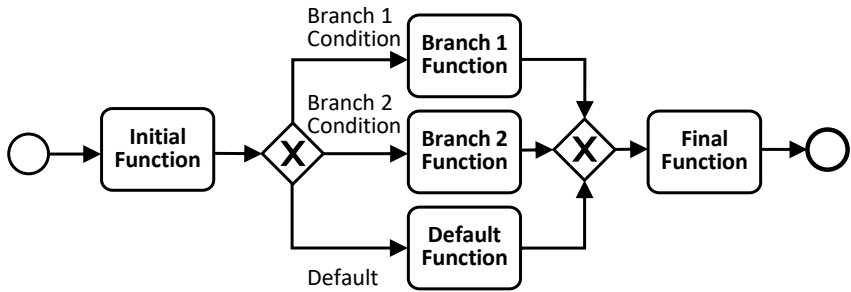


Figure 5.4: Example Conditional Branching modeled in BPMN.

Modeling Parallel Branching. Parallel execution of several FaaS functions is modeled using BPMN PARALLEL GATEWAYS, which enable representing multiple branches that must be run in parallel (see Figure 5.5). Each

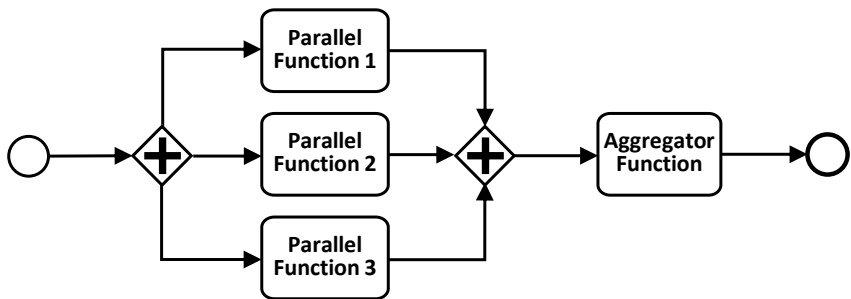


Figure 5.5: Example Parallel Branching modeled in BPMN.

parallel branch executes a separate copy of the input data received at the opening BPMN PARALLEL GATEWAY. After all branches are completed, the results are collected at the closing PARALLEL GATEWAY and sent to the next task in the control flow.

Modeling Fan-outs. The Fan-out (see Table 5.1) can be modeled using BPMN MULTI-INSTANCE MARKERS that enable specifying parallel execution of several instances of tasks marked with them (see Figure 5.6). This element can be used for array-like inputs: a new task instance is executed for each element in the input and the resulting outputs are returned as an array after all instances finish processing.

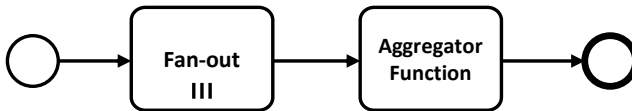


Figure 5.6: Example Fan-out modeled in BPMN.

Modeling Looping. Modeling of an iterative execution of a task is achieved using BPMN LOOP MARKERS, which enable marking tasks that must be repeated until a specified condition is satisfied (see Figure 5.7). In this



Figure 5.7: Example Looping modeled in BPMN.

element, the original input is used in the first iteration while each next execution receives the output of the preceding iteration as an input. After the condition is fulfilled, the iteration completes and the last output is passed to the next task in the modeled orchestration. Similar to Conditional Branching, a condition must be specified in an additional `loopCondition` attribute in a standardized manner to enable unambiguous transformation

into target function orchestrator formats. Such condition format is discussed in Section 5.1.5 in the context of the transformation framework supporting this method.

Modeling Delays. Delay in the execution of modeled orchestrations for a specific time interval is represented using BPMN `TIMER INTERMEDIATE CATCHING EVENTS` with one incoming and one outgoing BPMN `SEQUENCE FLOW` (see Figure 5.8). Additionally, the `milliseconds` attribute in the BPMN timer event must specify the desired duration of the delay after which the execution proceeds.

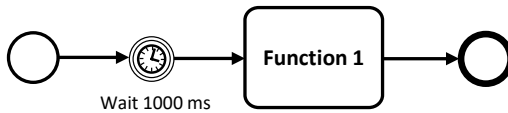


Figure 5.8: Example Delay modeled in BPMN.

Modeling Sub-Workflows. Modeling of sub-workflows within a parent function orchestration model is achieved using BPMN `SUB-PROCESSES`, which enable representing self-contained orchestration models with separate start and end events (see Figure 5.9). Similar to BPMN `TASK`, the

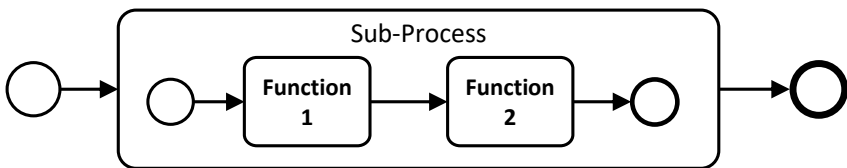


Figure 5.9: Example Sub-Workflow modeled in BPMN.

BPMN `SUB-PROCESS` must have one incoming and one outgoing BPMN `SEQUENCE FLOW` and a name attribute with the sub-orchestration name. Additionally, BPMN `MULTI-INSTANCE MARKER` and BPMN `LOOP MARKER` can be used to represent Fan-out and Looping, respectively.

Modeling Error Handling. To model fault handling flows in function orchestrations, BPMN ERROR BOUNDARY EVENTS are used to mark the error handler for BPMN TASKS or SUB-PROCESSES (Figure 5.10). The BPMN ERROR BOUNDARY EVENT has one outgoing BPMN SEQUENCE FLOW, which is connected to the branch executed when errors occur.

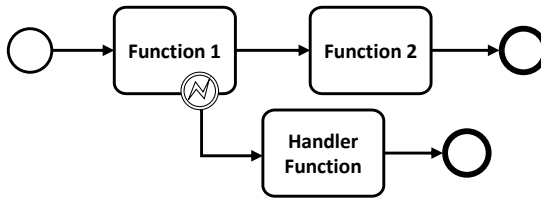


Figure 5.10: Example Error Handling modeled in BPMN.

5.1.5 BPMN4FO Transformation Framework

The BPMN4FO profile is designed based on the analysis of modeling languages and technical requirements in three prominent (at the time of writing) function orchestrators and enables modeling function orchestrations in an orchestrator-agnostic manner. This subsection presents the BPMN4FO Transformation Framework that enables generating technology-specific function orchestration models from BPMN4FO models. Orchestrator-specific realizations of GFOMCs vary significantly due to the different modeling approaches employed by function orchestrators. While more details on the analyzed orchestrators can be found in the original publications [YSB+21a; YSBL22], this subsection only summarizes these technologies in the context of BPMN4FO model transformation into technology-specific formats.

AWS Step Functions enables modeling function orchestrations as finite state machines specified using the Amazon State Language (ASL) [Ama22d], in which typed states representing different control flow constructs, inputs and outputs are modeled in JSON. In contrast, Azure Durable Functions

and Apache Openwhisk Composer rely on general-purpose programming languages for modeling function orchestrations. The former supports mod-

GFOMC Name	AWS Step Functions	Azure Durable Functions	Openwhisk Composer
<i>Task</i>	<i>ASL Task</i>	<i>Activity Function</i>	<i>composer.action</i>
<i>Sequence</i>	<i>Next</i> property	Subsequent synchronous calls	<i>composer.sequence</i>
<i>Conditional Branching</i>	<i>ASL Choice</i>	<i>if-else</i>	<i>composer.if</i>
<i>Parallel Branching</i>	<i>ASL Parallel</i>	Asynchronous invocation of sub-workflows	<i>composer.parallel</i>
<i>Fan-out</i>	<i>ASL Map</i>	Repeated and asynchronous invocation of a sub-workflow	<i>composer.map</i>
<i>Looping</i>	Loops with <i>ASL Choice</i> conditions ¹	<i>while</i>	<i>composer.while</i>
<i>Delay</i>	<i>ASL Wait</i>	<i>Durable Timer</i>	Custom delay functions ¹
<i>Sub-Workflow</i>	Call to a AWS Step Function workflow	Durable Function workflow call	<i>composer.action</i>
<i>Error Handling</i>	<i>Catch</i> property	<i>try-catch</i>	<i>composer.try</i>

¹ Requires custom implementation as no native construct is present.

Table 5.2: Mappings between GFOMCs and analyzed Proprietary Function Orchestration Modeling Constructs (PFOMCs).

eling orchestrations in C#, JavaScript, or Python [Mic22], whereas the latter relies on JavaScript [Apa22c]. With this modeling style, *orchestrating functions* represent the desired control flow and are responsible for executing orchestrated functions in that order. Typically, language-specific libraries need to be used for representing control flow constructs, e.g., combinators in Apache Openwhisk Composer. Despite providing native support for the majority of GFOMCs, the analyzed function orchestrators do not always offer exact matches, e.g., modeling of *Looping* for AWS Step Functions or *Delay* for Openwhisk Composer. Table 5.2 lists the identified mappings between generic and proprietary, orchestrator-specific modeling constructs. In cases when no exact mapping is possible, workarounds can be used, e.g., custom delay functions for Apache Openwhisk Composer instead of native constructs.

A system architecture supporting the use of BPMN4FO profile and the transformation of resulting orchestrator-agnostic models into proprietary function orchestration formats using the identified mappings is conceptualized in Figure 5.11. Firstly, to enable the use of the BPMN4FO profile, the BPMN4FO Transformation Framework in Figure 5.11 comprises a *Graphical BPMN Modeler* that provides a GUI for modeling with BPMN4FO,

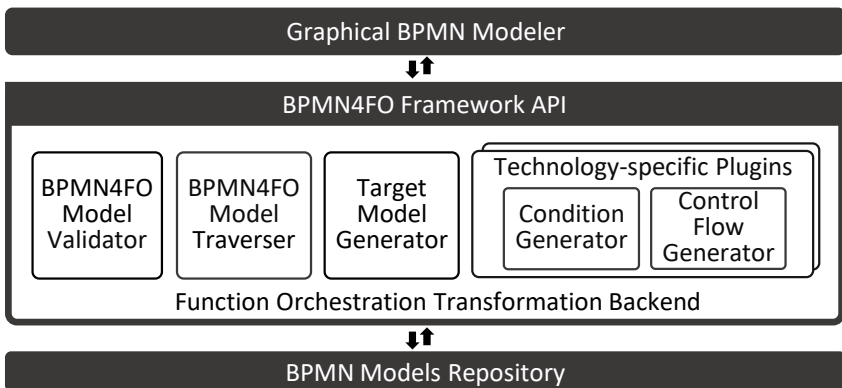


Figure 5.11: Architecture of the BPMN4FO Transformation Framework.

i.e., the GUI must support the specialized subset of BPMN constructs allowed by the profile and enable triggering the transformation of created models into supported target formats if they comply with the modeling requirements of the profile. Since BPMN4FO does not introduce any custom syntax and relies on standard BPMN elements, the core functionalities realized within the Graphical BPMN Modeler are related to the introduced specializations, e.g., custom element properties and specification of uniform conditions. Further, to enable generating target proprietary formats, models are checked for compliance with the profile modeling requirements discussed in Section 5.1.3 and the Graphical BPMN Modeler conveys the errors to modelers. Additionally, the framework relies on a *BPMN Models Repository*, such as a file-based repository or a relational database, for storing created models.

The requests to generate proprietary formats are issued using the BPMN Graphical Modeler and are processed by the *Function Orchestration Transformation Backend* via the *BPMN4FO Framework API*. The transformation is then performed using the following components in the *Function Orchestration Transformation Backend* shown in Figure 5.11: Firstly, the *BPMN4FO Model Validator* verifies the provided model for compliance with the BPMN4FO modeling requirements, e.g., allowed numbers of input and output BPMN SEQUENCE FLOWS for activities. Moreover, the *BPMN4FO Model Validator* is invoked on each change in the model by the *Graphical BPMN Modeler*, which enables warning modelers when the to-be-transformed BPMN4FO model is no longer compliant with the profile. Next, the *BPMN4FO Model Traverser* component traverses the valid BPMN4FO model and constructs an internal representation of the function orchestration graph. The *Target Model Generator* component is responsible for the generation of target function orchestration formats using respective *Technology-specific Plugins*, e.g., ASL models for AWS Step Functions. Each plugin comprises the *Control Flow Generator* sub-component that transforms the orchestration elements into the corresponding target format.

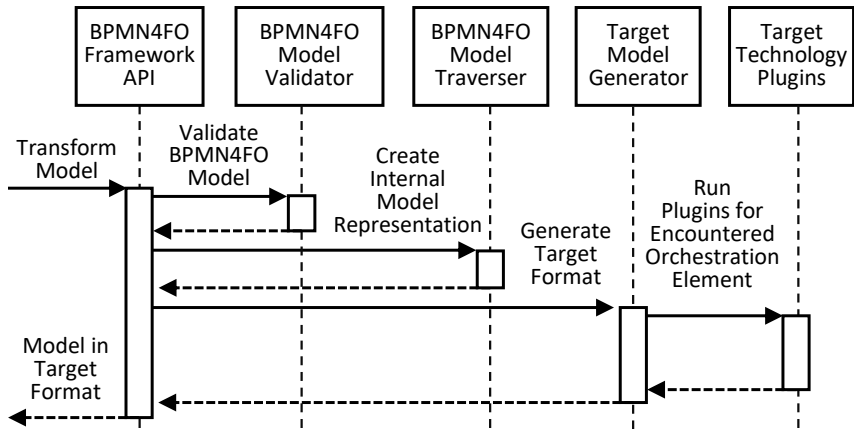


Figure 5.12: The interplay of components when transforming BPMN4FO models using the BPMN4FO Transformation Framework.

Further, the *Condition Generator* is called by the *Target Model Generator* when conditional control flow constructs are encountered to generate technology-specific conditions, e.g., for ASL models.

Figure 5.12 shows the described interplay of components when a transformation request is issued to generate a specific function orchestration format for a given BPMN4FO model. After the provided BPMN4FO model is validated and traversed, the generation starts and for every encountered orchestration element, the corresponding orchestrator-specific plugins are triggered, e.g., when a condition needs to be transformed the respective Condition Generator is invoked to produce the condition in the target format. The actual orchestration elements are transformed based on the identified mappings to the language-specific constructs shown in Table 5.2. The orchestrator-specific transformation logic is provided in a pluggable fashion, i.e., support for more function orchestrators can be added via additional technology-specific plugins realizing the same structure.

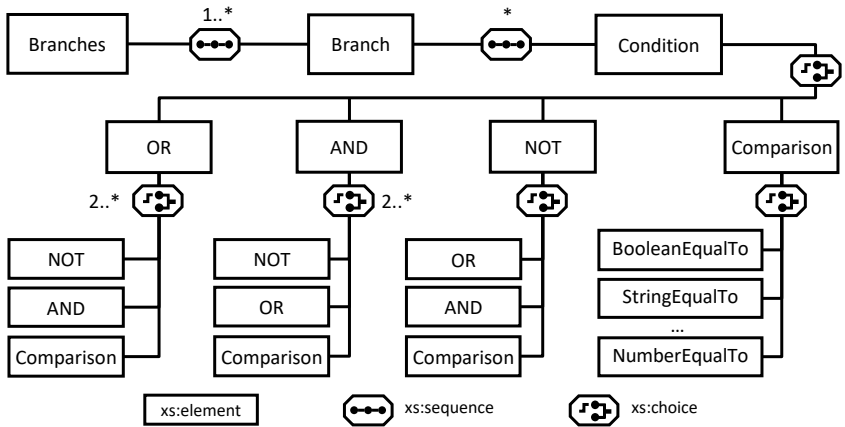


Figure 5.13: A simplified XSD diagram for the uniform condition format.

As discussed previously, to specify conditions in a uniform manner, an XML-based format was introduced to represent exit and transition conditions uniformly in BPMN4FO, hence enabling the use of orchestrator-specific condition generators of the framework. Figure 5.13 shows a simplified diagram visualizing the XML Schema Definition (XSD) of the introduced condition format. Conditional branching can be described using the `branches` element that comprises one or more `branch` elements, each comprising `condition` elements that describe how variables referenced in the condition should be compared, e.g., equality for numbers or strings. Further, conditions for Looping can be directly expressed using `condition` elements.

Listing 5.1 shows an example condition specified for a branch called “HappyPath”, which evaluates equality of two variables. More detailed examples about conditions is provided in Chapter 7 in which the open source implementation of the BPMN4FO Transformation Framework is discussed in the context of an integrated architecture supporting the presented concepts.

```
... <branch name="HappyPath">
    <condition>
```

```
<and>
  <comparison variable="num">
    <NumberEqualTo>42</NumberEqualTo>
  </comparison>
  <not>
    <comparison variable="s">
      <StringEqualTo>FML</StringEqualTo>
    </comparison>
  </not>
</and>
</condition>
</branch> ...
```

Listing 5.1: Example uniform condition specified for branch “HappyPath”.

5.1.6 Discussion and Limitations of the Approach

In the context of this work, the emphasis was put on the uniform transformations to enable transitioning between different levels of abstractions for modeled FaaS-based applications. This decision inherently requires supporting only the subset of features shared by all analyzed function orchestrators, which required imposing certain modeling restrictions discussed in Section 5.1.3. A disadvantage of this design decision is that unique capabilities of certain orchestrators cannot be used due to missing modeling constructs in BPMN4FO and orchestrator-specific mappings for orchestrators without such capabilities. Clearly, uniform modeling is not necessary when only one specific function orchestrator format is needed instead. To make the transformation more flexible, the BPMN4FO meta-model and the BPMN4FO Transformation Framework can additionally be extended with orchestrator-specific elements and mappings to desired formats. In addition, the transformation framework can be extended to support user configurations specifying which orchestrator-specific features to keep during the transformation, e.g., via GUI or external descriptors. Generated models can also be refined during a post-processing step to add extra features manually.

As discussed in Section 2.2.3, compared to approaches introducing function orchestration modeling languages relying on custom orchestrators, BPMN4FO models aim at utilizing existing function orchestrators without locking into one specific custom solution. Moreover, unlike approaches without visual notation, BPMN4FO facilitates comprehensibility of the produced function orchestration models to non-technical staff, thus, simplifying the collaboration between business users and IT users. Furthermore, the BPMN4FO profile and the underlying transformation framework can be easily combined with the aforementioned approaches by extending the metamodel with new constructs and implementing new language plugins, hence adding the support for transforming models into new formats.

The modeling restrictions introduced in BPMN4FO due to technical requirements of the analyzed function orchestrators affect the expressiveness of produced models. For example, since BPMN TASKS in BPMN4FO represent function calls, the modelers are not required to differentiate between different types of tasks, e.g., *Service Tasks* in BPMN represent service calls, whereas *Script Tasks* represent tasks executed by a business process engine. Another related example is handling of different fault types: the modeled fault handling path in BPMN4FO represents a call to a handler function without differentiating between fault types. Advanced features such as cancellation or compensation of tasks also cannot be expressed in BPMN4FO models. To improve the expressiveness of the profile, additional modeling capabilities and transformation logic can be added, e.g., to enable representing compensation functions that must be triggered whenever there is a need to revert the effects certain successfully completed tasks in the model.

Another limitation of this approach is related to changes in function orchestrations, e.g., the addition or modification of language and orchestrator features. To accommodate for such scenarios, the corresponding mappings might require modifications to reflect the changes in generated target formats. However, even for such scenarios there are benefits of using BPMN4FO. For example, existing BPMN4FO models can be reused to

generate updated versions of the target model formats. Further, only the orchestrator-specific plugins may require changes in the BPMN4FO Transformation Framework without affecting other parts of the framework.

While the introduced approach facilitates modeling function orchestrations independently of target technology, another issue is related to function implementations as they often rely on platform-specific libraries and packaging formats. As discussed in Section 2.2.4, this challenge can be addressed using FaaSification approaches that facilitate porting function implementations between FaaS platforms. In addition, general-purpose approaches such as Any2API [WBL15] could be employed to automatically wrap business logic for chosen FaaS platforms. The next subsection presents a source code extraction and packaging method that aims to abstract function implementations and is complementary to the BPMN4FO.

5.2 Serverless Parachutes for Code Abstraction

As an example abstraction technique applicable to function code, this section presents a FaaSification-based method that facilitates automatic extraction and packaging of general-purpose function code for different FaaS platforms. The presented concepts are based on the conference publication [YBHL19] that introduces the *serverless parachutes method*, which aims to use provider-managed FaaS platforms as backup routes for crucial functionalities. The name *serverless parachute* was introduced to emphasize the complementary nature of a duplicated functionality that can be used in emergency cases. The original work aimed to increase the availability of desired code snippets using FaaS platforms, even when the original application is unavailable. In this section, the concept of serverless parachutes and the underlying framework are discussed in a more general context, namely how to facilitate the reuse of provider-agnostic code for different platforms using automated code extraction mechanisms.

5.2.1 Method Steps

Figure 5.14 depicts the sequence of steps in the serverless parachutes method with respective step inputs and outputs that begins with the identification of a suitable code snippet and ends with a deployable FaaS function package for a specific platform. Each step is discussed next, showing how the method can be used for configuration of chosen code snippets for automated extraction and packaging for different FaaS platforms.

Step 1: Identify Suitable Functionality. The first step in Figure 5.14 is to select functionalities that are intended to be extracted as serverless parachutes. Generally, not all functionalities are easy to migrate to a cloud [KT14] and since the functionality is intended for a FaaS deployment, FaaS-specific requirements must be taken into account. This means that functionalities should preferably be fine-grained, easy to decouple, and stateless. As with other FaaSification approaches, functionalities with many dependencies that require extensive re-engineering efforts are rather unsuitable for such automated extraction and packaging method.

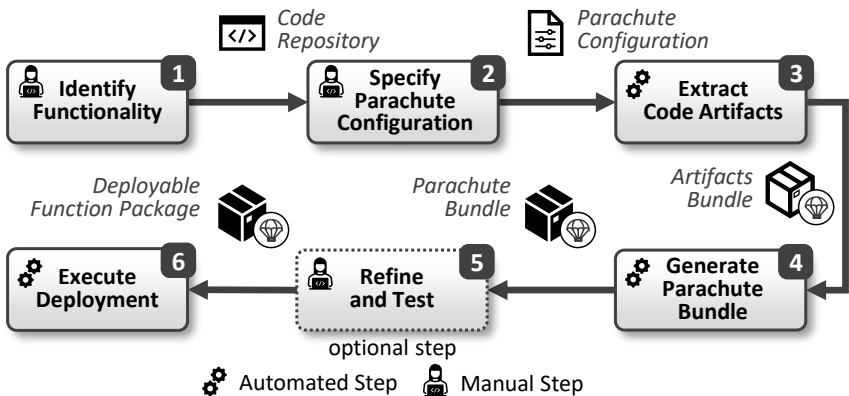


Figure 5.14: Steps of the Serverless Parachutes Method.

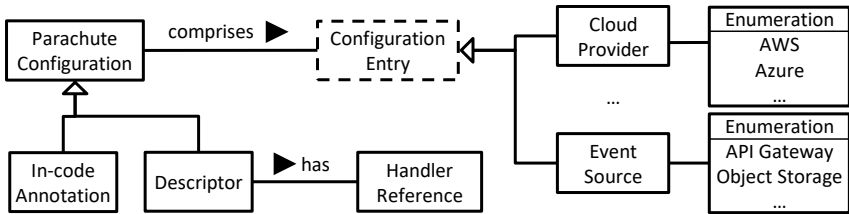


Figure 5.15: A meta-model for parachute configurations.

Step 2: Specify Parachute Configuration. In the second step, a *parachute configuration* for the chosen functionality needs to be specified, which describes the preferred extraction and packaging aspects. In the original publication [YBHL19], this configuration was specified directly in-code using annotations. This, however, requires introducing modifications in the repository including extra dependencies on annotation libraries, which could be not desirable or even not possible for third-party repositories. To make it more flexible, such configurations should be supported as both in-code annotations or external descriptors that provide the same configurations for a referenced functionality, i.e., the reference to the corresponding function handler should also be present in such descriptors. Figure 5.15 schematically shows a parachute configuration, which comprises a set of configuration entries that describe, e.g., a cloud provider and the event source type such as the API Gateway that should trigger the chosen functionality. The configuration entries could also be more specialized for specific extraction scenarios, e.g., describing failover behavior [YBHL19]. An example of in-code parachute configuration using Java annotations is shown in Listing 5.2 – a similar configuration could also be specified as a standalone YAML-based descriptor with a reference to the repository and the file containing the `myHandler` method with the same signature.

```

@ParachuteMethod(
    eventSource = "EventSource.API_Gateway", provider = "Provider.AWS",
    endpoint = "/myResourceName", httpMethod = "POST", ...)
public Response myHandler(Request data) {...}
  
```

Listing 5.2: Example parachute configuration using Java annotations.

Step 3: Extract Code Artifacts. In the third step, the respective *artifacts bundle* for the selected functionality is extracted using the specified parachute configuration, which comprises at least the function code with the list of required library dependencies. The extracted artifacts bundle is provider-agnostic and can be reused for generating platform-specific function packages for different supported targets. For instance, a plain Java function typically requires adding provider-specific code and configurations, e.g., for processing events from AWS offerings in the function code. The artifacts bundle extraction requires language- and technology-specific plugins due to the different characteristics of the underlying languages, e.g., strong vs. weak typing, and the use of different build tools such as Maven or npm. Examples of extracted information include (i) input and output types of the function with their definitions, (ii) class, method, and variable dependencies, and (iii) library dependencies. The resulting self-contained artifacts bundle can then be used for the generation and packaging of platform-specific function packages.

Step 4: Generate Parachute Bundle. In the fourth step, the *parachute bundle* with the extracted functionality for a specific platform is generated, e.g., a deployment package for AWS Lambda. Here, the previously-created artifacts bundle is used as an input to enrich the extracted function with platform-specific parts, e.g., adding required import and library call statements in the code. Additionally, platform-specific dependencies may need to be added in the underlying build script and, if needed, the compilation should be triggered, e.g., to generate a JAR for AWS Lambda. The parachute bundle can be generated for different scenarios, e.g., (i) to immediately deploy the function package to the target platform or (ii) to use it in the context of a larger deployment model, e.g., attach it as a Deployment Artifact to the corresponding Node Template representing a FaaS platform in a TOSCA model. For the former scenario, a deployment model can be generated, e.g., an AWS SAM [Ama22e] model, as a part of the parachute bundle to enable directly executing the deployment. In the latter case, the parachute bundle does not require generating a deployment model and only the function package should be provided as an output of this

step. The distinction between two scenarios can, for example, be implemented as separate generation modes available for selection by users of the method-enabling tooling.

