Master Thesis

Navigating the Space between Serverful and Serverless: Benefits, Drawbacks, Challenges and Mitigations

Author
Antonino Grasso

Supervisor
Prof. Dr. Johannes Ebke

Munich, November 13 2024

Hochschule München University of Applied Sciences
Department of computer science and mathematics

Device Insight GmbH

Abstract

This work explores the complexities of navigating the space between serverful and serverless for cloud-native applications. Initial deployment target choices involve requirements and assumptions, which can impact development experience, performance, and cost, as they change or no longer hold. As applications and requirements evolve, practitioners may want to reconsider deployment targets to meet performance and cost-efficiency needs. If left unaddressed, these can prevent the application from meeting its objectives, resulting in a deployment target shift that may require costly migrations if not facilitated beforehand. This research explores strategies to improve flexibility, allowing practitioners to *decide about deployment later* and shift to deployment targets as needed. By synthesizing various sources, this work presents a comprehensive definition of serverless and forms a new perspective, thereby broadening its applicability. Through literature review, interviews, technical exploration and performance benchmarks, two approaches to hybrid deployment are recommended: The *adapted serverful-first* approach leverages the adaptation of established frameworks for deployment targets, while the *portable serverless-first* approach focuses on serverless benefits and portability for seamless transitions across the serverful-serverless spectrum. The evaluation confirms that both approaches provide practical solutions for hybrid deployments, but practitioners need to consider factors such as traffic, usage patterns, cost and latency to select the most suitable deployment target. However, the limitations of the approaches, such as adaptation overhead and applicability constraints, still require careful consideration. Overall, this work provides a foundation for hybrid deployment strategies, guiding practitioners in navigating and optimizing the vast landscape of serverful and serverless deployment targets.

Contents

Chapter 1

Introduction

Software development is fundamentally about delivering value and functionality (Messerschmitt 2007, p. 1), whether through a service, a tool, or embedded in a product. For example, services can deliver value through web applications or APIs over the Internet

Traditionally, services have been built as monolithic applications where all functionality is bundled into a single executable and deployed on bare-metal servers — these are called serverful applications. While this example represents the more traditional case of serverful, serverful deployment remains relevant through virtual machines (VMs) and containerized applications (Deng et al. 2023, pp. 1, 9). With the rise of cloud computing (Sfondrini et al. 2018, p. 183; Deng et al. 2023, p. 1), this traditional model of software development and deployment has evolved.

The "cloud" refers to a network of remote servers, data centers, and hosted virtual infrastructure that allows data to be stored, processed, and delivered without relying on local hardware (Mell et al. 2011, pp. 2–3). It provides scalable, on-demand computing resources that are typically categorized into three service models: Infrastructure as a Service (IaaS), which provides virtualized computing resources such as virtual machines and storage; Platform as a Service (PaaS), which includes application development and deployment platforms; and Software as a Service (SaaS), which delivers fully functional applications over the Internet (Mell et al. 2011, pp. 2–3; Mohammed et al. 2021, pp. 20–21). These models allow developers to build and manage applications with varying degrees of control over infrastructure and software, from full management with IaaS to complete reliance on SaaS. As cloud infrastructure becomes more available and sophisticated, cloud-native applications designed to run in distributed environments using cloud services have become more common (Deng et al. 2023, p. 12). Serverful deployments are still used in the cloud through virtualization or containerization. However, serverless deployment models, where the cloud provider manages the underlying infrastructure and developers focus solely on the application code, are gaining popularity due to their pay-per-use model and reduced operational complexity (Hassan et al. 2021, p. 2; Deng et al. 2023, p. 12; Castro, Ishakian, et al. 2019, pp. 46, 47).

With serverless as an additional deployment target alternative, the application deployment landscape offers a wide range of deployment targets, from traditional serverful to serverless models. Although the business value does not directly depend on the underlying infrastructure on which it is deployed (Messerschmitt 2007, p. 11; Lannurien et al. 2023, p. 283), the choice of deployment target has a significant impact on the development and operation of the application. Early deployment target decisions can have long-term effects (Opara-Martins et al. 2016, pp. 2, 6). For example, when using serverless functions, the application code may be tightly coupled to platform-specific function handlers, potentially limiting the flexibility to adapt to future requirements or constraints (Yussupov, Breitenbücher, Leymann, et al. 2019, pp. 278, 279).

Serverless is chosen for its readily apparent benefits in the early stages of development, such as cost-effectiveness and reduced operational overhead (Leitner et al. 2019, pp. 14, 15, 18).

Depending on the application requirements and circumstances, it may be more beneficial to employ either serverful or serverless deployment targets, while later on the opposite may be more applicable. However, as the application grows in functionality and user demand, the performance and complexity overheads that serverless can bring, such as cold starts and execution time limits, may outweigh the initial benefits (Baldini et al. 2017, pp. 14, 15; Eivy et al. 2017, pp. 11, 12). Conversely, an application that starts with a serverful approach may later explore serverless options to reduce costs as traffic levels decline, and more elastic scaling is desired (Leitner et al. 2019, p. 14; Eismann et al. 2021b, p. 34; Castro, Ishakian, et al. 2019, p. 47).

Changing deployment targets can be complex and require a re-evaluation of architecture and deployment configurations (Eivy et al. 2017, p. 11). This complexity is compounded if the application is already in production, as any changes can have a direct impact on the availability and performance of the application. Flexibility and adaptability are key to effectively managing this complexity throughout the application lifecycle (Castro, Isahagian, et al. 2022, p. 12). Furthermore, identifying the right time to change strategies is not always clear-cut and requires careful consideration of both technical and business factors (Yussupov, Soldani, et al. 2021, p. 1; Leitner et al. 2019, p. 19).

1.1 Focus and Scope

This work aims to explore the spectrum of application deployment targets, from on-premises bare metal servers and virtual machines to the use of containers and serverless functions in the cloud. Conducted in collaboration with Device Insight GmbH[1] (DI), this work draws on DI's insights and experience with cloud-native applications and deployment targets. DI relies almost exclusively on cloud technologies to develop, maintain, and operate customer solutions. Given its strong focus on cloud-native applications, DI is particularly interested in exploring the range of application deployment targets. While grounded in DI's context, the findings aim to provide broadly applicable recommendations for practitioners and contribute to research on cloud-native deployments.

Applications typically consist of multiple components that interact to deliver the desired functionality (Lau et al. 2007; Messerschmitt 2007, p. 4). Infrastructure forms the backbone of an application, providing the essential resources — compute, network, and storage - needed to run it (Messerschmitt 2007, p. 4). The application code, which represents the business logic, is responsible for delivering value to the end user. Platform service offerings can play a supporting role, providing additional functionality such as authentication, message queues, or machine learning services.

Given the wide range of cloud-native application components available, this work prioritizes business logic application code over infrastructure or platform services. Specifically, the research focuses on deployment targets for compute components because they are responsible for executing the business logic and offer the most flexibility and control in deployment targets. Therefore, SaaS, which is fully managed by the provider, is not considered in this work, as services do not offer any level of control besides user-specific configuration (Hassan et al. 2021, p. 1; Kaushik et al. 2022, p. 95). While the networking and storage aspects of the infrastructure are recognized as important, they are not the primary focus of this work; rather, they are treated as influencing factors that affect the deployment targets for compute components. Similarly, platform services such as AWS Cognito[2] or AWS SQS[3], which are beyond the direct control of practitioners, are not considered for hybrid deployment in this research. Although they may also play a role in shaping deployment decisions. Compute therefore offers the most flexibility and control in terms of deployment targets, making it the most appropriate target for the approach.

[1]https://device-insight.com/
[2]https://aws.amazon.com/cognito/
[3]https://aws.amazon.com/sqs

1.2 Goals and Motivation

Navigating the space between serverful and serverless, developers face the challenge of selecting the most appropriate deployment target for their applications (Yussupov, Soldani, et al. 2021, p. 33). This decision is complicated by the need to balance performance, scalability, and cost while anticipating future changes in requirements and constraints (Castro, Isahagian, et al. 2022, p. 12). Traditional deployment targets can be rigid, and cloud offerings can be too locking-in (Baldini et al. 2017, pp. 8, 9). This would make it difficult for developers to adapt to changing circumstances without significant re-engineering.

The primary aim of this work is to enable developers to *decide about deployment later* by facilitating the development of applications capable of hybrid deployment. Such applications would provide the flexibility to respond to changing conditions and requirements, allowing developers to transition seamlessly between serverful and serverless deployment targets as needed, without being locked into a specific deployment target. The goal of this work is therefore to explore how developers can build such applications, allowing them to leverage the strengths of deployment targets as changing conditions dictate. This work also aims to provide practical recommendations for navigating the complexities of deployment targets. By uncovering the key factors that influence these decisions, the research aims to provide practitioners with the insights they need to make informed choices.

To address these goals, the following research questions guide this work:

RQ1 What are the key factors, and how do they influence the choice of deployment target for cloud-native applications?

RQ2 How can application components be enabled to shift within the spectrum of serverful and serverless deployment targets?

By answering these questions, this work aims to contribute both theoretical and practical knowledge to the field of cloud-native application deployment, ultimately improving the developer experience and the flexibility of applications in this dynamic environment.

1.3 Structure of this Work

The following chapters are structured as follows:

- Chapter 2 **Literature Review:** presents the approach to and the review of relevant literature and related work in the field of cloud-native applications and deployment targets.
- Chapter 3 **Methodology:** provides an insight into the methodology used throughout this work to answer the research questions posed.
- Chapter 4 **Results:** presents the contributions of this work, answering the research questions and evaluating them using the methods described.
- Chapter 5 **Conclusion:** summarizes the contributions and concludes this work and its research questions, discusses the results, and their limitations, and provides an outlook for future work.

Chapter 2

Literature Review

A literature review was conducted to provide a basis for further work. The following sections present the results of this review, which is structured as follows: first, the approach to the literature review is described, and then the literature relevant to the research questions is presented.

2.1 Approach

Given the research questions posed in section 1.2, a literature review was conducted, starting with a systematic but exploratory on-topic search for relevant literature in the field of cloud-native development and deployment. Databases and archives used for the search included Google Scholar[1], IEEE Xplore[2], SpringerLink[3], ACM Digital Library[4], ResearchGate[5] and arXiv[6]. In addition, Google Search[7] and Microsoft Copilot[8] were also used to find relevant literature. The search was conducted using a combination of keywords related to cloud-native development and deployment: cloud and cloud-native, development and deployment, serverless, microservices, challenges, FaaS, PaaS, IaaS, AWS, Azure. In addition, the search focused on, but was not limited to, conference papers, surveys, and studies, as these are considered to be the most relevant sources of information on the topic.

After an initial search of the primary literature, additional relevant literature was identified using the backward snowballing technique as outlined by Wohlin (2014). This method involves tracing the references of key literature to identify further relevant literature, on the assumption that references cited in valuable literature are also likely to be relevant. Most of the literature cited here was discovered using this approach.

2.2 Related Literature

The literature review was used to build a structured knowledge base on the topic of cloud-native development and deployment. In the following sections, the related literature is presented as a literary base by crystallized topics of interest for this work.

[1] https://scholar.google.com/
[2] https://ieeexplore.ieee.org/
[3] https://link.springer.com/
[4] https://dl.acm.org/
[5] https://www.researchgate.net/
[6] https://arxiv.org/
[7] https://Google.com/
[8] https://copilot.microsoft.com/

Cloud-native Development and Architecture

To set the scene, an understanding of cloud-native development, architecture, and deployment was gathered.

The FreeWheel Biz-UI Team (2024) in their book provides a comprehensive guide to cloud-native microservice application architecture, covering key principles, best practices, and strategies for developing, deploying, and maintaining such systems. While the primary focus is on microservices, the book also devotes a chapter to serverless architecture, discussing how serverless technologies relate to microservice-based development as a new application architecture.

Rajan (2020) and Taibi, Kehoe, et al. (2022) reviewed serverless architectures within serverless functions. The former focused on reviewing the literature to identify and categorize different patterns of serverless functions, discussing their characteristics and use cases.

In their work, Deng et al. (2023) surveyed cloud-native development, focusing on the issues faced by developers throughout the lifecycle of a service. They relate service-oriented software to microservices as the architecture of choice and relate the practice to serverless computing.

Deployment Targets (and the Cloud Service Stack)

The cloud service stack can be used as a model for classifying deployment targets at a basic level. Mohammed et al. (2021) clarify the distinctions between its base layers of IaaS, PaaS, and SaaS.

In addition, the literature (Castro, Ishakian, et al. 2019; FreeWheel Biz-UI Team 2024; Jonas et al. 2019; Kaushik et al. 2022) points to extensions of this model, including more specialized layers such as Container- (CaaS), Function- (FaaS), and Backend-as-a-service (BaaS). CaaS facilitates the deployment of containerized applications, such as with managed Kubernetes services, for example Azure Kubernetes Service (AKS)[9] while FaaS abstracts the infrastructure down to the function or code level, exemplified by AWS Lambda[10]. BaaS, on the other hand, includes backend services like authentication or storage, as seen with AWS Cognito or AWS S3[11].

In particular, Kaushik et al. (2022) distinguish the additional layers of CaaS and serverless computing (FaaS) from the base layers of the model. They also argue for a more detailed approach to the cloud service stack. They delve into the three categories of IaaS, PaaS, and SaaS, differentiating between compute, network, and storage to better categorize cloud products.

The extended cloud service stack and Kaushik et al. (2022)'s approach serve as a basic reference throughout this work for categorizing deployment targets.

In addition, Yussupov, Soldani, et al. (2021) explored different approaches to application hosting and propose a spectrum of patterns ranging from serverful to serverless deployments. They distinguish between *serverful*, *serverless*, and hybrid hosting patterns, with the latter further subdivided into sub-patterns: *consumer-* or *provider-managed containerization* and *provider-managed deployment stack*. The authors distinguish these patterns based on the management responsibility of the applications: whether either the consumer or the provider manages the scaling configuration and/or the deployment stack. They also argue for a broadening of the term serverless, which is usually used as a synonym for FaaS (Yussupov, Breitenbücher, Leymann, et al. 2019, p. 273; Castro, Isahagian, et al. 2022, p. 4). In their work, they propose a view of a *serverless zone* consisting of the patterns *provider-managed containerization*, *provider-managed deployment stack*, and *serverless*. This view defines serverless as a spectrum rather than a single deployment target. It is nuanced by how pronounced the two dimensions for a hosting pattern are towards serverless.

[9] https://azure.microsoft.com/en-us/products/kubernetes-service
[10] https://aws.amazon.com/lambda/
[11] https://aws.amazon.com/s3/

Later, Yussupov, Breitenbücher, Brogi, et al. (2022) complement their earlier work with further exploration of the two critical dimensions. They propose four new patterns and an approach to identifying appropriate hosting options and the rationale for the appropriate deployment stack.

Additional factors influencing these patterns, such as cost or workload characteristics, were left rather untouched and are explored in this work. Building on this foundation, this work delves deeper into the trade-offs between serverful and serverless approaches to develop a more nuanced understanding.

About Serverless

Roberts et al. (2017) provide an in-depth introduction to serverless, defining the term and distinguishing it from other cloud technologies. The authors also identify key characteristics of serverless, including automatic scaling, abstracted infrastructure management, pay-per-use pricing models, and high availability. In addition, they explored the characteristics of a typical serverless application, outlining both the benefits — such as reduced operational overhead — and the limitations, including issues of control and performance.

Castro, Ishakian, et al. (2019) highlight the rise and trace the emergence of serverless. They examine the adoption and practice of serverless to provide an overview of the current state of serverless computing.

Baldini et al. (2017), Jonas et al. (2019), and Lannurien et al. (2023) provide a more detailed insight into the current state of serverless computing: discussing advantages and drawbacks, exploring practices, examining challenges and mitigations, and concluding open issues within practice. In particular, Lannurien et al. (2023) explored the limitations of serverless and present approaches in the literature to mitigate limitations such as latency due to cold starts. In addition, Eismann et al. (2021b) further discuss the reasons for adopting serverless, when to use it, and how to implement it in practice.

Leitner et al. (2019) conducted a mixed-method empirical study using interviews, literature analysis, and surveys to gain a better understanding of FaaS adoption in industry practice. Similarly, Hassan et al. (2021) and Shafiei et al. (2021) surveyed serverless computing to understand the current state of serverless computing research and adoption. Both identified key issues and challenges in the practice of serverless computing, such as pricing and costs, cold starts, vendor lock-in, or its programming model in terms of testing or debugging.

In their work, Taibi, Kehoe, et al. (2022) interviewed 91 practitioners to explore developers' perceived bad practices when developing serverless applications. They proposed a catalog of bad practices, such as not adapting to the event-driven paradigm or misunderstanding the granularity of functionality.

Overall, the literature reviewed provides a comprehensive understanding of the current landscape of serverless computing. Works such as Roberts et al. (2017), Castro, Ishakian, et al. (2019) and Baldini et al. (2017) provide fundamental insights into the evolution, benefits, and challenges of serverless architectures, while specific studies such as those by Lannurien et al. (2023) and Eismann et al. (2021b) delved deeper into the limitations and practical considerations for adoption. In addition, the empirical research of Leitner et al. (2019), Taibi, Kehoe, et al. (2022), Shafiei et al. (2021) and Hassan et al. (2021) further enrich the understanding of serverless in both research and industry practice. Together, these sources have been invaluable in shaping this work, providing critical insights, and guiding the exploration of the serverless landscape.

Parameters, High-Level Project Aspects, and Cost

A key aspect of the literature review was to crystallize key parameters and their influence on the decision between serverful and serverless deployment targets, in addition to the benefits and drawbacks.

Villamizar et al. (2017) for example, look at cost comparisons between serverful, CaaS, and FaaS deployment targets, highlighting the importance of cost efficiency in the decision-making process. Reuter et al. (2020), on the other hand, explicitly explored and studied cost efficiency with mixed serverful and serverless deployment targets based on load distributions. Both validate cost as a key parameter influencing the decision between serverful and serverless deployment targets.

In related work, Fan et al. (2020) look at performance comparisons between microservices and serverless deployments. They found differences in the performance and cost implications between the two deployment targets, but emphasized that no single target was found to be superior in all cases.

Eivy et al. (2017) model economics of serverless cloud computing. They explored the cost implications of serverless deployment targets and parameters such as peak and average load, and the scalability versus economics of serverless deployments. They highlight the importance of considering the cost and potential scale of applications to suit the serverless deployment model.

In addition, Yussupov, Breitenbücher, Leymann, et al. (2019) examine the migration of serverless applications for different use cases. In addition to addressing the benefits and drawbacks of serverless deployment targets for such use cases, they explored and categorized different dimensions of the lock-in aspect for applications. They conclude that serverless does indeed amplify vendor lock-in.

In related work, Lloyd et al. (2018) present a case study on migrating a Java[12]-based application to AWS Lambda and investigated performance and cost implications. Migrating the application to their liking, they reason for saving costs and give insights into leveraging antipatterns such as keep-alive workloads to achieve better latency.

In addition, Sfondrini et al. (2018) investigated enterprise cloud adoption and tracked high-level business and technical issues in the adoption process.

The literature reviewed provides important insights into the factors that influence the choice between serverful and serverless deployment targets. Studies such as Villamizar et al. (2017) and Reuter et al. (2020) emphasize cost efficiency, while Fan et al. (2020) and Eivy et al. (2017) focus on performance and scalability. In addition, Yussupov, Breitenbücher, Leymann, et al. (2019) and Lloyd et al. (2018) address vendor lock-in and migration. Together, this literature highlights some complex trade-offs and has shaped the direction of the parameters explored in this work.

Serverful and Serverless Use-Cases

In addition, the serverful and serverless use cases discussed in the literature were examined, with a particular focus on application types and workload characteristics. While some of the previously mentioned literature provided information on this topic, the following literature was found to be particularly relevant.

Shrestha et al. (2022) conducted a case study using an image processing application. They compared microservices and serverless deployments, highlighting the suitability characteristics for each deployment target. They conclude that serverless would outperform in terms of performance and cost for their use case, while microservices would outperform in terms of controllability.

In their work, Eismann et al. (2021a) collected and reviewed serverless use cases to identify common characteristics and value propositions. They highlight key findings in terms of characteristics such as workload, applications, workflows, and requirements derived from their analysis.

[12]https://www.java.com/en/

Concluding the Literature Review

In conclusion, the literature review for this work has comprehensively explored the landscape of cloud-native development and architecture, with a particular focus on serverless deployment targets.

The review of cloud-native development has provided a solid understanding of microservices and serverless architectures, with works such as FreeWheel Biz-UI Team (2024) and Rajan (2020) providing critical insights into architectural patterns and service-oriented approaches. The review of deployment targets has clarified the distinctions between different layers of the cloud service stack, highlighting the roles of IaaS, PaaS, SaaS, CaaS, FaaS, and BaaS as described by Mohammed et al. (2021) and Kaushik et al. (2022).

A significant part of the review was devoted to serverless computing, with seminal works such as Roberts et al. (2017) and Baldini et al. (2017) providing basic definitions and insights into its benefits and challenges. Empirical studies from Leitner et al. (2019) and Taibi, Kehoe, et al. (2022) further enriched the understanding of serverless practices and limitations.

The review also addressed key decision parameters such as cost, performance, and vendor lock-in, drawing on sources such as Villamizar et al. (2017) and Yussupov, Breitenbücher, Leymann, et al. (2019). These factors, together with insights into use cases and workload characteristics from Shrestha et al. (2022) and Eismann et al. (2021a), provide a comprehensive basis for analyzing the choice of deployment targets.

Significantly, while the review has been organized into specific sections for clarity, it is important to note that the sources contribute valuable information across multiple topics. For example, foundational work on serverless computing not only provided insights into serverless architecture and deployment targets, but also offered perspectives on cost, performance, and use cases relevant to other aspects. Similarly, studies on deployment targets illuminated different layers of the cloud service stack, which also informed discussions on cost efficiency and migration challenges.

The interrelated nature of these topics means that much of the literature reviewed has broader implications and benefits beyond the specific sections in which it was discussed. This holistic approach has enriched the overall understanding of the subject and will support a more nuanced exploration of the decision-making processes in subsequent chapters of this work. The integration of these different insights will help to address the complex trade-offs between serverful and serverless deployment targets and effectively navigate their landscape.

Chapter 3

Methodology

The research described in this work was conducted using a combination of literature synthesis, qualitative interviews and surveys, technical experiments, and performance benchmarks to gather evidence, evaluate approaches, and assess the feasibility of findings. The GitHub repository linked to this work is available here. The following sections describe the methods used to answer the research questions posed in section 1.2.

3.1 Information Gathering

Based on the information gathered from the literature in section 2.2, an interpretive synthesis was conducted as outlined in Cooper et al. (2009). It summarizes and synthesizes the relevant information gathered into a coherent picture of the landscape of cloud-native deployment targets to answer **RQ1**.

The process involved identifying common themes, patterns, and trends in the literature and contrasting different viewpoints and approaches. It also involved identifying gaps in the literature and areas where further research is needed.

In addition, the synthesized knowledge of the landscape was combined with information gathered from interviews and surveys to reinforce a comprehensive view of the current state of cloud-native deployment targets and practices within the landscape.

3.2 Interviews and Surveys

In addition to gathering information from the literature and gaining practical insights, thematic-focused interviews were conducted with practitioners in different roles within DI. The interviews were semi-structured, with discussion points kept on topic and interviewees free to answer as they saw fit. Some interviewees were also interviewed several times but on different topics.

Regarding Information Gathering

The purpose of these interviews and discussions was to gather information, experiences, and expert opinions on various topics such as: developing and operating FaaS-based applications, deployment targets, development models, and application architecture. The interviews and discussions also identified potential pain points and solution requirements. Whilst focusing on the key issues, the interviews provided valuable qualitative data to help guide the research towards answering **RQ1** and **RQ2**.

The interview with the *Cloud Acceleration Team* (CAT) in section B.1 provided insights into development and deployment practices and high-level project aspects and requirements in practice.

Interviews with developers from *Project X* and *Project B* in section B.2 and section B.3 respectively, provided insights into the development and deployment practices of the respective teams within the project, as well as their experiences, expert opinions, and pain points with different deployment targets and development models. *Project X* is a project using FaaS to provide a REST API and typical Internet of Things (IoT) data processing; *Project B* uses a similar architectural approach.

Regarding Evaluation

In addition to the information-gathering interviews, practitioner interviews, and two surveys were conducted to evaluate the research findings and gather feedback on the proposed recommended approaches to **RQ2**. The survey results are also featured in the GitHub repository linked to this work[1].

The interview with *Practitioner S* in section B.5 provided valuable feedback on the approaches to hybrid deployment and insights into the practical feasibility of the research findings, as well as informed directions for evaluation.

The interviews with *Practitioner J* and *Practitioner A* - section B.4 and section B.6 respectively — focused on gaining insights and expert opinions, to elicit open points for discussion as a measure of evaluation.

In addition, the survey of *Project D* developers in Appendix C aimed to gain insight into development practices and experiences. The survey was conducted to assess the practicality of the research findings, although some findings are also informed by the survey.

The developer of *Project K* was interviewed in Appendix D after having accompanied a migration using one of the proposed recommended approaches, to assess the practical feasibility of the research findings in a real-world scenario. The migration consisted of moving a previously serverful application, deployed on Kubernetes, to a serverless architecture on a public cloud.

3.3 Technical Research

In addition to practitioner input to further enrich the information gathered, technical research was conducted to explore relevant technologies and frameworks, consisting of spikes and experiments to gain insights that may be useful in answering **RQ2**. The GitHub repository linked to this work[2] contains such spikes and experiments also.

This included experiments and field tests with programming languages such as Go[3], Java and TypeScript[4]. Frameworks experimented with included, for example Encore[5], Ampt[6], Spring Cloud Functions[7] or Quarkus Funqy[8]. Technologies tested include OpenAPI[9], Smithy 2.0[10], AsyncAPI[11], GOA[12]. Typical applications such as REST APIs with database backends were used to test the practical applicability of the technologies. A typical IoT domain reference application was developed and used as a basis for such discussions, considerations, and experiments. In addition, public cloud offerings and services, though mainly AWS[13], but also Azure[14], were used

[1] https://github.com/a-grasso/master-thesis-public/tree/main/google-forms-survey-results
[2] https://github.com/a-grasso/master-thesis-public/tree/main/spikes
[3] https://go.dev/
[4] https://www.typescriptlang.org/
[5] https://encore.dev/
[6] https://www.getampt.com/
[7] https://spring.io/projects/spring-cloud-function
[8] https://quarkus.io/guides/funqy
[9] https://www.openapis.org/
[10] https://smithy.io/
[11] https://www.asyncapi.com/
[12] https://goa.design/
[13] https://aws.amazon.com/
[14] https://azure.microsoft.com/en-us/

to deploy or experiment with applications. Private cloud offerings such as OpenFaaS[15] were also considered.

These experiments helped to evaluate the capabilities of existing tools and frameworks and to identify potential challenges and limitations to proposed recommended approaches. In a controlled but realistic environment, the performance, flexibility, and integration capabilities of different tools could be evaluated to inform the research findings and further research directions.

3.4 Benchmarks and Performance Testing

For evaluation purposes, performance benchmarks were conducted firstly to assess the feasibility and to validate and analyze the performance of the research findings towards **RQ2**. Secondly, the benchmarks were intended to provide further insight into the influencing factors and requirements identified in section 4.2 towards **RQ1** to further evaluate and validate the research findings. The full benchmark setup and details beyond the following description, including infrastructure provisioning, are available in the GitHub repository linked to this work[16].

Benchmarks were conducted using the K6[17] load testing tool to simulate workloads and collect performance data and benchmark metrics, taking into account the best practices detailed by Bermbach et al. (2017). The K6 tool was chosen for its ease of use and flexibility in simulating traffic behavior.

The benchmarks involved using two example use cases: a service to compute a Fibonacci number chosen so that the request duration itself would be short; and a service to encrypt, hash, and store files so that the request duration would be longer. The use cases were chosen to evaluate the impact of different request durations on the performance. The services were developed in Go and aimed to have an average computation time in the low milliseconds ($< 50ms$) for the Fibonacci service and in the low seconds ($\approx 10s$) for the File service. Both are CPU-bound services, with the File service being more CPU-intensive by design than the Fibonacci service.

Scenarios have been designed to simulate different traffic patterns and evaluate performance under different conditions. The scenarios are based on Grafana's load test types[18]. K6 offers customizable load scenarios, so different load scenarios were developed and used where appropriate, with scenario parameters changed depending on the service being tested:

Constant Rate: A fixed number of requests per second over a period of time:

- **Fibonacci**: 50 requests per second for one hour
- **File**: One request per second for one hour.

Spike: Within a short period of time, the load increases to a high number of requests per second:

- **Fibonacci**: Reaches 2000 requests per second within two minutes, then ramps down to no requests per second over one minute.
- **File**: Reaches 20 requests per second within two minutes, then ramps down to no requests per second over one minute.

Constant VU: A virtual user (VU) — the term comes from K6 and represents one concurrent user[19] — makes as many non-concurrent requests as possible for 30 minutes.

[15] https://www.openfaas.com/
[16] https://github.com/a-grasso/master-thesis-public/tree/main/evaluation-benchmarks
[17] https://k6.io/
[18] https://grafana.com/load-testing/types-of-load-testing/
[19] https://grafana.com/docs/k6/latest/using-k6/scenarios/executors/constant-vus/

The **Constant Rate** scenario is used to evaluate performance under a constant load and acts as the main scenario for further comparisons such as cost calculations. **Spike** is used to evaluate performance under sudden bursts of load to assess scalability and elasticity capabilities. **Constant VU** is used as an initial performance baseline to compare what the deployment targets would be capable of.

Infrastructure was set up to deploy the services on various deployment targets for the benchmarks. The deployment targets included AWS Lambda, AWS ECS[20] based on Fargate[21], AWS App Runner[22] and a Hetzner CCX23 virtual machine[23]. Benchmarks were run locally using the K6 client. In the case of the File service, an S3 bucket was used to store files. Figure 3.1 shows the deployment diagram of the benchmark setup for the File service as an example.

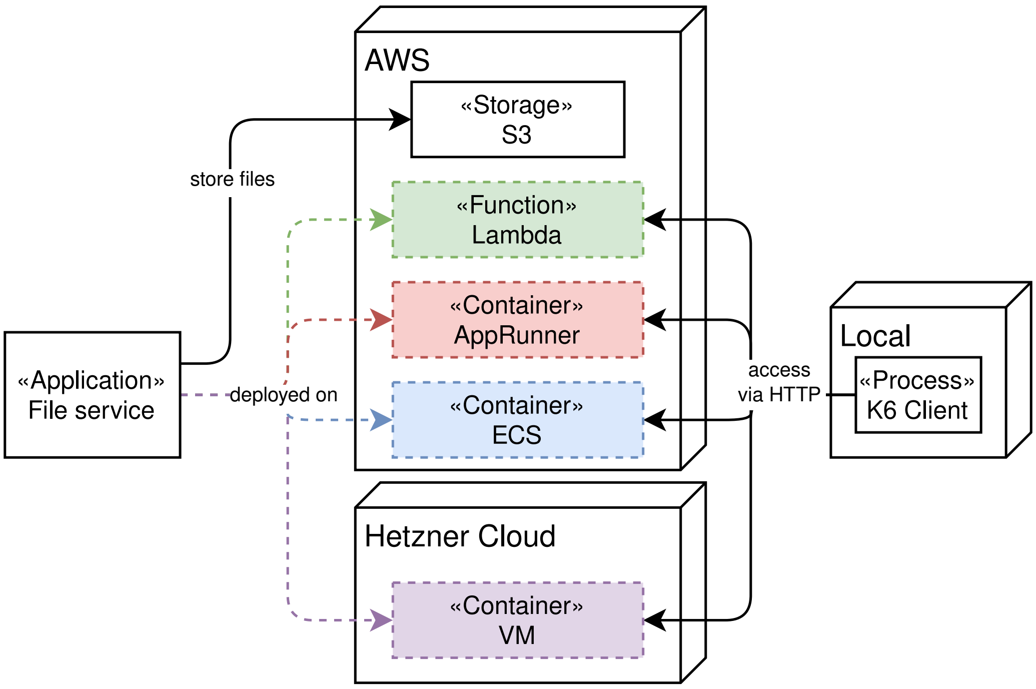


Figure 3.1: *Deployment diagram of the benchmark setup for performance testing the File service. The setup includes a K6 client running on a local machine, dashed and color-coded boxes indicate where the File service application is deployed on: AWS Lambda, AWS ECS based on Fargate, AWS App Runner or a Hetzner VM — files are stored in an AWS S3 bucket.*

The AWS services were configured in terms of memory and CPU to reach comparable performance levels. This involved:

- **Lambda**: Number of vCPUs depends on memory allocation[24]
  - **Fibonacci**: 1769 MB memory for one vCPU.
  - **File**: 1769 MB memory for one vCPU — kept the same as for the Fibonacci service, as concurrent requests are not possible with one Lambda instance[25], more resources would distort the results.
- **Fargate**:

[20] https://aws.amazon.com/ecs/
[21] https://aws.amazon.com/fargate/
[22] https://aws.amazon.com/apprunner
[23] https://www.hetzner.com/cloud/
[24] See here (docs.aws.amazon.com)
[25] https://docs.aws.amazon.com/lambda/latest/dg/lambda-concurrency.html

  - **Fibonacci**: 1vCPU and 2GB of memory.
  - **File**: 4vCPUs and 8GB of memory.

- **App Runner**:
  - **Fibonacci**: 1vCPU and 2GB of memory.
  - **File**: 4vCPUs and 8GB of memory.

- **Hetzner VM**: CCX23[26] with dedicated 4vCPUs and 16GB of memory
  - **Fibonacci**: Limited to 1vCPU with Docker resource constraints[27].
  - **File**: Full use of the 4vCPUs and 16GB of memory.

Scaling behavior has been configured to not skew the results and to achieve comparable performance levels. The deployment targets for the Fibonacci service benchmarks have been configured to scale automatically, except for the Hetzner VM: The App Runner has autoscaling capabilities that max out at 200 concurrent requests per instance before scaling out[28] to a maximum of 25 instances; This behavior has been replicated for ECS based on Fargate, but dependent on 200 concurrent Application Load Balancer[29] (ALB) requests per target for three data points within three minutes, up to 10 instances; The Lambdas would scale automatically based on the number of requests and allocated memory, no further scaling configuration was done[30].

For the File service benchmarks, more complex custom scaling was configured to handle concurrent CPU-intensive loads: The App Runner only scales based on the number of requests, so scaling was lowered to a maximum of three concurrent requests before scaling out to a maximum of 25 instances; The Fargate service has been configured to scale based on CPU usage, scaling up to a maximum of ten instances if CPU usage is above 80% for three data points within three minutes; The Lambda service retained its default scaling behavior; No scaling was configured for the Hetzner VM.

Data was collected using K6 as mentioned above, with the data stored in InfluxDB[31], which was also used for analysis and visualization. Metrics collected during the benchmarks include standard K6 metrics such as response time buckets, requests per second, and data received and sent. External metrics were also collected, some of which were manually read, tracked and stored, such as CPU usage, memory usage or concurrency, using AWS CloudWatch[32] and Hetzner Cloud monitoring[33]. The raw InfluxDB time series data referenced in this work is provided in the GitHub repository linked to this work also[34].

[26] https://www.hetzner.com/cloud/
[27] https://docs.docker.com/config/containers/resource_constraints
[28] https://docs.aws.amazon.com/apprunner/latest/dg/manage-autoscaling.html
[29] https://aws.amazon.com/elasticloadbalancing/application-load-balancer/
[30] https://docs.aws.amazon.com/lambda/latest/dg/lambda-concurrency.html
[31] https://www.influxdata.com/
[32] https://aws.amazon.com/cloudwatch/
[33] https://docs.hetzner.com/robot/dedicated-server/security/system-monitor
[34] https://github.com/a-grasso/master-thesis-public/tree/main/evaluation-benchmarks/influxdb-backup-tsm

Chapter 4

Results

4.1 What is Serverless anyway?

The term *serverless* lacks a universally acceptable definition (Shafiei et al. 2021, p. 3). No common single definition captures the full scope of serverless concisely. Instead, a variety of definitions exist which may overlap or complement each other while each emphasizing different aspects or characteristics about serverless or its technologies.