Steps 5 & 6: Refine, Test, and Deploy. Finally, in the last steps, the generated parachute bundle can be tested and refined to enable using it for automated deployments. Depending on the chosen generation mode, i.e., whether a deployment model for the extracted function was generated or not, the refinement may target different parts of the bundle. For example, the generated deployment model can be enriched with configurations for additional components intended to be used with the extracted functionality. In case the extracted functionality will be used in a larger application model it can be referenced as the respective artifact. For example, the functionality can be exposed via an endpoint in existing API, hence requiring to attach the parachute bundle to the deployment model.

5.2.2 Serverless Parachutes Framework Architecture

Figure 5.16 shows a conceptual architecture of the framework enabling the method discussed in Section 5.2.1. To enable using the framework, the presentation layer provides a REST API and a GUI. The business logic layer supports the method steps using the following core components. *Programming Language Annotation Libraries* are needed for defining in-code parachute configurations. *Parachute Configuration Processor* enables parsing the descriptor-based parachute configurations. *Function Extraction Manager* relies on language-specific plugins to enable static code analysis and extraction of artifact bundles, and *Parachute Deployment Bundle Generator* consumes the extracted platform-independent artifact bundles as an input for generating platform-specific parachute deployment bundles utilizing respective generation plugins, e.g., for AWS Lambda or Microsoft Azure Functions. Finally, the resource layer comprises an *Artifacts Repository* that enables persisting the extracted code artifact

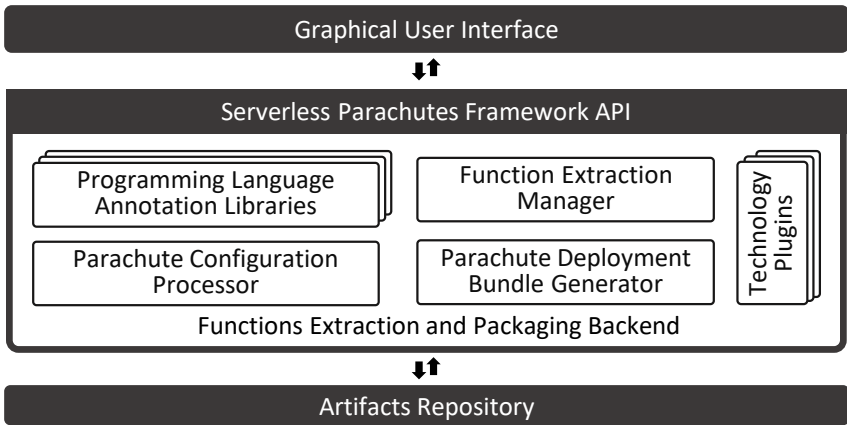


Figure 5.16: Architecture of the Serverless Parachutes Framework.

bundles and generated parachute deployment bundles for future reuse, e.g., to simplify generating deployable function packages for different platforms using already extracted code artifacts.

5.3 Associating Artifact Abstractions With Components in Application Models

Both introduced artifact abstraction approaches represent different kinds of FaaS-specific artifacts: function orchestration models and function code. Several modeling aspects need to be considered to enable associating such abstract artifacts with the corresponding components in application models, e.g., components that represent functions in architecture and deployment models. Clearly, components in application models need to be associated with different *artifact types* that can enable representing abstract artifacts such as BPMN4FO function orchestration models or parachute configurations. This requires that the modeling of typed artifacts is supported in the corresponding modeling language, e.g., components in TOSCA models

can be associated with different Artifact Types that can be utilized to represent function orchestration models created with the BPMN4FO profile. Additionally, typed artifacts can also be used to express various packaging formats for the same implementations. For instance, different packaging may be supported for FaaS functions, e.g., self-contained JARs or Docker images, since it was shown that archive packages are more beneficial to use in certain cases for reducing the cold start issues[DKL22].

Secondly, the artifact abstractions associated with components in application models need also to be linked with the tooling capable of refining them. For example, the approaches discussed in Sections 5.1 and 5.2 rely on transformation systems capable of producing target-specific formats based on provided abstract artifacts, i.e., transforming BPMN4FO models into orchestrator-specific models and using parachute configurations to generate platform-specific function packages. In addition, aside from generative techniques, artifact abstractions can be refined by selecting a suitable implementation from a repository of available concrete solutions [FBB+14b]. Therefore, tools that enable selecting available artifact refinements instead of generating them can be incorporated too.

Furthermore, another modeling aspect to consider is the specification of when the artifact refinement should be triggered, e.g., at what stage the call to the corresponding artifact refinement tool responsible for generating a target function orchestration model should happen. For example, this refinement process can be triggered at application model packaging time to generate a deployable application package. Additionally, to enable switching between abstract and concrete application models for explorative modeling scenarios, the link to the original artifact abstraction must be pertained, e.g., using versioning or by caching original abstract artifacts.

Therefore, on the application model level, different artifact abstractions need to be linked with various types of concrete solution selection or generation tools as well as refinement modes. Similar to the co-authored publication on modeling data transformations in service choreographies [HBL+18], the envisioned support for using artifact abstractions

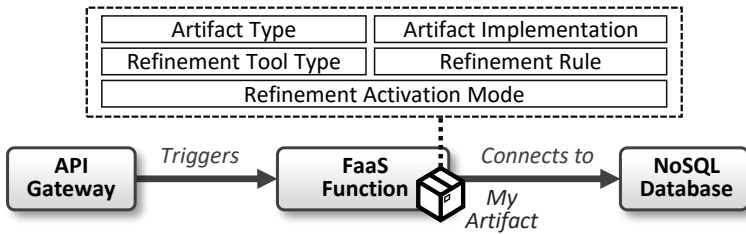


Figure 5.17: Configuration details needed for modeling abstract artifacts in application models.

in application models relies on additional *configuration details*. Figure 5.17 depicts an example of the discussed artifact configuration details for a FaaS function including the (i) *type of the artifact*, (ii) the actual *artifact implementation* such as a parachute configuration or a BPMN4FO model, (iii) *refinement tool type* referencing specific tool that will be used for refining the artifact, (iv) *refinement rule* specifying how the tool should be invoked to refine the referenced artifact, and (v) *refinement activation mode* that specifies when refinement should happen, e.g., at model export time. The BPMN4FO Transformation Framework and Serverless Parachutes Framework presented in this chapter are only two example tools that can be used for refining specific artifact types, i.e., function orchestration models and function code. However, the same configuration details can be used to enable more refinement options such as selecting suitable function implementations from existing repositories as long as the chosen modeling language provides means for associating typed artifacts with components in the model.

5.4 Chapter Summary

In this chapter, two artifact abstraction techniques were introduced: one for technology-agnostic modeling of function orchestrations (Section 5.1) and another for automated extraction and packaging of function code for

different FaaS platforms (Section 5.2). Additionally, the required configuration details for modeling such abstraction on the level of application models were discussed.

Similar to previous chapters, the introduced concepts can be used independently of previous contributions as specialized artifact abstraction and refinement techniques. For example, the function orchestration modeling with BPMN4FO can be used separately to create different target formats or explore the modeling capabilities of supported function orchestrators using a clear visual notation. The modeling restrictions aimed at portability also enable easily identifying the constraints for creating portable models. The caveat is, however, the portability of function implementations for such orchestration models. One way to overcome this is to combine BPMN4FO with function code abstraction techniques including the Serverless Parachutes method. On the other hand, the disadvantage of both approaches is their dependence on target technologies and languages, which requires implementing and maintaining multiple plugins that may result in an unnecessary overhead for some use cases.

GRADUAL REFINEMENT OF FAAS-BASED APPLICATIONS

INSPIRED by the concept of the gradient of abstraction [Flo08], this chapter introduces the Gradual Refinement of Architectures of Serverless applications (GRASP) method that aims to simplify transitioning between different abstractions for FaaS-based applications following the vision discussed in Section 1.2. To achieve this, the GRASP Method relies on a multi-layer application metamodel that enables transitioning between different levels of abstractions for FaaS-based applications. The presented metamodel and underlying multistep process of modeling and refinement of FaaS-based application models enable combined use of the concepts introduced in Chapters 3 to 5. The modeling concepts presented in this chapter are based on several peer-reviewed publications [WBK+18; YBB+22; YBKL20; YBK+20]. In the following, a step-wise overview of the GRASP Method is presented followed by a discussion of the underlying FaaS-based application metamodel and the gradual refinement process.

6.1 The GRASP Method

The GRASP Method aims to enable gradual refinement of FaaS-based application models following the vision presented in Section 1.2, i.e., how to transition from coarse-grained component-connector models to suitable, concrete deployment models with artifact implementations compatible with chosen target providers. This section first provides a general overview of the GRASP Method and its main actors followed by a more detailed discussion of each underlying step.

6.1.1 Method Overview

An overview of the method comprising five steps is depicted in Figure 6.1: Firstly, an abstract FaaS-based topology representing the desired connectivity among components needs to be created in *Step 1*. This abstract, provider-agnostic application topology is then used for exploring component hosting alternatives in *Step 2*. In this step, the modeling of stack hosting requirements is achieved by associating the components in the topology with the desired combinations of patterns presented in Chapter 3, and using the resulting pattern-based model fragments for exploring possible refinements into concrete deployment model fragments. In parallel with stack hosting alternatives exploration, abstract artifact representations can be modeled and attached to the respective components in *Step 3*. After deciding on component hosting, the modeled topology with attached artifact abstractions is refined into a concrete deployment model in *Step 4*. Finally, the obtained concrete deployment model is further tested and executed using a suitable deployment automation engine in *Step 5*.

The actors of the method can be separated into two general categories: (i) *method users* such as application designers and developers that perform the method steps to explore hosting alternatives and obtain concrete deployment models from coarse-grained descriptions of FaaS-based applications and (ii) *method software maintainers* such as domain experts

responsible for populating method-enabling data repositories and tool developers that provide and maintain the software toolchain that enables the method steps. Specifically, the method steps rely on different repositories that comprise (i) abstract modeling constructs and patterns and (ii) rules for refining topology fragments. Additionally, classification frameworks and the respective technology review data, e.g., for FaaS platforms, can facilitate choosing from the identified hosting alternatives based on various component-specific requirements, also using the initial technology query can be generated based on the user-provided abstract topology fragment as discussed in Chapter 4. Such repositories are intended to be used with dedicated software that either automates or facilitates using the repository data. For example, modeling constructs for specifying abstract FaaS-based component topologies can be used in combination with a graphical modeling tool such as Eclipse Winery [KBBL13]. Therefore, to make the GRASP Method attractive to the intended user audience, the access to different domain-specific knowledge corresponding to method steps needs to be augmented using a dedicated software. In the following, each step, its required repository dependencies, and the necessary software support are discussed in more detail.

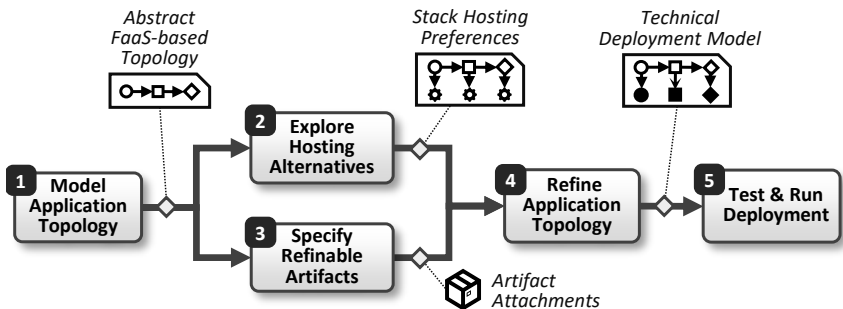


Figure 6.1: Overview of the steps of the GRASP Method.

6.1.2 Step 1: Model Application Topology

To provide an initial, coarse-grained view on the desired FaaS-based application structure, the first step of the GRASP Method shown in Figure 6.1 is concerned with creating an abstract topology of components without any technology- or provider-specific details. In its minimal form, this includes the information about (i) abstract component types such as functions and event sources of a FaaS-based application, (ii) abstract relations among these components, e.g., to represent event-driven or direct calls [WBK+18], and (iii) generic properties of components such as function runtime, and relations, e.g., event type for relations representing event-triggering semantics. Typed components and relations are represented using dedicated abstract modeling constructs, i.e., a repository of *Abstract Modeling Constructs* is required in this step. TOSCA is a good candidate for implementing such repositories: in this case TOSCA Requirements and Capabilities can be used to validate the connectivity between components, whereas TOSCA Node and Relationship Types can be used to flexibly specify abstract component types, their properties, and relations among them. In addition, existing graphical modeling tools supporting TOSCA can be employed to manage such repositories of modeling constructs. In the following steps, TOSCA is assumed to be used as the modeling language. Using the constructs from such repository, method users can, therefore, specify desired abstract topologies using source code editors or graphical modeling tools. For example, to specify the desired FaaS function runtime independently of specific FaaS platforms, an abstract TOSCA Node Type “FaaS Function” with a property “runtime” can be used.

6.1.3 Step 2: Explore Hosting Alternatives

In this step, the components in the specified abstract FaaS-based topology are first enriched with hosting requirements that are represented by associating components with the patterns (or their combinations) from the Component Hosting and Management pattern language introduced

in Contribution 1 (see Chapter 3). For instance, a FaaS Function component can be associated with the coarse-grained `FIXED DEPLOYMENT STACK` and `PROVIDER-MANAGED SCALING CONFIGURATION` patterns to represent the decisions on stack and scaling configuration management. Likewise, the same FaaS Function component can be associated directly with the finer-grained `SERVERLESS HOSTING` pattern instead.

Clearly, the expressed component-specific hosting requirements might be fulfilled by different hosting alternatives, e.g., a FaaS Function can be hosted on different proprietary and open source FaaS platforms. Therefore, to identify most suitable candidates for components, possible deployment stacks are explored following the sub-steps shown in Figure 6.2: after selecting a component and associating it with desired patterns from the Component Hosting and Management pattern language (A), the possible refinements are iteratively explored (B). As will be shown next, this includes two kinds of refinements, namely (i) *pattern refinements* that reflect transitions between patterns in the language as discussed in Chapter 3, and (ii) *technical deployment stack refinements* that reflect transitions from abstract, pattern-based models to concrete deployment model fragments. The available technical deployment stack refinements can then be optionally pruned (C) based on various technology-specific criteria requirements applied against the technology classification frameworks data as discussed in Chapter 4, e.g., FaaS platforms that do not support the desired event

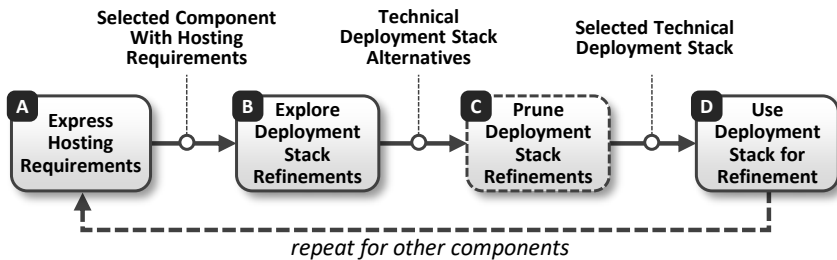


Figure 6.2: A sequence of sub-steps for exploring hosting alternatives that constitute the Step 2 of the GRASP Method.

trigger can be excluded from potential refinement options. Finally, the chosen technical deployment stack is used as a refinement preference (D) for the concrete deployment model. The same process can then be repeated for other components in the model.

To illustrate the process of exploration of hosting alternatives, Figure 6.3 illustrates it for different components in an example abstract FaaS-based topology comprising two functions, an object storage bucket, and a message queue. For each component in this abstract topology, three different layers of refinements can be explored, namely (1) *Management Requirements Layer*, (2) *Component Hosting Requirements Layer*, and (3) *Technical Deployment Stack Alternatives Layer*. Layers 1&2 represent refinements based on the relations between patterns introduced in Chapter 3, e.g., patterns representing management requirements can be refined into more concrete hosting patterns. For example, a *Java Application* component in Figure 6.3 requires no customization and is, thus associated with the `FIXED DEPLOYMENT STACK` and `PROVIDER-MANAGED SCALING CONFIGURATION` patterns in Layer 1, which must be refined into the `SERVERLESS HOSTING` pattern in Layer 2. Finally, Layer 3 represents suitable concrete technical deployment stack options for a component, e.g., the `SERVERLESS HOSTING` pattern for the *Java Application* can be further refined into `AWS Lambda` or `Azure Functions` deployment stacks.

Optionally, the available technical deployment stack refinements can be pruned as shown in Figure 6.3 using the concepts discussed in Chapter 4. Classification frameworks for different technology types as well as criteria mappings and architectural fact extractors compatible with the format of the abstract FaaS-based topology can be used for querying technologies complying with the specified requirements. An obvious way to employ these concepts is to manually search among available technology review data for the most suitable hosting target based on the available refinement options by querying the selection support system (see Section 4.4) using the desired criteria such as function runtime or supported event triggers. Another option is to employ the search query generation from the given abstract FaaS topology using the available architectural fact extractors

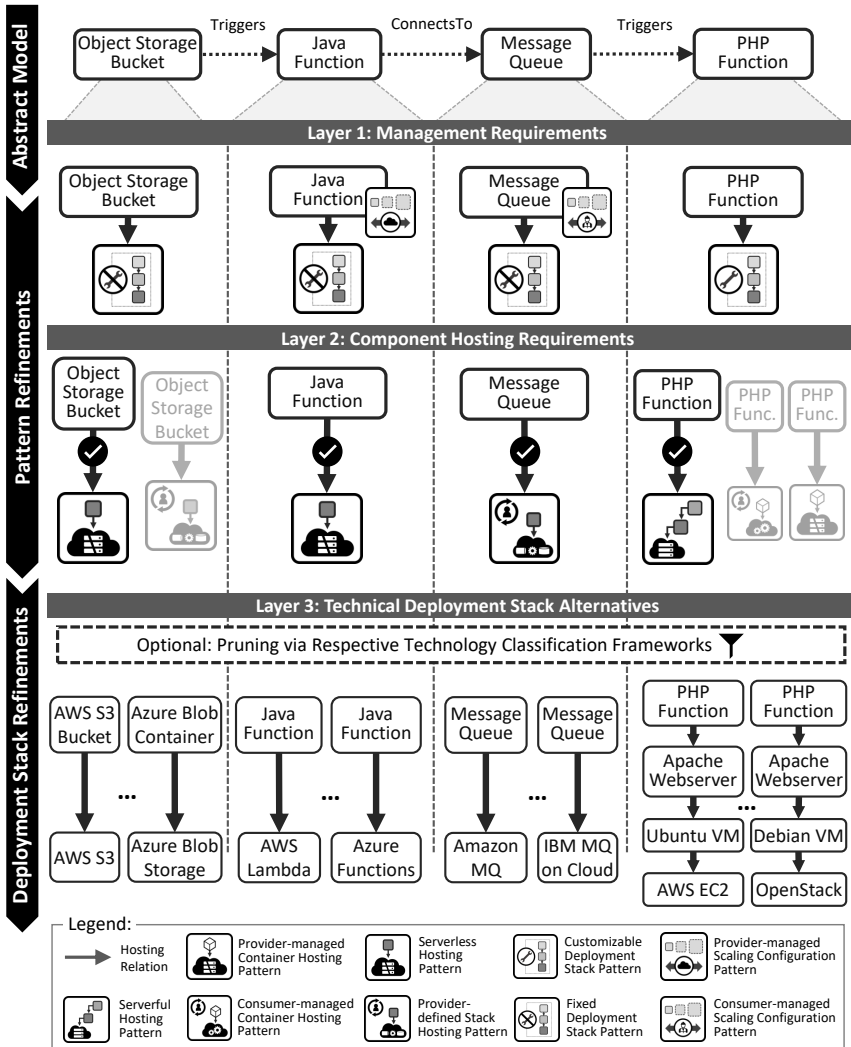


Figure 6.3: Examples of hosting alternatives exploration for distinct components in an abstract FaaS-based topology.

and criteria mappings. The exploration of alternatives is intended to be semi-automatic and iterative: an initial search query can be extracted from the provided abstract FaaS-based topology to explore the initial suitable hosting options for a specific component stack, e.g., FaaS platforms for a function in an abstract FaaS-based topology. However, the query can then be manually refined using the GUI of the underlying selection support system, which could also lead to possible changes in the initial design decisions and, hence the repetition of the previous steps. As an output of this step, the hosting preferences for modeled stacks are identified, which enables method users to refine the abstract component stacks contained in the abstract FaaS-based topology into concrete technical deployment model fragments using the identified hosting preferences.

As in Step 1, the *Abstract Modeling Constructs* repository is also needed for specifying patterns (as abstract representations of components) and hosting relations between patterns and components. For instance, in TOSCA, such pattern associations can be implemented using custom or normative Node and Relationship Types with TOSCA, e.g., a normative Relationship Type “HostedOn” can be used for representing hosting on SERVERFUL HOSTING and SERVERLESS HOSTING Node Types that express the respective patterns. In addition, the associations with hosting patterns that represent scaling configuration management decisions can be implemented as TOSCA Policies attached to abstract components. The abstract constructs in the FaaS-based topology may also cover type-specific and technology-agnostic characteristics such as required runtimes or event triggers that can be used for technology query generation as discussed in Chapter 4. After the abstract FaaS-based topology is specified in Step 1, the property values can be updated with preferred values in this step, e.g., a decision to use Java as a runtime for one of the functions in the topology, to facilitate pruning the available hosting alternatives. Identification and definition of such abstract properties is the responsibility of domain experts that populate and maintain the underlying repository. Method users can specify

stack hosting requirements using source code editors or compatible graphical modeling tools, e.g., abstract TOSCA types can be created with code editors or modeling tools such as Eclipse Winery [KBBL13].

Additionally, the exploration of technical deployment stacks relies on the repositories of *Refinement Rules* and *Classification Frameworks*, which are required to be populated and maintained by domain experts. The former repository is used to explore available stack refinements for a given component in the abstract FaaS-based topology, whereas the latter enables pruning those alternatives based on technology-specific classification criteria. The toolchain enabling the GRASP method that was implemented in the context of this work (see Chapter 7), utilizes the concept of pattern-based refinement models [HBF+18] to realize a TOSCA-based refinement models repository, and the concepts from Chapter 4 are employed to create the classification frameworks repository.

6.1.4 Step 3: Model Refinable Artifacts

After identifying suitable hosting alternatives, the given FaaS-based topology needs to be further enriched with the component-specific artifacts information in Step 3. At this stage, abstract representations for respective artifacts can be created and attached to the components in the topology. The concepts for abstract modeling of function orchestrations using BPMN4FO and functions code abstraction using the serverless parachutes framework discussed in Chapter 5 are example abstractions that can be used in this step. In the context of this work, Section 7.3 demonstrates how BPMN4FO models are used to generate provider-specific function orchestration models realizing a data processing use case. As was also described in Chapter 5, such representations can also include other approaches, e.g., FaaSification discussed in Section 2.2.4 to abstract away function code. The order of steps depicted in Figure 6.1 mainly highlights a scenario of creating reusable FaaS-based application models that are supposed to be refined into more than one target format, e.g., for AWS and Azure cloud infrastructures.

However, the GRASP Method might also be considered for scenarios in which one preferred deployment model is intended to be constructed semi-automatically, i.e., iterative composition of suitable deployment fragments based on high-level component hosting requirements. In such cases, Step 3 can be considered optional, since only one target host is needed – artifact implementations for the chosen targets can, therefore, be attached to the refined application model before Step 5.

6.1.5 Step 4: Refine Into Technical Deployment Model

After deciding on hosting preferences for the components and attaching the abstract artifact representations, a suitable technical deployment model can be obtained in Step 4 by refining these abstract entities into concrete target-specific representations. For instance, a generic FaaS Function component triggered by a generic Object Storage Bucket can be refined into an AWS-based deployment model by composing model fragments for AWS Lambda and AWS S3, respectively. The attached artifact abstractions are also refined in this step by invoking the corresponding refinement tools, e.g., a function can be extracted from an existing repository and packaged for AWS Lambda using an abstract parachute configuration discussed in Chapter 5 attached to the FaaS Function component. This step also relies on a repository of *Refinement Rules* that describe how to transform generic stack representations into concrete, technical deployment model fragments. Note that the relationships between stacks that represent, e.g., event-driven or direct calls, need to be refined too using the corresponding refinement rules. For example, the implementation of event bindings between an AWS Lambda function and an AWS S3 bucket would differ for Azure offerings. In TOSCA terms, these technology-specific implementations of event bindings could inherit from a generic “Triggers” Relationship Type and the corresponding pattern refinement model [HBF+18] would then describe how to transform it into a technology-specific Relationship

Type, e.g., “AzureBlobTriggers”. Following the concepts by Harzenetter et al. [HBF+18; HBF+20], the envisioned refinement of abstract stack representations happens in a semi-automated fashion.

6.1.6 Step 5: Test and Execute Deployment Model

Finally, the obtained technical deployment model with the refined artifacts needs to be tested and potentially enriched with execution-related details in Step 5. Here, existing methods for testing deployment models [DHY+22; WBKL18] can be utilized to verify the correct execution of obtained models. Depending on the employed modeling language, the enactment of the produced technical deployment model could be possible with different deployment automation engines. For example, in case of TOSCA models, multiple TOSCA-compliant deployment automation engines can be used, e.g., OpenTOSCA Container [BBH+13] or xOpera [LSC20]. Additionally, transformation tooling can be utilized to enable using other deployment engines, e.g., the TOSCA Lightning [WBH+20a] toolchain can be used to generate other model formats using TOSCA models as an input.

6.2 The GRASP Meta-Model and Language Support

To support the steps of the GRASP Method discussed previously, this section introduces a FaaS-based application metamodel that combines different levels of abstractions for such applications and enables transitioning between them. As seen in Figure 6.4, a coarse-grained GRASP model represents a set of *Component Stacks* that may comprise different *Stack Elements*, i.e., typed *Components* and *Hosting Patterns* used as abstract representations of hosting requirements. These elements can be interconnected using various typed *Connectors*, e.g., representing hosting relationships or event-driven calls. Additionally, some patterns discussed in Chapter 3, e.g., for scaling configuration management, can be used as

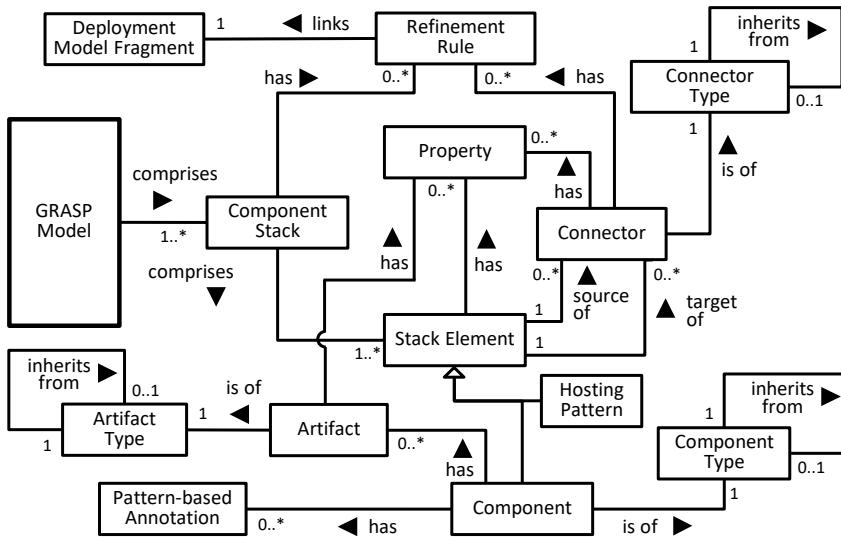


Figure 6.4: An overview of the GRASP metamodel.

component *Annotations* to represent management requirements for deployment stack and scaling configuration for a component in the stack – examples of such pattern-based stacks are depicted in Figure 6.3. Both Stack Elements and Connectors may include zero or more *Properties* that describe technology-agnostic characteristics of the corresponding element type using primitive and composite data types, and can be used to specify requirements for modeled FaaS-based applications. Further, to support modeling artifacts, Components can have typed *Artifact* attachments. Finally, to enable refinement into technical deployment model fragments, the GRASP metamodel comprises *Refinement Rules* for Component Stacks and Connectors that describe how these abstract model elements can be transformed into deployment model fragments. As the definition of a novel modeling language is not in the scope of this work, in the following, an application of GRASP Method is discussed in the context of TOSCA.

As mentioned previously, abstract FaaS-based topologies in GRASP can leverage pattern-based refinement models in TOSCA [HBF+18; HBF+20]: pattern-based deployment models enable representing abstract component topologies with patterns supported as distinct components or component annotations. In this case, the repository of Abstract Modeling Constructs discussed in Steps 1&2 would comprise a collection of abstract TOSCA Types including Node and Relationship Types for modeling generic components in FaaS-based applications and relationships among them. Additionally, TOSCA Policy Types and Artifact Types can be used to represent pattern-based annotations and component-specific artifact attachments shown in Figure 6.4, respectively.

The refinement into concrete technical deployment model fragments relies on graph transformation rules that describe how abstract TOSCA model fragments can be substituted with concrete TOSCA model fragments, hence, implementing the hosting alternatives exploration process depicted in Figure 6.3. Refinement rules are specified using triple graph grammars [HBF+18; HBF+20], i.e., the Refinement Rules repository is

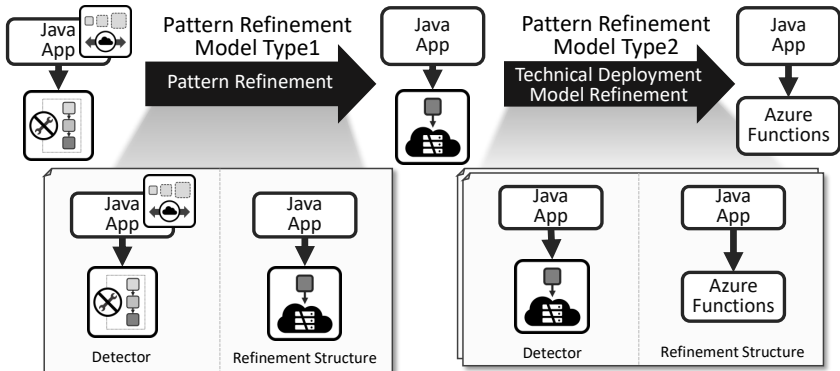


Figure 6.5: Implementation of GRASP refinement rules using different types of pattern refinement models using the concepts by Harzenetter et al. [HBF+18; HBF+20].

populated with the so-called Pattern Refinement Models that define how an input TOSCA graph is transformed into an output TOSCA graph. Each Pattern Refinement Model comprises (i) a *detector* – a deployment model fragment describing a refinable combination of components and patterns, and (ii) a *refinement structure* describing the desired outcome of the refinement. An example illustrating the required types of refinement rules implemented using the concept of Pattern Refinement Models [HBF+18; HBF+20] is shown in Figure 6.5: two different kinds of pattern refinement models are implemented to enable pattern refinements (Pattern Refinement Model Type 1) and technical deployment model refinements (Pattern Refinement Model Type 2). Thus, Type 1 refinements cover the transitions within the categories of the Component Hosting and Management pattern language, i.e., the transition from the Management Requirements Layer to the Component Hosting Requirements Layer. Type 2 covers the refinements into concrete technical deployment stacks corresponding to the transition from the Component Hosting Requirements Layer to the Technical Deployment Stack Alternatives Layer as shown in Figure 6.3. The Pattern Refinement in Figure 6.5 is applicable since it specifies the modeled combination of a generic *Java Application* component and the two patterns as detector. It enables obtaining a more concrete model fragment, i.e., a *Java Application* hosted on the `SERVERLESS HOSTING` pattern as it is a refinement of the modeled pattern combination. Next, the resulting model is further refined using the Technical Deployment Model Refinement shown in Figure 6.5: its detector matches the current model state and can be refined into a technical deployment model fragment, i.e., a *Java Application* hosted on *Azure Functions*. In practice, the Technical Deployment Model Refinement provides a more detailed technical deployment model fragment with required properties, which are omitted here for brevity. Furthermore, there could be more available pattern refinement models with matching detectors, e.g., enabling the refinement into an AWS Lambda stack.

6.3 Chapter Summary

In this chapter, the GRASP Method for gradual refinement of FaaS-based application models was introduced to support transitioning from abstract, provider-agnostic application models to executable, provider-specific deployment models. The introduced method combines contributions presented in Chapters 3 to 5 in a five-step process. In the first step, to represent FaaS-based applications in a provider-agnostic manner, the GRASP method relies on pattern-based modeling: abstract application topologies are specified as typed DAGs that contain the patterns from the Component Hosting and Management pattern language introduced in Chapter 3 as first-class model elements for representing component hosting and scaling configuration management requirements. In the second step, the classification framework concepts introduced in Chapter 4 are used to facilitate searching for suitable hosting alternatives. To support provider-agnostic representation of FaaS-specific artifacts and their refinement into target formats, the method relies on contributions presented in Chapter 5 in the third step. To derive provider-specific deployment models, in step four, the produced abstract application topologies are refined into target-specific deployment models using the ideas presented in this chapter that rely on pattern-based refinement models [HBF+18; HBF+20]. Finally, the produced deployment models are then tested and executed using compatible deployment automation technologies in step five.

INTEGRATED ARCHITECTURE AND PROTOTYPICAL VALIDATION

MODELING concepts and abstractions for FaaS-based applications introduced in Chapters 3 to 5 can be used as standalone contributions or as the parts of GRASP Method presented in Chapter 6. To enable using the introduced method and underlying concepts in practical scenarios, this chapter presents Contribution 5 that describes an architecture of the GRASP Toolchain and elaborates on the prototypical implementation of the underlying tools. The discussion is structured as follows. Firstly, an integrated architecture supporting the method is presented in Section 7.1. As conceptual architectures of some tools were previously discussed in Chapters 4 and 5, Section 7.2 further elaborates on their prototypical implementation and use in the context of the integrated architecture and an example FaaS-based application focusing on a data analytics scenario. Finally, Section 7.4 discusses the integration with other existing research concepts outside the context of this work.

7.1 Architecture of the GRASP Toolchain

Figure 7.1 provides an overview of an integrated architecture of the GRASP toolchain, which enables using the GRASP Method and the underlying concepts introduced in Chapters 3 to 5 in practice. Following the discussion in Chapter 6, the YAML version of TOSCA [OAS20] is used as a modeling language for specifying both coarse-grained FaaS-based application models and technical, technology-specific deployment model fragments. Therefore, for modeling and enactment of produced technical deployment models, TOSCA-compliant software is used as a basis. While the creation of GRASP models in TOSCA can also be achieved using code editors, *Eclipse Winery* [BEK+16; KBBL13] is utilized as the core modeling tool as it also supports the refinement of produced models by processing refinement rules discussed in Chapter 6 and transforming detected model fragments using the specified refinement structures. To enable executing the produced TOSCA models packaged as TOSCA Cloud Service Archives (CSARs) [OAS20], *xOpera* [LSC20] is employed as a TOSCA-compliant deployment automation engine. To support the artifact

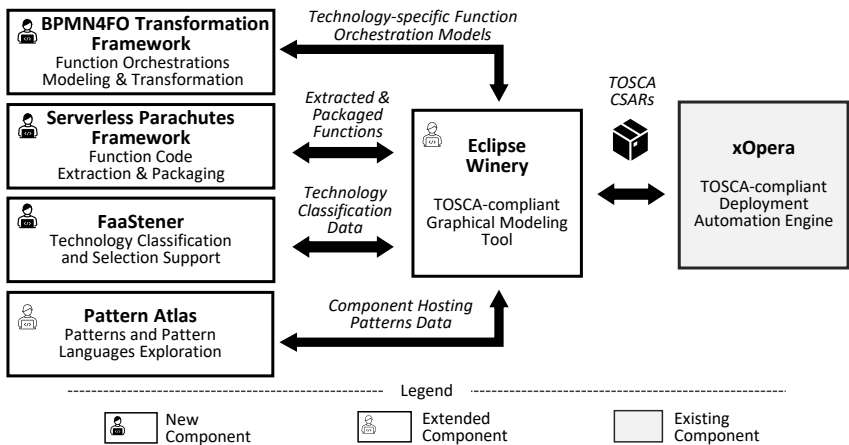


Figure 7.1: Integrated architecture of the GRASP toolchain.

abstraction mechanisms discussed in Chapter 5, Winery enables modeling abstract artifacts produced with the prototypical implementations of (i) *BPMN4FO Transformation Framework* to enable generating orchestrator specific models from BPMN4FO models and (ii) *Serverless Parachutes Framework* to support extraction and packaging of FaaS functions using parachute configurations. Furthermore, to facilitate the exploration of technology classification data, the *FaaSter* prototype implementing the selection support system architecture discussed in Chapter 4 is integrated with Winery to enable building search queries from provided TOSCA models. In addition, to explore and use the patterns introduced in Chapter 3, an existing tool for exploring patterns and pattern languages called *Pattern Atlas* [LB21] is enriched with the patterns from Component Hosting and Management pattern language. Moreover, since Pattern Atlas is integrated with Winery, the added patterns are automatically extracted by Winery to generate TOSCA Node Types and Policy Types representing these patterns, thus enabling using them for modeling hosting requirements as discussed in Chapter 6.

7.2 Prototypical Implementation

This section elaborates on the new prototypical implementations and extensions to existing tools for each of the component described in the GRASP toolchain architecture.

7.2.1 Specification and Execution of GRASP Models

Eclipse Winery¹ was extended to enable modeling and refinement of FaaS-based applications using the GRASP Method. Winery [BEK+16; KBBL13] is a well-established TOSCA-compliant graphical modeling

¹<https://github.com/eclipse/winery>

tool implemented in Java that provides a plethora of advanced features for deployment modeling of applications including pattern-based deployment modeling and refinement [HBF+18; HBF+20], application topology splitting [SBKL17], or even blockchain-based accountability mechanisms for collaborative model development scenarios [FBF+18]. Originally, Winery supported only the XML TOSCA specification. However, over the course of the EU Horizon 2020 Project RADON [CAH+19] Winery was extended to support the TOSCA YAML specification. Additionally, Winery was extended to support the aforementioned pattern refinement modeling concepts [HBF+18; HBF+20] for the YAML version of TOSCA. Figure 7.2 shows a simplified architecture of Eclipse Winery that depicts only components relevant in the context of this work. To support the TOSCA YAML specification [OAS20], Winery was extended to use a *Canonical Data Model* that unifies the concepts from both XML and YAML TOSCA specifications, hence, enabling the same modeling process for different

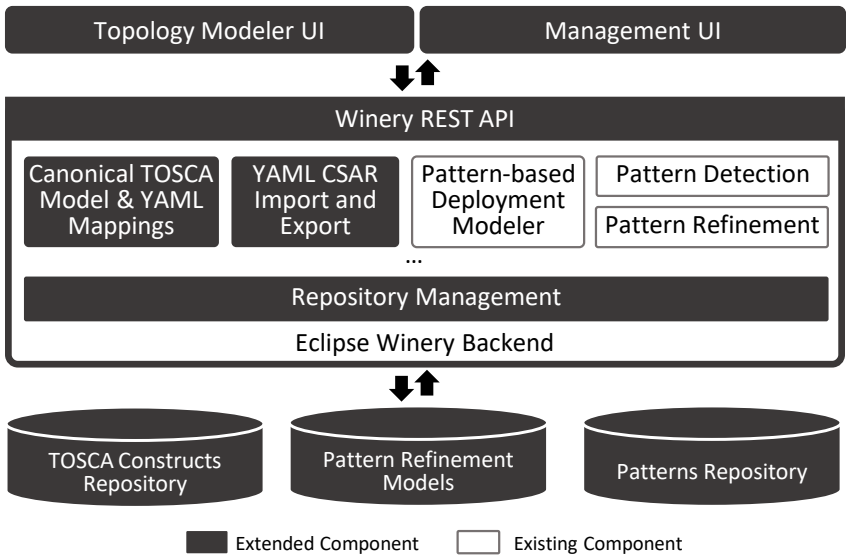


Figure 7.2: Simplified architecture of Eclipse Winery.

specification versions. The transformation between the canonical model and actual specifications happens transparently for modelers – these modifications also required extensions of *Repository Management* functionalities in Winery. Moreover, to support export and import of CSARs in YAML format, the *Export and Import* components were also extended.

Graphical modeling in Winery happens by means of two components implemented in Angular (see Winery documentation² for more details), namely (i) the Topology Modeler, which enables creating application topologies graphically, and (ii) the Management UI, which is responsible for creating TOSCA modeling constructs such as Node Types or Relationship Types. Both components interact with the backend via the REST API – all these components had to be also extended in the context of the EU Horizon 2020 Project RADON [CAH+19] to enable YAML-based modeling. Figure 7.3 shows an example YAML TOSCA model created graphically in

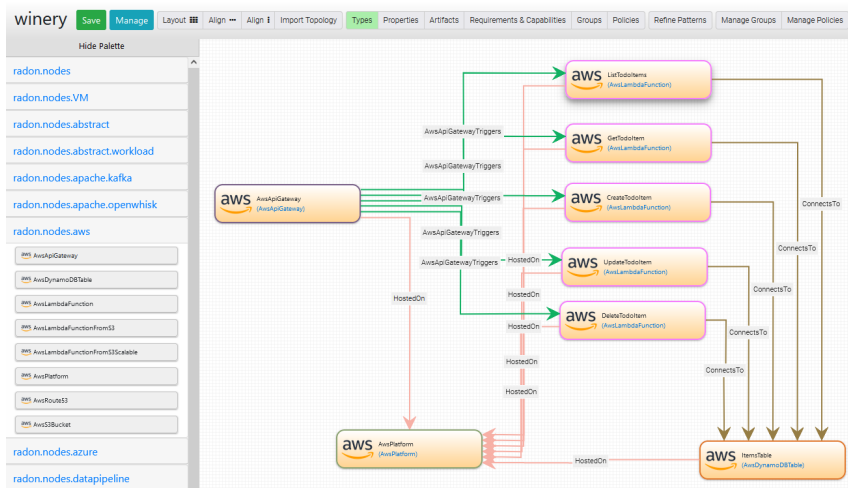


Figure 7.3: An example YAML TOSCA model created in Winery.

²<https://winery.readthedocs.io/en/latest/index.html>

Eclipse Winery that describes a FaaS-based application hosted on AWS that implements a serverless API use case. In this example, a set of AWS Lambda functions implementing basic CRUD operations for managing To-Do items, which are persisted to AWS DynamoDB, are exposed via the AWS API Gateway.

To support the pattern-based modeling and exploration of patterns, the repository of the Pattern Atlas [LB21] was enriched with the Component Hosting and Management pattern language as shown in Figure 7.4 – the existing integration with Winery also enables using these pattern descriptions for generating the corresponding Node and Policy Types for pattern-based modeling. As a TOSCA-compliant deployment automation engine, xOpera [LSC20] consumes CSARs as an input and is capable of deriving the required order of operations to enact the deployment of modeled applications. TOSCA implementation artifacts in xOpera are

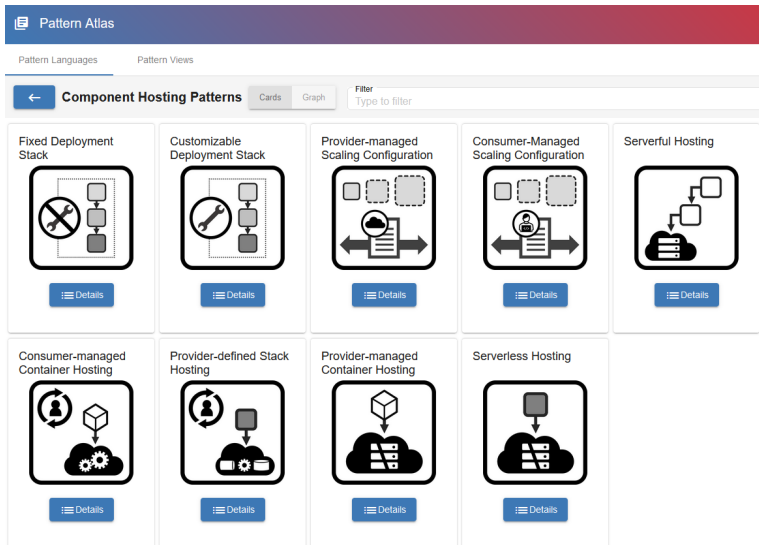


Figure 7.4: The patterns from the Component Hosting and Management pattern language accessible via the Pattern Atlas GUI.

developed using Ansible [Red22], thus, all required concrete deployment fragments used in the implemented Pattern Refinement Models rely on Ansible playbooks. Since xOpera is used as-is, it is not further discussed (see <https://github.com/xlab-si/xopera-opera> for more details). The *TOSCA Constructs Repository* and *Pattern Refinement Models* were enriched with (i) abstract and concrete (deployable) TOSCA types based on the work conducted in the EU Horizon 2020 project RADON and (ii) pattern refinement models using the concepts by Harzenetter et al. [HBF+18; HBF+20] as discussed in Chapter 7.

7.2.2 FaaSStener: A Technology Selection Support Tool

To support the use of technology classification data for FaaS functions and other component types in FaaS-based applications, the selection support tool called FaaSStener³ was implemented following the conceptual architecture presented in Section 4.4. FaaSStener comprises two parts, namely (i) a frontend component implemented in Angular⁴ using the Angular Material component library and (ii) a backend component implemented using the Java Spring Boot framework⁵. The frontend component is responsible for the visual exploration of the technology classification data including visualizing the framework and filter representations discussed in Section 4.3 and using those filters for executing manual queries against the data. A demo version of this component with static FaaS platforms review data is also hosted online⁶. Figure 7.5 shows a screenshot of a FaaSStener-generated framework representation, in which the dimensions and categories can be interactively explored by clicking on the corresponding card elements. Likewise, Figure 7.6 shows a screenshot of a FaaSStener-generated filter representation and the data exploration interface,

³<https://github.com/faastener>

⁴<https://angular.io>

⁵<https://spring.io/projects/spring-boot>

⁶<https://faastener.github.io>

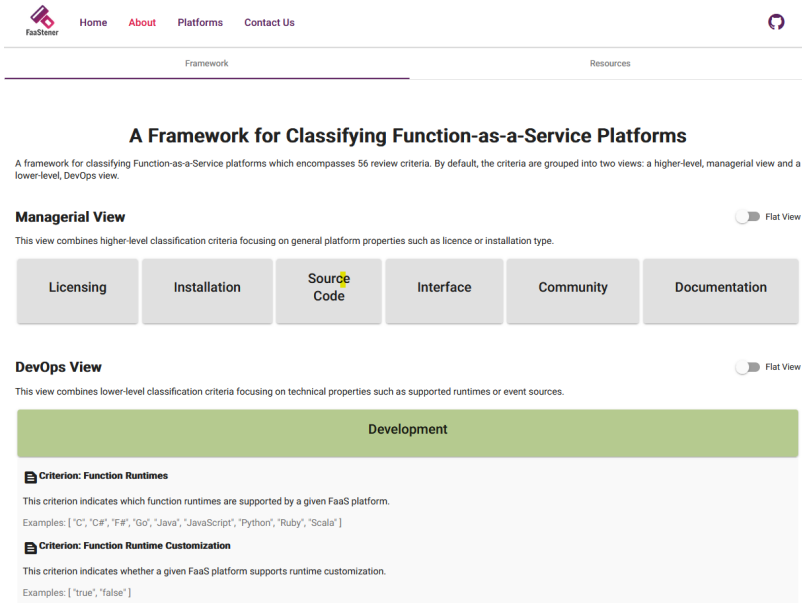


Figure 7.5: Example framework representation generated in FaaSter.

in which FaaS platforms can be search based on user-provided criteria requirements. The frontend component interacts with the Java backend via a REST API to retrieve the framework- and technology classification-related data. The generated OpenAPI specification for the REST API can be explored using Swagger UI⁷. Moreover, the backend implements an example set of fact extractors for TOSCA models to extract architectural facts for FaaS functions and map them to the classification criteria as discussed in Section 4.3. This enables generating search queries for specific components in TOSCA models sent to the FaaSter backend via its REST API. Specifically, search queries are generated in RSQL, a query language for parametrized filtering of data in REST APIs (see the RSQL

⁷See <https://github.com/faastener/faastener-core> for more details.

The screenshot shows the FaaSter Platform Browser interface. On the left is a 'Filter Panel' with sections for Licensing, License, License Type, Installation, and Source Code. The 'License' section has 'Apache 2.0' selected. The 'License Type' section has 'permissive' selected. The 'Installation' section has 'installable' selected. The 'Source Code' section has 'open source' selected. The main area is a table titled 'Platform Browser' with columns: Name, Licensing, Installation, Source Code, and Interface. The table lists four Apache 2.0 platforms: Apache Operator, OpenFaaS, FaaS Project, and Apache. Each row provides details on license type, installation targets, source code availability, programming language, and interface types.

Name	Licensing		Installation		Source Code			Interface		GHUB Stars	GHUB	
	License Name	License Type	Installation Type	Installation Targets	Source Code Availability	Source Code Repository	Main Programming Language	Interface Type	Interface Operations for Application Management			Interface Operations for Platform Administration
Apache Operator	Apache 2.0	permissive	installable	Docker Kubernetes Linux MacOS Micos	open source	GHUB	Scala	CLI API	create retrieve update delete	deployment configuration enrichment	4800	932
OpenFaaS	Apache 2.0	permissive	installable	Kubernetes	open source	GHUB	Go	CLI API	create retrieve update delete	deployment configuration enrichment	5200	487
FaaS Project	Apache 2.0	permissive	installable	Docker Linux MacOS Windows	open source	GHUB	Go	CLI API GUI	create retrieve update delete	deployment configuration enrichment termination	4900	363
Apache	Apache 2.0	permissive	installable	Kubernetes	open source	GHUB	Go	CLI API	create retrieve update delete	deployment configuration enrichment	3600	764

Figure 7.6: Example filter representation generated in FaaSter.

parser documentation <https://github.com/jirutka/rsql-parser> for more details). Thus, the required integration between Winery and FaaSter enables sending a TOSCA topology to the FaaSter backend for exploring suitable hosting alternatives for a selected FaaS function as discussed in Chapter 6. To show the applicability of the classification framework metamodel introduced in Section 4.3, in the context of this thesis, FaaS-ter comprises a framework for selection support of FaaS platforms that includes data for ten platforms [YSB+21a] presented in Chapter 4.

7.2.3 Uniform Function Orchestrations Models with BPMN4FO

To enable creating technology-agnostic function orchestration models using the BPMN4FO profile introduced in Section 5.1 and transforming these models into orchestrator-specific formats, the BPMN4FO Transformation

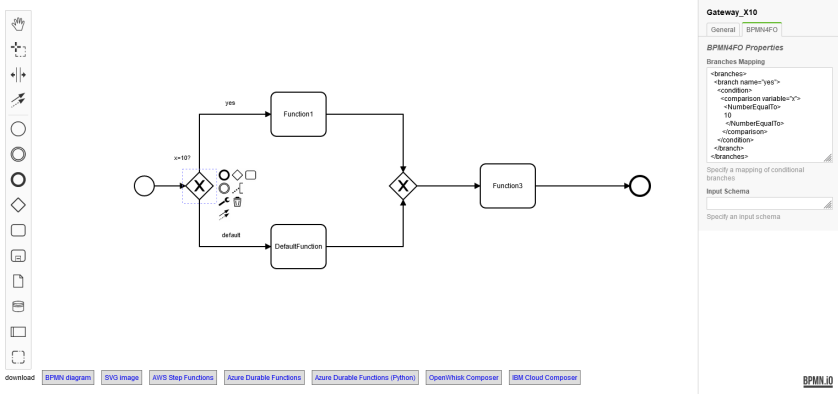


Figure 7.7: Example modeling in the BPMN4FO Framework GUI.

Framework⁸ was implemented following the conceptual architecture described in Section 5.1.5. The BPMN4FO Transformation Framework is a JavaScript-based application that enables (i) graphical modeling of function orchestrations using the BPMN4FO profile and (ii) transforming these orchestrator-agnostic models into orchestrator-specific formats such as ASL [Ama22d]. Figure 7.7 shows a screenshot of the graphical editor in the BPMN4FO Transformation Framework implemented based on the open source BPMN 2.0 web modeler by Camunda bpmn-js⁹. Additionally, the modeling of artifacts with Eclipse Winery is implemented by means of the *BPMN4FOModel* TOSCA Artifact Type as discussed in Section 5.3.

Several modeling restrictions were imposed on the created BPMN4FO models due to technology-specific requirements of the analyzed function orchestrators – these restrictions are only needed to enable the uniform transformation into the analyzed target formats: Firstly, to ensure the uniform transformation into the analyzed function orchestrator formats, modeling of fault handling in parallel branches is not allowed, i.e., no

⁸<https://github.com/iaas-splab/matoswo#bpmn4fo>

⁹<https://bpmn.io>

BPMN ERROR BOUNDARY EVENTS can be attached to activities (i) modeled in parallel branches or (ii) marked with BPMN MULTI-INSTANCE MARKERS. In addition, BPMN MULTI-INSTANCE MARKER and BPMN LOOP MARKER cannot be used in combination in the same activity to unambiguously represent the desired GFOMC. Finally, since AWS Step Functions uses JSON objects as input and output states, the data containers are assumed to contain elements in JSON format. Assumptions are also made for the structure of inputs and outputs for Parallel Branching and Fan-out orchestration elements due to constraints of Openwhisk Composer: inputs/outputs are assumed to be arrays stored in the `value` field of a JSON object.