The Cloud Native Computing Foundation[1] (CNCF) definition[2] for example presents an array of characteristics regarding infrastructure, scaling and pricing within serverless. Roberts et al. (2017, pp. 40–42) focus their serverless definition on high-level requirements for a service to be considered as such. Rajan (2020) on the other hand, focuses on *serverless computing* and associates FaaS as the main serverless platform. The association with FaaS is widespread among literature and industry (Baldini et al. 2017, p. 18; Jonas et al. 2019, p. 4; Hassan et al. 2021, p. 2; Yussupov, Breitenbücher, Leymann, et al. 2019, p. 1). The survey conducted by Leitner et al. (2019) specifically had 58 percent of practitioners defining serverless as FaaS. Interviews with practitioners also revealed a strong association between serverless and FaaS (B.1, B.2, B.3, B.5). Furthermore, Kaushik et al. (2022, pp. 93–94) describe serverless generally as "convenience to deploy application code [. . .] without the hassle [. . .]", whilst exclusively giving FaaS services such as AWS Lambda, Google Cloud Run functions[3] or Apache OpenWhisk[4] as an example. Otherwise, Baldini et al. (2017, p. 3) see FaaS as merely a version of serverless computing, but not the only one. In relation, Shafiei et al. (2021, p. 3) and Jonas et al. (2019, p. 4) but also Roberts et al. (2017, p. 6) argue that serverless is a generalization and incorporates both FaaS and also BaaS offerings.

With all these definitions and perspectives, it is clear that serverless is a complex term with a variety of interpretations and evidently a notion towards FaaS specifically. However, all these definitions share a common thread that runs through them, pointing to the core attributes of serverless. This section further explores such definitions, characteristics, and perspectives to synthesize a comprehensive understanding of the term *serverless*.

4.1.1 SP1: *costs nothing when not in use*

A core attribute linked to serverless is the principle of *costs nothing when not in use*, referred to as *pay-per-use*. Castro, Ishakian, et al. (2019, p. 47) quotes Paul Johnston, co-founder of 'ServerlessDays'[5], defining serverless as: "costs you nothing to run if nobody is using it (excluding data storage)". This is widely acknowledged in both literature and industry as a

[1] `https://www.cncf.io/`
[2] `https://glossary.cncf.io/serverless/`
[3] `https://cloud.google.com/functions?hl=en`
[4] `https://openwhisk.apache.org/`
[5] `https://serverlessdays.io/`

defining attribute of serverless (Baldini et al. 2017, p. 6; Hassan et al. 2021, pp. 9, 16; Castro, Isahagian, et al. 2022, p. 10; Jonas et al. 2019, p. 8; Roberts et al. 2017, pp. 19, 41). The principle of *costs nothing when not in use* reflects a pricing model where resources are only billed based on actual usage. This model is central to serverless, ensuring that users are only charged for the exact resources consumed during activity.

Detrimental to this principle, is the question of what constitutes *usage*. In terms of compute, this is straightforward: usage can be measured by execution time, CPU or memory, or the number of requests (Baldini et al. 2017, p. 6). For storage, usage could be measured in two aspects: *a)* the amount of data persisted over a given time and *b)* the number of read and write operations. Network usage could be measured by the amount of data transmitted or received, or the number of requests made.

Johnston's definition of serverless notably excludes data storage. The *costs nothing when not in use* principle applies primarily to compute resources, which can be dynamically allocated and de-allocated (Hassan et al. 2021, p. 9). When compute is not in use, resources are released, incurring no further costs. This behavior is referred to as *scale-to-zero*. It is the pay-per-use related behavior associated with serverless computing (Baldini et al. 2017, p. 6). However, scale-to-zero is generally less applicable to storage and network infrastructure (Castro, Isahagian, et al. 2022, p. 11) and depends on the definition of usage for a given resource to denote such behavior.

Public cloud providers, such as AWS, have used the "serverless" label to market a variety of services, though the degree to which these services adhere to the *costs nothing when not in use* principle can vary. For example, AWS Lambda is marketed as a serverless platform, with its pricing model based on the number of requests and execution duration. When not in use, Lambda incurs no cost, it scales to zero aligning well with the pay-per-use characteristic. On the other hand, AWS S3, a storage service, operates on a fully *pay-per-use* basis. Data stored in S3 remains persistent, and users are charged based on the amount of data stored and accessed rather than the underlying infrastructure. This is in line with the *pay-per-use* principle, by which S3 can be considered serverless. It is worth noting that S3 can also be considered to scale to zero with the way storage usage has been defined and acts as an exception to the rule for storage. The amount of data persisted can be reduced to zero and storage resources are released, so there are no costs when S3 is not in use. Further complicating the serverless term is Amazon Aurora Serverless[6] (Castro, Ishakian, et al. 2019, p. 46), a database service marketed as serverless. Despite its name, Aurora Serverless does not fully adhere to the *costs nothing when not in use* principle. The service charges are based on Aurora Capacity Units (ACUs), with a minimum configuration of 0.5 ACUs, meaning there is always some cost, even when the service is idle or has never been used. It contradicts the *costs nothing when not in use* principle, as it is not scalable to zero and does not fully embody *pay-per-use*, skewing the perception of serverless[7]. Furthermore, AWS Route 53[8], a DNS service, is marketed as pay-per-use. Again, how usage is defined and how the service is billed is important. For its main use, DNS queries, Route 53 is billed based on the number of queries made to the service. If no DNS queries are served, Route 53 incurs no cost, aligning with the *costs nothing when not in use* principle, one could also argue it scales to zero in this regard. However, customers are charged for managing hosted zones, regardless of the number of queries made. How the sole managing of hosted zones itself counts towards usage is up to debate, which may contradict *pay-per-use*.

The examples discussed above highlight the nuanced complexity of the *costs nothing when not in use* principle and the challenges associated with its applicability in cloud environments. In

[6]https://aws.amazon.com/rds/aurora/serverless/

[7]This work does not presume to make a judgment on the inclusion of "serverless" in the name of the database service, rather it remains open to debate

[8]https://aws.amazon.com/route53/

contrast, self-hosted web applications on Kubernetes, a PostgreSQL[9] database, or an NGINX[10] reverse proxy can incur costs by existing, regardless of actual usage, simply by the fact that they are self-hosted and resources are being allocated to them, even when idle. Cloud providers may use a similar underlying infrastructure for the services discussed. However, the key difference lies in how these services are managed by the provider and how users are billed. The billing models used by cloud providers do not always follow the principle of *costs nothing when not in use*. Instead, they charge based on pre-defined pricing structures, which can include fixed costs for provisioning and resource allocation, regardless of actual usage.

In defining serverless, it is important to recognize the underlying principle of *costs nothing when not in use* and how it applies to different types of services and infrastructure. While the concepts of scale-to-zero and pay-per-use are integral to serverless, they are arguably not the same and their relevance varies depending on the nature of the resource — be it compute, storage, or network. As shown, compute resources are more in line with the *costs nothing when not in use* principle, whereas storage and networking can incur costs regardless of usage, raising debates about what qualifies as serverless in this regard. This will depend on the specific service, its pricing model, and how the underlying infrastructure is managed, as well as the definition of usage and how resources are billed, which may or may not be consistent with the *costs nothing when not in use* principle.

Nevertheless, the principle of *costs nothing when not in use* remains a critical feature in characterizing serverless. While its application is not universal across all infrastructure types, it is a fundamental concept for understanding the economic and functional dimensions of serverless. It serves the purpose as a sufficient attribute to define serverless. If a component adheres to this single principle, it can be considered serverless already.

4.1.2 SP2: *no responsibility for infrastructure management*

In their requirements for referring to a service as serverless, Roberts et al. (2017) denotes "does not require managing a long-lived host or application instance" as "the heart of Serverless"(Roberts et al. 2017, p. 40). This aspect is consistently echoed in related works, which emphasize the importance of abstracting infrastructure management responsibilities in serverless (Baldini et al. 2017, p. 2; Hassan et al. 2021, pp. 2, 9; FreeWheel Biz-UI Team 2024, p. 180) to allow developers to focus on building and deploying application logic (Castro, Isahagian, et al. 2022, p. 4; Deng et al. 2023, p. 12). While Roberts et al. (2017) explain this requirement as an indication that developers should not be concerned with or be responsible for servers, the statement itself lacks depth and specificity. Exploring related literature expands this aspect, leading to the more comprehensive principle of *no responsibility for infrastructure management* as a defining attribute of serverless.

Understanding this principle requires examining its key aspect: the implications of having no responsibility for the scope of infrastructure management. No responsibility highlights that while infrastructure management describes the *what*, the core defining attribute regarding serverless lies in the *who* — who is responsible for it?

In a serverful environment, the principle of *no responsibility for infrastructure management* does not apply. Developers, operators, or internal IT teams are responsible for provisioning, maintaining, and operating resources (Jonas et al. 2019, p. 5; Baldini et al. 2017, pp. 2, 3; Castro, Ishakian, et al. 2019, p. 46; Eivy et al. 2017, p. 12). Now, whether internal IT teams offer resources in a self-service manner or not, similar to cloud providers, and how this deviates from serverful, is a different question for later discussions in subsection 4.1.6. Within the public cloud, as the cloud provider takes on more infrastructure management, the responsibility shifts: from IaaS over CaaS or PaaS to FaaS or BaaS.

[9] `https://www.postgresql.org/`
[10] `https://nginx.org/en/`

The first stage where *no responsibility* can emerge is in *provisioning* resources (Deng et al. 2023, p. 12; Baldini et al. 2017, pp. 2, 5, 15; Jonas et al. 2019, pp. 4, 6). The cloud simplifies this process with its IaaS offerings. Rather than developers purchasing hardware, cloud providers or data center operators (e.g., Hetzner[11]) offer virtual machines that can be rented and configured to then run applications on, removing much of the provisioning part and reducing it to mostly configuration (Baldini et al. 2017, p. 3). For example, AWS EC2[12] and Azure Virtual Machines[13] allow developers to select VM sizes and configurations while the cloud provider manages the underlying infrastructure. However, this alone does not constitute *no responsibility for infrastructure management,* as operation and maintenance are still required.

With IaaS, developers remain responsible for operation and maintenance tasks (Castro, Ishakian, et al. 2019, p. 48). CaaS offerings like AKS or AWS Elastic Kubernetes Service[14] (EKS) further reduce operational and maintenance responsibilities when deploying containerized applications (Kaushik et al. 2022, p. 95; Jonas et al. 2019, p. 8; FreeWheel Biz-UI Team 2024, p. 181). AWS ECS based on EC2[15] is also a CaaS offering, where developers deploy containerized applications on EC2 instances. However, developers remain responsible for the EC2 instance in terms of operation and maintenance.

As cloud providers take on more responsibilities, the model shifts towards PaaS (Kaushik et al. 2022, p. 93). With EC2, the cloud provider does not check for performance issues, nor do they update operating systems or software. With managed services like AWS RDS[16] or Azure SQL Database[17], cloud providers ease operational and maintenance burdens. RDS, for example, is a managed database service that handles the provisioning, operation, and maintenance of databases. Developers can use such services without having to worry about the underlying infrastructure.

In their work, Yussupov, Soldani, et al. (2021) propose a dimension regarding the responsibility of the deployment stack, which aligns with the principle of *no responsibility for infrastructure management*. They argue that PaaS services managed by cloud providers fulfill this principle (Yussupov, Soldani, et al. 2021, p. 32). In that regard, CaaS services like AKS or EKS, excluding ECS based on EC2, also fulfill the principle of *no responsibility for infrastructure management* for developers. However, most PaaS offerings are generally not considered serverless (Leitner et al. 2019, p. 8).

FaaS and BaaS offerings exemplify the principle of *no responsibility for infrastructure management* best (Roberts et al. 2017, p. 6). While PaaS or CaaS may abstract some concerns, they do not fully embody this principle as FaaS or BaaS do. According to Kaushik et al. (2022), FaaS differentiates itself from PaaS by requiring developers to manage only application code, while the cloud provider handles everything else. This aligns with BaaS, in that developers, according to Roberts et al. (2017, p. 10), focus on outsourcing and configuring services instead of managing any infrastructure. In contrast to PaaS, FaaS and BaaS offerings abstract away much of the work regarding application scaling, security, compliance, and other infrastructure management concerns to the cloud provider. With services such as Lambda or S3, developers configure their compute or storage resources and the cloud provider takes care of the rest (Lannurien et al. 2023, p. 289; Roberts et al. 2017, pp. 20, 30). There is no need for resource provisioning, system upkeep, or maintaining security and performance updates for the underlying infrastructure. Both FaaS and BaaS fully abstract infrastructure management responsibilities from developers, aligning with the principle.

In essence, the principle of *no responsibility for infrastructure management* offloads infras-

[11]https://www.hetzner.com/
[12]https://aws.amazon.com/ec2/
[13]https://azure.microsoft.com/en-us/services/virtual-machines/
[14]https://aws.amazon.com/eks/
[15]https://docs.aws.amazon.com/AmazonECS/latest/developerguide/create-capacity.html
[16]https://aws.amazon.com/rds/
[17]https://azure.microsoft.com/en-us/services/sql-database/

tructure management tasks from developers. While they may still be aware of cloud resources and configurations (Eivy et al. 2017, p. 7), provisioning, operational, and maintenance tasks are handled by the provider. FaaS and BaaS, in contrast to PaaS and CaaS, fully embody this principle, enabling developers to concentrate on building and deploying application logic (Castro, Isahagian, et al. 2022, p. 4; Deng et al. 2023, p. 12). Thus, the principle of *no responsibility for infrastructure management* is a necessary attribute for defining serverless, but not a sufficient one.

4.1.3 SP3: *no responsibility for scaling configuration*

PaaS or CaaS offerings such as RDS or EKS are managed services and conform to the *no responsibility for infrastructure management*, but they are not serverless. Not to mention that such services are not *pay-per-use*, Yussupov, Soldani, et al. (2021, p. 38) further argue that such services fall short of fulfilling their second dimension: developers are still required to manage scaling configurations. Based on their work, the third key principle of serverless architecture, *no responsibility for scaling configuration*, can be established as a clear-cut differentiator.

The principle of *no responsibility for scaling configuration* strengthens the notion that developers should not need to concern themselves with the scaling of their applications — whether it is configuring scaling behavior or managing the underlying provisioning of resources (Roberts et al. 2017, p. 40; Castro, Isahagian, et al. 2022, p. 5). Literature groups scaling responsibilities with infrastructure management and operations, reinforcing the idea that developers do not need to consider how scaling is handled (Castro, Isahagian, et al. 2022, p. 10; Eismann et al. 2021b, p. 34). Developers though might configure desired scaling behavior on a high level, but they are not responsible for the actual execution or management of the scaling process itself (Yussupov, Soldani, et al. 2021, p. 37; Castro, Ishakian, et al. 2019, p. 47).

This principle distinctly differentiates serverless from PaaS and traditional CaaS offerings, where developers must still manage scaling. PaaS in particular is criticized for its lack of automatic scaling capabilities (Castro, Ishakian, et al. 2019, p. 47; Kaushik et al. 2022, p. 94).

ECS based on EC2 was already presented, though, AWS also offers AWS Fargate for CaaS offerings, with which they can be considered serverless. Fargate is a serverless compute engine for containers[18] that works with both ECS and EKS. As already mentioned, with ECS based on EC2, developers are still responsible for and operating the EC2 instances. With ECS or EKS based on Fargate however, developers only need to worry about the application and configure their containers, the cloud provider manages the rest. No need to maintain or operate underlying infrastructure, and no need to manage the scaling of applications. In addition, but unrelated to Fargate, AWS also offers the App Runner service, which is a fully managed service that automatically scales web applications specifically. Both Fargate and App Runner serve as examples of CaaS offerings that could be considered serverless, as they are fully managed services capable of automatically scaling applications without requiring developer intervention besides configuration. Similarly, Azure and Google Cloud[19] provide comparable services, such as Azure Container Apps[20] and Google Cloud Run[21] respectively.

The epitome of the principle of *no responsibility for scaling configuration* again is FaaS, where developers are not responsible for any scaling configuration. Developers can expect to have their functions automatically scaled by the cloud provider based on demand, without any effort (Castro, Ishakian, et al. 2019, p. 46; Lannurien et al. 2023, p. 285; Roberts et al. 2017, p. 40).

Thus, the principle of *no responsibility for scaling configuration* acts as a nuanced yet fundamental criterion for distinguishing serverless services from PaaS or traditional CaaS offerings.

[18] AWS markets this also as such: 'serverless compute for containers'.
[19] https://cloud.google.com/?hl=en
[20] https://azure.microsoft.com/en-us/products/container-apps
[21] https://cloud.google.com/run

FaaS and BaaS services are further solidified in their serverless standing by adhering to this principle, and offerings like Fargate can now be categorized as serverless based on the fact that developers have no responsibility for managing scaling configuration or infrastructure. With that, the principle of *no responsibility for scaling configuration* is established as a key differentiator and necessary attribute for serverless.

4.1.4 What Serverless should not be concerned with

Having defined attributes of serverless that arguably cover the distinctions needed to define it, it is also important to examine attributes and characteristics that the already established view does not benefit from or already incorporates: *what serverless should not be concerned with.*

Runtime management is a key distinguishing feature of FaaS within the context of serverless computing. Building on the association of FaaS with serverless (Baldini et al. 2017, p. 18; Hassan et al. 2021, p. 2), developers select the necessary runtime environments and build their applications on top of these environments (Yussupov, Soldani, et al. 2021, p. 37). The responsibility for managing the runtime itself is abstracted away from the developers (Yussupov, Soldani, et al. 2021, p. 37). This abstraction elevates FaaS within the serverless landscape.

In contrast, other offerings such as CaaS which can be classified as serverless under the broader established definition, do not offer the same level of runtime abstraction. For instance, with AWS Fargate, developers are responsible for containerizing their applications, effectively taking ownership of the runtime in the form of containers (Yussupov, Soldani, et al. 2021, p. 36).

This aspect of runtime management, while prominent in FaaS, is not included in the general definition of serverless to avoid restricting the concept to FaaS alone. Moreover, it can be argued that managing the runtime is not a heavy responsibility to be considered a defining characteristic. Instead, it is a feature that is emphasized in FaaS but is not central to established principles of serverless.

Appearance of infinite resources available on demand is listed in Castro, Isahagian, et al. (2022, p. 5) as a characteristic of serverless computing. However, this characteristic is not extensive to established principles of serverless, as it is more closely associated with cloud computing in general. The National Institute of Standards and Technology[22] (NIST) definition incorporates this within *rapid elasticity* (Mell et al. 2011, p. 2). In a broader stretch, it can be argued that the *no responsibility for scaling configuration* principle already implicitly incorporates it. Developers would not need to worry about scaling and therefore not worry if enough resources are available (Eismann et al. 2021a, p. 13, 2021b, p. 34). In that sense, this aspect is already included.