7.2.4 Serverless Parachutes Framework

To enable the automated extraction of code snippets from open source code repositories and packaging them as FaaS functions using the concept of serverless parachutes presented in Section 5.1, the Serverless Parachutes Framework was implemented following the conceptual architecture described in Section 5.2. The framework¹⁰ is implemented in Java and supports extracting Java code snippets marked with custom annotations¹¹ and packaging them for AWS Lambda. In terms of trigger support, only invocation via API Gateways is supported, i.e., the extracted function is assumed to be capable of processing incoming HTTP request and returning its result as outputs. The REST API of the framework is realized using Jersey, an implementation of the JAX-RS specification. In addition, a web-based visualization of the provided API is available using the Swagger tools. Similar to the BPMN4FO Transformation Framework, the modeling in Eclipse Winery is achieved by means of the dedicated TOSCA Artifact Type *ParachuteFunction* – artifacts of this type containing a parachute configuration file can then be attached to TOSCA components and refined via Winery as discussed in Section 5.3.

¹⁰<https://github.com/v-yussupov/parachutesmethod-framework>

¹¹<https://github.com/v-yussupov/parachutesmethod-annotations>

7.3 The GRASP Toolchain by Example

This section discusses how the GRASP Method presented in Chapter 6 can be employed in the context of a FaaS-based use case motivated by an existing function orchestration scenario [Con20] for processing the open air quality dataset (see <https://registry.opendata.aws/openaq>) that comprises physical air quality data aggregated from public data sources, e.g., data provided by government, research, and other institutions. The resulting Extract-Transform-Load (ETL) function orchestration generates the minimum, maximum, and average ratings for air quality measurements on a daily basis and store them in a corresponding cloud object storage bucket. The toolchain presented in Section 7.1 is employed for modeling and gradual refinement of the FaaS-based application, which results in its deployment to three public cloud providers, namely AWS, Azure, and IBM Cloud. For conciseness, only some results are illustrated in the following, whereas the implementation of the use case business logic and all the required modeling elements including abstract modeling constructs and pattern refinement models are available on GitHub (see <https://github.com/v-yussupov/tosca-grasp-modeling>).

7.3.1 Use Case: An ETL Orchestration for Air Quality Data

Figure 7.8 shows the aforementioned function orchestration model, which represents an ETL orchestration for extracting and transforming the publicly-available air quality data using FaaS functions, and storing the results in a serverless object storage offering. In the shown model, a function orchestrator such as AWS Step Functions or Azure Durable Functions coordinates the execution of four FaaS functions that also need to interact with object storage buckets. The *Get Files* function lists the files that contain air quality data for the previous day, which are stored in a public object storage bucket *Open Air Quality Dataset*, and then splits them into chunks that will be processed in parallel using multiple instances of the function *Transform Data*. Here, the files in each chunk are first downloaded

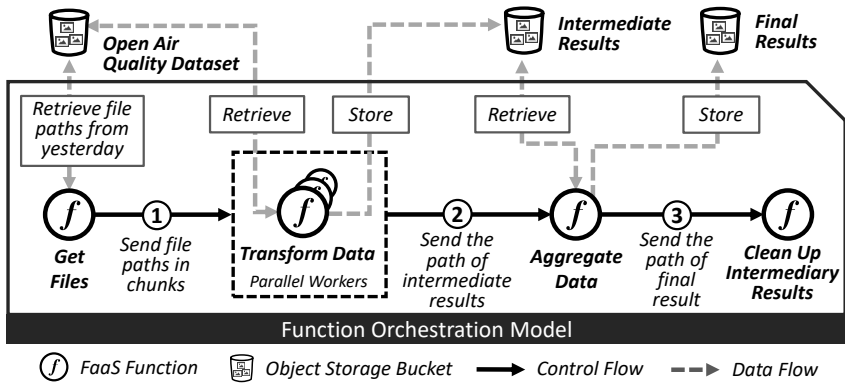


Figure 7.8: A function orchestration for processing open air quality data based on the ETL orchestration for AWS Step Functions [Con20].

and then transformed into an intermediary representation that is persisted to the *Intermediate Results* bucket. After receiving all intermediary results, the function *Aggregate Data* merges the results into a single file with normalized structure and stores this final result in the *Final Results* bucket. Finally, the *Clean Up Intermediary Results* function removes the intermediary results.

7.3.2 Modeling Abstract FaaS-based Application Topology

While the described function orchestration can be seen as a standalone FaaS-based application, it may be necessary to execute it as a part of a larger application. For example, since the air quality data is processed only for the previous day, the ETL orchestration can be scheduled using a timer. Moreover, after the final result is ready, a notification event may need to be emitted to notify external components that require the final

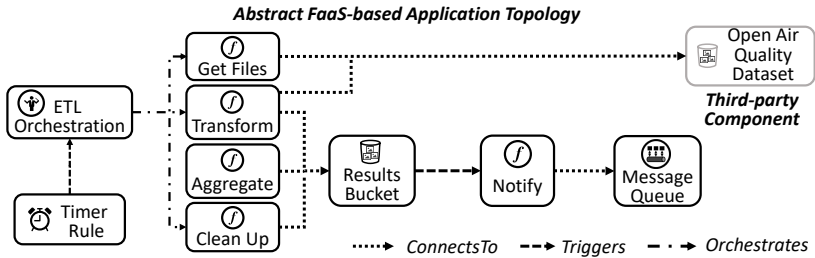


Figure 7.9: An abstract FaaS-based application topology of the ETL function orchestration from Figure 7.8.

result. Following the Step 1 of the GRASP Method, Figure 7.9 shows an example abstract FaaS-based topology that represents the aforementioned ETL function orchestration.

Firstly, the shown topology comprises the (i) *ETL Orchestration* that need to be modeled for the chosen function orchestrator and (ii) four orchestrated FaaS functions that need to be implemented for a compatible FaaS platform, e.g., AWS Step Functions and AWS Lambda. The *ETL Orchestration* is triggered using a *Timer Rule* that can be configured using scheduling services such as Amazon EventBridge [Ama22b]. The *Results Bucket* can be hosted on object storage services such as IBM Cloud Object Storage and is used to store the final result.

The event emitted by the bucket after the final result is stored triggers the standalone *Notify* function, which generates a notification message and publishes it to a *Message Queue*, which can be created on a provider-managed messaging service such as AWS SQS [Ama22b]. External clients can access the message queue to process generated messages, e.g., a data analytics application running on-premises. Figure 7.10 shows the same abstract topology modeled in Winery using abstract constructs such as “Function” and “Message Queue”. Please, note that the timer rule is modeled as a property of the ETL Orchestration component, and, thus is not an explicit part of the topology. This topology can be utilized

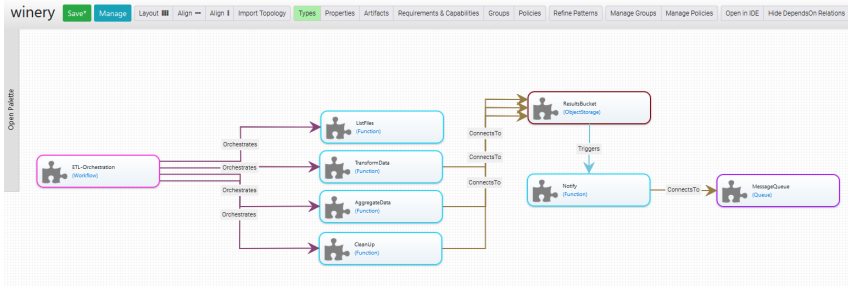


Figure 7.10: An abstract topology for a FaaS-based application of the ETL function orchestration modeled in Eclipse Winery.

for expressing hosting requirements using the patterns from Component Hosting and Management pattern language and exploring the available technology-specific hosting alternatives as discussed next.

7.3.3 Exploring Hosting Alternatives

To express the component hosting requirements, each component in the topology is associated with a preferred combination of patterns. In this scenario, the `SERVERLESS` pattern is used for each component in the model to reduce the deployment stack and scaling configuration management overhead as shown in Figure 7.11. Specifically, each abstract component is associated with this pattern using the normative TOSCA Relationship Type “HostedOn”.

Clearly, the resulting abstract topology can be refined into different target-specific variants, e.g., all components can be hosted using offerings from a single cloud provider such as AWS or Azure. Distinct parts of the topology could also be distributed across different cloud providers, e.g., based on the data placement requirements. In this example scenario, only refinements within a single cloud offering were considered, resulting in three possible pattern refinement models being explored, namely for AWS, Azure, and

7 Integrated Architecture and Prototypical Validation

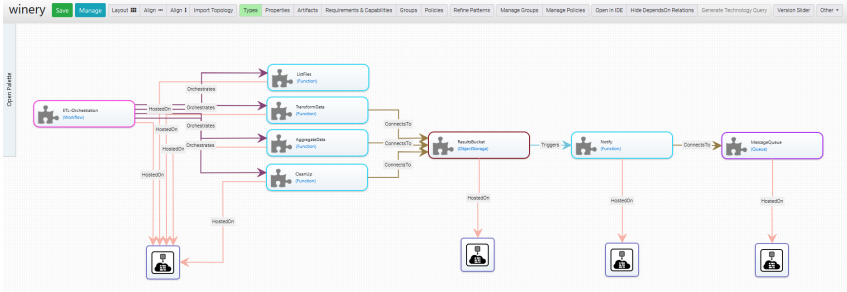


Figure 7.11: Component hosting requirements for the ETL function orchestration modeled with patterns from the Component Hosting and Management pattern language in Eclipse Winery.

IBM Cloud. By clicking on the “Refine Patterns” button in Winery as depicted in Figure 7.12, a list of available refinement options is shown to modelers that now can visualize each refinement variant.

As mentioned previously, within a single provider context, the modeled topology can be refined using compatible service offerings. For example, in AWS, AWS Lambda can host the functions, AWS Step Functions can host the function orchestration model, while AWS S3 and AWS SQS can be used as object storage and message queue offerings, respectively.

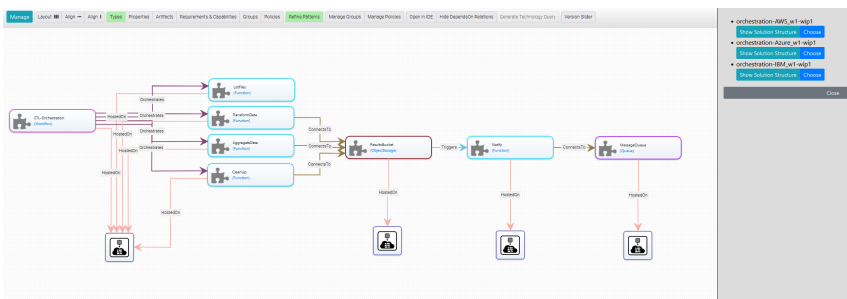


Figure 7.12: Available refinements for the modeled FaaS-based topology.

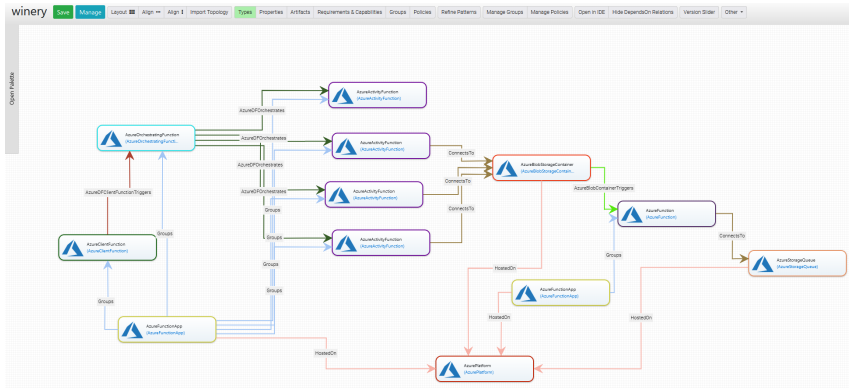


Figure 7.14: The modeled FaaS-based topology refined for Azure.

customization requirements for deployment stack are needed – in such cases, open source FaaS platforms that support function orchestrations such as Fission can be utilized too. Therefore, the exploration process is intended to be iterative as modelers are always able to update the initial requirements and explore the updated list of suitable hosting alternatives.

7.3.4 Specify Abstract Artifacts

Complementary to the exploration of hosting alternatives, the ETL function orchestration was modeled using the BPMN4FO Framework. The BPMN4FO model shown in Figure 7.15 is a Sequence that begins with the call to the *ListFiles* function modeled as a Task to prepare the file

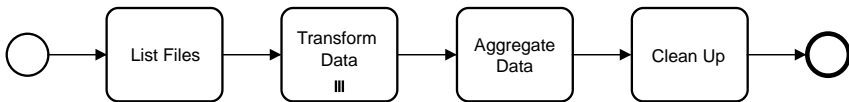


Figure 7.15: A BPMN4FO model representing the ETL function orchestration for processing air quality data.

paths with air quality data for the previous day split in chunks. Next, the *TransformData* function is run in parallel for each chunk, which is modeled using the Fan-out construct from the BPMN4FO profile. As a result, each chunk is mapped to a separate function instances that downloads and transforms the files in this chunk into an intermediary format. Afterwards, the *AggregateData* function modeled as a Task reduces all intermediary files into one final result. Finally, also modeled as a Task, the *CleanUp* function deletes unnecessary files.

Other kinds of abstract artifacts could be modeled at this stage too. For example, the serverless parachute configuration can be specified to extract the implementations of the orchestrated functions and package them for FaaS platforms of choice. Since the functions for this particular example FaaS-based application were implemented in Python, the prototype discussed in Section 7.2.4 could not be utilized due to its support for Java. An example use of the Serverless Parachutes framework for a Java application and extraction of its functionality for AWS Lambda with an evaluation of the approach can be found in the original publication [YBHL19].

7.3.5 Refine, Test, and Deploy

After deciding on the target provider, the abstract FaaS-based topology can be refined into the concrete, technical deployment model. For conciseness, this subsection covers this process only briefly, whereas a detailed description of the deployment and execution results can be found in the original publication [YSBL22].

To illustrate the process, consider that AWS was chosen as the target cloud, hence the abstract topology should be refined into the AWS-specific deployment model shown in Figure 7.13. Thus, the BPMN4FO model shown in Figure 7.15 was refined for the AWS Step Functions orchestrator, i.e., the ASL model that will be deployed to the orchestrator. In addition, since no function code abstraction mechanisms were used in this example, the function code needed to be implemented for AWS as the target provider,

which in this case was not needed since the function code for AWS was reused from the original repository [Con20]. The refined deployment model with the attached artifacts and specified component properties can then be tested and deployed to the target provider. The process of deployment is shown in Figure 7.16: the TOSCA orchestrator xOpera (i) executes the implementation artifacts created in Ansible in the correct order derived based on the relations among components in the model and (ii) reports about the successful completion of the deployment process. Finally, the application is triggered using the timer, which results in the execution of the ETL orchestration as shown in Figure 7.17. More details on the deployment and execution of this application for all three mentioned cloud providers can be found in the original publication [YSBL22].

```
(.venv) yussupv@DESKTOP-K1EDR3:/mnt/c/dev/opera/ETL-FunctionOrchestration$ opera deploy _definitions/iaaso
[Worker_0] Deploying AwsPlatform_0_0
[Worker_0]   Executing configure on AwsPlatform_0_0
[Worker_0] Deployment of AwsPlatform_0_0 complete
[Worker_0] Deploying AwsLambdaFunction_0_0
[Worker_0]   Executing create on AwsLambdaFunction_0_0
[Worker_0] Deployment of AwsLambdaFunction_0_0 complete
[Worker_0] Deploying AwsSQSQueue_0_0
[Worker_0]   Executing create on AwsSQSQueue_0_0
[Worker_0] Deployment of AwsSQSQueue_0_0 complete
[Worker_0] Deploying AwsLambdaFunction_4_0
[Worker_0]   Executing create on AwsLambdaFunction_4_0
[Worker_0] Deployment of AwsLambdaFunction_4_0 complete
[Worker_0] Deploying AwsS3Bucket_0_0
[Worker_0]   Executing create on AwsS3Bucket_0_0
[Worker_0]   Executing post_configure_source on AwsS3Bucket_0_0--AwsLambdaFunction_4_0
[Worker_0] Deployment of AwsS3Bucket_0_0 complete
[Worker_0] Deploying AwsLambdaFunction_1_0
[Worker_0]   Executing create on AwsLambdaFunction_1_0
[Worker_0] Deployment of AwsLambdaFunction_1_0 complete
[Worker_0] Deploying AwsLambdaFunction_2_0
[Worker_0]   Executing create on AwsLambdaFunction_2_0
[Worker_0] Deployment of AwsLambdaFunction_2_0 complete
[Worker_0] Deploying AwsLambdaFunction_3_0
[Worker_0]   Executing create on AwsLambdaFunction_3_0
[Worker_0] Deployment of AwsLambdaFunction_3_0 complete
[Worker_0] Deploying AwsSFOrchestration_0_0
[Worker_0]   Executing create on AwsSFOrchestration_0_0
[Worker_0]   Executing pre_configure_source on AwsSFOrchestration_0_0--AwsLambdaFunction_0_0
[Worker_0]   Executing pre_configure_source on AwsSFOrchestration_0_0--AwsLambdaFunction_1_0
[Worker_0]   Executing pre_configure_source on AwsSFOrchestration_0_0--AwsLambdaFunction_2_0
[Worker_0]   Executing pre_configure_source on AwsSFOrchestration_0_0--AwsLambdaFunction_3_0
[Worker_0]   Executing configure on AwsSFOrchestration_0_0
[Worker_0] Deployment of AwsSFOrchestration_0_0 complete
(.venv) yussupv@DESKTOP-K1EDR3:/mnt/c/dev/opera/ETL-FunctionOrchestration$ _
```

Figure 7.16: Deployment to AWS using xOpera.

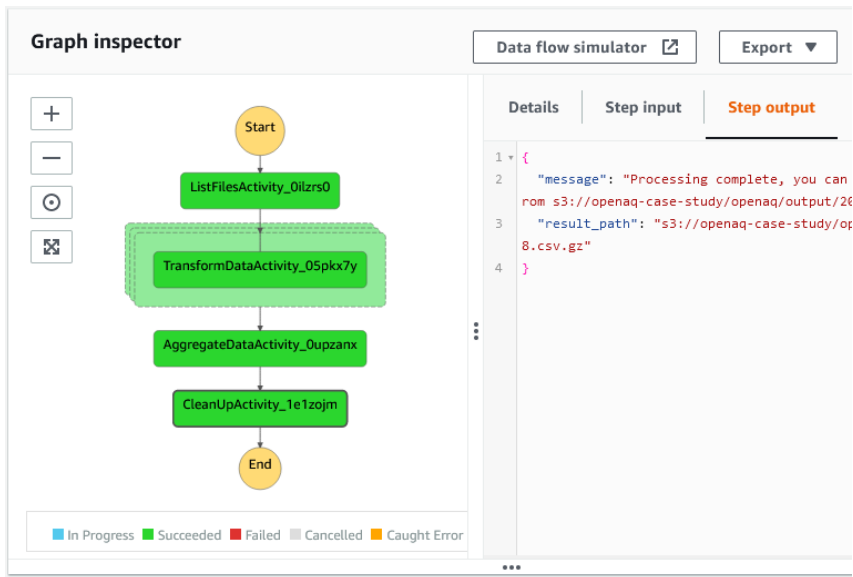


Figure 7.17: Execution of the ETL orchestration for processing air quality data using AWS Step Functions.

7.4 Integration with Other Systems

The concepts presented in this thesis and their implementations as the GRASP Toolchain can also be combined with other existing co-authored approaches. Firstly, the GRASP method can be combined with the approach that transforms TOSCA models into other deployment model formats using the TOSCA Light toolchain [WBB+19; WBH+20b]. Additionally, the TOSCA-based models implementing the GRASP concepts can be combined with the Continuous Testing Tool [DHY+22] implemented in the context of the EU Horizon 2020 Project RADON.

7.4.1 TOSCA Light Toolchain

Wurster et al. [WBH+20b] introduce an end-to-end toolchain for transforming TOSCA models into the variety of deployment automation technology-specific model formats such as Ansible [Red22] or Kubernetes [The22c]. To achieve this, authors rely on the concept of Essential Deployment Meta-model (EDMM) [WBF+19] – a metamodel that captures the core characteristics of major deployment automation technologies and their underlying model formats. As one outcome of this work, the so-called TOSCA Light profile specifying an EDMM-compliant subset of TOSCA to enable transforming models into formats supported by EDMM is derived. To enable using the profile in practice, the TOSCA Light toolchain is implemented that supports graphical specification of deployment models using Eclipse Winery and transformation of resulting TOSCA models into proprietary formats using the EDMM Transformation Framework [WBB+19]. Being based on Winery, the GRASP toolchain is compatible with the aforementioned format transformations, hence, enabling also using other deployment automation engines for the refined technical deployment models obtained by following the GRASP method steps, e.g., Kubernetes deployment specifications can be generated for FaaS-based applications that are intended to be hosted on Kubernetes.

7.4.2 Continuous Testing Tool

Düllmann et al. [DHY+22] present the Continuous Testing Tool (CTT) that facilitates (i) specification of tests, e.g., load tests for modeled FaaS-based applications, and test infrastructures as well as cloud systems under test and (ii) automated deployment and execution of such tests against deployed applications. CTT utilizes TOSCA-based modeling to enable modeling and execution of tests continuously. As both Eclipse Winery and CTT are parts of the EU Horizon 2020 Project RADON, the integration between these tools was implemented in the context of the project. In particular, CTT is compliant with the format of exported CSARs and TOSCA modeling

constructs created in Winery, hence, enabling to produce models for both FaaS-based applications and testing infrastructures as well as desired tests that are modeled using TOSCA Policy Types. Resulting models produced following the GRASP method steps can, thus, be combined with the CTT and the underlying concepts to enable continuous testing of produced technical deployment models.

CONCLUSIONS AND OUTLOOK

ENGINEERING FaaS-based applications, as any other software, is a multi-faceted task that requires considering multiple functional and non-functional aspects. This thesis introduces several research contributions that aim to facilitate the design and implementation of FaaS-based applications. The susceptibility to various kinds of vendor lock-in and the gap in existing research on abstraction mechanisms for FaaS-based applications identified during the preliminary research phase and highlighted in Section 1.1 served as the core motivation for this work that investigated which concepts can facilitate the process of technology-agnostic design of FaaS-based application models and their refinement into concrete technical deployment models. A detailed analysis of state-of-the-art research literature presented in Chapter 2 further reinforced the identified research questions and the vision of the work (see Section 1.2). To conclude this thesis, this chapter summarizes each introduced contribution and discusses them with respect to their limitations. Furthermore, potential future research directions for engineering FaaS-based applications related to the presented contributions are discussed.

8.1 Summary of Contributions

The research contributions introduced in the context of this thesis focus on various aspects of the design and implementation of FaaS-based applications. As an abstract way to document possible kinds of hosting solutions for components in FaaS-based applications, Contribution 1 (see Chapter 3) introduces the Component Hosting and Management pattern language that documents different component hosting trade-offs in the context of deployment stack and scaling configuration management, which addresses one part of the Research Question 1 related to the abstract representation of component hosting decisions as described in Section 1.2. Single patterns or their combinations can facilitate the selection of specific kinds of cloud offerings, e.g., provider-managed or self-hosted FaaS platforms. This pattern language is generic enough to be used for designing general cloud applications, which makes this contribution usable not only as a part of the GRASP method, but also as a standalone pattern language that can be combined with other pattern languages such as Cloud Computing patterns [FLR+14] and Enterprise Integration patterns [HW04a].

To further facilitate transitioning from abstract component hosting decisions in FaaS-based applications to concrete deployment options, Contribution 2 (see Chapter 4) presents a technology classification framework for FaaS platforms derived using a systematic analysis of academic literature and documentation of major existing FaaS platforms. This classification framework captures various kinds of higher-level, managerial criteria as well as technical, development and operations criteria. Moreover, as a part of this contribution, the classification framework metamodel is specified and discussed in detail together with the conceptual architecture of a selection support tool supporting these concepts to facilitate classification and selection of other component types in FaaS-based applications, which addresses the Research Question 2 formulated in Section 1.2. This contribution is also generic-enough to be used on its own, e.g., for selection of suitable FaaS platforms or implementing selection support tools for

other kinds of components. However, these concepts are also helpful in the context of application design and gradual refinement of FaaS-based application models envisioned in this thesis.

Next, to address the Research Question 3 formulated in Section 1.2, two distinct function-related abstractions were introduced as a part of the Contribution 3 (see Chapter 5): Firstly, Section 5.1 introduces the BPMN4FO Profile that enables modeling function orchestrations independently of specific function orchestrators such as AWS Step Functions. The resulting BPMN-compliant models can be transformed into target formats using the BPMN4FO Transformation Framework – a conceptual architecture and the underlying transformation concepts are introduced in the context of this abstraction approach too. Secondly, Section 5.2 presents an abstraction method and a conceptual architecture that enable extracting and packaging FaaS functions for platforms by annotating the desired code fragments in open source repositories. In addition, this contribution also discusses how such abstraction techniques can be used on the level of application models, hence, enabling using these techniques not only as standalone approaches, but also as a part of application design processes.

Contribution 4 (see Chapter 6) presents the GRASP Method for gradual refinement of FaaS-based applications from coarse-grained architectural models to concrete, technical deployment models that addresses the second part of the Research Question 1 as discussed in Section 1.2. The GRASP Method envisioned after conducting the preliminary research phase builds on top of the introduced GRASP application metamodel for FaaS-based applications and combines the previously-introduced contributions to enable creating abstract application models and refining them into technical deployment models. Finally, to address the Research Question 4 formulated in Section 1.2, Contribution 5 (see Chapter 7) introduces the GRASP Toolchain supporting this method using the TOSCA CML. To prove the practical feasibility of the introduced concepts, the GRASP Toolchain comprises prototypical implementations for the conceptual architectures

presented in Sections 4.4, 5.1.5 and 5.2.2 as well as extends/enriches existing open source tools Eclipse Winery [KBBL13] and Pattern Atlas [LB21] to enable the envisioned modeling and refinement processes.

8.2 Research Opportunities

The contributions presented in this thesis open multiple future research directions. Clearly, each distinct contribution presented in this work can be extended in multiple ways. For example, the Component Hosting and Management pattern language presented in Chapter 3 can be extended with more patterns related to component hosting aspects, essentially, resulting in a more general deployment-centric language. Another direction is to identify and add pattern links with more existing pattern languages to make the Pattern-based Decision models in the GRASP Method more flexible and powerful, i.e., generalizing the pattern usage for decision support and combining it further with the runtime-related concepts by Harzenetter et al. [HBB+21; HBKL19]. The pattern-based deployment modeling in GRASP can also be connected to the topic of architectural compliance checking [KBF+20], hence, further simplifying the use of introduced concepts in such architectural scenarios.

The technology classification and selection support concepts introduced in Chapter 4 can further benefit from reuse of existing taxonomies and ontologies related to different kinds of component types and service offerings. The introduced framework for classification of FaaS platforms can also be further refined and enriched with more managerial and technical criteria and new platform data. Furthermore, the topic of automated data collection is very important for such classification frameworks since the technology landscape evolves rapidly, making the collected technology classification data obsolete very quickly. Additionally, the search for suitable component hosting alternatives in this work focuses on semi-automatic exploration process utilized for distinct components in a given application model, whereas follow-up research can extend this task to groups of

components, i.e., find suitable hosting alternatives for a specific group of components in a given abstract FaaS-based application topology. Likewise, the FaaS-related abstraction techniques presented in Chapter 5 may benefit from various enhancements, e.g., support for additional technologies and language. The topic of FaaSification is, in general, limited in its applicability as the as-is extraction and packaging often limits the ways existing code can be reused. For example, existing general-purpose functionalities may need to be wrapped not only in provider-specific code parts, e.g., working with specific event types in AWS, but also used as a part of larger functions that include extra business logic. In such cases, parachute annotations can be extended to enable substitution of annotations in newly-created functions with the code extracted from other repositories.

Since the presented concepts focus on a minor share of well-known quality attributes such as deployability and portability, a variety of topics targeting other attributes are possible, e.g., modeling and enforcement of performance, availability, and security requirements. One straightforward research direction is further integration of contributions in this work with research concepts introduced by other groups in the context of the EU Horizon 2020 Project RADON, e.g., defect prediction in deployment models [DDPT21] or deployment optimization for FaaS [ZGTC21b]. Finally, the rapidly increasing capabilities of Large Language Models (LLMs) open up exciting research opportunities for architectural decision-making. The use of specialized LLMs can facilitate many of the discussed functionalities including generation of initial architectures and provider-specific code based on the user-provided requirements, search for suitable component hosting alternatives, or refinement of abstract models into technical deployment models for the chosen deployment automation engine.

BIBLIOGRAPHY

- [AHKB03] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, A. P. Barros. “Workflow Patterns”. In: *Distributed and Parallel Databases* 14 (July 2003), pp. 5–51 (cit. on pp. 54, 70).
- [AFHI21] C. Abad, I. T. Foster, N. Herbst, A. Iosup. “Serverless Computing (Dagstuhl Seminar 21201)”. In: *Dagstuhl Reports* 11.4 (2021). Ed. by C. Abad, I. T. Foster, N. Herbst, A. Iosup, pp. 34–93. ISSN: 2192-5283 (cit. on p. 49).
- [ÁCJ+16] E. Abraham, F. Corzilius, E. B. Johnsen, G. Kremer, J. Mauro. “Zephyrus2: on the fly deployment optimization using SMT and CP technologies”. In: *International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*. Springer. 2016, pp. 229–245 (cit. on p. 60).
- [AKR+19] A. P. Achilleos, K. Kritikos, A. Rossini, G. M. Kapitsaki, J. Domaschka, M. Orzechowski, D. Seybold, F. Griesinger, N. Nikolov, D. Romero, et al. “The cloud application modelling and execution language”. In: *Journal of Cloud computing* 8.1 (2019), pp. 1–25 (cit. on p. 61).
- [Ada17] Adam Wiggins. *The Twelve-Factor App*. 2017 (cit. on p. 55).

- [AC17] G. Adzic, R. Chatley. “Serverless Computing: Economic and Architectural Impact”. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 884–889. ISBN: 978-1-4503-5105-8 (cit. on p. 53).
- [ABMT17] Y. M. Afify, N. L. Badr, I. F. Moawad, M. F. Tolba. “Evaluation of cloud service ontologies”. In: *2017 Eighth International Conference on Intelligent Computing and Information Systems (ICICIS)*. IEEE. 2017, pp. 144–153 (cit. on p. 80).
- [AB14] A. Ahmad, M. A. Babar. “Towards a pattern language for self-adaptation of cloud-based architectures”. In: *Proceedings of the WICSA 2014 Companion Volume*. 2014, pp. 1–6 (cit. on p. 87).
- [AU22] A. Aho, J. Ullman. “Abstractions, Their Algorithms, and Their Compilers”. In: *Commun. ACM* 65.2 (Jan. 2022), pp. 76–91. ISSN: 0001-0782 (cit. on p. 64).
- [AIS77] C. Alexander, S. Ishikawa, M. Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, Aug. 1977 (cit. on pp. 41, 84, 92).
- [Ama22a] Amazon Web Services. *AWS Solutions Constructs patterns*. <https://aws.amazon.com/solutions/constructs/patterns>. 2022 (cit. on pp. 86, 88).
- [Ama22b] Amazon Web Services. *AWS Solutions Library*. <https://aws.amazon.com/solutions/browse-all>. 2022 (cit. on pp. 16, 53, 64, 99, 100, 102, 104–107, 109, 110, 112, 114, 115, 117, 120, 127, 133, 137–139, 141, 162, 224).
- [Ama22c] Amazon Web Services. *Serverless Application Repository*. <https://aws.amazon.com/serverless/serverlessrepo>. 2022 (cit. on p. 141).

-
- [Ama22d] Amazon Web Services, Inc. *Amazon States Language*. <https://states-language.net/spec>. 2022 (cit. on pp. 177, 220).
- [Ama22e] Amazon Web Services, Inc. *AWS Serverless Application Model*. Sept. 2022. URL: <https://aws.amazon.com/serverless/sam> (cit. on pp. 66, 188).
- [ABLS13] V. Andrikopoulos, T. Binz, F. Leymann, S. Strauch. “How to Adapt Applications for the Cloud Environment”. In: *Computing* 95.6 (June 2013), pp. 493–535 (cit. on p. 55).
- [AGLW14] V. Andrikopoulos, S. Gómez Sáez, F. Leymann, J. Wettinger. “Optimal distribution of applications in the cloud”. In: *International Conference on Advanced Information Systems Engineering*. Springer, 2014, pp. 75–90 (cit. on p. 61).
- [ARSL14] V. Andrikopoulos, A. Reuter, S. G. Sáez, F. Leymann. “A GENTL Approach for Cloud Application Topologies”. In: *Proceedings of the Third European Conference on Service-Oriented and Cloud Computing (ESOCC 2014)*. Lecture Notes in Computer Science (LNCS). Springer, Sept. 2014, pp. 1–11 (cit. on p. 65).
- [ASL13] V. Andrikopoulos, S. Strauch, F. Leymann. “Decision Support for Application Migration to the Cloud: Challenges and Vision”. In: *Proceedings of the 3rd International Conference on Cloud Computing and Service Science (CLOSER’13)*. SciTePress, May 2013, pp. 1–7 (cit. on p. 39).
- [AVS12] D. Androcec, N. Vrcek, J. Seva. “Cloud computing ontologies: A systematic review”. In: *Proceedings of the third international conference on models and ontology-based design of protocols, architectures and services*. 2012, pp. 9–14 (cit. on p. 80).

- [AIVP18] L. Ao, L. Izhikevich, G. M. Voelker, G. Porter. “Sprocket: A Serverless Video Processing Framework”. In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 263–274 (cit. on p. 53).
- [Apa22a] Apache Software Foundation. *Apache Kafka*. <https://kafka.apache.org>. 2022 (cit. on p. 138).
- [Apa22b] Apache Software Foundation. *Apache OpenWhisk*. <https://openwhisk.apache.org>. 2022 (cit. on pp. 53, 127, 133, 135).
- [Apa22c] Apache Software Foundation. *Apache OpenWhisk Composer*. <https://github.com/apache/openwhisk-composer>. 2022 (cit. on pp. 54, 139, 162, 179).
- [ALS+21] A. Arjona, P. G. López, J. Sampé, A. Slominski, L. Villard. “Triggerflow: Trigger-based orchestration of serverless workflows”. In: *Future Generation Computer Systems* 124 (2021), pp. 215–229. issn: 0167-739X (cit. on p. 72).
- [AEK+07] W. Arnold, T. Eilam, M. Kalantar, A. V. Konstantinou, A. A. Totok. “Pattern Based SOA Deployment”. In: *Proceedings of the Fifth International Conference on Service-Oriented Computing (ICSOC 2007)*. Springer, Sept. 2007, pp. 1–12 (cit. on pp. 86, 87).
- [ARI+18] T. Asghar, S. Rasool, M. Iqbal, Z. u. Qayyum, A. N. Mian, G. Ubakanma. “Feasibility of Serverless Cloud Services for Disaster Management Information Systems”. In: *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. June 2018, pp. 1054–1057 (cit. on p. 54).

-
- [AG17] M. Ast, M. Gaedke. “Self-contained Web Components Through Serverless Computing”. In: *Proceedings of the 2Nd International Workshop on Serverless Computing*. WoSC '17. New York, NY, USA: ACM, 2017, pp. 28–33 (cit. on p. 54).
- [BKJS21] A. F. Baarzi, G. Kesidis, C. Joe-Wong, M. Shahrads. “On Merits and Viability of Multi-Cloud Serverless”. In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '21. Association for Computing Machinery, 2021, pp. 600–608. ISBN: 9781450386388 (cit. on p. 76).
- [BA18] T. Back, V. Andrikopoulos. “Using a Microbenchmark to Compare Function as a Service Solutions”. In: *Service-Oriented and Cloud Computing*. Springer International Publishing, 2018, pp. 146–160. ISBN: 978-3-319-99819-0 (cit. on p. 84).
- [BCC+17] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, et al. “Serverless computing: Current trends and open problems”. In: *Research Advances in Cloud Computing*. Springer, 2017, pp. 1–20 (cit. on pp. 16, 22–24, 52, 53).
- [BCF+17] I. Baldini, P. Cheng, S. J. Fink, N. Mitchell, V. Muthusamy, R. Rabbah, P. Suter, O. Tardieu. “The Serverless Trilemma: Function Composition for Serverless Computing”. In: *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Onward! 2017. ACM, 2017, pp. 89–103 (cit. on p. 71).
- [BG21] D. Barcelona-Pons, P. García-López. “Benchmarking parallelism in FaaS platforms”. In: *Future Generation Computer Systems* 124 (2021), pp. 268–284. ISSN: 0167-739X (cit. on p. 84).

- [BSS+22] D. Barcelona-Pons, P. Sutra, M. Sánchez-Artigas, G. París, P. García-López. “Stateful Serverless Computing with Crucial”. In: *ACM Trans. Softw. Eng. Methodol.* 31.3 (Mar. 2022). ISSN: 1049-331X (cit. on p. 54).
- [BFG17] L. Baresi, D. Filgueira Mendonça, M. Garriga. “Empowering Low-Latency Applications Through a Serverless Edge Computing Architecture”. In: *Service-Oriented and Cloud Computing*. Springer International Publishing, 2017, pp. 196–210 (cit. on p. 54).
- [BCK21] L. Bass, P. Clements, R. Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, July 2021 (cit. on pp. 17, 55, 57, 58).
- [BSG+18] N. Bassiliades, M. Symeonidis, P. Gouvas, E. Kontopoulos, G. Meditskos, I. Vlahavas. “PaaSport semantic model: An ontology for a platform-as-a-service semantically interoperable marketplace”. In: *Data & Knowledge Engineering* 113 (2018), pp. 81–115 (cit. on p. 81).
- [BSM+17] N. Bassiliades, M. Symeonidis, G. Meditskos, E. Kontopoulos, P. Gouvas, I. Vlahavas. “A semantic recommendation algorithm for the PaaSport platform-as-a-service marketplace”. In: *Expert Systems with Applications* 67 (2017), pp. 203–227 (cit. on p. 81).
- [BZ08] I. Bayley, H. Zhu. “On the composition of design patterns”. In: *2008 The Eighth International Conference on Quality Software*. IEEE. 2008, pp. 27–36 (cit. on p. 120).
- [BEDM98] P. Benjamin, M. Erraguntla, D. Delen, R. Mayer. “Simulation modeling at multiple levels of abstraction”. In: *1998 Winter Simulation Conference. Proceedings (Cat. No. 98CH36274)*. Vol. 1. IEEE. 1998, pp. 391–398 (cit. on p. 74).

- [BBF+18] A. Bergmayr, U. Breitenbücher, N. Ferry, A. Rossini, A. Solberg, M. Wimmer, G. Kappel, F. Leymann. “A Systematic Review of Cloud Modeling Languages”. In: *ACM Computing Surveys (CSUR)* 51.1 (Feb. 2018), pp. 1–38 (cit. on pp. 18, 22, 56, 59, 65).
- [BBSR13] A. Beslic, R. Bendraou, J. Sopenal, J.-Y. Rigolet. “Towards a solution avoiding Vendor Lock-in to enable Migration Between Cloud Platforms”. In: *MDHPCL@ MoDELS*. Cite-seer. 2013, pp. 5–14 (cit. on p. 75).
- [BDR21] O. Bibartiu, F. Dürr, K. Rothermel. “Clams: A Cloud Application Modeling Solution”. In: *2021 IEEE International Conference on Services Computing (SCC)*. IEEE. 2021, pp. 1–10 (cit. on p. 88).
- [BBH+13] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, S. Wagner. “OpenTOSCA – A Runtime for TOSCA-based Cloud Applications”. In: *Proceedings of the 11th International Conference on Service-Oriented Computing (ICSOC 2013)*. Vol. 8274. LNCS. Springer, Dec. 2013, pp. 692–695 (cit. on pp. 63, 73, 205).
- [BBKL14] T. Binz, U. Breitenbücher, O. Kopp, F. Leymann. “Migration of enterprise applications to the cloud”. In: *it - Information Technology, Special Issue: Architecture of Web Application* 56.3 (May 2014), pp. 106–111 (cit. on p. 75).
- [BLS11] T. Binz, F. Leymann, D. Schumm. “CMotion: A Framework for Migration of Applications into and between Clouds”. In: *Proceedings of the 2011 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE Computer Society Conference Publishing Services, Dec. 2011 (cit. on p. 75).

- [BFFB21] A. Bocci, S. Forti, G.-L. Ferrari, A. Brogi. “Secure FaaS orchestration in the fog: how far are we?” In: *Computing* 103.5 (May 2021), pp. 1025–1056. ISSN: 0010-485X, 1436-5057 (cit. on p. 72).
- [BGF+22] A. Bocci, R. Guanciale, S. Forti, G.-L. Ferrari, A. Brogi. “Secure Partitioning of Composite Cloud Applications”. In: *Service-Oriented and Cloud Computing*. Vol. 13226. Series Title: Lecture Notes in Computer Science. Springer International Publishing, 2022, pp. 47–64 (cit. on p. 62).
- [BSNB20] M. Bogo, J. Soldani, D. Neri, A. Brogi. “Component-aware orchestration of cloud-based enterprise applications, from TOSCA to Docker and Kubernetes”. In: *Software: Practice and Experience* 50.9 (2020), pp. 1793–1821 (cit. on p. 64).
- [BDJ18] N. Bommadevara, A. Del Miglio, S. Jansen. “Cloud adoption to accelerate IT modernization”. In: *McKinsey Digital* (2018) (cit. on p. 15).
- [BO19] D. Bortolini, R. R. Obelheiro. “Investigating Performance and Cost in Function-as-a-Service Platforms”. In: *International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*. Springer. 2019, pp. 174–185 (cit. on pp. 83, 130).
- [BGS20] N. E. H. Bouzerzour, S. Ghazouani, Y. Slimani. “A survey on the service interoperability in cloud computing: Client-centric and provider-centric perspectives”. In: *Software: Practice and Experience* 50.7 (2020), pp. 1025–1060 (cit. on p. 53).
- [Box79] G. E. Box. “Robustness in the strategy of scientific model building”. In: *Robustness in statistics*. Elsevier, 1979, pp. 201–236 (cit. on p. 57).
- [BC10] J. Bozman, G. Chen. “Cloud computing: The need for portability and interoperability”. In: *IDC Executive Insights* (2010) (cit. on pp. 74, 75).

- [BE14] R. Braun, W. Esswein. “Classification of domain-specific BPMN extensions”. In: *IFIP Working Conference on The Practice of Enterprise Modeling*. Springer. 2014, pp. 42–57 (cit. on p. 170).
- [BSBE14] R. Braun, H. Schlieter, M. Burwitz, W. Esswein. “BPMN4CP: Design and implementation of a BPMN extension for clinical pathways”. In: *2014 IEEE international conference on bioinformatics and biomedicine (BIBM)*. IEEE. 2014, pp. 9–16 (cit. on p. 169).
- [BEK+16] U. Breitenbücher, C. Endres, K. Képes, O. Kopp, F. Leymann, S. Wagner, J. Wettinger, M. Zimmermann. “The OpenTOSCA Ecosystem - Concepts & Tools”. In: *European Space project on Smart Systems, Big Data, Future Internet - Towards Serving the Grand Societal Challenges - Volume 1: EPS Rome 2016* (Dec. 2016), pp. 112–130 (cit. on pp. 212, 213).
- [BCS17] A. Brogi, P. Cifariello, J. Soldani. “DrACO: Discovering available cloud offerings”. In: *Computer Science - Research and Development* 32.3 (2017), pp. 269–279. issn: 1865-2042 (cit. on p. 82).
- [BFI+15] A. Brogi, M. Fazzolari, A. Ibrahim, J. Soldani, J. Carasco, J. Cubo, F. Durán, E. Pimentel, E. Di Nitto, F. D Andria. “Adaptive management of applications across multiple clouds: The SeaClouds Approach”. In: *CLEI Electronic Journal* 18.1 (2015), pp. 1–14 (cit. on p. 82).
- [BRS18] A. Brogi, L. Rinaldi, J. Soldani. “TosKer: A synergy between TOSCA and Docker for orchestrating multicomponent applications”. In: *Software: Practice and Experience* 48.11 (2018), pp. 2061–2079 (cit. on p. 64).

- [BCG+22] S. Burckhardt, B. Chandramouli, C. Gillum, D. Justo, K. Kallas, C. McMahon, C.S. Meiklejohn, X. Zhu. “Netherite: Efficient Execution of Serverless Workflows”. In: *Proc. VLDB Endow.* 15.8 (Apr. 2022), pp. 1591–1604. issn: 2150-8097 (cit. on p. 73).
- [BPB+21] K. Burkat, M. Pawlik, B. Balis, M. Malawski, K. Vahi, M. Rynge, R. F. da Silva, E. Deelman. “Serverless Containers – Rising Viable Approach to Scientific Workflows”. In: *2021 IEEE 17th International Conference on eScience (eScience)*. Sept. 2021, pp. 40–49 (cit. on p. 73).
- [BO16] B. Burns, D. Oppenheimer. “Design patterns for container-based distributed systems”. In: *8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. 2016 (cit. on p. 100).
- [BHS07] F. Buschmann, K. Henney, D. C. Schmidt. *Pattern-oriented software architecture, on patterns and pattern languages*. Vol. 5. John wiley & sons, 2007 (cit. on pp. 92, 93, 121).
- [Cam22] Camunda. *BPMN Workflow Engine*. <https://camunda.com/products/camunda-platform/bpmn-engine>. 2022 (cit. on p. 70).
- [CLT20] M. Cankar, A. Luzar, D. A. Tamburri. “Auto-scaling Using TOSCA Infrastructure as Code”. In: *Software Architecture*. Vol. 1269. Series Title: Communications in Computer and Information Science. Springer International Publishing, 2020, pp. 260–268 (cit. on p. 69).
- [CAH+20] G. Casale, M. Artač, W.-J. van den Heuvel, A. van Hoorn, P. Jakovits, F. Leymann, M. Long, V. Papanikolaou, D. Prezenza, A. Russo, S. N. Srirama, D. A. Tamburri, M. Wurster, L. Zhu. “RADON: rational decomposition and orchestration for serverless computing”. In: *SICS Software-Intensive Cyber-Physical Systems* 35.1-2 (Aug. 2020), pp. 77–87 (cit. on pp. 48, 68).

-
- [CAH+19] G. Casale, M. Artač, W.-J. van den Heuvel, A. van Hoorn, P. Jakovits, F. Leymann, M. Long, V. Papanikolaou, D. Prezenza, A. Russo, S. N. Srirama, D. A. Tamburri, M. Wurster, L. Zhu. “RADON: Rational Decomposition and Orchestration for Serverless Computing”. In: *SICS Software-Intensive Cyber-Physical Systems* (Aug. 2019) (cit. on pp. 75, 214, 215).
- [CIMS19] P. Castro, V. Ishakian, V. Muthusamy, A. Slominski. “The Rise of Serverless Computing”. In: *Commun. ACM* 62.12 (Nov. 2019), pp. 44–54. ISSN: 0001-0782 (cit. on p. 54).
- [CBL+20] R. Chard, Y. Babuji, Z. Li, T. Skluzacek, A. Woodard, B. Blaiszik, I. Foster, K. Chard. “FuncX: A Federated Function Serving Fabric for Science”. In: *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*. HPDC ’20. Association for Computing Machinery, 2020, pp. 65–76. ISBN: 9781450370523 (cit. on p. 75).
- [CA20] R. Chatley, T. Allerton. “Nimbus: Improving the Developer Experience for Serverless Applications”. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. ICSE ’20. Association for Computing Machinery, 2020, pp. 85–88. ISBN: 9781450371223 (cit. on p. 78).
- [CT12] M. Chinosi, A. Trombetta. “BPMN: An introduction to the standard”. In: *Computer Standards & Interfaces* 34.1 (2012), pp. 124–134 (cit. on pp. 70, 170).
- [CLFG15] J. Cito, P. Leitner, T. Fritz, H. C. Gall. “The making of cloud applications: An empirical study on software development for the cloud”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 2015, pp. 393–403 (cit. on pp. 55, 56).

- [Clo18] Cloud Native Computing Foundation (CNCf). *CNCf Serverless Whitepaper v1.0*. Sept. 2018. URL: <https://github.com/cncf/wg-serverless/tree/master/whitepapers/serverless-overview> (cit. on pp. 16, 23, 52, 130).
- [Clo22a] Cloud Native Computing Foundation (CNCf). *CNCf Serverless Landscape*. Sept. 2022. URL: <https://landscape.cncf.io/serverless> (cit. on p. 66).
- [Clo22b] CloudEvents Authors 2022. *CloudEvents*. Sept. 2022. URL: <https://cloudevents.io> (cit. on pp. 30, 75).
- [Clo22c] Cloudstate. *Distributed State Management for Serverless*. <https://cloudstate.io>. 2022 (cit. on p. 73).
- [Col89] M. I. Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989 (cit. on p. 77).
- [COZ17] C. Combi, B. Oliboni, F. Zerbatò. “Modeling and Handling Duration Constraints in BPMN 2.0”. In: *Proceedings of the Symposium on Applied Computing*. Ed. by D. Shin, M. Lencastre. Vol. 2017 edition. SAC. Marrakech, Morocco. ACM, New York, NY, USA, 2017, pp. 727–734 (cit. on p. 70).
- [CFH+02] A. Conte, M. Fredj, I. Hassine, J.-P. Giraudin, D. Rieu. “A tool and a formalism to design and apply patterns”. In: *International conference on object-oriented information systems*. Springer. 2002, pp. 135–146 (cit. on p. 120).
- [Con20] Contributors to AWS Samples Repository. *Serverless Reference Architecture: Extract Transfer Load*. <https://github.com/aws-samples/aws-lambda-etl-ref-architecture>. 2020 (cit. on pp. 222, 223, 230).
- [Cop96] J. O. Coplien. *Software patterns*. 1996 (cit. on pp. 92, 93).

- [CDN+21] G. Cordasco, M. D’Auria, A. Negro, V. Scarano, C. Spagnuolo. “Toward a domain-specific language for scientific workflow-based applications on multicloud system”. In: *Concurrency and Computation: Practice and Experience* 33.18 (2021), e5802 (cit. on pp. 76, 79).
- [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*. Vol. 11997. Series Title: Lecture Notes in Computer Science. Springer International Publishing, 2020, pp. 531–544 (cit. on pp. 76, 79).
- [CL22] R. Cordingly, W. Lloyd. “FaaSSET: A Jupyter Notebook to Streamline Every Facet of Serverless Development”. In: *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*. ICPE ’22. New York, NY, USA: Association for Computing Machinery, 2022, pp. 49–52. ISBN: 9781450391597 (cit. on p. 78).
- [CH01] B. Councill, G. T. Heineman. “Definition of a software component and its elements”. In: *Component-based software engineering: putting the pieces together* (2001), pp. 5–19 (cit. on p. 94).
- [DDPT21] S. Dalla Palma, D. Di Nucci, F. Palomba, D. A. Tamburri. “Within-project defect prediction of infrastructure-as-code using product and process metrics”. In: *IEEE Transactions on Software Engineering* 48.6 (2021), pp. 2086–2104 (cit. on p. 239).
- [DGD+21] S. Dalla Palma, M. Garriga, D. Di Nucci, D. A. Tamburri, W.-J. Van Den Heuvel. “DevOps and Quality Management in Serverless Computing: The RADON Approach”. In: *Advances in Service-Oriented and Cloud Computing*. Vol. 1360. Springer International Publishing, 2021, pp. 155–160 (cit. on p. 68).

- [DBC+12] F. DAndria, S. Bocconi, J.G. Cruz, J. Ahtes, D. Zeginis. “Cloud4SOA: Multi-cloud Application Management Across PaaS Offerings”. In: *2012 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. 2012, pp. 407–414 (cit. on p. 82).
- [DKL22] J. Dantas, H. Khazaei, M. Litoiu. “Application Deployment Strategies for Reducing the Cold Start Delay of AWS Lambda”. In: *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*. July 2022, pp. 1–10 (cit. on p. 191).
- [DGRB15] A. V. Dastjerdi, S. K. Garg, O. F. Rana, R. Buyya. “Cloud-Pick: a framework for QoS-aware and ontology-based service deployment across clouds”. In: *Software: Practice and Experience* 45.2 (2015), pp. 197–231 (cit. on p. 82).
- [Dav19] C. Davis. *Cloud Native Patterns: Designing change-tolerant software*. Manning Pub., 2019. ISBN: 9781617294297 (cit. on p. 85).
- [DBK20] N. Daw, U. Bellur, P. Kulkarni. “Xanadu: Mitigating Cascading Cold Starts in Serverless Function Chain Deployments”. In: *Proceedings of the 21st International Middleware Conference*. Middleware ’20. Association for Computing Machinery, 2020, pp. 356–370. ISBN: 9781450381536 (cit. on p. 72).
- [DJS+20] C. Dehury, P. Jakovits, S. N. Srirama, V. Tountopoulos, G. Giotis. “Data Pipeline Architecture for Serverless Platform”. In: *Software Architecture*. Vol. 1269. Series Title: Communications in Computer and Information Science. Cham: Springer International Publishing, 2020, pp. 241–246 (cit. on p. 69).