Implicit high availability is another requirement identified by Roberts et al. (2017, pp. 39–41) for a service to be considered serverless. It refers to the ability to remain operational and accessible even in the face of failures or unexpected disruptions. They argue that serverless offerings are expected to provide high availability by expectation, which Castro, Ishakian, et al. (2019, p. 46) support. The FreeWheel Biz-UI Team (2024, p. 190) also supports this by stating: "[s]erverless is naturally highly available [...]". However, this sentiment hints therefore onto *no responsibility* for developers and is arguably incorporated by the *no responsibility for infrastructure management* principle. More specifically, related and partly the same literature hints that availability is a task of operational responsibility in infrastructure management. Therefore, similar to the notion of infinite resources available on demand, the implicit high availability is acknowledged but already included.

[22] https://www.nist.gov/

Capability parameterization beyond hosts finally is the remaining requirement identified by Roberts et al. (2017, pp. 39–41) for a service to be considered serverless. Serverless services must be capable of configuration that is independent or not in the form of correlating to infrastructure means. Developers configure capacity for resources without correlation to the underlying infrastructure. The authors give the example that AWS Lambda can be configured by memory size[23], abstracted from the underlying instances. Again, this is covered by the *no responsibility for infrastructure management* principle. While configuration can be constrained at a high level in this respect, i.e., CPU and RAM, it is in no way related to the underlying infrastructure, nor is it the responsibility of the developer. Thus, the ability to parameterize beyond hosts is less important in terms of the established principles of serverless.

4.1.5 Summarizing the Principles of Serverless

Now, having examined literature and industry perspectives on serverless, principles of serverless have been established and differentiated to form a comprehensive understanding of the term *serverless*. Building on Yussupov, Soldani, et al. (2021), its applicability is not limited to FaaS or BaaS as commonly associated in literature (Shafiei et al. 2021, p. 3; Roberts et al. 2017, p. 6; Jonas et al. 2019, p. 4), but also can include CaaS- or PaaS- based offerings that would adhere to these principles. Furthermore, as the term *serverless* is expressed through cross-cutting principles, it is specifically not confined to Any-as-a-Service (XaaS) but applicable to any deployment target; not only XaaS-based services but application components in general in the way they are deployed can have at least the potential to be considered serverless.

As shown, FaaS offerings are arguably the epitome of serverless, as they fully adhere to all the principles of serverless. However, non-FaaS offerings such as ECS based on Fargate or the App Runner can also be considered serverless, as they adhere to the principles of serverless — they are serverless but to a lesser degree. *Costs nothing when not in use*, thereby acts as a sufficient attribute for a component to be considered serverless; while *no responsibility for infrastructure management* and *no responsibility for scaling configuration* are necessary attributes. As there is no constraint on the degree of adherence to the principles, services can be more or less serverless. With that, serverless is not a binary concept, but rather a *spectrum* — or as Yussupov, Soldani, et al. (2021, p. 38) have put it: a *zone*. Table A.1 presents selected deployment targets to illustrate the spectrum, highlighting their adherence to the serverless principles and the diversity of deployment target options.

To form a concluding sentence about the term *serverless*, the following is proposed:

> In terms of application components or deployment targets, where
> **(i)** the practitioner has no responsibility for managing the infrastructure or scaling configurations, and where
> **(ii)** costs are incurred if and only if the component is in use,
> such a component or deployment target is considered to be serverless;
> and in cases where these criteria are not fully satisfied, it can still be considered as part of the serverless spectrum.

4.1.6 The Serverless Experience

From the discussion of the *no responsibility for infrastructure management* principle, an open question remained whether internal IT teams can provide resources to deliver applications similarly to cloud providers, and how this differs from traditional serverful deployments.

As an example, an internal IT team offering a self-hosted Kubernetes cluster running OpenFaaS can provide FaaS for their developers and their projects. Now, the question is whether

[23]Memory size then correlates with the number of vCPU cores: see here (docs.aws.amazon.com)

this deployment target for developers is serverful or serverless, or where it resides in the spectrum between the two. Although the principles are derived from the context of cloud providers, they can be evaluated for any deployment target, including those outside the XaaS model or cloud environments. Thus, they are also applicable to the most serverful environments, such as bare-metal hardware or on-premise data centers. However, what makes a difference is the perspective from which the principles are evaluated. The principles are evaluated in the context of the internal IT team, but from the perspective of the developers deploying their applications on the FaaS offering.

With that, *no responsibility for infrastructure management* is achieved, as the internal IT team manages the infrastructure and the cluster running OpenFaaS for their developers. *No responsibility for scaling configuration* is also achieved, as the internal IT team manages the scaling configuration of the cluster, and the scaling of running applications is abstracted by OpenFaaS itself. However, *costs nothing when not in use* is a case of perspective and how the actual costs of the internal IT team's FaaS offering are distributed within the organization. From a strictly separate view, costs for the developers are zero when their applications are not in use: the functions are not running and resources can be allocated to other applications. However, the internal IT team has to maintain and operate the infrastructure and resources, incurring costs for the organization. To keep the focus on developers and the application itself, the costs of the internal IT team are not considered part of the context of the application but rather in the context of the organization. With that, the principle *costs nothing when not in use* is fulfilled, though it is ultimately a matter of perspective, opinion, and billing in this case. As such, this scenario can be considered part of the serverless spectrum, albeit rather at the lower end of it. When it comes to rapid elasticity, for instance, cloud providers offer richer possibilities for developers to scale their applications. The internal IT team may not offer the same possibilities and have greater limitations as cloud providers do (FreeWheel Biz-UI Team 2024, p. 181).

Nevertheless, this scenario opens up a new perspective on serverless, where serverless is not a deployment target but rather a term that describes deploying applications under certain characteristics. With the principles of serverless established, the term *serverless* can be abstracted from any deployment target and act as an adjective to describe a conforming deployment target and how it behaves from the outside when users deploy applications on such targets. Be it on AWS Lambda, the internal IT team's OpenFaaS, or some other deployment target, the serverless experience is based on the same three principles for developers. Whereby the underlying infrastructure and how it is managed is not of concern to the developers, but rather the experience they have when deploying their applications and if the principles of serverless are fulfilled. In other words, the term *serverless* describes the *experience* of a deployment target for developers.

4.1.7 An Interface for Deployment Targets

From a technical perspective, a unified interface for serverless deployment targets can be derived from the examples presented in Table A.1. The serverless experience, guided by its principles, characterizes how deployment targets behave when developers deploy applications, while the interface specifies the deployment process and sets expectations for developers.

The serverless deployment targets can be grouped into three categories based on what developers need to do:

- **Container and Execution Configuration**: Developers deploy applications as container images and configure the execution environment, including CPU and memory.
- **Artifact and Trigger**: Developers deploy applications as binaries, *.zip* or *.jar* files or container images and set up triggers to invoke the application.
- **Service Configuration**: Developers create and configure services to match their specific needs, e.g. storage or message queues.

This interface complements the serverless experience by providing a standardized way for developers to deploy applications across various serverless targets. In its simplicity, it accommodates serverless deployment targets while keeping alignment with the serverless principles. A deployment target that requires developers to configure underlying infrastructure or its scaling would not fully meet serverless principles. Optional configurations that influence scaling or infrastructure, however, do not disqualify a target from being serverless.

For example, configuring memory and CPU is common in FaaS and necessary in CaaS, and while these settings affect the infrastructure, they remain the responsibility of the service provider, not the developer. AWS Lambda, for instance, allows memory configuration tied to vCPUs, but AWS manages the underlying infrastructure. Additionally, Lambda offers features like provisioned concurrency[24], which keeps instances pre-warmed and may make a function implicitly serverful. However, these settings are optional and configurable by developers. Deployment targets that would break this interface include those requiring developers to connect via SSH into their servers and manually execute deployment scripts or manage instance scaling, as could be necessary with VMs or self-hosted Kubernetes, for example. These do not align with the principles of serverless deployment.

4.2 Practitioners Experiences Deploying Applications

The following section explores and synthesizes practitioners' experiences in deploying applications across the serverful-serverless spectrum. It maps benefits and drawbacks to the spectrum and examines where and why practitioners start and move within it. Furthermore, this section structures its discussion around the timeline of an application, from planning to its development and operation phase, highlighting key factors that either pose an influence on decisions or constitute rethinking of deployment targets.

Serverless deployment targets are inherently selected for their serverless experience and established principles. SP1 brings cost-savings and cost-efficiency, (Roberts et al. 2017, p. 19; FreeWheel Biz-UI Team 2024, pp. 182, 192; Villamizar et al. 2017, p. 246); and SP2 & SP3 lets developers focus on their application instead of managing underlying infrastructure (Castro, Isahagian, et al. 2022, p. 4; Deng et al. 2023, p. 12; Hassan et al. 2021, p. 2; Roberts et al. 2017, p. 30). FaaS is specially chosen for its purpose of "gluing cloud services together" to boost development (Leitner et al. 2019, p. 10; Roberts et al. 2017, p. 25; FreeWheel Biz-UI Team 2024, pp. 183, 191; Baldini et al. 2017, p. 12; Castro, Isahagian, et al. 2022, p. 11; Eismann et al. 2021b, p. 34).

On the other hand, serverful deployment targets are chosen because of the need for control and ownership over infrastructure and therefore the flexibility they provide (Hassan et al. 2021, p. 8). Developers may need to configure infrastructure or customize applications for their needs, for example, to optimize performance (Castro, Ishakian, et al. 2019, p. 48). Though serverless deployment targets may expose configuration options, serverful deployment targets will have to take over where these are not sufficient (FreeWheel Biz-UI Team 2024, p. 184). Vendor lock-in and independence of cloud providers relate closely to this aspect, serverful deployment targets may be chosen for exactly this reason (Opara-Martins et al. 2016).

However, such reasoning is made during the planning phase to favor later development and operation and is based on information available to developers and susceptible to derived assumptions that may change during the application's lifecycle. Developers cannot avoid making assumptions, based on information or experience, about future aspects such as expected traffic

[24] https://docs.aws.amazon.com/lambda/latest/dg/provisioned-concurrency.html

and therefore expected costs (Eivy et al. 2017, p. 11). As a result, the assumptions made are the reason developers may need to reconsider their choice of deployment target later during development and operations.

The above rationale is not exhaustive and will be expanded on later in this section, but these were the most common reasons for practitioners choosing serverful or serverless deployment targets that emerged from the literature and interviews. In any case, these aspects represent one of the four key factors identified as influencing the first early decision on the deployment target: *non-functional requirements*, alongside *top-down constraints*, *know-how* and *use cases, workloads & architecture*.

4.2.1 Planning

Non-Functional Requirements

As mentioned, non-functional requirements are first and foremost what practitioners will have to face when deploying applications to decide upon which deployment target to choose, as they influence such choice (Baldini et al. 2017, p. 11). Functional requirements, in the form of describing business value features, do not pose an influence on deployment targets: *users do not care whether their login page is hosted on-premises, in the cloud, or on a serverless platform*. However, they do care about *non-functional requirements*: *how fast the login page loads, how reliable and secure it is, or how many users can log in at the same time* (FreeWheel Biz-UI Team 2024, p. 185). They describe the quality attributes of a system and how the system should behave, such as how performant and secure it operates or how fast it scales or is available after failure. The arc42 quality model[25] is a reference to specify such non-functional requirements. The following examines the non-functional requirements most commonly raised in the literature, research, and interviews, and how they influence the choice of deployment targets:

Ownership and control, as mentioned, favors serverful and bears importance in the choice of deployment target (Baldini et al. 2017, pp. 3, 8, 11). Developers must be in the clear about this early on, as it plays an over-arching role. It implies a choice rather between private and public clouds within the serverful-serverless spectrum. As serverless can also be on-premises or in a private cloud, the choice between private and public clouds is orthogonal to the serverful-serverless spectrum and thus independently made. With a private cloud, the infrastructure is either deployed on-premises or procured from a third-party provider exclusively for a single organization (Mell et al. 2011, p. 3). This arrangement grants organizations full control and management over their resources, enabling them to manage decisions, limitations, constraints, and custom features (Mell et al. 2011, p. 3). Such control is particularly important for security and compliance (Andrikopoulos et al. 2014, p. 3) or data protection considerations, one interviewee gave reasoning (B.1). Organizations dealing with sensitive data or in-house IT assets may hesitate to migrate to the cloud due to potential security or privacy issues (Opara-Martins et al. 2016, p. 3). In contrast, with public cloud and its infrastructure available to the public (Mell et al. 2011, p. 3), users are limited to the offerings and restrictions imposed by the cloud provider and have no control beyond capabilities provided by the cloud providers (Roberts et al. 2017, pp. 30, 31; FreeWheel Biz-UI Team 2024, p. 184; Yussupov, Breitenbücher, Leymann, et al. 2019, p. 281). This reliance can lead to vendor lock-in, as organizations become dependent on the cloud provider. In their work, Sfondrini et al. (2018) surveyed organizations and concluded that practitioners make the informed choice to relinquish some degree of ownership to the cloud providers in exchange for a more relaxed operational environment, i.e., the serverless experience. The trade-off results in reduced commitment and responsibility, as the cloud provider assumes responsibility for the infrastructure and services, along with the associated costs of operation

[25] https://quality.arc42.org/

and maintenance. Sfondrini et al. (2018) support that the most detrimental aspect to choosing private over public cloud boils down to the need for *control* and *ownership* over infrastructure and services. This aligns with the findings of Hassan et al. (2021), where the need for control and ownership is a key factor in choosing serverful deployment targets that coincide mostly with the private cloud. However, it is important to note that for example a decision for the private cloud is not a decision against serverless, and vice versa.

Vendor lock-in and independence is a concern for practitioners and organizations to be aware of when choosing a deployment target, especially if they want to remain flexible from cloud providers in the future (Opara-Martins et al. 2016, p. 2; Castro, Isahagian, et al. 2022, p. 12). Opara-Martins et al. (2016, p. 2) analyzed vendor lock-in and described it as: *"where customers are dependent (i.e., locked-in) on a single cloud provider's technology implementation and cannot easily move in the future without substantial costs, legal constraints, or technical incompatibilities"*. The need to migrate from a service or cloud provider can come overnight as the example of VMWare[26] showcases, where since its overtake by Broadcom[27], the company has raised its prices drastically leaving customers with no choice but to accept the new terms or face significant expenses to migrate to another provider [28]. This example illustrates the risk of future entrapment, similarly, cloud providers may raise prices as one interviewee indicated (B.1) or remove offerings as it was the case with Azure Time Series Insights[29] for instance. Similar to *ownership and control*, vendor lock-in and the independence from cloud provider ecosystems push practitioners towards serverful deployment targets (or the private cloud). With serverful deployment targets, organizations can avoid vendor lock-in and foster independence by maintaining more control over their infrastructure and services. For example, organizations may choose a managed Kubernetes service instead of a FaaS offering to be able to migrate to other cloud provider-managed Kubernetes services or host Kubernetes themselves. The underlying interface of developing containerized services remains the same and thus independent of any ecosystem. In contrast, with FaaS and BaaS, practitioners are tied to the cloud provider's ecosystem and services and its proprietary systems and standards (Jonas et al. 2019, p. 21; Yussupov, Breitenbücher, Leymann, et al. 2019, p. 278; Opara-Martins et al. 2016, p. 16; Hassan et al. 2021, p. 16; FreeWheel Biz-UI Team 2024, p. 184). Vendor-specific solutions not available elsewhere or requirements for applications that might not be compatible with other cloud providers, consequent such risk of vendor lock-in (Yussupov, Breitenbücher, Leymann, et al. 2019, p. 279) - especially the case with BaaS offerings (Lannurien et al. 2023, p. 303). As Opara-Martins et al. (2016, p. 5) surveyed, however, organizations remain accepting of and go into such lock-in for reasons of cost savings, IT flexibility, and business agility. Furthermore, rich cloud ecosystems (Baldini et al. 2017, p. 9) and well-integrated component offerings (Roberts et al. 2017, p. 36) of cloud providers, especially present with FaaS, can pose vendor lock-in bearable but are difficult to exit from, as also mentioned in one of the interviews (B.6). Due to the serverless experience and therefore the necessary responsibility that is relinquished, arguably, organizations trade non-management and ease-of-use at the price of vendor lock-in. Thus, based on research and interviews conducted, vendor lock-in is one of the non-functional requirements that is identified as one of the current most influential factors with which practitioners decide against serverless deployment targets.

Cost drives consideration as the most influential non-functional requirement (Eismann et al. 2021a, pp. 6, 21). It does so from the beginning of a project, but also requires assumptions that may change later and cause a shift of deployment target (Eivy et al. 2017, p. 11). For

[26]https://www.vmware.com/
[27]https://www.broadcom.com/
[28]see here (heise.de) or here (itassetmanagement.net)
[29]see here (azure.microsoft.com)

instance, two IoT projects studied, *Project X* and *Project B* had an operational cost per device budget, or their particular customer had a strict sensitivity regarding cost. Cost influences deployment targets, as practitioners look for the more cost-effective choice that will deliver the more appropriate total cost of ownership (Eivy et al. 2017, p. 11; Fan et al. 2020, p. 205) (B.1, B.2, B.3). Total Cost of Ownership (TCO) is the sum of all costs associated with an application over its lifecycle: a composite of *development* and *operational* costs. The former are costs associated with the development of the application, such as the cost of developers, tools, and services used during development, while the latter are costs associated with the operation of an application, such as the cost of infrastructure in addition to maintenance and operating expenses for both (B.1). Development costs tend to be more fixed or predictable because they are generally incurred upfront and are largely independent of the volume of use after completion; they can be extrapolated within reasonable limits based on experience and expectations. In contrast, operating costs tend to be more variable because they scale with usage patterns, maintenance, and performance requirements of the system, which are harder to predict and control over time (Eivy et al. 2017, p. 12; Eismann et al. 2021a, p. 21).

Considering the cost of serverless deployments, they are inherently lower than serverful deployments. BaaS, used in conjunction with FaaS (Jonas et al. 2019, p. 4; Eismann et al. 2021b, p. 37; Lannurien et al. 2023, p. 303), simplifies development and reduces costs because components can be outsourced and developers do not need to develop them themselves (Lannurien et al. 2023, pp. 289, 303). Given the serverless experience and its principles, public cloud consumers benefit from economy of scale (Roberts et al. 2017, p. 3) and practitioners can expect lower operational costs than with serverful deployment targets (Eivy et al. 2017, p. 12; Eismann et al. 2021a, p. 20) [30]. This makes serverless deployment targets a lower commitment cost burden and a more cost-effective choice than serverful deployment targets (Castro, Isahagian, et al. 2022, p. 5). However, costs can explode with serverless deployment targets, as there are stories of practitioners and projects that have run up huge bills overnight[31]. Wrong assumptions about operational costs in terms of unexpected usage patterns, traffic, and load, especially in pay-per-use serverless models, cause costs to skyrocket and make serverless a less cost-effective solution than serverful deployment targets (Eivy et al. 2017, p. 11). Interviews shared the common perspective that serverless loses its applicability under constant or sufficiently high load — if services are running 24/7 anyway (B.1).

With serverful deployment targets, costs are more rigid, and with it practitioners are less susceptible to unexpected costs. Provisioned serverful resources, such as VMs or managed databases, are paid for whether they are used or not. As a result, costs are more predictable and independent of operational factors that can affect costs, as is the case with serverless deployment targets. This might be beneficial in cases of constant or predictable load, where serverless deployment targets would be more expensive, as shown before. However, costs are higher initially and require a greater commitment. Such upfront investment in infrastructure and resources can be a barrier to entry for some organizations (Castro, Isahagian, et al. 2022, p. 5). Similar to serverless, assumptions about future operational costs can be wrong and lead to overprovisioning, where resources are provisioned for higher load and traffic than occurs, resulting in unnecessary costs. In the worst case, practitioners may invest in a serverful deployment target, for instance, big Kubernetes clusters, in anticipation of high user demand, only to find that these investments have been wasted as the expected need does not materialize.

Latency is one of the performance-related non-functional requirements that practitioners consider, particularly for serverless deployment targets (Eismann et al. 2021b, p. 36). It is the total

[30] Economy of scale is heavily influenced by sheer size of infrastructure and therefore harder to achieve for private cloud, thus rather reserved for public cloud providers

[31] See for example reports of practitioners here (news.ycombinator.com) and here (techcrunch.com) (discussion about the same bill), or here (serverlesshorrors.com)

delay between sending a request and receiving a response, and is, therefore, a key factor for user-facing applications (Leitner et al. 2019, p. 2). As serverless deployment targets are increasingly used, for example for web APIs (FreeWheel Biz-UI Team 2024, p. 187), latency is a concern for practitioners. For serverless deployment targets — especially for FaaS-based applications — one needs to be aware of this. Roberts et al. (2017, p. 29) argue: *"latency will always be a concern"* and interviewees shared that they would avoid serverless platforms altogether due to high latency (B.2, B.3, B.6). Leitner et al. (2019, p. 11) support this, arguing that serverless deployment targets are not suitable for applications with high-performance requirements that are affected by higher latency. Relatedly, the same applies to startup times as a non-functional requirement in general: where high startup times are an issue, serverless is the inferior choice.

With serverful deployment targets, latency tends not to be a concern (Fan et al. 2020, pp. 210, 211) because serverful applications — *in the best case* — start once and run continuously, or can be optimized for low latency if required (Roberts et al. 2017, p. 28). This is not the case with serverless deployment targets, which, due to *scale-to-zero* and therefore the potential for frequent scaling up and down (Lannurien et al. 2023, p. 292), face the problem commonly referred to as *cold-starts* (Roberts et al. 2017, p. 32). Cold starts, using FaaS as an example, refer to the time it takes for the cloud provider to: allocate and start the instance, i.e., the function, initialize the execution environment and the application-specific initialization time (Jonas et al. 2019, p. 14). This can result in higher latency on the very first request or after a period of inactivity as the function needs to be started and initialized again. With the frequent scaling up and down typical of serverless, the cold start problem is exacerbated.

Time-To-Market emerged as a non-functional requirement in the interview B.1. The interviewee emphasized the importance of rapid deployment for customer-focused Proof of Concepts (POCs), where time-to-market was crucial. As a result, cloud and serverless deployment targets were almost exclusively chosen as deployment targets for their POCs (B.1). This aligns with the findings of Leitner et al. (2019), who note that "getting it to run in the shortest time possible" (Leitner et al. 2019, p. 11) was the primary concern in their survey for early application prototyping.

Time-to-market is seen by practitioners as a factor that strongly favors a public cloud solution. In their work, Sfondrini et al. (2018, p. 181) had companies selecting this as the main benefit of the public cloud. The interviewee also highlighted the advantages of the public cloud, such as high availability in terms of service level agreements (SLAs) and geo-redundancy (B.1). Moreover, the principles of serverless allow for significantly faster development cycles, enabling practitioners to accelerate their work (FreeWheel Biz-UI Team 2024, p. 183). FaaS relates to this especially, as mentioned before, "gluing together services" boosts development and therefore time-to-market. To quote the same interviewee: "[w]ith FaaS, we can focus on business value for customers" (B.1).

Scalability and elasticity are non-functional requirements that practitioners paraphrase as *handling sudden traffic spikes* (Lloyd et al. 2018, p. 195; FreeWheel Biz-UI Team 2024, p. 190; Fan et al. 2020, p. 214). Scalability refers to a system's ability to adjust resources to meet demand, while elasticity focuses on the dynamic scaling of resources in the face of dynamic demand. In terms of performance, both have a direct impact on throughput, which measures the rate at which a system processes requests, and is a key performance indicator for practitioners (Castro, Isahagian, et al. 2022, p. 12; Eismann et al. 2021a, p. 20). As demand grows, the system can add resources to maintain throughput while allowing for real-time adjustments to handle traffic spikes or changes. The literature agrees that serverless deployment targets are particularly well suited to meeting scalability and elasticity requirements, as they can automatically scale resources up and down based on demand (FreeWheel Biz-UI Team 2024, pp. 187, 190; Lloyd et al. 2018, p. 195; Eismann et al. 2021a, p. 20). The serverless experience and its no responsibility principles mean that practitioners do not have to worry about scalability and elasticity, which is appreciated by practitioners (Sfondrini et al. 2018, p. 181; Leitner et al. 2019, pp. 15, 18) and mentioned in two of the interviews (B.1, B.5). The latter most appreciated the scalability of serverless, especially within the IoT domain. With serverful deployment targets, this is an issue that needs to be addressed by practitioners (Baldini et al. 2017, p. 8) to ensure that the requirements are met, and throughput performance is maintained. Although manageable, manually scaling resources based on demand requires more effort and cost: practitioners must plan, provision, monitor, and adjust resources to meet demand (FreeWheel Biz-UI Team 2024, p. 183; Castro, Isahagian, et al. 2022, p. 5). This aspect makes serverful less desirable than serverless deployment targets in this regard.

Top-down Constraints

In addition to non-functional requirements, top-down constraints such as company policies or regulations influence the choice of deployment target in a more restricting way. These can have the weight to prohibit or dictate deployment targets, can come from either customers or organizations, and can be driven by business cases, strategies, or regulation (Castro, Isahagian, et al. 2022, p. 13) (B.1). Arguably, developers have little to no room to maneuver in this regard.

An example used throughout interviews to illustrate a top-down constraint is a bank as a customer: *they may require that applications run on-premises on their own hardware or data center for any reason*, denying the choice of public cloud providers. One interviewee also gave military contract work as an example of this illustration. Less extreme examples in practice include:

C1 Applications and infrastructure must be deployed using Terraform[32].

C2 Applications must run on Azure infrastructure.

C3 Applications must be hosted within the EU.

All three either prohibit or enforce certain deployment targets: **C1** mandates a deployment method that may not work with on-premise or private cloud environments. **C2** confines deployments to a specific cloud provider, namely Azure, though it doesn't specify serverful or serverless. Lastly, **C3** demands locality, requiring applications to be hosted within the EU, which may limit the choice of cloud providers or regions.

Know-How

Across the literature, projects, and interviews, know-how emerged as a key factor, that influences the choice of deployment target from a general perspective. The interviewees and projects surveyed (B.2, B.3, C) consistently emphasized the importance of having the right expertise from the outset when choosing between serverful and serverless architectures. Know-how refers to the skills, experience, and understanding that developers and teams have of specific practices, technologies, and methods. In the general context of cloud-native deployment, it includes familiarity with development, the ability to troubleshoot, and the knowledge required to effectively maintain and operate both the application and its infrastructure. This expertise is particularly important when dealing with serverless and addressing the issues and challenges that differ from traditional serverful approaches (Baldini et al. 2017; Lannurien et al. 2023; Leitner et al. 2019).

Serverless is relatively new compared to already established serverful solutions (Hassan et al. 2021, p. 1; Baldini et al. 2017, p. 2; Jonas et al. 2019, p. 9; Castro, Isahagian, et al. 2022, p. 11). Missing know-how complicates the evaluation of serverless deployment targets and their suitability for an application's architecture and operational needs, resulting in suboptimal

[32] https://www.terraform.io/

decisions during planning that would affect both later phases. During planning, for example, developers may not fully understand the implications of choosing a serverless deployment target, leading to unforeseen problems later on. This could be trade-offs in tooling, local debugging or testing, vendor-specific limitations that were previously unknown, or a lack of supporting frameworks (Leitner et al. 2019, p. 15; Baldini et al. 2017, pp. 14–15).

For production-ready environments, it is safer to stick with established, well-understood technologies rather than opting for the latest, 'flashy' solutions that the team may not be well-equipped to handle (B.1). One interviewee supported the importance of know-how when choosing a deployment target. They agreed that serverless requires a certain level of expertise to be used effectively (B.1). However, they also noted that serverless can abstract required expertise, making it more accessible to developers. As the serverless experience abstracts infrastructure from developers, there is no need for in-depth knowledge about operating a virtual machine, for example. This abstraction can make serverless more appealing to teams with limited expertise, as it simplifies the development process and reduces the operational burden (B.1) — although it remains a double-edged sword, as it can also lead to unexpected problems if developers lack the necessary know-how, as discussed before.

Use Cases, Workloads, & Architecture

To make an informed decision about the deployment target, practitioners can consider the application's use cases, workloads, and architecture. Literature and interviews suggest that these factors play a significant role in determining the suitability of serverful or serverless deployment targets. Both share opinions on where either a serverful or serverless deployment target approach will be more beneficial. This advantage is measured, for example, by the cost or performance requirements mentioned above, or simply by ease of use during development or operations. It provides a more technical perspective on how practitioners make decisions during planning:

Different uses cases and types of applications are referenced in literature to fit either serverful or serverless deployment targets. As serverful offers the greater flexibility of the two, it is theoretically more applicable to any type of application or use case. Serverless is a more specialized deployment target — arguably a subset of what can be achieved with serverful — and therefore rather suitable for rather specific types of applications or use cases. Web services are mainly where serverless deployment targets are used (Eismann et al. 2021a, p. 6; Jonas et al. 2019, p. 9). Utility functionality, e.g., processing monitoring data, or scientific computing, e.g., simulations, are also mentioned as suitable use cases (Eismann et al. 2021b, p. 36). However, backend or core functionality, i.e., business logic, is the most common use case for serverless deployment targets (Leitner et al. 2019, pp. 2, 10; Eismann et al. 2021b, p. 36). This includes use cases such as integrating a REST API (Lloyd et al. 2018, p. 195; Jonas et al. 2019, p. 9; FreeWheel Biz-UI Team 2024, p. 185), running scheduled jobs (Leitner et al. 2019, p. 10; FreeWheel Biz-UI Team 2024, p. 185; Lloyd et al. 2018, p. 195) or processing data (Lloyd et al. 2018, p. 195; Leitner et al. 2019, p. 10). The latter gives reason as to why the IoT domain is a common use case for serverless deployment targets (Leitner et al. 2019, p. 11; Lloyd et al. 2018, p. 195), which was also supported in interviews (B.1, B.3). Practitioners choose serverless for its inherent scalability, cost-effectiveness, and lack of management, or to benefit from the 'gluing together' of cloud services that benefit the above use cases (Leitner et al. 2019, pp. 10, 11; FreeWheel Biz-UI Team 2024, p. 191; Lloyd et al. 2018, p. 195). As mentioned, serverful deployment targets can achieve the same but require more effort and cost, which is why practitioners choose serverless for these use cases (Eismann et al. 2021b, p. 34).

From an architectural perspective, the decision between serverful and serverless deployment targets is also driven by the nature of application state management and event handling.

Serverless architectures are inherently stateless and ephemeral, making them ideal for applications that follow an event-driven design, where instances are short-lived and respond to specific triggers without maintaining persistent state (Leitner et al. 2019, p. 11; Lannurien et al. 2023, pp. 286, 288). This statelessness allows serverless platforms to scale seamlessly in response to unpredictable events, fitting scenarios where scalability, cost efficiency, and reduced operational management are critical. This aligns well with scenarios that require rapid scaling based on events or data streams, as serverless platforms excel at scaling automatically and efficiently to handle spikes in demand (Baldini et al. 2017, p. 11; Castro, Ishakian, et al. 2019, p. 50). Which makes it particularly suitable for processing events or telemetry data within IoT use cases (Leitner et al. 2019, p. 11; Baldini et al. 2017, p. 11; Castro, Ishakian, et al. 2019, p. 50). However, serverful architectures are more appropriate when handling stateful applications that require persistent state management and heavy reliance on consistent, long-running processes (Leitner et al. 2019, p. 11; Eismann et al. 2021a, p. 6; Lannurien et al. 2023, p. 288). While microservices are associated with serverful deployments due to their need for fine-grained control over state and resources, they can also be implemented using serverless, especially when individual microservices are designed to respond to specific events and remain stateless according to FreeWheel Biz-UI Team (2024).

Workloads have extensively been explored by literature and research and how they influence the choice of serverful or serverless deployment targets. In terms of traffic frequency and intensity, serverless, given the pay-per-use and scalability and elasticity, was more likely to be suitable for bursty and infrequent, unpredictable or irregular workloads (Lannurien et al. 2023, p. 288; FreeWheel Biz-UI Team 2024, p. 185; Leitner et al. 2019, p. 11; Eismann et al. 2021b, p. 34; Hassan et al. 2021, p. 8; Eismann et al. 2021a, p. 6; Castro, Ishakian, et al. 2019, p. 50). On the other hand, serverful deployment targets are therefore more suitable for regular and steady traffic, mainly for cost reasons (Fan et al. 2020, p. 214; Baldini et al. 2017, p. 14; Hassan et al. 2021, p. 8). However, this is one point where practitioners cannot avoid making assumptions about future traffic and load, which are subject to change and may lead to a shift in the deployment target later during operations (Eivy et al. 2017, p. 11). In terms of compute, Baldini et al. (2017, p. 11) argue that I/O-intensive workloads, for example, exchange with third-party APIs, are better suited to serverful deployment targets. Meanwhile, compute-intensive workloads such as image processing are better suited to serverless deployment targets (Castro, Ishakian, et al. 2019, p. 50). This is because I/O-related tasks can be handled more efficiently by serverful deployment targets, as they can be optimized for I/O operations (Baldini et al. 2017, p. 11). In contrast, serverless deployment targets would incur costs for compute resources while being idle for I/O operations, resulting in much worse cost efficiency for such workload.

4.2.2 Development

Given the practitioner's decision on the serverful-serverless spectrum, this section focuses on the development phase and what leads therein to the decision to move within the spectrum. The development phase is where a subset of assumptions made during planning are put to the test and where the reality of the chosen deployment target becomes apparent.

The various reasons that practitioners have for wanting to move within the serverful-serverless spectrum during development can be summed up in a single observation: *when developers spend more time fighting the stack than developing features.*

As serverless bears lesser development flexibility than serverful, developers may find themselves fighting the serverless ecosystem or technologies therein. Cloud provider specifications, runtimes, requirements, or limitations are some of these battlegrounds that would fall under the assumption of encountered degrees of vendor lock-in (Opara-Martins et al. 2016, p. 4; Baldini et al. 2017, p. 6; Yussupov, Breitenbücher, Leymann, et al. 2019, pp. 274, 279, 280, 283; Leitner et al. 2019, p. 20). Furthermore, developers may find themselves fighting the deployment target due to missing know-how or experience. A lack of tooling or local debugging or testing capability leads to developers spending more time fighting the stack than developing features (Lannurien et al. 2023, p. 303). Coincidentally, the reasons listed are the same reasons that practitioners choose serverful over serverless deployment targets. The difference is that these reasons are only discovered during development, and not during planning, or if predictable, overseen due to missing know-how.

Assumptions about the level of vendor lock-in and its impact, the suitability of the application for the paradigms or development aspects encountered, and the tools available would lead developers to want to move away from the previously chosen deployment target. In combination, missing know-how about a particular deployment target or the consequences of decisions would exacerbate the situation.

In the case of *Project X*, developers interviewed in B.2, struggled with the Azure Functions[33] ecosystem. Developers had to implement functionality not provided by the ecosystem or resort to workarounds to overcome (hard) runtime or resource constraints. The project involved running machine learning (ML) jobs that were not suitable for Azure Functions as is. However, the developers had to adapt the ML jobs to fit into short-running FaaS-ready workloads at the cost of introducing complexity and overhead. Also, the development experience within Azure Functions can be described as raw development in the sense that there was little to no development abstraction. Furthermore, developers fully embraced the Azure Function development model, which meant a higher degree of vendor lock-in too. Developers were implementing common functionality for their common use case that would have been pre-packaged in common but for them not applicable frameworks. Interviewees shared that they had to implement custom request validation and that for instance, API documentation generation was not given due to the raw development experience with Azure Functions. For example, as the project uses an API with Azure Functions written in Java, Spring Boot[34] would be one such framework, which with its ecosystem would provide the required functionality through extensions, but is used more for serverful deployment targets and therefore not suitable within Azure Functions. To sum up *Project X*'s perspective, missing know-how led to suboptimal development choices, resulting in an unguided therefore bad experience and mental overhead in development with Azure Functions, which in turn led them to avoid serverless platforms with REST APIs altogether. The obstacles were overcome, but the developers were unable to shift the deployment target, and to this day *Project X* still uses Azure Functions as a deployment target — despite the bad experience. Changing the deployment target is not possible, because the application is working and running in production and no time or resources would be allocated to change deployment targets. Even if so, it would involve re-engineering the implemented Azure Functions and confining them within a more comfortable framework for APIs to be run inside a containerized environment, which would require time and resources. In hindsight, the developers interviewed would have decided against FaaS for their REST API and would have chosen a containerized and therefore more serverful deployment target for *Project X*, given the experience they had.

Project B* and *Project D both have similar project architectures to *Project X*, and interviews with their developers shared similar, if not the same, struggles with serverless deployment targets during development (B.3, C). Both struggled with implementing REST APIs with Azure Functions in a typical backend scenario, with similar problems of implementing custom functionality to better the development experience that would otherwise be pre-packaged in common frameworks for serverful deployment targets.