- [DJS+22] C. K. Dehury, P. Jakovits, S. N. Srirama, G. Giotis, G. Garg. “TOSCAdata: Modeling data pipeline applications in TOSCA”. In: *Journal of Systems and Software* 186 (2022), p. 111164. ISSN: 0164-1212 (cit. on p. 69).
- [DKA+21] J. DesLauriers, T. Kiss, R. C. Ariyattu, H.-V. Dang, A. Ullah, J. Bowden, D. Krefting, G. Pierantoni, G. Terstyanszky. “Cloud apps to-go: Cloud portability with TOSCA and MiCADO”. In: *Concurrency and Computation: Practice and Experience* 33.19 (2021), e6093 (cit. on p. 64).
- [DMZZ14] R. Di Cosmo, J. Mauro, S. Zacchiroli, G. Zavattaro. “Aeolus: A component model for the cloud”. In: *Information and Computation* 239 (2014), pp. 100–121 (cit. on p. 60).
- [DLM19] P. Di Francesco, P. Lago, I. Malavolta. “Architecting with microservices: A systematic mapping study”. In: *Journal of Systems and Software* 150 (2019), pp. 77–97 (cit. on p. 18).
- [DCE14] B. Di Martino, G. Cretella, A. Esposito. “Towards a unified owl ontology of cloud vendors’ appliances and services at paas and saas level”. In: *2014 Eighth International Conference on Complex, Intelligent and Software Intensive Systems*. IEEE. 2014, pp. 570–575 (cit. on p. 81).
- [DCE17] B. Di Martino, G. Cretella, A. Esposito. “Cloud services composition through cloud patterns: a semantic-based approach”. In: *Soft Computing* 21.16 (2017), pp. 4557–4570 (cit. on p. 86).
- [DPC+10] B. Di Martino, D. Petcu, R. Cossu, P. Goncalves, T. Máhr, M. Loichate. “Building a Mosaic of Clouds”. In: *Proceedings of the 2010 Conference on Parallel Processing*. Euro-Par 2010. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 571–578 (cit. on p. 82).

- [DT15] G. Di Modica, O. Tomarchio. “Matching the business perspectives of providers and customers in future cloud markets”. In: *Cluster Computing* 18.1 (2015), pp. 457–475 (cit. on p. 80).
- [DMT15] DMTF. *Open Virtualization Format Specification (Version: 2.1.1)*. Distributed Management Task Force (DMTF). 2015 (cit. on p. 74).
- [DS19] S. Dorodko, J. Spillner. “Selective Java code transformation into AWS Lambda functions”. In: *ESSCA, Zurich, 21 December 2018*. CEUR Workshop Proceedings. 2019, pp. 9–17 (cit. on pp. 78, 79).
- [DJ13] R. Dukaric, M. B. Juric. “Towards a unified taxonomy and architecture of cloud frameworks”. In: *Future Generation Computer Systems* 29.5 (2013), pp. 1196–1210 (cit. on p. 80).
- [DHY+22] T. F. Düllmann, A. van Hoorn, V. Yussupov, P. Jakovits, M. Adhikari. “CTT: Load Test Automation for TOSCA-Based Cloud Applications”. In: *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*. Association for Computing Machinery, July 2022, pp. 89–96 (cit. on pp. 47, 69, 205, 231, 232).
- [ESE+21] S. Eismann, J. Scheuner, E. van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. L. Abad, A. Iosup. “Serverless Applications: Why, When, and How?” In: *IEEE Software* 38.1 (2021), pp. 32–39 (cit. on p. 53).
- [ESV+21] S. Eismann, J. Scheuner, E. Van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. Abad, A. Iosup. “The State of Serverless Applications: Collection, Characterization, and Community Consensus”. In: *IEEE Transactions on Software Engineering* (2021) (cit. on p. 53).

-
- [Eiv17] A. Eivy. “Be Wary of the Economics of SServerless”Cloud Computing”. In: *IEEE Cloud Computing* 4.2 (Mar. 2017), pp. 6–12. ISSN: 2325-6095 (cit. on p. 54).
- [ESNA18] T. Elgamal, A. Sandur, K. Nahrstedt, G. Agha. “Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement”. In: *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. 2018, pp. 300–312 (cit. on p. 66).
- [EJBE19] A. Elhabbash, A. Jumagaliyev, G. S. Blair, Y. Elkhatib. “SLO-ML: A Language for Service Level Objective Modelling in Multi-Cloud Applications”. In: *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*. UCC’19. Association for Computing Machinery, 2019, pp. 241–250. ISBN: 9781450368940 (cit. on p. 76).
- [EBF+17] C. Endres, U. Breitenbücher, M. Falkenthal, O. Kopp, F. Leymann, J. Wettinger. “Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications”. In: *Proceedings of the 9th International Conference on Pervasive Patterns and Applications (PATTERNS 2017)*. Xpert Publishing Services, Feb. 2017, pp. 22–27 (cit. on pp. 64, 85, 92, 110, 112, 115, 117, 120, 125, 141).
- [Erl+15] T. Erl et al. *Cloud Computing Design Patterns*. 1st. Prentice Hall Press, 2015. ISBN: 0133858561 (cit. on p. 85).
- [EGE+19] E. van Eyk, J. Grohmann, S. Eismann, A. Bauer, L. Versluis, L. Toader, N. Schmitt, N. Herbst, C. L. Abad, A. Iosup. “The SPEC-RG Reference Architecture for FaaS: From Microservices and Containers to Serverless Platforms”. In: *IEEE Internet Computing* 23.6 (Nov. 2019), pp. 7–18. ISSN: 1941-0131 (cit. on p. 54).
- [EIA+18] E. van Eyk, A. Iosup, C. L. Abad, J. Grohmann, S. Eismann. “A SPEC RG Cloud Group’s Vision on the Performance Challenges of FaaS Cloud Architectures”. In: *Companion*

- of the 2018 ACM/SPEC International Conference on Performance Engineering. ICPE '18. ACM, 2018, pp. 21–24. ISBN: 978-1-4503-5629-9 (cit. on p. 55).*
- [FG13] D. Fahland, C. Gierds. “Analyzing and completing middleware designs for enterprise integration using coloured petri nets”. In: *International Conference on Advanced Information Systems Engineering*. Springer. 2013, pp. 400–416 (cit. on p. 87).
- [FBD+20] G. Falazi, U. Breitenbücher, F. Daniel, F. Lamparelli, F. Leymann, V. Yussupov. “Smart Contract Invocation Protocol (SCIP): A Protocol for the Uniform Integration of Heterogeneous Blockchain Smart Contracts”. In: *Proceedings of the 32nd Conference on Advanced Information Systems Engineering (CAiSE 2020)*. Vol. 12127. Lecture Notes in Computer Science. Springer International Publishing, June 2020, pp. 134–149 (cit. on p. 48).
- [FBF+18] G. Falazi, U. Breitenbücher, M. Falkenthal, L. Harzenetter, F. Leymann, V. Yussupov. “Blockchain-based Collaborative Development of Application Deployment Models”. In: *On the Move to Meaningful Internet Systems: OTM 2018 Conferences (CoopIS 2018)*. Springer International Publishing, Oct. 2018, pp. 40–60 (cit. on p. 48, 214).
- [FHBL19] G. Falazi, M. Hahn, U. Breitenbücher, F. Leymann. “Modeling and execution of blockchain-aware business processes”. In: *SICS Software-Intensive Cyber-Physical Systems* 34.2 (Feb. 2019), pp. 105–116 (cit. on p. 71).
- [FHB+19] G. Falazi, M. Hahn, U. Breitenbücher, F. Leymann, V. Yussupov. “Process-Based Composition of Permissioned and Permissionless Blockchain Smart Contracts”. In: *Proceedings of the 2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC 2019)*. IEEE, Oct. 2019, pp. 77–87 (cit. on p. 48).

-
- [FBB+14a] M. Falkenthal, J. Barzen, U. Breitenbücher, C. Fehling, F. Leymann. “Efficient Pattern Application: Validating the Concept of Solution Implementations in Different Domains”. In: *International Journal On Advances in Software 7.3&4* (Dec. 2014), pp. 710–726 (cit. on p. 87).
- [FBB+14b] M. Falkenthal, J. Barzen, U. Breitenbücher, C. Fehling, F. Leymann. “From Pattern Languages to Solution Implementations”. In: *Proceedings of the Sixth International Conferences on Pervasive Patterns and Applications (PATTERNS 2014)*. Xpert Publishing Services, May 2014, pp. 12–21 (cit. on pp. 86, 191).
- [FBB+15] M. Falkenthal, J. Barzen, U. Breitenbücher, C. Fehling, F. Leymann, A. Hadjakos, F. Hentschel, H. Schulze. “Leveraging Pattern Application via Pattern Refinement”. In: *Proceedings of the International Conference on Pursuit of Pattern Languages for Societal Change (PURPLSOC 2015)*. epubli, June 2015, pp. 38–61 (cit. on p. 86).
- [FBL18] M. Falkenthal, U. Breitenbücher, F. Leymann. “The Nature of Pattern Languages”. In: *Pursuit of Pattern Languages for Societal Change*. tredition, Oct. 2018, pp. 130–150 (cit. on pp. 84, 120).
- [Feh15] C. Fehling. “Cloud computing patterns : identification, design, and application”. Dissertation. Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Nov. 2015, p. 270 (cit. on pp. 84, 92, 93).
- [FBBL14] C. Fehling, J. Barzen, U. Breitenbücher, F. Leymann. “A Process for Pattern Identification, Authoring, and Application”. In: *Proceedings of the 19th European Conference on Pattern Languages of Programs (EuroPLoP 2014)*. ACM, Jan. 2014 (cit. on p. 92).

- [FEL+12] C. Fehling, T. Ewald, F. Leymann, M. Pauly, J. Rütschlin, D. Schumm. “Capturing Cloud Computing Knowledge and Experience in Patterns”. In: *Proceedings of the 5th IEEE International Conference on Cloud Computing (CLOUD 2012)*. IEEE, June 2012, pp. 726–733 (cit. on p. 92).
- [FLR+11] C. Fehling, F. Leymann, R. Retter, D. Schumm, W. Schupeck. “An Architectural Pattern Language of Cloud-based Applications”. In: *Proceedings of the 18th Conference on Pattern Languages of Programs (PLoP 2011)*. ACM, Oct. 2011 (cit. on p. 87).
- [FLR+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, Jan. 2014, p. 367 (cit. on pp. 16, 55, 85, 86, 92, 100, 102, 105, 107, 110, 112, 115, 125, 236).
- [FLR+13] C. Fehling, F. Leymann, S. T. Ruehl, M. Rudek, S. Verclas. “Service Migration Patterns – Decision Support and Best Practices for the Migration of Existing Service-Based Applications to Cloud Environments”. In: *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications*. 2013, pp. 9–16 (cit. on p. 85).
- [FLRS12] C. Fehling, F. Leymann, J. Rütschlin, D. Schumm. “Pattern-Based Development and Management of Cloud Applications”. In: *Future Internet 4.1* (Mar. 2012), pp. 110–141 (cit. on p. 87).
- [FKDH18] L. Feng, P. Kudva, D. Da Silva, J. Hu. “Exploring Serverless Computing for Neural Network Training”. In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. 2018, pp. 334–341 (cit. on p. 53).

- [FCS+18] N. Ferry, F. Chauvel, H. Song, A. Rossini, M. Lushpenko, A. Solberg. “CloudMF: Model-driven management of multi-cloud applications”. In: *ACM Transactions on Internet Technology (TOIT)* 18.2 (2018), pp. 1–24 (cit. on p. 61).
- [FDS22] N. Ferry, R. Dautov, H. Song. “Towards a Model-Based Serverless Platform for the Cloud-Edge-IoT Continuum”. In: *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. May 2022, pp. 851–858 (cit. on p. 68).
- [FGZ+18] K. Figiela, A. Gajek, A. Zima, B. Obrok, M. Malawski. “Performance evaluation of heterogeneous cloud functions”. In: *Concurrency and Computation: Practice and Experience* 30.23 (2018) (cit. on p. 84).
- [Fle22] Flexera. *State of the Cloud Report*. Mar. 2022 (cit. on p. 15).
- [Flo08] L. Floridi. “The method of levels of abstraction”. In: *Minds and machines* 18.3 (2008), pp. 303–329 (cit. on pp. 43, 74, 90, 195).
- [Fow02] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, Nov. 2002 (cit. on p. 84).
- [Fow10] M. Fowler. *Domain-specific languages*. Pearson Education, 2010 (cit. on p. 59).
- [FIMS17] G. C. Fox, V. Ishakian, V. Muthusamy, A. Slominski. “Status of serverless computing and function-as-a-service (faas) in industry and research”. In: *arXiv preprint arXiv:1708.08028* (2017) (cit. on pp. 23, 52).
- [FCSK03] R. France, S. Chosh, E. Song, D.-K. Kim. “A metamodeling approach to pattern-based model refactoring”. In: *IEEE software* 20.5 (2003), pp. 52–58 (cit. on p. 86).

- [FH11] S. Frey, W. Hasselbring. “The cloudmig approach: Model-based migration of software systems to cloud-optimized applications”. In: *International Journal on Advances in Software* 4.3 and 4 (2011), pp. 342–353 (cit. on p. 75).
- [GGL+19] M. Gabbrielli, S. Giallorenzo, I. Lanese, F. Montesi, M. Peressotti, S. P. Zingaro. “No More, No Less”. In: *Coordination Models and Languages*. Springer International Publishing, 2019, pp. 148–157 (cit. on p. 71).
- [GM05] A. Gaffar, N. Moha. “Semantics of a pattern system”. In: *STEP 2005* (2005), p. 211 (cit. on p. 122).
- [Gam+95] E. Gamma et al. “Elements of reusable object-oriented software”. In: *Addison-Wesley* (1995) (cit. on pp. 84, 86, 92).
- [GFE+20] F. Gand, I. Fronza, N. El Ioini, H. R. Barzegar, C. Pahl. “Serverless Container Cluster Management for Lightweight Edge Clouds”. In: *Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER 2020)*. SciTePress, 2020, pp. 302–311 (cit. on pp. 83, 130).
- [GSP+18] P. García López, M. Sánchez-Artigas, G. París, D. Barcelona Pons, Á. Ruiz Ollobarren, D. Arroyo Pinto. “Comparison of FaaS Orchestration Systems”. In: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. Dec. 2018, pp. 148–153 (cit. on pp. 16, 26, 27, 54, 83, 130, 139).
- [GAO94] D. Garlan, R. Allen, J. Ockerbloom. “Exploiting style in architectural design environments”. In: *ACM SIGSOFT software engineering notes* 19.5 (1994), pp. 175–188 (cit. on p. 58).
- [GMW10] D. Garlan, R. Monroe, D. Wile. “Acme: An architecture description interchange language”. In: *CASCON First Decade High Impact Papers*. 2010, pp. 159–173 (cit. on p. 58).

- [GFM19] V. Garousi, M. Felderer, M. V. Mäntylä. “Guidelines for including grey literature and conducting multivocal literature reviews in software engineering”. In: *Information and Software Technology* 106 (2019), pp. 101–121 (cit. on p. 130).
- [Ger19] N. Gerasimov. “The DSL for composing functions for FaaS platform”. In: *Fourth Conference on Software Engineering and Information Management (SEIM-2019)(full papers)*. 2019, p. 13 (cit. on p. 71).
- [GGB20] D. Gesvindr, O. Gasior, B. Buhnova. “Architecture design evaluation of PaaS cloud applications using generated prototypes: PaaSArch Cloud Prototyper tool”. In: *Journal of Systems and Software* 169 (2020). ISSN: 0164-1212 (cit. on p. 60).
- [GLM+20] S. Giallorenzo, I. Lanese, F. Montesi, D. Sangiorgi, S. P. Zingaro. “The servers of serverless computing: a formal revisitation of functions as a service”. In: *Recent Developments in the Design and Implementation of Programming Languages*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2020 (cit. on p. 71).
- [GHZ+20] A. U. Gias, A. van Hoorn, L. Zhu, G. Casale, T. F. Düllmann, M. Wurster. “Performance Engineering for Microservices and Serverless Applications: The RADON Approach”. In: *Companion of the ACM/SPEC International Conference on Performance Engineering*. ICPE ’20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 46–49 (cit. on p. 69).
- [GMC19] V. Giménez-Alventosa, G. Moltó, M. Caballer. “A framework and a performance assessment for serverless MapReduce on AWS Lambda”. In: *Future Generation Computer Systems* 97 (2019), pp. 259–274. ISSN: 0167-739X (cit. on p. 53).

- [Goo22] Google. *Google Cloud products*. <https://cloud.google.com/products/>. 2022 (cit. on pp. 64, 105, 110, 117, 120, 127, 133).
- [Gre97] S. M. Greenstein. “Lock-in and the costs of switching main-frame computer vendors: What do buyers see?” In: *Industrial and Corporate Change* 6.2 (1997), pp. 247–273 (cit. on p. 56).
- [GJGN13] S. Gudenkauf, M. Josefiok, A. Göring, O. Norkus. “A reference architecture for cloud service offers”. In: *2013 17th IEEE International Enterprise Distributed Object Computing Conference*. IEEE. 2013, pp. 227–236 (cit. on p. 80).
- [GSN+19] R. Guidotti, J. Soldani, D. Neri, A. Brogi, D. Pedreschi. “Helping Your Docker Images to Spread Based on Explainable Models”. In: *Machine Learning and Knowledge Discovery in Databases*. Ed. by U. Brefeld, E. Curry, E. Daly, B. MacNamee, A. Marascu, F. Pinelli, M. Berlingerio, N. Hurley. Springer International Publishing, 2019, pp. 205–221 (cit. on p. 135).
- [GL19] J. Guth, F. Leymann. “Pattern-based rewrite and refinement of architectures using graph theory”. In: *Software-Intensive Cyber-Physical Systems (SICS)* (Aug. 2019), pp. 1–12 (cit. on p. 87).
- [HBLW17] M. Hahn, U. Breitenbücher, F. Leymann, A. Weiß. “TraDE - A Transparent Data Exchange Middleware for Service Choreographies”. In: *On the Move to Meaningful Internet Systems. OTM 2017 Conferences: Confederated International Conferences: CoopIS, C&TC, and ODBASE 2017, Rhodes, Greece, October 23-27, 2017, Proceedings, Part I*. Ed. by H. Panetto, C. Debruyne, W. Gaaloul, M. Papazoglou, A. Paschke, C. A. Ardagna, R. Meersman.

-
- Vol. 10573. Lecture Notes in Computer Science. Springer International Publishing, Oct. 2017, pp. 252–270 (cit. on p. 69).
- [HBL+18] M. Hahn, U. Breitenbücher, F. Leymann, M. Wurster, V. Yussupov. “Modeling Data Transformations in Data-Aware Service Choreographies”. In: *Proceedings of the IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC 2018)*. IEEE Computer Society, Oct. 2018, pp. 28–34 (cit. on pp. 47, 191).
- [HBLY18] M. Hahn, U. Breitenbücher, F. Leymann, V. Yussupov. “Transparent Execution of Data Transformations in Data-Aware Service Choreographies”. In: *On the Move to Meaningful Internet Systems. OTM 2018 Conferences (CoopIS 2018)*. Vol. 11230. Lecture Notes in Computer Science. Springer International Publishing AG, Oct. 2018, pp. 117–137 (cit. on p. 47).
- [HS10] T. Han, K. M. Sim. “An ontology-enhanced cloud service discovery system”. In: *Proceedings of the International MultiConference of Engineers and Computer Scientists*. Vol. 1. 2010. 2010, pp. 17–19 (cit. on p. 80).
- [HBB+21] L. Harzenetter, T. Binz, U. Breitenbücher, F. Leymann, M. Wurster. “Automated Generation of Management Workflows for Running Applications by Deriving and Enriching Instance Models”. In: *Proceedings of the 11th International Conference on Cloud Computing and Services Science (CLOSER 2021)*. SciTePress, May 2021, pp. 99–110 (cit. on p. 238).
- [HBF+18] L. Harzenetter, U. Breitenbücher, M. Falkenthal, J. Guth, C. Krieger, F. Leymann. “Pattern-based Deployment Models and Their Automatic Execution”. In: *11th IEEE/ACM*

- International Conference on Utility and Cloud Computing (UCC 2018)*. IEEE Computer Society, Dec. 2018, pp. 41–52 (cit. on pp. 87, 88, 203–205, 207–209, 214, 217).
- [HBF+20] L. Harzenetter, U. Breitenbücher, M. Falkenthal, J. Guth, F. Leymann. “Pattern-based Deployment Models Revisited: Automated Pattern-driven Deployment Configuration”. In: *Proceedings of the Twelfth International Conference on Pervasive Patterns and Applications (PATTERNS 2020)*. Xpert Publishing Services, Oct. 2020, pp. 40–49. ISBN: 978-1-61208-783-2 (cit. on pp. 87, 88, 205, 207–209, 214, 217).
- [HBKL19] L. Harzenetter, U. Breitenbücher, K. Képes, F. Leymann. “Freezing and Defrosting Cloud Applications: Automated Saving and Restoring of Running Applications”. In: *Software-Intensive Cyber-Physical Systems (SICS) 35* (Aug. 2019), pp. 101–114 (cit. on p. 238).
- [Has22] HashiCorp. *Terraform*. <https://www.terraform.io>. 2022 (cit. on pp. 64, 65).
- [HFG+18] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, C. Wu. “Serverless computing: One step forward, two steps back”. In: *arXiv preprint arXiv:1812.03651* (2018) (cit. on pp. 23, 26, 54).
- [HKR13] N. R. Herbst, S. Kounev, R. Reussner. “Elasticity in cloud computing: What it is, and what it is not”. In: *10th international conference on autonomic computing (ICAC 13)*. 2013, pp. 23–27 (cit. on p. 52).
- [Hil09] D. Hilley. *Cloud computing: A taxonomy of platform and infrastructure-level offerings*. Tech. rep. Georgia Institute of Technology, 2009 (cit. on p. 81).
- [HK10] C. N. Hoefer, G. Karagiannis. “Taxonomy of cloud computing services”. In: *2010 IEEE Globecom Workshops*. IEEE. 2010, pp. 1345–1350 (cit. on p. 56).

- [HK11] C. Höfer, G. Karagiannis. “Cloud computing services: taxonomy and comparison”. In: *Journal of Internet Services and Applications 2.2* (2011), pp. 81–94 (cit. on p. 80).
- [Hoh19] G. Hohpe. *Don’t get locked up into avoiding lock-in*. Sept. 2019. URL: <https://martinfoowler.com/articles/oss-lockin.html> (cit. on pp. 29, 33, 56).
- [Hoh20] G. Hohpe. *The Software Architect Elevator: Transforming Enterprises with Technology and Business Architecture*. O’Reilly Media, Inc., Apr. 2020 (cit. on pp. 17, 74).
- [Hoh22a] G. Hohpe. *Cloud Strategy: A Decision-based Approach to Successful Cloud Migration*. Leanpub, Feb. 2022 (cit. on pp. 15, 17).
- [Hoh22b] G. Hohpe. *Concerned about Serverless Lock-in? Consider Patterns!* Aug. 2022. URL: <https://architectelevator.com/cloud/serverless-design-patterns> (cit. on pp. 54, 79).
- [HW04a] G. Hohpe, B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2004 (cit. on pp. 84, 87, 236).
- [HW04b] G. Hohpe, B. Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional, 2004 (cit. on p. 94).
- [Hon+18] S. Hong et al. “Go Serverless: Securing Cloud via Serverless Design Patterns”. In: *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*. 2018 (cit. on p. 85).
- [HF10] J. Humble, D. Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010 (cit. on p. 56).

- [IBM21] IBM. *IBM Cloud Solutions*. <https://www.ibm.com/cloud/solutions>. 2021 (cit. on pp. 53, 100, 102, 107, 110, 112, 120, 138, 163).
- [ILM+20] J. Ichnowski, W. Lee, V. Murta, S. Paradis, R. Alterovitz, J. E. Gonzalez, I. Stoica, K. Goldberg. “Fog Robotics Algorithms for Distributed Motion Planning Using Lambda Serverless Computing”. In: *2020 IEEE International Conference on Robotics and Automation (ICRA)*. 2020, pp. 4232–4238 (cit. on p. 54).
- [Igu22] Iguazio. *Nuclio*. <https://nuclio.io>. 2022 (cit. on p. 133).
- [INS+14] C. Inzinger, S. Nastic, S. Sehic, M. Vögler, F. Li, S. Dustdar. “MADCAT: A methodology for architecture and deployment of cloud application topologies”. In: *2014 IEEE 8th international symposium on service oriented system engineering*. IEEE. 2014, pp. 13–22 (cit. on p. 61).
- [Iro22] Iron.io. *Iron Worker*. www.iron.io/worker. 2022 (cit. on p. 117).
- [IMS18] V. Ishakian, V. Muthusamy, A. Slominski. “Serving Deep Learning Models in a Serverless Platform”. In: *2018 IEEE International Conference on Cloud Engineering (IC2E)*. Apr. 2018, pp. 257–262 (cit. on p. 53).
- [IB18] Y. Izrailevsky, C. Bell. “Cloud Reliability”. In: *IEEE Cloud Computing* 5.3 (2018), pp. 39–44 (cit. on p. 52).
- [Jam+15] P. Jamshidi et al. “Cloud Migration Patterns: A Multi-cloud Service Architecture Perspective”. In: *Service-Oriented Computing - ICSOC 2014 Workshops*. Cham: Springer International Publishing, 2015, pp. 6–19 (cit. on p. 85).
- [Jam+17] P. Jamshidi et al. “Pattern-based multi-cloud architecture migration”. In: *Software: Practice and Experience* 47.9 (2017), pp. 1159–1184 (cit. on p. 85).