[33] https://learn.microsoft.com/en-us/azure/azure-functions/functions-overview
[34] https://spring.io/projects/spring-boot

4.2.3 Operation

When operating applications, practitioners may find themselves moving across the serverful-serverless spectrum, even after making an initial choice. This shift is typically driven by assumptions they initially made regarding the *non-functional requirements*, which later turn out to be inaccurate or false (Castro, Isahagian, et al. 2022, p. 12). These assumptions are crucial to the decision-making process, as they influence how well the deployment target can support the application's intended operations.

Specifically, such assumptions relate to two key areas: **costs** and **performance**. For example, practitioners might assume that a serverless deployment target would offer significant cost savings due to its pay-per-use model, only to later discover that, at scale, the costs are higher than anticipated (Eivy et al. 2017, p. 11). Alternatively, they might expect a certain level of performance, such as low latency or high availability, but find that the chosen architecture cannot meet these demands under specific workloads (Villamizar et al. 2017, p. 234; Lannurien et al. 2023, p. 308). In either case — whether due to costs, performance, or both — the initially chosen deployment target may fail to meet the actual needs of the application.

Missing know-how, again, relates to both aspects of **cost** and **performance**. In the operational phase, missing know-how can lead to unexpected costs or insufficient performance, as developers may not be able to optimize the application or infrastructure for the chosen deployment target, troubleshoot issues effectively, or make informed configuration or maintenance decisions about the application and infrastructure (B.2). In some instances, the deployment target may never have been suitable for the application's requirements to begin with. This becomes apparent only after practical use but was overlooked or underestimated during planning due to missing know-how.

However, taking it out of the picture, the assumptions made during planning are still susceptible to change regardless. It may just be that with the right know-how, potential fallout could be anticipated and minimized or mitigated.

Cost is the key driver behind the decision to move within the serverful-serverless spectrum during operations, as practitioners typically choose their deployment target based on economic considerations (Eismann et al. 2021a, pp. 6, 21). This makes cost-efficiency a big factor as practitioners aim to minimize operational expenses while maximizing the value of their applications. However, the assumptions made during planning about the cost-effectiveness of a deployment target may not always align with the reality of the application's operational costs (Eivy et al. 2017, p. 11). Different usage patterns exist that all would affect the cost of serverless or serverful deployment targets. They influence when and how the application encounters load or traffic, and therefore how resources are provisioned and billed. Some examples may be: *constant load 24/7, the latter but additional high spikes, office hour bound load (e.g., nine-to-five), peak load during certain times of the day, week or month, bursty and unpredictable traffic, or mostly none with some occasional spikes.* Each of these usage patterns can lead to different cost outcomes for serverless or serverful deployment targets, and assumptions made during planning may not always align with the actual usage patterns of the application.

The less consistent and the more random the usage pattern, the more cost-effective serverless deployment targets become, due to pay-per-use (Fan et al. 2020, p. 214). However, steady or predictable usage patterns may still be viable for serverless deployment targets in the face of detrimental idle periods, as this could offset the cost savings of serverful deployment targets during non-idle periods. Practitioners would make assumptions about what their applications would encounter and make decisions based on that. However, if the actual usage patterns of an application do not match these initial cost-benefit assumptions - perhaps due to higher or lower-than-expected usage - practitioners may be forced to reconsider their decision.

Performance is the other driver behind the decision to move within the serverful-serverless spectrum during operations. Practitioners select deployment targets based on performance requirements such as latency, throughput, and scalability to ensure their application meets operational demands and deliver a high-quality user experience (FreeWheel Biz-UI Team 2024, p. 185).

Serverless deployments offer advantages in handling bursty or unpredictable traffic through automatic scaling and managed infrastructure. Again, assumptions about usage patterns and therefore necessary scaling ability may consequent in a shift of deployment target to meet throughput needs (Andrikopoulos et al. 2014, p. 2). While serverful deployment targets allow practitioners to optimize performance for specific workloads, especially those with more predictable or high-performance requirements, the need to manually provision and scale resources can introduce inefficiencies, especially under varying load conditions.

As already mentioned, one significant challenge with serverless is cold-start induced latency (Roberts et al. 2017, p. 32; Fan et al. 2020, p. 210), which can be detrimental to applications requiring low-latency responses. For instance, developers of *Project X* faced latency issues related to cold starts (B.2). Although their main pain point was development-wise, the cold-start problem would have led to a switch in deployment targets. Instead, they implemented the antipattern of using keep-alive mechanisms to keep functions warm, thereby mitigating latency issues (Lloyd et al. 2018, p. 2; Taibi, Ioini, et al. 2020, p. 187). While this approach reduced cold-start latency, it also resulted in higher costs, as idle functions were continuously charged. Alternatively, they could have switched to a more serverful deployment target to counteract the cold starts, developers mentioned Azure Container Instances[35] as an alternative. However, this would have required the application to be re-engineered, which would have taken more time and resources (B.2).

As real-world performance needs evolve, practitioners reassess their deployment choices, transitioning between serverless and serverful deployment targets to better align with actual performance demands — demands that frequently differ from their initial assumptions (Yussupov, Breitenbücher, Leymann, et al. 2019, p. 274; Castro, Isahagian, et al. 2022, p. 12).

4.3 Towards Hybrid Deployment Models

In section 4.2, a comprehensive view of where and why practitioners start and move within the serverful-serverless spectrum is presented. It showcases that practitioners face trade-offs when first deciding about deployment targets and that yet such decisions may be subject to change. Hybrid deployment models would counteract the need for a strict decision about deployment targets. With such, practitioners are not tied to a specific deployment target and can shift between serverful and serverless deployment targets as the need arises. In this section, approaches towards hybrid deployment models are explored, i.e., how applications can be deployed across the spectrum to enable practitioners to *decide about deployment later*.

4.3.1 Component Specifications

Before diving into hybrid deployment models, it is important to understand what type of application components exist that can come into question to be suitable for hybrid deployment. In section 1.1, the focus is set on compute resources, so storage or network components such as databases or load balancers are not considered. With that, the focus is on business logic applications providing business value, i.e., applications that process data and serve requests. And yet, not all applications fitting this description may be suitable for hybrid deployment.

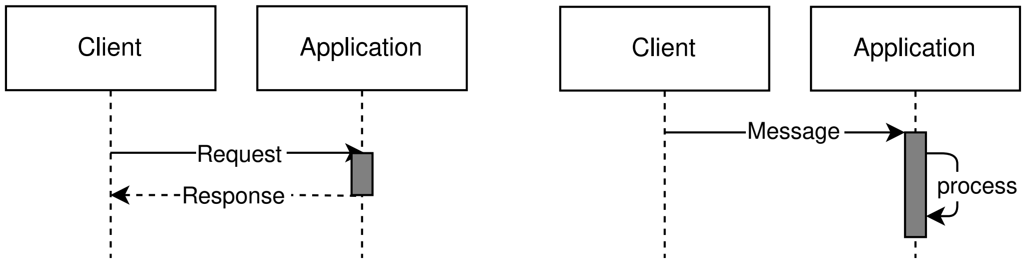
[35] `https://azure.microsoft.com/en-us/products/container-instances/`

Serving and Processing Applications

Applications for this work can be broadly categorized into two distinct application interface types: *serving* and *processing* applications (FreeWheel Biz-UI Team 2024, p. 37). Figure 4.1 illustrates the difference between the two application interface types. The distinction between these two types is crucial when considering hybrid deployment models, as each has different requirements and constraints. Understanding their differences allows for more informed decisions about deployment targets and the trade-offs involved in moving between serverful and serverless environments.

Serving applications follow a synchronous request-response pattern, where clients send requests (e.g., HTTP requests) and expect immediate responses. These are typical web services or APIs that provide direct user interaction.

Processing applications are designed for asynchronous data processing. They typically consume data from messaging systems (e.g., message queues), process it, and produce some output. These types of applications are prevalent in use cases like telemetry processing in IoT systems, where data is processed in the background without direct user involvement.



(a) *Serving Applications* ***(b)*** *Processing Applications*

Figure 4.1: *Illustration of the two application interface types: serving applications (a) and processing applications (b). In the former, clients send requests and expect immediate responses, while in the latter, the request is processed asynchronously with no immediate response.*

Stateless Applications

For applications that need to be both serverful *and* serverless deployed, the latter is the more restrictive (Lannurien et al. 2023, p. 308) and most limitations presented stem from it. With that, applications, to be deployed serverful *and* serverless, must be *stateless* (Jonas et al. 2019, p. 12; Lannurien et al. 2023, p. 288; Roberts et al. 2017, p. 28). It is a core characteristic required for serverless applications, as it allows applications to scale seamlessly by treating each request as independent of the others (Lannurien et al. 2023, pp. 282, 283). The twelve-factor app methodology[36], which is a set of best practices for building modern, cloud-native applications, also emphasizes the importance of statelessness in the design of applications.

Therefore, in a stateless design:

- The application does not retain any internal state between requests. Any necessary data or context must be passed along with each request or stored somewhere externally.
- Each instance of the application can handle any incoming request, enabling better load distribution and scalability.

[36] `https://12factor.net/`

This statelessness aspect reaches into a different aspect for processing applications to be suitable for hybrid deployment. It is crucial to manage connections to message queues and other resources externally, rather than allowing the business logic application to handle them. By restricting the application's responsibility to only processing business-specific inputs and outputs as of Figure 4.1, the application can remain stateless and therefore scalable down to zero. Therefore, processing-related event handling or transport, such as connection management, polling, or parsing, should be managed externally by the platform (e.g., as with FaaS), middleware, or some other form of integration, thereby separating it from the core business logic. If the application were to manage its connections or polling, it would inevitably maintain state and resource usage, rendering it incompatible with serverless deployment targets and therefore viable for a hybrid deployment.

4.3.2 Sample Architecture

Now that the types of applications that are suitable for hybrid deployment have been defined, a sample architecture can be built as a basis for further exploration. To cover both serving and processing applications, a very general sample system architecture depicting a common use case in IoT was elaborated and is presented in Figure 4.2. In this sample architecture, the focus is set on the component *service s*, which is a serving application that provides a service for users to fetch data from a database. The component *service p*, which is a processing application that consumes data from a device, processes them, and stores the results in the same database. Both services present in Figure 4.2 thereby are generally suitable for hybrid deployment. The sample architecture is designed to be cloud-agnostic, meaning that it can be deployed on any cloud provider that supports serverless and containerized deployment targets. This allows for a more general exploration of hybrid deployment models that are not tied to a specific cloud provider.

The sample architecture is inspired by real use cases from *Project X*, *Project D* and *Project B*. Notably, as all three of these projects are inherently built on Azure Functions, both *service s* and *service p* are represented within these three projects: Azure Functions, which provide a REST API, and Azure Functions, which handle events and messages. At the other end of the serverful-serverless spectrum, both *service s* and *service p* could be implemented as self-hosted web services running on dedicated hardware. Technically possible, both extremes can also be mixed, e.g., *service s* as a self-hosted web server and *service p* as a serverless function.

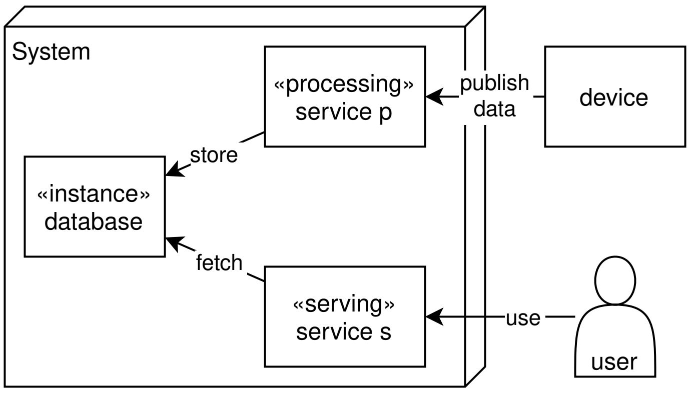


Figure 4.2: *Sample system architecture diagram of an IoT use case that incorporates both a serving application and a processing application: some device publishes data that is processed by some service and stored in a database; a user-facing service with which a user can fetch the same data stored in the database.*

4.3.3 Requirements

With the sample architecture set in Figure 4.2, consideration about requirements and challenges of deploying such sample architecture in a hybrid deployment manner can be made. Challenges identified have to be kept in mind when exploring hybrid deployment models. The requirements are set to ensure that applications can be deployed in a hybrid manner without compromising the quality of the applications before and after a switch between serverful and serverless deployment targets. As the sample architecture is designed to be cloud-agnostic, the requirements are to be general enough to apply to any cloud provider. With the set requirements for hybrid deployment of such two applications *service s* and *service p*, further exploration and evaluation can be supported.

Non-Functional Requirements

The sample architecture originates from real-world use cases in *Project X*, *Project D*, and *Project B*, so requirements from such IoT use cases can be considered. Also supported by synthesis of section 4.2, the following non-functional requirements are considered to be the most important for the sample architecture on any deployment target:

NFR1 Applications need to be **highly available and reliable** to ensure consistent access.

NFR2 Applications must prioritize **low latency** to provide a smooth experience, especially for user-facing applications.

NFR3 Applications need to prioritize **scalability and elasticity** to handle high volumes of data and adapt to changing requirements, ensuring availability and performance.

NFR4 **Cost-efficiency** is crucial: in general, running applications should be as cost-effective as possible. Ideally, costs should match usage, so that low usage results in low costs.

Applications leveraging hybrid deployment models should be able to meet these requirements, regardless of the deployment target.

The Most Optimal Experience

Based on section 4.2, and drawing on the interviews with *Project X*, *Project B*, and *Project D*, the development experience was a big part of the negative perception of serverless development. With this in mind, considerations can be made as to what the most optimal development experience would look like. Furthermore, these can be enriched by considering the optimal deployment and operational experience of serverful and serverless deployment targets, as also discussed in section 4.2. The following requirements are considered to ensure that an overall optimal experience is not compromised by hybrid deployment models:

R1 **Standards:** Use common and open standards and conventions as a basis, avoiding vendor lock-in as far as possible.

R2 **Abstraction:** Prioritize high-level abstraction over raw development experience to promote lower mental overhead in development.

R3 **Development:** Ensure that applications can be developed, tested, and debugged locally to promote a flexible development experience.

R4 **Maturity:** Prioritize mature and well-documented tools and frameworks over custom-built solutions to promote lower complexity in development and address missing know-how.

R5 **Separation:** Keep infrastructure and integration separate and abstract from business logic to allow for flexible and adaptable development and deployment.

R6 Modularity: Prioritize modularity and separation of concerns to facilitate the transition between serverful and serverless deployment targets.

R7 Portability: Prioritize deployment artifacts that are portable between deployment targets to enable easy transition between serverful and serverless environments.

With such requirements, approaches toward hybrid deployment models can be explored and evaluated, or new approaches can be derived. Ultimately, the goal is to enable hybrid deployment models to be leveraged with a seamless experience when switching between serverful and serverless deployment targets.

4.3.4 Existing Solutions and Approaches

Given the sample architecture and the requirements set, approaches towards hybrid deployment models can be explored. Technical research was conducted to identify existing solutions, technologies, or current practices that would enable or could be used for hybrid deployment models. These were evaluated against the requirements set to further inform the exploration of hybrid deployment models and are presented below.

Infrastructure From Code

An upcoming paradigm in the cloud computing space is Infrastructure from Code[37] (IfC) (Aviv et al. 2023). IfC is an approach that abstracts and automates the creation and management of infrastructure based on and directly from within application code and function signatures and annotations. Unlike Infrastructure as Code (IaC), which requires manually defining infrastructure in configuration files (e.g., Terraform or CloudFormation[38]), IfC embeds infrastructure directly into the development process.

Following that, listing 1 shows a simplified example given on the IfC website, where the application code directly defines the infrastructure requirements alongside the application code.

```typescript
// File: src/index.ts
import { api, db } from 'sdk';

api.get('/users', async (req, res) => {
  const users = await db.find();
  res.json(users);
});

api.post('/users', async (req, res) => {
  const { email, name } = req.body;
  const user = await db.insert({ email, name });
  res.json(user);
});
```

Listing 1: *Example of Infrastructure from Code (IfC): where the application code directly defines the infrastructure requirements alongside business logic. In this example, the application code defines two API endpoints that fetch from and insert user data into a database. Whereby API and the database are abstracted within the* `sdk` *module. When deployed, such an SDK module would then provision and configure the necessary infrastructure resources, i.e., the API and the database.*

For hybrid deployment models, IfC seems to be a promising approach, as it abstracts infrastructure requirements directly into the application code. This would allow for a transition between serverful and serverless deployment targets, as developers may influence under-

[37] https://infrastructurefromcode.com/; *note: this website is hosted by Ampt, a paid platform for IfC*
[38] https://aws.amazon.com/cloudformation/

lying infrastructure created at deployment time. With this developers could define infrastructure requirements in the application code, and the deployment platform would then provision and configure the necessary infrastructure resources. For instance, given the API endpoints in listing 1, such IfC framework could either provision a serverless function or embed the API into a self-hosted web server, based on the deployment target chosen during deployment: `ifc deploy --target [az-faas|aws-faas|az-vm|aws-fargate]` or the like.

The IfC paradigm exists and, with its additional layer of automation, brings benefits to development productivity and deployment speed (Aviv et al. 2023, p. 45). They argue that with IfC, automation takes over once developers have implemented their business logic, reducing the overhead of managing separate infrastructure configuration files. It has the potential to simplify deployment to just automating infrastructure requirements, as developers can focus on writing business logic and let the IfC take care of infrastructure requirements and deployment specifics.

However, while the IfC paradigm is promising, its ecosystem is still in its infancy at the time of writing (Aviv et al. 2023, p. 43). Tools and frameworks would need time to mature and become more versatile and flexible for different needs. There exist cloud deployment platform services for IfC, such as Shuttle[39], Ampt, or Encore. Although such platforms are paid subscription-based services, the main concern is vendor lock-in and independence for the same reasons as discussed in section 4.2. Developers would shift vendor lock-in from cloud providers to such services, as they would be tied to the specific IfC platform and its ecosystem. For enterprise or production use it is not feasible to rely on such a new and yet immature ecosystem, but rather on established and proven cloud hyperscalers or on-premise infrastructure. Furthermore, it would be difficult for such a third-party IfC provider to deploy on-premises or on self-hosted infrastructure, if required, as such presented IfC platforms are cloud-based. IfC as a paradigm is promising but not yet mature enough to be used for this work and existing frameworks are not suitable for further use in this work. Thus, **R1** and **R4** would not be met by the IfC paradigm and its ecosystem *yet*.

Serverless Function Frameworks

Serverless function frameworks such as Spring Cloud Functions and Quarkus Funqy build on top of established frameworks for function-based programming. Both are serverless function frameworks for Java that abstract deployment specifics to be handled by the frameworks' adapters and extensions. They provide a common interface, allowing developers to write business logic as functions decoupled from any specific deployment target. It allows developers to focus on the business logic instead of how to integrate and deploy the business logic to a specific deployment target. The frameworks then can integrate such business-level functions to be deployed onto different FaaS offerings of cloud providers, such as AWS Lambda or Azure Functions, or be deployed as standalone applications. Quarkus Funqy to be deployed on AWS Lambda for instance, is based on Quarkus's respective REST and HTTP extensions[40].

See listing 2 for an example of the Quarkus Funqy framework, where the business level application code — *greeting a person* — is implemented on a functional level independent of any deployment target.

Both frameworks, with what they are capable of, including their ecosystems and features, are promising for hybrid deployment models. They abstract deployment specifics and allow for a transition between serverful and serverless deployment targets.

To further explore the frameworks, firstly, a simple serving application is implemented with Quarkus Funqy: a REST API that greets a person. The example in listing 2 is used to be deployed as either a serverless function on AWS Lambda or as a standalone application on the AWS App Runner. The App Runner instead of a self-hosted web server for example is chosen for the sake of simplicity, as the resulting artifact of Quarkus Funqy for serverful deployment

[39] https://shuttle.dev/
[40] https://quarkus.io/guides/aws-lambda-http

```java
// File: src/main/java/com/example/Function.java
// ...
import io.quarkus.funqy.Funq;

public class Function {

  @Funq
  public String function(Person person) {
    return "Hello " + person.name + "!";
  }

  public static class Person{
    public String name;
  }
}
```

Listing 2: *Example of Quarkus Funqy: where the business level application code is implemented as a function independent of any deployment target. The function greets a person by name.*

can be a container image, and no difference for this argumentation would be made for either of the two deployment targets.

Hybrid deployment for serving applications with Quarkus Funqy was straightforward via its extensions. The deployment target is chosen at the build time of the deployment artifact, but has to be defined in the application's build dependencies. Based on the extensions, dependencies, and configurations, the Quarkus Funqy framework generates the necessary deployment artifacts for a given deployment target — Spring Cloud Functions works similarly. With the example in listing 2, the Quarkus Funqy framework can generate for instance an HTTP web server container image deployable on the App Runner or an application code package (*.zip*) for Lambda that could be used with the AWS API Gateway. The API Gateway is a prerequisite stemming from how Quarkus Funqy packages the application to be deployed on Lambda. However, the framework itself explains[41] that it is rather limited and trades flexibility for broad applicability across multiple integrations, i.e., different cloud providers and their protocols. This is a trade-off that has to be made when using such frameworks, as they have to keep a simple API and sacrifice flexibility and features. For simple HTTP serving applications — somewhat resembling RPC — Quarkus Funqy is a valid choice, enabling hybrid deployment. For more complex serving applications — REST APIs for instance — Quarkus Funqy is not suitable, as it lacks the depth of necessary features and integrations for applications such as complex path routing or middleware. Serving applications in *Project X*, *Project B*, and *Project D* for example all are REST APIs, and Quarkus Funqy would not have been suitable for their applications — neither would have Spring Cloud Functions. Points of discussion above also apply to Spring Cloud Functions, although Spring Cloud Functions has a larger ecosystem as it is built on top of the Spring Framework[42] and is therefore more feature rich.

In conclusion, both Quarkus Funqy and Spring Cloud Functions are suitable for hybrid deployment of serving applications, but only for rather simple applications that do not require complex features or integrations.

Hybrid deployment for processing applications with both frameworks is not inherently possible in the same way as shown before with serving applications. With serving applications, the same business level function is packaged differently depending on extensions for the respective deployment targets — *its integration is abstracted by the framework*. For processing applications, given the same example in listing 2, the function itself can be reused, but the integration

[41]https://quarkus.io/guides/funqy
[42]https://spring.io/projects/spring-framework

for other than FaaS deployment targets, is not handled by the frameworks as is. Instead, additional implementation is necessary to integrate the business-level function with the necessary infrastructure, such as for instance message queues.

For example, a function that consumes messages from an AWS SQS queue and processes them can be deployed with Spring Cloud Function onto Lambda with similar code as in listing 2 and respective extensions. Now, given the same function, to deploy it on a serverful deployment target, as a standalone containerized application for example, additional implementation is necessary to integrate the function with the SQS queue. Due to the Spring's mature ecosystem, Spring Cloud[43] provides an abstraction for SQS polling, which can be used to connect the prior serverless deployed business level function to the SQS queue and deploy it as a standalone application. This scenario was implemented and can be found on the GitHub repository linked to this work[44]. Quarkus Funqy lacks such an abstraction and would require the developer to implement the SQS polling and connection management in a more manual way.

In conclusion, both Quarkus Funqy and Spring Cloud Functions would be suitable for the hybrid deployment of processing applications. However, shifting deployment targets for processing applications with such frameworks is more complex than for serving applications and requires additional implementation effort.

In terms of set requirements, both frameworks align with **R1**, **R2**, **R3**, **R5** and **R6**. For serving applications, they rather enhance the development experience and deployment process with further abstraction according to requirements **R1** and **R2**. This is a step in the right direction, in contrast to the development experience described in the interviews. However, limited flexibility and features are traded for broad applicability, which may not be suitable for a wide range of use cases. In particular, for processing applications, the frameworks would break with requirement **R4**, as additional implementation can be necessary. However, such frameworks explicitly reinforce requirements **R5** and **R6** which are crucial for enabling hybrid deployment models. Depending on the complexity necessary for serverful integration, the frameworks at least foster the transition between serverful and serverless deployment targets.

Requirement **R7** is not met, as the resulting deployment artifacts are highly specific for either serverful or serverless deployment targets and cannot be reused. Related to this is the fact that with such frameworks, developers are faced with the decision about deployment rather early on, as the deployment target is chosen at the build time of the deployment artifact. Development thereby is influenced by the deployment target, which is contrary to the goal of enabling hybrid deployment models. As shown, more strongly so within processing applications than with serving applications.

Service Description and Code Generation

Different from the IfC paradigm and serverless function frameworks, service description and code generation were explored as an alternative approach. This idea originated from the concept of Service Oriented Architecture (SOA) and Interface Driven Development (IDD) (Papazoglou 2003).

It could consist of a two-step generation process, where the first step would be to define a service contract description that describes the interface and behavior of the service. The service contract description could be used to generate a service scaffold that complies with the defined contract, and the business logic would be implemented within the generated code. Finally, integration logic would be generated to integrate the service with a selected deployment target.

This approach is based on generalized integration logic for deployment targets and the separation of business and integration logic with interfaces. Listing 3 illustrates the difference

[43]https://spring.io/projects/spring-cloud
[44]https://github.com/a-grasso/master-thesis-public/tree/main/spring-cloud-functions-sqs

between business and integration logic. Regarding the two-step generation process: the service contract description would be used to generate the scaffolding for a service that implements the processing service interface, the business logic of which is then up to the developer to implement. The deployment target would then be selected, in this case, AWS Lambda and its specific integration logic would then be generated to integrate the service with the deployment target, as seen in listing 3.

```go
// File: main.go
// ...

type ProcessingService interface{ // - *generated*
  ProcessMessage(Message)
}

type Message struct{ // - *generated*
  // ...
}

// deployment specific integration logic - *generated*
func handleRequest(ctx context.Context, sqsEvent events.SQSEvent) error {
  // parse input event
  message := ...

  // business logic
  service := ...
  service.ProcessMessage(message)

  fmt.Println("Message processed")
}

func main() { // - *generated*
  lambda.Start(handleRequest)
}
```

Listing 3: *Example to illustrate the difference between business logic and integration logic. The business logic is implemented behind the `ProcessingService` interface, and the integration logic is thereby deployment target specific before and after the business logic is called. Here: the example showcases the integration logic for AWS Lambda to process messages from an AWS SQS queue.*

The approach abstracts deployment target specifics, i.e., integration logic, behind configuration and code generation. By separating business and integration logic, the former could be reused across different deployment targets, enabling hybrid deployment models. The separation aspect is similar to how serverless function frameworks abstract deployment specifics. However, the deployment target would be chosen at generation time, before the deployment artifact is built. With the flexibility of code generation, almost endless possibilities and customizations could be achieved, as the code generation step for integration would be modified and extended as needed. Nevertheless, such extension and customization would require a great deal of engineering effort to enable different applications, whether they are serving or processing applications and their potentially highly specific requirements, to be compatible with different deployment targets and their specificities across the serverful-serverless spectrum. It is not feasible for developers to implement such code generation and integration logic for each deployment target themselves, and therefore a tool or framework that provides capabilities for such service description and code generation is required.

At the time of writing, no such tool or framework has been found that would provide such comprehensive capabilities. Therefore, parts of this approach have been explored and spiked to generate insights.

Service description languages exist and some have been considered, such as Smithy2.0, AsyncAPI, GOA and OpenAPI. Smithy2.0 is a service description language for defining services and SDKs in a language-independent way. The specification itself is comprehensive and powerful, but its relative newness and lack of stable code generation tools made it difficult to base further work on its ecosystem and made it unsuitable for this work. AsyncAPI is a specification for defining event-driven APIs, similar to Smithy2.0, its specification may be usable, but the surrounding tooling and ecosystem were not mature enough to be used for this work. Both would break requirements **R1** and **R4**. GOA is a design-first approach to building APIs and services in Go. In favor of OpenAPI and its wider applicability, GOA was not considered further.

OpenAPI was chosen from the four options because of its extensive ecosystem and support for a wide range of tools and frameworks[45]. OpenAPI is a widely adopted and mature specification for defining RESTful APIs (Di Lauro 2024). The OpenAPI specification is used to define the service contract, and code generation tools such as Swagger Codegen[46] can be used to generate the service scaffolding. Its service description and code generation are comprehensive enough, and the latter is extensible via templates, albeit the former is limited to serving applications. The OpenAPI Generator project[47] is a collection of code generators that can generate code for various programming languages and frameworks based on an OpenAPI specification.

Generated service scaffolds need to be adaptable to different deployment targets. Depending on the generator used, this was possible, but not all were suitable for deployment across different targets, depending on what the scaffolding included. For example, if the service scaffolding propagates the HTTP request into the business logic, it is not suitable for serverless deployment targets, as the business logic should not need to handle integration specifics. Thus, the interfaces and business logic need to be kept separate from the integration logic, which can be ensured depending on how the code is generated. However, no tool or framework was found that would generate the integration logic for different deployment targets based on the OpenAPI specification. Therefore, the approach and service description capabilities (for RESTful serving applications) of OpenAPI *could* be suitable for hybrid deployment models, but to date are not fully capable of hybrid deployment. In terms of requirements, OpenAPI and its ecosystem conforms to **R1**, **R2**, **R3**, **R5** and **R6**. Requirements **R5** and **R6** are particularly highlighted as OpenAPI promotes such separation of concerns and modularity, and thus the potential to enable hybrid deployment models. However, it does not fully satisfy **R4** because the OpenAPI ecosystem does not currently provide code generation in terms of integration logic, and such would need to be implemented manually or the OpenAPI ecosystem extended by appropriate generators. It does not satisfy **R7** because the resulting deployment artifacts are specific to a deployment target and cannot be reused.

Lambda Runtime Substitute Sidecars

Another concept thought of and explored is the idea of a Lambda Runtime Substitute Sidecar (LRSS). This concept revolves around replicating the internal event handling mechanisms used by AWS Lambda within serverful deployments, offering a potential "lift and shift" solution that would allow deploying Lambda functions on serverful deployment targets.

At its core, Lambda operates by using a runtime environment that communicates with the Lambda service via its runtime API. This runtime continuously checks for new events by querying the runtime API, while the Lambda service internally connects to event sources such as SQS queues or S3 buckets. In essence, the runtime serves as an intermediary between the Lambda service and the business logic executed within the Lambda function.

[45] `https://openapi.tools/`
[46] See here (swagger.io) or here (openapi-generator.tech)
[47] `https://openapi-generator.tech/`

The idea behind the LRSS is to replace the internal event mechanisms by introducing a sidecar container that handles event management. A sidecar is a companion container that runs alongside the primary container — in this case, the Lambda function — and it would take over the task of continuously polling an event source like SQS for new events, bypassing the native Lambda service infrastructure. Figure 4.3 illustrates this concept using an SQS event source as an example.

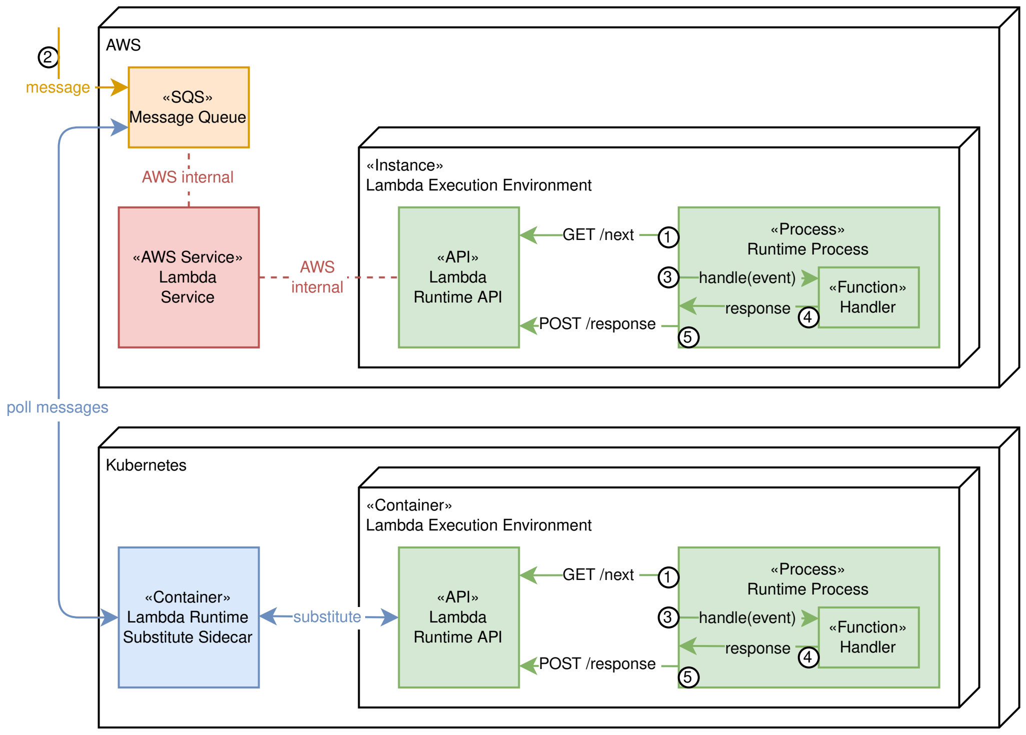


Figure 4.3: *Illustration about the concept of Lambda Runtime Substitute Sidecars (LRSS): a sidecar container is introduced to handle event polling with AWS SQS, replacing the native AWS Lambda service infrastructure to "lift and shift" the Lambda processing functions to a serverful deployment target — here: Kubernetes. Green-colored boxes and arrows indicate Lambda function-related components and their interactions within the system; red and blue indicate AWS-specific infrastructure and its replacement by the LRSS, respectively. Orange is used to color-code the SQS component. The sequenced numbering demonstrates the flow of events from the SQS queue to the Lambda function:* ***(1)*** *the Lambda runtime queries the Runtime API and waits for its next invocation;* ***(2)*** *a message is put on the SQS queue;* ***(3-5)*** *the runtime API passes the SQS event to the Lambda function to be processed.* ***NOTE:*** *This illustration, as it is, does not reflect all details and is for illustrative purposes only. It is not 100 % accurate, AWS internals are also only hinted at.*

The LRSS brings flexibility in situations where Lambda processing functions need to be transitioned to a serverful environment without significant refactoring. It enables a "lift and shift" approach for use cases where constraints such as budget limitations or legacy dependencies prevent the continued use of Lambda in its original form. For instance, an organization that needs to migrate from a serverless to a serverful deployment target due to cost constraints could leverage this solution to avoid rewriting their Lambda functions and incurring additional expenses. However, there are significant limitations to this approach. One primary constraint is that the infrastructure supporting the Lambda function must still be accessible by third-party services outside the AWS ecosystem. For example, S3 as event source is AWS exclusive[48] with AWS-specific event destinations, i.e., AWS Lambda or AWS SQS. This means that applications

[48]`https://docs.aws.amazon.com/AmazonS3/latest/userguide/EventNotifications.html`

outside the AWS ecosystem can not be triggered by S3 directly, and therefore, the sidecar would not be able to poll S3 events. REST APIs for instance are not suitable for this approach either, even though the sidecar could theoretically handle HTTP requests if implemented resembling an HTTP proxy. However, the AWS API Gateway is not accessible for the LRSS too (Lannurien et al. 2023, p. 303), which is why native serving applications built on Lambda typically involve such API Gateway (Eismann et al. 2021a, p. 6; Leitner et al. 2019, p. 9), are disregarded for the LRSS. The focus for the LRSS would be to provide a generic solution that can be used to substitute the internal event sourcing mechanisms of Lambda, enabling Lambda processing functions to be deployed on serverful deployment targets. At the time of writing, no such framework or container exists, nor have any approaches been found, therefore a proof of concept LRSS implementation has been developed for SQS[49].

While this method provides a quick workaround for certain scenarios, it is far from a universal solution and to be treated as more of a workaround. None of the requirements are crucially broken or significantly fulfilled, but, the LRSS provides an interesting way to migrate Lambda processing applications to serverful deployment targets when time or resources are limited.

Similar to LRSS, is the concept of the Distributed Application Runtime (Gatev 2021) (DAPR)[50]. Both use a sidecar model to handle external event sourcing and service communication. The LRSS concept aims to replace Lambda's internal event handling, while DAPR provides a broader, platform-agnostic framework for event-driven architectures, supporting various message brokers and services. Both approaches aim to decouple application logic from infrastructure-specific mechanisms, although DAPR offers more sophisticated flexibility and is cloud-agnostic, making it a more general and versatile solution. However, DAPR would need to be present in the development from the start. In terms of FaaS in the context of hybrid deployment, DAPR during testing was only usable in conjunction with Azure Functions and their DAPR extension[51], so at the risk of vendor lock-in.