-
- [JPBG19] A. Jangda, D. Pinckney, Y. Brun, A. Guha. “Formal Foundations of Serverless Computing”. In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019) (cit. on p. 71).
- [JLZ17] Q. Jiang, Y. C. Lee, A. Y. Zomaya. “Serverless Execution of Scientific Workflows”. In: *Service-Oriented Computing*. Springer International Publishing, 2017, pp. 706–721. ISBN: 978-3-319-69035-3 (cit. on p. 72).
- [JFCG21] A. Jindal, J. Frielinghaus, M. Chadha, M. Gerndt. “Courier: Delivering Serverless Functions within Heterogeneous FaaS Deployments”. In: *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing*. UCC ’21. Association for Computing Machinery, 2021. ISBN: 9781450385640 (cit. on p. 75).
- [JGC+21] A. Jindal, M. Gerndt, M. Chadha, V. Podolskiy, P. Chen. “Function delivery network: Extending serverless computing for heterogeneous platforms”. In: *Software: Practice and Experience* 51.9 (2021), pp. 1936–1963 (cit. on p. 76).
- [JAM+19] A. John, K. Ausmees, K. Muenzen, C. Kuhn, A. Tan. “SWEEP: Accelerating Scientific Research Through Scalable Serverless Workflows”. In: *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing Companion*. UCC ’19 Companion. Association for Computing Machinery, 2019, pp. 43–50. ISBN: 9781450370448 (cit. on p. 73).
- [JSS+19] E. Jonas, J. Schleier-Smith, V. Sreekanti, C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. J. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, D. A. Patterson. “Cloud Programming Simplified: A Berkeley View on Serverless Computing”. In: *CoRR* abs/1902.03383 (2019). arXiv: [1902.03383](https://arxiv.org/abs/1902.03383) (cit. on p. 83).

- [JMK+13] G. Jung, T. Mukherjee, S. Kunde, H. Kim, N. Sharma, F. Goetz. “CloudAdvisor: A Recommendation-as-a-Service Platform for Cloud Configuration and Pricing”. In: *2013 IEEE Ninth World Congress on Services*. 2013, pp. 456–463 (cit. on p. 82).
- [KSHD13] S. Kächele, C. Spann, F. J. Hauck, J. Domaschka. “Beyond IaaS and PaaS: An extended cloud taxonomy for computation, storage and networking”. In: *2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*. IEEE. 2013, pp. 75–82 (cit. on p. 80).
- [KS19] D. Kalnauz, V. Speranskiy. “Productivity Estimation of Serverless Computing”. In: *Applied Aspects of Information Technology* 1 (2019), pp. 20–28 (cit. on pp. 83, 130).
- [KY17] A. Kanso, A. Youssef. “Serverless: Beyond the Cloud”. In: *Proceedings of the 2Nd International Workshop on Serverless Computing*. WoSC ’17. New York, NY, USA: ACM, 2017, pp. 6–10 (cit. on p. 54).
- [Kap19] A. Kaplunovich. “ToLambda: Automatic Path to Serverless Architectures”. In: *Proceedings of the 3rd International Workshop on Refactoring*. IWOR ’19. IEEE Press, 2019, pp. 1–8 (cit. on p. 78).
- [Kav14] M. J. Kavis. *Architecting the cloud: design decisions for cloud computing service models (SaaS, PaaS, and IaaS)*. John Wiley & Sons, 2014 (cit. on p. 52).
- [KZSB21] S. Kehrler, D. Zietlow, J. Scheffold, W. Blochinger. “Self-tuning serverless task farming using proactive elasticity control”. In: *Cluster Computing* 24.2 (June 2021), pp. 799–817. ISSN: 1386-7857, 1573-7543 (cit. on pp. 77, 79).
- [Ken02] S. Kent. “Model Driven Engineering”. In: *Integrated Formal Methods*. Springer Berlin Heidelberg, 2002, pp. 286–298 (cit. on p. 57).

-
- [KB13] B. Kitchenham, P. Brereton. “A systematic review of systematic review process research in software engineering”. In: *Information and software technology* 55.12 (2013), pp. 2049–2075 (cit. on p. 18).
- [KTS21] R. Klingler, N. Trifunovic, J. Spillner. “Beyond CloudFunction: Powerful Code Annotations to Capture Serverless Runtime Patterns”. In: *Proceedings of the Seventh International Workshop on Serverless Computing (WoSC7) 2021*. 2021, pp. 23–28 (cit. on p. 78).
- [Kol19] S. Kolb. “On the Portability of Applications in Platform as a Service”. PhD thesis. University of Bamberg, Germany, 2019. ISBN: 978-3-86309-631-1 (cit. on pp. 41, 80, 81, 128, 129, 150, 151, 155).
- [KBBL13] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann. “Winery – A Modeling Tool for TOSCA-based Cloud Applications”. In: *Proceedings of the 11th International Conference on Service-Oriented Computing (ICSOC 2013)*. Springer, Dec. 2013, pp. 700–704 (cit. on pp. 63, 197, 203, 212, 213, 238).
- [KKL08] O. Kopp, R. Khalaf, F. Leymann. “Deriving Explicit Data Links in WS-BPEL Processes”. In: *Proceedings of the International Conference on Services Computing (SCC 2008)*. IEEE, July 2008, pp. 367–376 (cit. on p. 165).
- [KLW11] O. Kopp, F. Leymann, S. Wagner. “Modeling Choreographies: BPMN 2.0 versus BPEL-based Approaches”. In: *Enterprise Modelling and Information Systems Architectures - EMISA 2011*. Lecture Notes in Informatics. Gesellschaft für Informatik e.V. (GI), Sept. 2011 (cit. on p. 69).
- [KAC+22] G. Kousiouris, S. Ambroziak, D. Costantino, S. Tsarsitalidis, E. Boutas, A. Mamelli, T. Stamati. “Combining NodeRED and Openwhisk for Pattern-based Development and Execution of Complex FaaS Workflows”. In: *arXiv preprint arXiv:2202.09683* (2022) (cit. on p. 73).

- [KGTS22] G. Kousiouris, C. Giannakos, K. Tserpes, T. Stamati. “Measuring Baseline Overheads in Different Orchestration Mechanisms for Large FaaS Workflows”. In: *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering. ICPE '22*. Association for Computing Machinery, 2022, pp. 61–68. ISBN: 9781450391597 (cit. on p. 73).
- [KBF+20] C. Krieger, U. Breitenbücher, M. Falkenthal, F. Leymann, V. Yussupov, U. Zdun. “Monitoring Behavioral Compliance with Architectural Patterns Based on Complex Event Processing”. In: *Proceedings of the 8th European Conference on Service-Oriented and Cloud Computing (ESOCC 2020)*. Springer International Publishing, Mar. 2020, pp. 125–140 (cit. on pp. 48, 238).
- [KS18] K. Kritikos, P. Skrzypek. “A Review of Serverless Frameworks”. In: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. Dec. 2018, pp. 161–168 (cit. on pp. 82, 83, 130).
- [KSMM19] K. Kritikos, P. Skrzypek, A. Moga, O. Matei. “Towards the Modelling of Hybrid Cloud Applications”. In: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. July 2019, pp. 291–295 (cit. on p. 67).
- [KZI+19] K. Kritikos, C. Zeginis, J. Iranzo, R. S. Gonzalez, D. Seybold, F. Griesinger, J. Domaschka. “Multi-cloud provisioning of business processes”. In: *Journal of Cloud Computing* 8.1 (Dec. 2019), p. 18. ISSN: 2192-113X (cit. on p. 73).
- [Kru04] P. Kruchten. “An ontology of architectural design decisions in software intensive systems”. In: *2nd Groningen workshop on software variability*. Groningen, The Netherlands. 2004, pp. 54–61 (cit. on p. 151).

- [KWB+19] J. Kuhlenkamp, S. Werner, M. C. Borges, K. El Tal, S. Tai. “An Evaluation of FaaS Platforms as a Foundation for Serverless Big Data Processing”. In: *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*. Association for Computing Machinery, 2019, pp. 1–9 (cit. on p. 84).
- [Küh06] T. Kühne. “Matters of (meta-) modeling”. In: *Software & Systems Modeling* 5.4 (2006), pp. 369–385 (cit. on p. 59).
- [KP10] K. Kumar, T. V. Prabhakar. “Design decision topology model for pattern relationship analysis”. In: *AsianPLoP '10*. 2010 (cit. on p. 122).
- [Kum19] M. Kumar. “Serverless Architectures Review, Future Trend and the Solutions to Open Problems”. In: *American Journal of Software Engineering* 6.1 (2019), pp. 1–10 (cit. on pp. 83, 130).
- [KQR+22] I. Kumara, G. Quattrocchi, D. Radolović, K. Tokmakov, J. R. Rivas, W.-J. Van Den Heuvel. “The SODALITE Runtime Environment”. In: *Deployment and Operation of Complex Software in Heterogeneous Execution Environments*. Ed. by E. Di Nitto, J. Gorroñoigoitia Cruz, I. Kumara, D. Radolović, K. Tokmakov, Z. Vasileiou. Series Title: SpringerBriefs in Applied Sciences and Technology. Springer International Publishing, 2022, pp. 67–92 (cit. on p. 64).
- [KT14] Y.-W. Kwon, E. Tilevich. “Cloud refactoring: automated transitioning to cloud-based services”. In: *Automated Software Engineering* 21.3 (2014), pp. 345–372 (cit. on p. 186).
- [LPJ10] M. M. Lankhorst, H. A. Proper, H. Jonkers. “The anatomy of the archimate language”. In: *International Journal of Information System Modeling and Design (IJISMD)* 1.1 (2010), pp. 1–32 (cit. on p. 58).

- [LW07] K.-K. Lau, Z. Wang. “Software component models”. In: *IEEE Transactions on software engineering* 33.10 (2007), pp. 709–724 (cit. on p. 94).
- [Lau04] A. M. S. Laurent. *Understanding Open Source and Free Software Licensing*. O’Reilly Media, Inc., 2004 (cit. on p. 133).
- [LSF18] H. Lee, K. Satyam, G. Fox. “Evaluation of Production Serverless Computing Environments”. In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. July 2018, pp. 442–450 (cit. on pp. 83, 130).
- [LHB18] S. Lehrig, M. Hilbrich, S. Becker. “The architectural template method: templating architectural knowledge to efficiently conduct quality-of-service analyses”. In: *Software: Practice and Experience* 48.2 (2018), pp. 268–299 (cit. on p. 59).
- [LMM18] J. Lehvä, N. Mäkitalo, T. Mikkonen. “Case Study: Building a Serverless Messenger Chatbot”. In: *Current Trends in Web Engineering*. Ed. by I. Garrigós, M. Wimmer. Cham: Springer International Publishing, 2018, pp. 75–86 (cit. on p. 53).
- [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 (cit. on pp. 16, 18, 34, 52).
- [LB21] F. Leymann, J. Barzen. “Pattern Atlas”. In: Springer International Publishing, Apr. 2021, pp. 67–76 (cit. on pp. 86, 213, 216, 238).

- [LBWW17] F. Leymann, U. Breitenbücher, S. Wagner, J. Wettinger. “Native Cloud Applications: Why Monolithic Virtualization Is Not Their Foundation”. In: *Cloud Computing and Services Science*. Springer International Publishing, July 2017, pp. 16–40 (cit. on p. 55).
- [LFM+11] F. Leymann, C. Fehling, R. Mietzner, A. Nowak, S. Dustdar. “Moving Applications to the Cloud: An Approach based on Application Model Enrichment”. In: *International Journal of Cooperative Information Systems* 20.3 (Sept. 2011), pp. 307–356 (cit. on pp. 60, 90).
- [LFWW16] F. Leymann, C. Fehling, S. Wagner, J. Wettinger. “Native Cloud Applications: Why Virtual Machines, Images and Containers Miss the Point!” In: *Proceedings of the 6th International Conference on Cloud Computing and Service Science (CLOSER 2016)*. Rome: SciTePress, Apr. 2016, pp. 7–15 (cit. on p. 55).
- [LR00] F. Leymann, D. Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall PTR, 2000 (cit. on pp. 16, 26, 36, 54, 64, 69, 139, 163–169, 171–173).
- [LLTY21] B. Li, Z. Li, Y. Tan, J. Yu. “CoFunc: A unified development framework for heterogeneous FaaS computing platforms”. In: *2021 International Conference on Communications, Information System and Computer Engineering (CISCE)*. May 2021, pp. 726–730 (cit. on p. 78).
- [LK21] C. Lin, H. Khazaei. “Modeling and Optimization of Performance and Cost of Serverless Applications”. In: *IEEE Transactions on Parallel and Distributed Systems* 32.3 (Mar. 2021), pp. 615–632. ISSN: 1558-2183 (cit. on p. 66).
- [Lip12] P. Lipton. “Escaping vendor lock-in with toasca, an emerging cloud standard for portability”. In: *CA Labs Research* 49 (2012) (cit. on pp. 33, 56, 62, 75).

- [LLM11] C. Liu, B. T. Loo, Y. Mao. “Declarative Automated Cloud Resource Orchestration”. In: *Proceedings of the 2nd ACM Symposium on Cloud Computing*. SOCC ’11. Association for Computing Machinery, 2011. ISBN: 9781450309769 (cit. on p. 59).
- [LKBT11] N. Loutas, E. Kamateri, F. Bosi, K. Tarabanis. “Cloud computing interoperability: the state of play”. In: *2011 IEEE Third International Conference on Cloud Computing Technology and Science*. IEEE. 2011, pp. 752–757 (cit. on pp. 74, 75).
- [LKT11] N. Loutas, E. Kamateri, K. Tarabanis. “A semantic interoperability framework for cloud platform as a service”. In: *2011 IEEE Third International Conference on Cloud Computing Technology and Science*. IEEE. 2011, pp. 280–287 (cit. on p. 81).
- [LL16] D. Lübke, T. van Lessen. “Modeling Test Cases in BPMN for Behavior-Driven Development”. In: *IEEE Software* 33.5 (2016), pp. 15–21 (cit. on p. 71).
- [LL17] D. Lübke, T. v. Lessen. “BPMN-based model-driven testing of service-based processes”. In: *Enterprise, Business-Process and Information Systems Modeling*. Springer, 2017, pp. 119–133 (cit. on pp. 71, 161, 169).
- [LSC20] A. Luzar, S. Stanovnik, M. Cankar. “Examination and comparison of toasca orchestration tools”. In: *European Conference on Software Architecture*. Springer. 2020, pp. 247–259 (cit. on pp. 205, 212, 216).
- [LRLE17] T. Lynn, P. Rosati, A. Lejeune, V. Emeakaroha. “A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms”. In: *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. Dec. 2017, pp. 162–169 (cit. on pp. 83, 130).

- [MDEK95] J. Magee, N. Dulay, S. Eisenbach, J. Kramer. “Specifying distributed software architectures”. In: *European Software Engineering Conference*. Springer. 1995, pp. 137–153 (cit. on p. 58).
- [MPP+15] K. Magoutis, C. Papoulas, A. Papaioannou, F. Karniavoura, D.-G. Akestoridis, N. Parotsidis, M. Korozi, A. Leonidis, S. Ntoa, C. Stephanidis. “Design and implementation of a social networking platform for cloud deployment specialists”. In: *Journal of Internet Services and Applications* 6.1 (2015), pp. 1–26 (cit. on p. 81).
- [MGZ+17] M. Malawski, A. Gajek, A. Zima, B. Balis, K. Figiela. “Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions”. In: *Future Generation Computer Systems* (2017). issn: 0167-739X (cit. on p. 73).
- [MEHW18] J. Manner, M. Endreß, T. Heckel, G. Wirtz. “Cold Start Influencing Factors in Function as a Service”. In: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. Dec. 2018, pp. 181–188 (cit. on p. 54).
- [MTB18] Y. Mansouri, A. N. Toosi, R. Buyya. “Data storage management in cloud environments: Taxonomy, survey, and future directions”. In: *ACM Computing Surveys (CSUR)* 50.6 (2018), p. 91 (cit. on p. 137).
- [MT00] N. Medvidovic, R. Taylor. “A classification and comparison framework for software architecture description languages”. In: *IEEE Transactions on Software Engineering* 26.1 (2000), pp. 70–93 (cit. on p. 58).
- [MMP00] N. R. Mehta, N. Medvidovic, S. Phadke. “Towards a taxonomy of software connectors”. In: *Proceedings of the 22nd international conference on Software engineering*. 2000, pp. 178–187 (cit. on pp. 24, 57).

- [MG+11] P. Mell, T. Grance, et al. “The NIST definition of cloud computing”. In: (2011) (cit. on p. 15).
- [MG11] P. M. Mell, T. Grance. *SP 800-145. The NIST Definition of Cloud Computing*. Tech. rep. Gaithersburg, MD, United States, 2011 (cit. on p. 52).
- [MR12] M. Menzel, R. Ranjan. “CloudGenius: Decision Support for Web Server Cloud Migration”. In: *Proceedings of the 21st International Conference on World Wide Web. WWW ’12*. New York, NY, USA: Association for Computing Machinery, 2012, pp. 979–988 (cit. on p. 82).
- [Mes07] D. G. Messerschmitt. “Rethinking components: From hardware and software to systems”. In: *Proceedings of the IEEE 95.7* (2007), pp. 1473–1496 (cit. on p. 94).
- [Mic22] Microsoft. *Directory of Azure Services*. <https://azure.microsoft.com/en-us/services/>. 2022 (cit. on pp. 16, 53, 64, 99, 102, 104, 105, 107, 110, 112, 114, 117, 120, 127, 133, 137, 162, 179).
- [MUL09] R. Mietzner, T. Unger, F. Leymann. “Cafe: A Generic Configurable Customizable Composite Cloud Application Framework”. In: *On the Move to Meaningful Internet Systems: OTM 2009 (CoopIS 2009)*. Springer, Nov. 2009, pp. 357–364 (cit. on p. 61).
- [Mli21] K. Mlitz. *Distribution of cloud computing market revenues worldwide from 2015 to June 2019, by vendor*. <https://www.statista.com/statistics/540511/worldwide-cloud-computing-revenue-share-by-vendor/>. 2021 (cit. on p. 162).
- [MM18] P. Moczurad, M. Malawski. “Visual-Textual Framework for Serverless Computation: A Luna Language Approach”. In: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. 2018, pp. 169–174 (cit. on p. 67).

-
- [MPd18] S. K. Mohanty, G. Premsankar, M. di Francesco. “An Evaluation of Open Source Serverless Computing Frameworks”. In: *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. Dec. 2018, pp. 115–120 (cit. on pp. 83, 130).
- [Moo09] D. Moody. “The “physics” of notations: toward a scientific basis for constructing visual notations in software engineering”. In: *IEEE Transactions on software engineering* 35.6 (2009), pp. 756–779 (cit. on pp. 59, 160).
- [Mor20] K. Morris. *Infrastructure as Code*. O’Reilly Media, 2020 (cit. on pp. 64, 85).
- [MAD+11] F. Moscato, R. Aversa, B. Di Martino, T.-F. Fortiș, V. Munteanu. “An analysis of mosaic ontology for cloud resources annotation”. In: *2011 federated conference on computer science and information systems (FedCSIS)*. IEEE. 2011, pp. 973–980 (cit. on p. 80).
- [New15] S. Newman. *Building microservices: designing fine-grained systems*. O’Reilly Media, Inc., 2015 (cit. on p. 55).
- [Nob98] J. Noble. “Classifying relationships between object-oriented design patterns”. In: *Proceedings 1998 australian software engineering conference (cat. no. 98ex233)*. IEEE. 1998, pp. 98–107 (cit. on pp. 121, 122, 124).
- [NBF+12] A. Nowak, T. Binz, C. Fehling, O. Kopp, F. Leymann, S. Wagner. “Pattern-driven Green Adaptation of Process-based Applications and their Runtime Infrastructure”. In: *Computing* (Feb. 2012), pp. 463–487 (cit. on p. 87).
- [NT20] J. Nupponen, D. Taibi. “Serverless: What it Is, What to Do and What Not to Do”. In: *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*. Mar. 2020, pp. 49–50 (cit. on p. 71).

- [OAS07] OASIS. *Web Services Business Process Execution Language (WS-BPEL) Version 2.0*. Organization for the Advancement of Structured Information Standards (OASIS). 2007 (cit. on pp. 69, 161, 165).
- [OAS20] OASIS. *TOSCA Simple Profile in YAML Version 1.3*. Organization for the Advancement of Structured Information Standards (OASIS). 2020 (cit. on pp. 62, 63, 75, 212, 214).
- [OMG11] OMG. *Business Process Model and Notation (BPMN) Version 2.0*. Object Management Group (OMG). 2011 (cit. on pp. 27, 42, 64, 69, 70, 160, 161, 171).
- [OMG14] OMG. *Model Driven Architecture (MDA) MDA Guide rev. 2.0*. Object Management Group (OMG). 2014 (cit. on p. 57).
- [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 pp. 23, 30, 56).
- [Ope22] OpenFaaS Project. *OpenFaaS*. <https://www.openfaas.com>. 2022 (cit. on pp. 53, 127, 133, 135).
- [Ora22] Oracle. *Oracle Database Classic Cloud Service*. [docs . oracle.com/en/cloud/paas/database-dbaas-cloud/index.html](https://docs.oracle.com/en/cloud/paas/database-dbaas-cloud/index.html). 2022 (cit. on p. 115).
- [Pah+18] C. Pahl et al. “Architectural Principles for Cloud Software”. In: *ACM Transactions on Internet Technology (TOIT)* 18.2 (2018). issn: 1533-5399 (cit. on p. 85).
- [PKC19] A. Palade, A. Kazmi, S. Clarke. “An Evaluation of Open Source Serverless Computing Frameworks Support at the Edge”. In: *2019 IEEE World Congress on Services (SERVICES)*. Vol. 2642. IEEE. 2019, pp. 206–211 (cit. on pp. 83, 130).

- [PKYY20] J. Park, U. Kim, D. Yun, K. Yeom. “Approach for selecting and integrating cloud services to construct hybrid cloud”. In: *Journal of Grid Computing* 18.3 (2020), pp. 441–469 (cit. on p. 87).
- [PLMR19] P. C. Paul, J. Loane, F. McCaffery, G. Regan. “A Serverless Architecture for Wireless Body Area Network Applications”. In: *Model-Based Safety and Assessment*. Vol. 11842. Series Title: Lecture Notes in Computer Science. Springer International Publishing, 2019, pp. 239–254 (cit. on p. 79).
- [PRF22] S. Pedratscher, S. Ristov, T. Fahringer. “M2FaaS: Transparent and fault tolerant FaaSification of Node.js monolith code blocks”. In: *Future Generation Computer Systems* 135 (2022), pp. 57–71. issn: 0167-739X (cit. on p. 78).
- [PPSC21] I. Pelle, F. Paolucci, B. Sonkoly, F. Cugini. “Latency-Sensitive Edge/Cloud Serverless Dynamic Deployment Over Telemetry-Based Packet-Optical Network”. In: *IEEE Journal on Selected Areas in Communications* 39.9 (Sept. 2021), pp. 2849–2863. issn: 1558-0008 (cit. on p. 68).
- [Pel03] C. Peltz. “Web services orchestration and choreography”. In: *Computer* 36.10 (2003), pp. 46–52 (cit. on p. 69).
- [PZL+09] J. Peng, X. Zhang, Z. Lei, B. Zhang, W. Zhang, Q. Li. “Comparison of Several Cloud Computing Platforms”. In: *2009 Second International Symposium on Information Science and Engineering*. 2009, pp. 23–27 (cit. on p. 82).
- [PP18] K. J. P. G. Perera, I. Perera. “A Rule-based System for Automated Generation of Serverless-Microservices Architecture”. In: *2018 IEEE International Systems Engineering Symposium (ISSE)*. 2018, pp. 1–8 (cit. on p. 58).
- [Pet11] D. Petcu. “Portability and interoperability between clouds: challenges and case study”. In: *European Conference on a Service-Based Internet*. Springer. 2011, pp. 62–74 (cit. on p. 56).

- [PFMM08] K. Petersen, R. Feldt, S. Mujtaba, M. Mattsson. “Systematic mapping studies in software engineering.” In: *Ease*. Vol. 8. 2008, pp. 68–77 (cit. on pp. 18, 20, 130).
- [PVK15] K. Petersen, S. Vakkalanka, L. Kuzniarz. “Guidelines for conducting systematic mapping studies in software engineering: An update”. In: *Information and Software Technology* 64 (2015), pp. 1–18 (cit. on pp. 18, 130).
- [Pet95] M. Petre. “Why looking isn’t always seeing: readership skills and graphical programming”. In: *Communications of the ACM* 38.6 (1995), pp. 33–44 (cit. on p. 92).
- [PO09] R. Prodan, S. Ostermann. “A survey and taxonomy of infrastructure as a service and web hosting cloud providers”. In: *2009 10th IEEE/ACM International Conference on Grid Computing*. IEEE. 2009, pp. 17–25 (cit. on p. 81).
- [QRD16] C. Quinton, D. Romero, L. Duchien. “SALOON: a platform for selecting and configuring cloud environments”. In: *Software: Practice and Experience* 46.1 (2016), pp. 55–78 (cit. on pp. 79, 82).
- [Raj18] R. A. P. Rajan. “Serverless Architecture-A Revolution in Cloud Computing”. In: *2018 Tenth International Conference on Advanced Computing (ICoAC)*. IEEE. 2018, pp. 88–93 (cit. on pp. 83, 130).
- [RSL+] V. Rampérez, J. Soriano, D. Lizcano, S. Aljawarneh, J. A. Lara. “From SLA to vendor-neutral metrics: An intelligent knowledge-based approach for multi-cloud SLA-based broker”. In: *International Journal of Intelligent Systems* () (cit. on p. 76).
- [Red22] Red Hat, Inc. *Ansible*. <https://www.ansible.com>. 2022 (cit. on pp. 65, 217, 232).

-
- [RBF+16] L. Reinfurt, U. Breitenbücher, M. Falkenthal, F. Leymann, A. Riegg. “Internet of Things Patterns”. In: *Proceedings of the 21st European Conference on Pattern Languages of Programs (EuroPLoP)*. ACM, July 2016, pp. 1–21 (cit. on p. 92).
- [RFL19] L. Reinfurt, M. Falkenthal, F. Leymann. “Where to Begin - On Pattern Language Entry Points”. In: *SICS Software-Intensive Cyber-Physical Systems* (Aug. 2019), pp. 1–12 (cit. on pp. 120, 122).
- [RWZT12] J. Repschlaeger, S. Wind, R. Zarnekow, K. Turowski. “A reference guide to cloud computing dimensions: infrastructure as a service classification framework”. In: *2012 45th Hawaii International Conference on System Sciences*. IEEE. 2012, pp. 2178–2188 (cit. on pp. 80, 129).
- [RF20] M. Richards, N. Ford. *Fundamentals of Software Architecture: An Engineering Approach*. O’Reilly Media, 2020 (cit. on p. 34).
- [Ric18] C. Richardson. *Microservices patterns*. Manning Publications Company, 2018 (cit. on pp. 24, 55, 85, 120).
- [RCL09] B. P. Rimal, E. Choi, I. Lumb. “A taxonomy and survey of cloud computing systems”. In: *2009 Fifth International Joint Conference on INC, IMS and IDC*. Ieee. 2009, pp. 44–51 (cit. on p. 80).
- [RMNB21] S. Risco, G. Moltó, D. M. Naranjo, I. Blanquer. “Serverless Workflows for Containerised Applications in the Cloud Continuum”. In: *Journal of Grid Computing* 19.3 (Sept. 2021), p. 30. issn: 1570-7873, 1572-9184 (cit. on p. 68).
- [RPF21a] S. Ristov, S. Pedratscher, T. Fahringer. “AFCL: An Abstract Function Choreography Language for serverless workflow specification”. In: *Future Generation Computer Systems* 114 (2021), pp. 368–382. issn: 0167-739X (cit. on p. 72).

- [RPF21b] S. Ristov, S. Pedratscher, T. Fahringer. “xAFCL: Run Scalable Function Choreographies Across Multiple FaaS Systems”. In: *IEEE Transactions on Services Computing* (2021), pp. 1–1. ISSN: 1939-1374 (cit. on p. 72).
- [RPWF21] S. Ristov, S. Pedratscher, J. Wallnoefer, T. Fahringer. “DAF: Dependency-Aware FaaSifier for Node.js Monolithic Applications”. In: *IEEE Software* 38.1 (Jan. 2021), pp. 48–53. ISSN: 1937-4194 (cit. on p. 78).
- [RFS22] P. Rodrigues, F. Freitas, J. Simão. “QuickFaaS: Providing Portability and Interoperability between FaaS Platforms”. In: *Future Internet* 14.12 (2022), p. 360 (cit. on p. 77).
- [RWLN89] J. Rothenberg, L. E. Widman, K. A. Loparo, N. R. Nielsen. “The nature of modeling”. In: *Artificial Intelligence, Simulation and Modeling* (1989) (cit. on p. 57).
- [Rue20] B. Ruecker. *How to Orchestrate AWS Lambda using Camunda Cloud: Powerful Serverless Function Orchestration using BPMN and Cloud-Native Workflow Technology*. <https://camunda.com/blog/2020/05/how-to-orchestrate-aws-lambda-using-camunda-cloud>. 2020 (cit. on pp. 70, 71, 160).
- [RTVM06] N. Russell, A. H. Ter Hofstede, W. M. Van Der Aalst, N. Mulyar. “Workflow control-flow patterns: A revised view”. In: *BPM Center Report BPM-06-22, BPMcenter.org* (2006), pp. 06–22 (cit. on pp. 54, 70, 161).
- [SBKL17] K. Saatkamp, U. Breitenbücher, O. Kopp, F. Leymann. “Topology Splitting and Matching for Multi-Cloud Deployments”. In: *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017)*. SciTePress, Apr. 2017, pp. 247–258 (cit. on p. 214).

- [EB14] N. El-Saber, A. Boronat. “BPMN formalization and verification using Maude”. In: *Proceedings of the 2014 Workshop on Behaviour Modelling-Foundations and Applications*. 2014, pp. 1–12 (cit. on p. 171).
- [SAA+19] F. Samea, F. Azam, M. W. Anwar, M. Khan, M. Rashid. “A UML Profile for Multi-Cloud Service Configuration (UMLPMSC) in Event-Driven Serverless Applications”. In: *Proceedings of the 2019 8th International Conference on Software and Computer Applications*. ICSCA '19. Association for Computing Machinery, 2019, pp. 431–435. ISBN: 9781450365734 (cit. on p. 67).
- [SGS+21] J. Sampe, P. Garcia-Lopez, M. Sanchez-Artigas, G. Vernik, P. Roca-Llaberia, A. Arjona. “Toward Multicloud Access Transparency in Serverless Computing”. In: *IEEE Software* 38.1 (Jan. 2021), pp. 68–74. ISSN: 1937-4194 (cit. on p. 77).
- [SVSG18] J. Sampé, G. Vernik, M. Sánchez-Artigas, P. García-López. “Serverless Data Analytics in the IBM Cloud”. In: *Proceedings of the 19th International Middleware Conference Industry*. Middleware '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 1–8 (cit. on p. 53).
- [Sch09] H. E. Schaffer. “X as a service, cloud computing, and the need for good judgment”. In: *IT professional* 11.5 (2009), pp. 4–5 (cit. on p. 56).
- [SL19] J. Scheuner, P. Leitner. “Transpiling Applications into Optimized Serverless Orchestrations”. In: *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS*W)*. June 2019, pp. 72–73 (cit. on p. 76).
- [Ser22a] Serverless Workflow Specification Authors. *Serverless Workflow Specification*. <https://serverlessworkflow.io>. 2022 (cit. on pp. 73, 75).

- [Ser22b] Serverless, Inc. *Serverless Framework*. <https://serverless.com>. 2022 (cit. on pp. 66, 83, 141).
- [SL22] K. R. Sheshadri, J. Lakshmi. “QoS aware FaaS for Heterogeneous Edge-Cloud continuum”. In: *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*. July 2022, pp. 70–80 (cit. on p. 76).
- [SRC13] G. C. Silva, L. M. Rose, R. Calinescu. “A systematic review of cloud lock-in solutions”. In: *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*. Vol. 2. IEEE. 2013, pp. 363–368 (cit. on p. 56, 75).
- [SEM+21] F. Smirnov, C. Engelhardt, J. Mittelberger, B. Pourmohseni, T. Fahringer. “Apollo: Towards an Efficient Distributed Orchestration of Serverless Function Compositions in the Cloud-Edge Continuum”. In: *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing*. UCC ’21. Association for Computing Machinery, 2021. ISBN: 9781450385640 (cit. on p. 72).
- [SWS21] D. Sokolowski, P. Weisenburger, G. Salvaneschi. “Automating Serverless Deployments for DevOps Organizations”. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021. Association for Computing Machinery, 2021, pp. 57–69. ISBN: 9781450385626 (cit. on p. 67).
- [SMG+22] M. Son, S. Mohanty, J. R. Gunasekaran, A. Jain, M. T. Kandemir, G. Kesidis, B. Urgaonkar. “Splice: An Automated Framework for Cost-and Performance-Aware Blending of Cloud Services”. In: *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. May 2022, pp. 119–128 (cit. on p. 77).

-
- [Spi17] J. Spillner. “Transformation of Python Applications into Function-as-a-Service Deployments”. In: *arXiv preprint arXiv:1705.08169* (2017) (cit. on pp. 22, 78).
- [Spi19] J. Spillner. “Quantitative Analysis of Cloud Function Evolution in the AWS Serverless Application Repository”. In: *arXiv preprint arXiv:1905.04800* (2019) (cit. on p. 83).
- [Spi21] J. Spillner. “Self-Balancing Architectures Based on Liquid Functions across Computing Continuums”. In: *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion*. UCC ’21. Association for Computing Machinery, 2021. ISBN: 9781450391634 (cit. on p. 76).
- [SD17] J. Spillner, S. Dorodko. “Java code analysis and transformation into AWS lambda functions”. In: *arXiv preprint arXiv:1702.05510* (2017) (cit. on pp. 22, 78).
- [SGBV20] J. Spillner, P. Gkikopoulos, A. Buzachis, M. Villari. “Rule-Based Resource Matchmaking for Composite Application Deployments across IoT-Fog-Cloud Continuums”. In: *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. Dec. 2020, pp. 336–341 (cit. on p. 82).
- [SMM18] J. Spillner, C. Mateos, D. A. Monge. “FaaSter, Better, Cheaper: The Prospect of Serverless Scientific Computing and HPC”. In: *High Performance Computing*. Vol. 796. Series Title: Communications in Computer and Information Science. Springer International Publishing, 2018, pp. 154–168 (cit. on pp. 53, 78).
- [SRF16] K.-J. Stol, P. Ralph, B. Fitzgerald. “Grounded Theory in Software Engineering Research: A Critical Review and Guidelines”. In: *Proceedings of the 38th International*

- Conference on Software Engineering*. ICSE '16. Association for Computing Machinery, 2016, pp. 120–131. ISBN: 9781450339001 (cit. on p. 51).
- [SKLU11] S. Strauch, O. Kopp, F. Leymann, T. Unger. “A Taxonomy for Cloud Data Hosting Solutions”. In: *Proceedings of the International Conference on Cloud and Green Computing (CGC '11)*. IEEE Computer Society, Dec. 2011, pp. 577–584 (cit. on p. 80).
- [SPSP14] K. Stravoskoufos, A. Preventis, S. Sotiriadis, E. G. Petrakis. “A Survey on Approaches for Interoperability and Portability of Cloud Computing Services”. In: *CLOSER*. 2014, pp. 112–117 (cit. on p. 30).
- [SDH+14] L. Sun, H. Dong, F. K. Hussain, O. K. Hussain, E. Chang. “Cloud service selection: State-of-the-art and future research directions”. In: *Journal of Network and Computer Applications* 45 (2014), pp. 134–150 (cit. on pp. 81, 150).
- [SSL12] S. Sundareswaran, A. Squicciarini, D. Lin. “A Brokerage-Based Approach for Cloud Service Selection”. In: *2012 IEEE Fifth International Conference on Cloud Computing*. 2012, pp. 558–565 (cit. on p. 82).
- [SF15] M. H. Syed, E. B. Fernandez. “The software container pattern”. In: *Proceedings of the 22nd Conference on Pattern Languages of Programs*. The Hillside Group. 2015, pp. 24–26 (cit. on pp. 112, 117).
- [SF17] M. H. Syed, E. B. Fernandez. “The container manager pattern”. In: *Proceedings of the 22nd European Conference on Pattern Languages of Programs*. 2017, pp. 1–9 (cit. on pp. 112, 117).
- [SGM02] C. Szyperski, D. Gruntz, S. Murer. *Component software: beyond object-oriented programming*. Pearson Education, 2002 (cit. on p. 94).

-
- [Tai+20] D. Taibi et al. “Patterns for Serverless Functions (Function-as-a-Service): A Multivocal Literature Review.” In: *CLOSER*. 2020, pp. 181–192 (cit. on pp. 16, 85, 161).
- [TSL08] C. C. Tan, B. Sheng, Q. Li. “Secure and serverless RFID authentication and search protocols”. In: *IEEE Transactions on Wireless Communications* 7.4 (2008), pp. 1400–1407 (cit. on p. 52).
- [TVGB21] V. Tankov, D. Valchuk, Y. Golubev, T. Bryksin. “Infrastructure in Code: Towards Developer-Friendly Cloud Applications”. In: *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*. ASE ’21. IEEE Press, 2021, pp. 1166–1170. ISBN: 9781665403375 (cit. on p. 68).
- [The22a] The Fission Authors. *Fission*. <https://fission.io>. 2022 (cit. on pp. 133, 135).
- [The22b] The Knative Authors. *Knative*. <https://knative.dev>. 2022 (cit. on p. 135).
- [The22c] The Kubernetes Authors. *Kubernetes*. <https://kubernetes.io>. 2022 (cit. on pp. 53, 64, 111, 127, 133, 232).
- [The22d] The Linux Foundation. *Open Container Initiative*. 2022 (cit. on p. 74).
- [TCBR21] R. Tolosana-Calasanz, G. G. Castañé, J. Á. Bañares, O. Rana. “Modelling Serverless Function Behaviours”. In: *Economics of Grids, Clouds, Systems, and Services*. Vol. 13072. Series Title: Lecture Notes in Computer Science. Springer International Publishing, 2021, pp. 109–122 (cit. on p. 67).
- [TVC+21] A. Tsagkaropoulos, Y. Verginadis, M. Compastié, D. Apostolou, G. Mentzas. “Extending TOSCA for Edge and Fog Deployment Support”. In: *Electronics* 10.6 (2021), p. 737 (cit. on p. 64).

- [TTK21] G. Tzouros, M. Tsenos, V. Kalogeraki. “Portable Intermediate Representation for Efficient Big Data Analytics”. In: *Distributed Applications and Interoperable Systems*. Vol. 12718. Series Title: Lecture Notes in Computer Science. Springer International Publishing, 2021, pp. 74–80 (cit. on p. 77).
- [VCL+20] J. Vandebon, J.G.F. Coutinho, W. Luk, E. Nurvitadhi, M. Naik. “SLATE: Managing Heterogeneous Cloud Functions”. In: *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. July 2020, pp. 141–148 (cit. on p. 75).
- [Var10] J. Varia. “Architecting for the cloud: Best practices”. In: (Jan. 2010) (cit. on pp. 55, 56).
- [VGO+17] M. Villamizar, O. Garcés, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, M. Lang. “Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures”. In: *Service Oriented Computing and Applications* 11.2 (June 2017), pp. 233–247. ISSN: 1863-2394 (cit. on p. 54).
- [VFD+16] M. Villari, M. Fazio, S. Dustdar, O. Rana, R. Ranjan. “Osmotic computing: A new paradigm for edge/cloud integration”. In: *IEEE Cloud Computing* 3.6 (2016), pp. 76–83 (cit. on p. 82).
- [WLZ+18] L. Wang, M. Li, Y. Zhang, T. Ristenpart, M. Swift. “Peeking Behind the Curtains of Serverless Platforms”. In: *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '18. Berkeley, CA, USA: USENIX Association, 2018, pp. 133–145. ISBN: 978-1-931971-44-7 (cit. on p. 84).

- [WBB+20] M. Weigold, J. Barzen, U. Breitenbücher, M. Falkenthal, F. Leymann, K. Wild. “Pattern Views: Concept and Tooling of Interconnected Pattern Languages”. In: *Proceedings of the 14th Symposium and Summer School on Service-Oriented Computing (SummerSOC 2020)*. Springer International Publishing, Dec. 2020, pp. 86–103 (cit. on p. 86).
- [WF11] T. Wellhausen, A. Fiesser. “How to write a pattern? A rough guide for first-time pattern authors”. In: *Proceedings of the 16th European Conference on Pattern Languages of Programs*. 2011, pp. 1–9 (cit. on pp. 84, 92).
- [WCL+21] J. Wen, Z. Chen, Y. Liu, Y. Lou, Y. Ma, G. Huang, X. Jin, X. Liu. “An Empirical Study on Challenges of Application Development in Serverless Computing”. In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021*. Association for Computing Machinery, 2021, pp. 416–428. ISBN: 9781450385626 (cit. on p. 55).
- [WL21] J. Wen, Y. Liu. “A Measurement Study on Serverless Workflow Services”. In: *2021 IEEE International Conference on Web Services (ICWS)*. Sept. 2021, pp. 741–750 (cit. on p. 83).
- [WLC+] J. Wen, Y. Liu, Z. Chen, J. Chen, Y. Ma. “Characterizing commodity serverless computing platforms”. In: *Journal of Software: Evolution and Process* () (cit. on p. 83).
- [WT21] S. Werner, S. Tai. “Application-Platform Co-design for Serverless Data Processing”. In: *Service-Oriented Computing*. Vol. 13121. Series Title: Lecture Notes in Computer Science. Springer International Publishing, 2021, pp. 627–640 (cit. on p. 83).

- [WBL15] J. Wettinger, U. Breitenbücher, F. Leymann. “Any2API - Automated APIfication”. In: *Proceedings of the 5th International Conference on Cloud Computing and Services Science (CLOSER 2015)*. SciTePress, May 2015, pp. 475–486 (cit. on p. 185).
- [WW19] S. Winzinger, G. Wirtz. “Model-Based Analysis of Serverless Applications”. In: *Proceedings of the 11th International Workshop on Modelling in Software Engineerings. MiSE '19*. IEEE Press, 2019, pp. 82–88 (cit. on p. 66).
- [Woh14] C. Wohlin. “Guidelines for snowballing in systematic literature studies and a replication in software engineering”. In: *Proceedings of the 18th international conference on evaluation and assessment in software engineering*. Citeseer. 2014, p. 38 (cit. on pp. 19, 130).
- [WS07] C. Wolter, A. Schaad. “Modeling of task-based authorization constraints in BPMN”. In: *International Conference on Business Process Management*. Springer. 2007, pp. 64–79 (cit. on p. 169).
- [WBB+19] M. Wurster, U. Breitenbücher, A. Brogi, G. Falazi, L. Harzenetter, F. Leymann, J. Soldani, V. Yussupov. “The EDMM Modeling and Transformation System”. In: *Service-Oriented Computing – ICSOC 2019 Workshops*. Springer, Dec. 2019 (cit. on pp. 47, 65, 231, 232).
- [WBF+19] M. Wurster, U. Breitenbücher, M. Falkenthal, C. Krieger, F. Leymann, K. Saatkamp, J. Soldani. “The Essential Deployment Metamodel: A Systematic Review of Deployment Automation Technologies”. In: *SICS Software-Intensive Cyber-Physical Systems 35* (Aug. 2019), pp. 63–75 (cit. on pp. 65, 68, 130, 232).
- [WBH+20a] M. Wurster, U. Breitenbücher, L. Harzenetter, F. Leymann, J. Soldani. “TOSCA Lightning: An Integrated Toolchain for Transforming TOSCA Light into Production-Ready De-

- ployment Technologies”. In: *Advanced Information Systems Engineering (CAiSE Forum 2020)*. Vol. 386. Lecture Notes in Business Information Processing. Springer International Publishing, Aug. 2020, pp. 138–146 (cit. on p. 205).
- [WBH+20b] M. Wurster, U. Breitenbücher, L. Harzenetter, F. Leymann, J. Soldani, V. Yussupov. “TOSCA Light: Bridging the Gap between the TOSCA Specification and Production-ready Deployment Technologies”. In: *Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER 2020)*. SciTePress, May 2020, pp. 216–226 (cit. on pp. 47, 65, 231, 232).
- [WBK+18] M. Wurster, U. Breitenbücher, K. Képes, F. Leymann, V. Yussupov. “Modeling and Automated Deployment of Serverless Applications using TOSCA”. In: *Proceedings of the IEEE 11th International Conference on Service-Oriented Computing and Applications (SOCA 2018)*. IEEE Computer Society, Nov. 2018, pp. 73–80 (cit. on pp. 47, 67, 195, 198).
- [WBKL18] M. Wurster, U. Breitenbücher, O. Kopp, F. Leymann. “Modeling and Automated Execution of Application Deployment Tests”. In: *Proceedings of the IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC 2018)*. IEEE Computer Society, Oct. 2018, pp. 171–180 (cit. on p. 205).
- [YCCI16] M. Yan, P. Castro, P. Cheng, V. Ishakian. “Building a Chatbot with Serverless Computing”. In: *Proceedings of the 1st International Workshop on Mashups of Things and APIs. MOTA ’16*. New York, NY, USA: ACM, 2016, 5:1–5:4. ISBN: 978-1-4503-4669-6 (cit. on p. 53).

- [YKK03] W. Ye, A. I. Khan, E. A. Kendall. “Distributed network file storage for a serverless (p2p) network”. In: *The 11th IEEE International Conference on Networks, 2003. ICON2003*. IEEE, 2003, pp. 343–347 (cit. on p. 52).
- [YBB+22] V. Yussupov, U. Breitenbücher, A. Brogi, L. Harzenetter, F. Leymann, J. Soldani. “Serverless or Serverful? A Pattern-based Approach for Exploring Hosting Alternatives”. In: *Proceedings of the 16th Symposium and Summer School on Service-Oriented Computing (SummerSOC 2022)*. Springer, Oct. 2022, pp. 45–67 (cit. on pp. 45, 91, 98, 103, 195).
- [YBHL19] V. Yussupov, U. Breitenbücher, M. Hahn, F. Leymann. “Serverless Parachutes: Preparing Chosen Functionalities for Exceptional Workloads”. In: *Proceedings of the 23rd International Enterprise Distributed Object Computing Conference (EDOC 2019)*. IEEE Computer Society, Oct. 2019, pp. 226–235 (cit. on pp. 46, 79, 159, 185, 187, 229).
- [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 (cit. on pp. 45, 195).
- [YBK+20] V. Yussupov, U. Breitenbücher, C. Krieger, F. Leymann, J. Soldani, M. Wurster. “Pattern-based Modelling, Integration, and Deployment of Microservice Architectures”. In: *Proceedings of the 24th International Enterprise Distributed Object Computing Conference (EDOC 2020)*. IEEE, Oct. 2020, pp. 40–50 (cit. on pp. 45, 195).
- [YBLM19] V. Yussupov, U. Breitenbücher, F. Leymann, C. Müller. “Facing the Unplanned Migration of Serverless Applications: A Study on Portability Problems, Solutions, and Dead

- Ends”. In: *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2019)*. Ed. by ACM. ACM, Dec. 2019, pp. 273–283 (cit. on pp. 18, 46).
- [YBLW19] V. Yussupov, U. Breitenbücher, F. Leymann, M. Wurster. “A Systematic Mapping Study on Engineering Function-as-a-Service Platforms and Tools”. In: *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2019)*. Ed. by ACM. ACM, Dec. 2019, pp. 229–240 (cit. on pp. 18, 46, 53).
- [YFFL19] V. Yussupov, G. Falazi, M. Falkenthal, F. Leymann. “Protecting Deployment Models in Collaborative Cloud Application Development”. In: *International Journal On Advances in Security* 12.1&2 (June 2019), pp. 79–94. ISSN: 1942-2636 (cit. on p. 46).
- [YFK+18] V. Yussupov, M. Falkenthal, O. Kopp, F. Leymann, M. Zimmermann. “Secure Collaborative Development of Cloud Application Deployment Models”. In: *Proceedings of The Twelfth International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2018)*. Xpert Publishing Services, Sept. 2018, pp. 48–57 (cit. on p. 46).
- [YSB+21a] V. Yussupov, J. Soldani, U. Breitenbücher, A. Brogi, F. Leymann. “FaaSSten your decisions: A classification framework and technology review of Function-as-a-Service platforms”. In: *Journal of Systems and Software* 175 (May 2021). ISSN: 0164-1212 (cit. on pp. 45, 84, 128, 132, 134, 138, 140, 177, 219).
- [YSB+21b] V. Yussupov, J. Soldani, U. Breitenbücher, A. Brogi, F. Leymann. “From Serverful to Serverless: A Spectrum of Patterns for Hosting Application Components”. In: *Proceed-*

- ings of the 11th International Conference on Cloud Computing and Services Science (CLOSER 2021)*. SciTePress, May 2021, pp. 268–279 (cit. on pp. 45, 91, 108).
- [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 (June 2022), pp. 1454–1495 (cit. on pp. 45, 159, 160, 177, 227, 229, 230).
- [Zam18] B. Zambrano. *Serverless Design Patterns and Best Practices: Build, Secure, and Deploy Enterprise Ready Serverless Applications with AWS to Improve Developer Productivity*. Packt Publishing, 2018. ISBN: 178862064X (cit. on p. 85).
- [ZHS+18] H. Zhou, Y. Hu, J. Su, C. de Laat, Z. Zhao. “CloudsStorm: An Application-Driven Framework to Enhance the Programmability and Controllability of Cloud Virtual Infrastructures”. In: *Cloud Computing – CLOUD 2018*. Springer International Publishing, 2018, pp. 265–280 (cit. on p. 60).
- [ZGTC21a] L. Zhu, G. Giotis, V. Tountopoulos, G. Casale. “RDOF: Deployment Optimization for Function as a Service”. In: *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. Sept. 2021, pp. 508–514 (cit. on p. 69).
- [ZGTC21b] L. Zhu, G. Giotis, V. Tountopoulos, G. Casale. “RDOF: Deployment Optimization for Function as a Service”. In: *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. IEEE. 2021, pp. 508–514 (cit. on p. 239).
- [ZBH+20] M. Zimmermann, U. Breitenbücher, L. Harzenetter, F. Leymann, V. Yussupov. “Self-Contained Service Deployment Packages”. In: *Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER 2020)*. SciTePress, May 2020, pp. 371–381 (cit. on p. 48).

- [Zim09] O. Zimmermann. “An architectural decision modeling framework for service oriented architecture design”. PhD thesis. University of Stuttgart, 2009 (cit. on p. 17).
- [Zim17] O. Zimmermann. “Microservices tenets”. In: *Computer Science-Research and Development* 32.3 (2017), pp. 301–310 (cit. on p. 55).
- [ZLZ+20] O. Zimmermann, D. Lübke, U. Zdun, C. Pautasso, M. Stocker. “Interface Responsibility Patterns: Processing Resources and Operation Responsibilities”. In: *Proceedings of the European Conference on Pattern Languages of Programs 2020*. EuroPLoP ’20. Association for Computing Machinery, 2020. ISBN: 9781450377690 (cit. on p. 85).

All links were last accessed on February 17, 2024.

ACRONYMS

- ADL** Architecture Description Language. 58
- API** Application Programming Interface. 23
- ASL** Amazon States Language. 27
- AWS** Amazon Web Services. 16
- BPEL** Business Process Execution Language. 69
- BPM** Business Process Management. 69
- BPMN** Business Process Model and Notation. 42
- BPMN4FO** BPMN for Function Orchestration. 163
- CaaS** Container-as-a-Service. 112
- CAMEL** Cloud Application Modelling and Execution Language. 61
- CDMI** Cloud Data Management Interface. 74
- CI/CD** Continuous Integration/Continuous Delivery. 22
- CLI** Command Line Interface. 135

CML Cloud Modeling Language. 22

CNCF Cloud Native Computing Foundation. 52

CRUD Create-Retrieve-Update-Delete. 24

CSAR Cloud Service Archive. 212

DAG Directed Acyclic Graph. 57

DBaaS Database-as-a-Service. 98

DSL Domain-specific Language. 23

EDA Event-driven Architecture. 34

FaaS Function-as-a-Service. 3

GFOMC Generic Function Orchestration Modeling Construct. 161

GUI Graphical User Interface. 31

HPC High Performance Computing. 64

HTML HyperText Markup Language. 147

HTTP Hypertext Transfer Protocol. 137

IaaS Infrastructure-as-a-Service. 3

IaC Infrastructure-as-Code. 64

IoT Internet of Things. 25

JAR Java Archive. 91

JRE Java Runtime Environment. 94

LoA Level of Abstraction. 43

MCDM Multi-Criteria Decision Making. 81

MDA Model Driven Architecture. 57

MDE Model Driven Engineering. 57

MSA Microservice-based Architecture. 34

OCCI Open Cloud Computing Interface. 74

OCI Open Container Initiative. 74

OS Operating System. 54

OVF Open Virtualization Format. 74

PaaS Platform-as-a-Service. 3

PBDM Pattern-Based Deployment Model. 87

PFOMC Proprietary Function Orchestration Modeling Construct. 178

PIM Platform Independent Model. 57

PL Pattern Language. 41

PRM Pattern Refinement Model. 87

PSM Platform Specific Model. 57

QoS Quality of Service. 56

RDBMS Relational Database Management System. 26

SaaS Software-as-a-Service. 16

SAR Serverless Application Repository. 83

SDK Software Development Kit. 27

SLO Service Level Objective. 76

SNS Simple Notification Service. 138

SOA Service-oriented Architecture. 34

TOSCA Topology and Orchestration Specification for Cloud Applications.
62

UML Unified Modeling Language. 67

VM Virtual Machine. 35

WAR Web Application Archive. 113

XML Extensible Markup Language. 63

XSD XML Schema Definition. 182

YAML YAML Ain't Markup Language. 63