Recap and Summary

To summarize the requirements set and the exploration of existing solutions and approaches regarding hybrid deployment models:

Requirements R1 to R4 focus on ensuring that solutions are robust, scalable, and maintainable and emphasize the importance of a solid foundation of development or deployment experience.

Although solutions exhibit potential, none have yet achieved the level of functionality necessary to fully support hybrid deployment models as required. For example, the IfC paradigm shows promise but remains too immature for practical application in this context. Similarly, serverless function frameworks are effective for simpler use cases, but face limitations when handling more complex scenarios. OpenAPI-based service descriptions and code generation tools also show potential, yet they are not fully capable of supporting hybrid deployments. Lambda Runtime Substitute Sidecars offer a workaround but are not a universal solution and as shown only applicable in a specific context under specific constraints.

Requirement **R4** with its maturity aims to address the issue of missing know-how and its complexities of development and deployment. The maturity of existing approaches and solutions was not difficult to focus on and was mostly given if the right choices were made. Thus, missing know-how can be weakened and hints that indeed such aspect plays a role in development and

[49]https://github.com/a-grasso/master-thesis-public/tree/main/lambda-runtime-substitute-sidecar-sqs
[50]https://dapr.io/
[51]https://learn.microsoft.com/en-us/azure/azure-functions/functions-bindings-dapr

deployment experience. However, development or deployment experience is of secondary importance if the capability for hybrid deployment is not present in the first place. As with most of the solutions discussed, they have potential but lack the necessary extensions for such capability. Existing solutions would therefore need to be extended to encourage this. Requirements **R1**, **R2** and **R3** are more focused on development experience and are not as critical to the capability of hybrid deployment models themselves. Similar to **R4**, they are important downstream of hybrid deployment throughout development.

Requirements R5, R6, and R7 are crucial for hybrid deployment models. Requirement **R7** aims to delay deployment target decisions to later stages, offering greater flexibility. However, with existing solutions, deployment targets are selected before or at build time, which means that approaches, except for the LRSS, interfere with the development process. This contradicts the goal of enabling truly flexible hybrid deployment models. Although no solution explored meets **R7**, the following requirements **R5** and **R6**, however, provide a foundation that helps minimize the negative impact of deployment decisions on the development process. In particular, **R5** highlights the importance of solution flexibility to accommodate the diverse and dynamic nature of hybrid deployment models.

4.3.5 Recommended Approach: Serving Applications

After exploring existing solutions and approaches in subsection 4.3.4, the following recommended approach for hybrid deployment of serving applications was conceptualized.

Now, given the optimal experience in section 4.3.3, a possible approach would, in essence, combine the insights of explored solutions and approaches and address the requirements set in section 4.3.3. Such an approach incorporates all requirements and lessons learned and is designed to enable hybrid deployment models in a seamless and flexible manner. That means specifically, requirements and so far gained knowledge can be puzzled together to form such an approach that would involve the following:

1. **Requirements R1-R4**: the approach enables the most mature, stable and ergonomic development (and deployment) experience possible.
   - For this, for serving applications, HTTP frameworks meet the requirements. For example, but not exhaustive: Go-Gin[52], Spring Boot or NestJS[53].
   - Such HTTP frameworks are: common and well-known; mature and well-documented; provide a high-level abstraction and are inherently serverful so locally testable.
2. **Requirements R5-R6**: the approach ensures that business logic is separated from integration logic and that flexibility and adaptability is fostered.
   - Again, serverful HTTP frameworks can inherently conform to both requirements.
   - They provide a high-level abstraction and separation of concerns, which allows modularization and adaptation.
   - Together with an OpenAPI specification and code generation, for example, these requirements can be even better met.
3. **Requirement R7**: the approach results in a deployment artifact, that is deployable to any deployment target or the broadest range of it.
   - The broadest applicable artifact could be a container to cover serverful and serverless deployment targets (Li et al. 2023, p. 1523).

[52] https://github.com/gin-gonic/gin
[53] https://nestjs.com/

   - Containerized applications are inherently serverful, but can be serverless deployable (Li et al. 2023, p. 1528).

4. To meet the prerequisite specifications explained in subsection 4.3.1, the *serving application* must be ***stateless***.
   - This can be achieved, however, it is to be noted that this is mostly in the hands of the developers, but frameworks can be utilized to foster such statelessness.

Combining all the above results in a *serverful yet stateless containerized HTTP web service application* to be capable of being serverless deployed. Such an application is developed with a *serverful-first* approach but with serverless constraints considered to be deployed onto such deployment targets. While the application is inherently serverful, it would be *adapted* for serverless deployment targets. It is similar to the lift and shift approach with LRSSs, but the other way around.

Deploying such applications on serverful deployment targets is straightforward, the containerized applications can be deployed to any target capable of running containers. The HTTP framework used for the application is put into a container image and deployed as is. Be it Kubernetes, AWS ECS based on EC2, Azure Container Instances, a VM, or a bare-metal server running Docker for instance. For CaaS offerings well within the serverless spectrum, such as ECS based on Fargate, Azure Container Apps or self-hosted OpenFaaS, the containerized applications can also be deployed as if serverful, but with the serverless experience facilitated.

To enable FaaS deployment specifically, the application is adapted to be able to run within the FaaS environment. Given the example HTTP web service in listing 4, different approaches for adaptation of HTTP frameworks to FaaS were explored while conceptualizing the recommended approach.

```go
// File: src/main.go
import "net/http"

func handler(ctx context.Context, w http.ResponseWriter, _ *http.Request) {
  fmt.Fprintf(w, "Hello!")
}

func main() {
  http.HandleFunc("/", handler)

  http.ListenAndServe(":8080", nil)
}
```

Listing 4: *Example of a simple HTTP web service in Go. The service, using the Go standard HTTP library, listens on port 8080 and responds with a greeting message when a request is made to the root path.*

Adaptation of HTTP Frameworks for FaaS

Adaptation to FaaS can be achieved in different ways, but the core strategy is simple: the FaaS integration logic passes the HTTP request through to the HTTP framework and passes through its HTTP response as the function's response. The adapter acts as middleware between the FaaS environment and the HTTP framework.

In-code adaptation is available for HTTP frameworks with adapter libraries that enable running them within FaaS environments. For instance, adapter libraries like Azure Function Adapter for NestJS[54] or the AWS Lambda Go net/http server adapter (ALGNHSA)[55] provide

[54] https://github.com/nestjs/azure-func-http
[55] https://github.com/akrylysov/algnhsa

middleware-like functionalities to allow HTTP-based applications to run within specific FaaS environments as seen in listing 5 for the latter.

```go
// File: src/main.go
import (
  "net/http"
  "github.com/akrylysov/algnhsa"
)

func handler(w http.ResponseWriter, _ *http.Request) {
  fmt.Fprintf(w, "Hello!")
}

func main() {
  http.HandleFunc("/", handler)
  service := http.DefaultServeMux

  // the adapter wraps the framework for AWS Lambda compatibility
  algnhsa.ListenAndServe(service, nil)
}
```

Listing 5: *Example of a simple HTTP web service in Go that is adapted with the ALGNHSA adapter for AWS Lambda. The adapter wraps the HTTP framework to make it compatible with the Lambda environment. Internally, the adapter starts a Lambda handler — see listing 3 — but proxies the Lambda request to the HTTP framework.*

With such adapter libraries, the HTTP framework can be used as is, and the adapter takes care of the integration with the FaaS environment. In the *Project D* survey in Appendix C, the Azure Function Adapter for NestJS was later used to develop and deploy their new REST API endpoints with Azure Functions. According to the survey, this was very successful, as their development experience before was similar to that of *Project X* or *Project B*.

The serverless function frameworks as discussed in subsection 4.3.4 work in similar ways, though with more abstraction. Quarkus's REST and HTTP extension dependencies for AWS Lambda[56] for instance, could also be used in conjunction with any Quarkus HTTP framework apart from Quarkus Funqy to adapt to Lambda[57].

However, the *in-code adaptation* approach has development implications, as the same container image of listing 5 for instance would not be deployable across the spectrum, i.e., to non-Lambda deployment targets, without *conditional integration branching*. See listing 6 for a demonstration of how the integration would have to branch between a specific FaaS integration with its adapter and the normal execution of the HTTP service for the same example. Although business logic is kept separate and abstracted through the HTTP framework, platform-specific adjustments for FaaS and specific integration logic are still part of development. While this adaptation approach works, it introduces conditional deployment logic, requiring platform-specific adjustments for each FaaS offering. Though kept to a minimum, *conditional integration logic branching* introduces complexity and reduces maintainability. A more robust solution could involve abstracting this logic into a shared library that handles conditional branching automatically. However, this doesn't fully mitigate the underlying issue of code interference and fragmentation.

Out-of code adaptation is a possible diverting approach to not interfere with the development of the application and its *serverful-first* approach at all. Instead of adapting the applications' execution, the FaaS platform takes the containerized applications as-is, starting the web server within the FaaS environment and querying it via HTTP. With it, the same core strategy as before applies: the FaaS environment proxies external requests to the web service running

[56]https://quarkus.io/guides/aws-lambda-http
[57]See here (github.com) for a Maven archetype example covering this

```go
// File: src/main.go
...

func inLambdaEnv() bool {
  ...
}

func main() {
  http.HandleFunc("/", handler)

  if (inLambdaEnv()){
    algnhsa.ListenAndServe(http.DefaultServeMux, nil)
  } else {
    http.ListenAndServe(":8080", nil)
  }
}
```

Listing 6: *Example of the simple HTTP web service in Go, with conditional integration branching for AWS Lambda with ALGNHSA (listing 5) or standalone execution (listing 4). The* ***inLambdaEnv()*** *function checks if the service is running inside the Lambda environment. Depending on the result, the service is either started with the ALGNHSA adapter or as a standalone HTTP web server.*

inside the FaaS environment. The *out-of-code adaptation* builds on the clean interface of an HTTP service, so the underlying implementation of such service can be as flexible as necessary. So to speak, deployment artifacts have to fulfill the sole contract of "a stateless HTTP service reachable on port x" and the FaaS environment should take care of the rest.

The Lambda Web Adapter[58] (LWA), an official open source AWS Lambda extension provided by AWS, is one possible solution that encapsulates an HTTP server within a Lambda function. It allows the web service to be FaaS platform-agnostic, simplifying hybrid deployment while keeping the *serverful-first* application's container image truly consistent across serverful and serverless environments. There is no need to adapt the application itself, the FaaS environment takes care of its integration. The LWA extension is added to the container image that is created, the application code is not affected. This approach is possible for Lambda because of the way the Lambda execution environment and its internals work with the Lambda runtime, extensions, and the Lambda service. The LWA extension includes a custom runtime that launches *any* HTTP server within the Lambda function, and the Lambda runtime proxies the HTTP requests to the internally running HTTP server. However, it is important to note that it is not possible to deploy the application using the LWA on other FaaS offerings, as the LWA is specific to Lambda. For more information on the LWA, the reader is referred to its official GitHub repository[59] or official AWS documentation[60].

The Azure Function custom handler[61] is the same approach for Azure Functions that allows any HTTP web server executable to run inside an Azure Function. The custom handler is a wrapper around the web server executable that is launched by the Azure Function runtime and the HTTP requests are proxied to the web server. As with the LWA, the Azure Function custom handler is specific to Azure Functions and cannot be used for other FaaS offerings.

In addition, OpenFaaS has built-in support for HTTP web server workloads as functions[62], so the same approach can be used when deploying on OpenFaaS. OpenFaaS is designed to run HTTP microservices natively[63], no adaptation is required — although the developers note that

[58] `https://github.com/awslabs/aws-lambda-web-adapter`

[59] `https://github.com/awslabs/aws-lambda-web-adapter`

[60] The presentation of its introduction can be found here (d1.awsstatic.com) or see AWSLabs' explanation here (github.com) or this AWS community blog post here (community.aws)

[61] `https://learn.microsoft.com/en-us/azure/azure-functions/functions-custom-handlers`

[62] See the example for Go/http here (docs.openfaas.com)

[63] `https://docs.openfaas.com/reference/workloads/`

native functions are better supported than containerized services

To have a practical example, the GitHub repository linked to this work, features an example project based on the sample architecture in Figure 4.2 in AWS, featuring both *in-code* and *out-of-code adaptation* with the ALGNHSA adapter and the LWA respectively[64].

Limits, Constraints, and Contra-Indications

While the recommended hybrid deployment approach offers significant benefits, particularly in terms of flexibility and adaptability, it also presents notable limitations and constraints that must be considered and contra-indications that emerged during exploration.

A contra-indication to the adaptation approach stems from the mismatch between serverless paradigms and certain application behaviors. Any deviation from statelessness mentioned in subsection 4.3.1, such as using persistent data storage within the application layer, can lead to performance bottlenecks and operational failures in serverless environments (Lannurien et al. 2023, pp. 286, 304, 308; Baldini et al. 2017, p. 4). Developers must be mindful of best practices, including avoiding asynchronous background tasks that extend beyond the request-response lifecycle. As discussed in section 4.2, developers unfamiliar with serverless paradigms may struggle with debugging, monitoring, or optimizing performance, potentially slowing down development or leading to inefficient deployments. Asynchronous background tasks were found especially restrictive for this approach. For example, during testing with the LWA, execution environment freezing during idle periods led to unexpected behavior, which for instance degraded observability. Asynchronous tasks that extended beyond the request lifecycle failed to complete in time or continued at a later stage, resulting in incomplete executions. Hidden asynchronous features within frameworks that may be unknown to developers could exacerbate these problems. On the one hand, it would have to be known in advance, and on the other, would have to be made possible to be disabled; in addition to the fact that it was difficult to debug such misbehavior. This contra-indicates the approach, it is less effective or even unsuitable for workloads requiring such functionalities.

Regarding *in-code adaptation*, if multiple FaaS deployment targets must be supported, the conditional integration logic can negatively impact maintainability. This adaptation variant relies on conditional integration logic to meet **R7**, which can become increasingly complex, especially when targeting multiple distinct deployment targets. The use of conditional logic can fragment the codebase, ultimately reducing the maintainability of the project. While offloading this logic to an open-source library or framework can alleviate some of these issues, ensuring that it remains up-to-date with specifications may remain a challenge. A potential mitigation would be waiving requirement **R7**, a trade-off that developers might consider. However, if complicated special adaptation or configuration is required for specific FaaS deployment targets, the in-code customization approach may not be appropriate to promote each of these.

A major limitation of the recommended approach is its dependency on adapters or adapter libraries for specific FaaS platforms. While adapters or adapter libraries are available for platforms such as AWS Lambda, Azure Functions, or OpenFaaS, they may not be available for other FaaS platforms. This could limit deployment target options, forcing developers to either create custom solutions, exclude certain platforms, or bet on cloud providers to foster such adapters. Moreover, developers are limited by the features facilitated by the adapters and what they lack. The *out-of-code adaptation* approach for instance is available for AWS Lambda with the LWA, Azure Functions with their custom handler, or natively in OpenFaaS. Though available for the two most popular public cloud providers[65] and a private cloud alternative, *out-of-code adaptation* is not available for other platforms such as Google Cloud Functions. The latter though supports hosting such HTTP web service as in listing 4 directly, though with *in-code adaptation*

[64] `https://github.com/a-grasso/master-thesis-public/tree/main/recommended-serving-approach`

[65] According to the 2024 developer survey of StackOverflow: see here (survey.stackoverflow.co)

required[66]. *Out-of-code adaptation* is not directly weakened in terms of the broadest applicability of hybrid deployment, although depending on whether the application requires compatibility with certain FaaS platforms and depending on the FaaS platforms chosen, it may not be applicable, leaving *in-code adaptation* as the only viable option, despite its potential added complexity. As mentioned, conditional integration logic can become cumbersome, increasing both complexity and maintenance overhead.

While *adapted serverful-first* satisfies **NFR4** by allowing to shift deployment targets as needed, the remaining non-functional requirements discussed in section 4.3.3 may contradict this approach. However, potential strictness of the **NFR1-3** requirements may lead developers to avoid this approach (B.2). In terms of latency and performance overhead, discussed later in detail in section 4.4, the approach inherently introduces overhead. Both *in-code* and *out-of-code adaptation* methods will introduce additional resource usage and potential performance degradation, particularly regarding cold-start latency. This can be especially problematic for latency-sensitive applications. Incorporating a full framework into serverless will contribute to overhead, while the adaptation layer may introduce further latency. In particular, the *out-of-code adaptation* method adds additional overhead through containerization and intercommunication with the FaaS environment, resulting in potentially noticeable latency. Although the NFRs do not directly contradict this approach, stricter specifications of the NFRs may make it less appropriate.

Recap: Hybrid Deployment for Serving Applications

The *serverful-first* approach for serving applications to enable hybrid deployment brings what satisfies requirements **R1**-**R6**. Incorporating serverless constraints during development and with the *adaptation* of such applications for FaaS in particular finally satisfies requirement **R7** and the full serverful-serverless spectrum is accessible for deploying serving applications. Thus, *adapted serverful-first* is the recommended approach for the hybrid deployment of serving applications. With it, developers develop an *adaptable HTTP web server executable* and with it *can decide about deployment later.*

The foundation of established HTTP frameworks counteracts what was previously highlighted in section 4.2 as a negative for serverless environments: their poor development experience. Lack of expertise can also be mitigated, as the development experience is similar to that of serverful development, and facilitated by familiar and well-known frameworks. Meanwhile, the separation of business and integration logic and the flexibility of containerized applications provide the necessary adaptability for hybrid deployment to ensure that the serverless experience does not negatively impact development.

Depending on whether it is facilitated, the *out-of-code adaptation* fully satisfies the **R7** requirement, as the same container image would be deployable across the serverful-serverless spectrum. However, *in-code adaptation* will remain with greater flexibility in this term and will be applicable for any FaaS deployment target customization with a slight trade-off in terms of development maintainability.

Though, as mentioned in section 4.3.5, limitations and constraints remain to be considered that may or may not affect applicability as discussed nonetheless.

4.3.6 Recommended Approach: Processing Applications

Exploring hybrid deployment for processing applications prominently faced the issue of heterogeneous integration components. Unlike serving applications, which benefit from the commonality of HTTP-based web services, processing applications face a more complex landscape. These applications use a variety of technologies, protocols, and patterns, making integration more

[66] `https://cloud.google.com/functions/docs/create-deploy-http-go`

difficult. For example, in the architecture shown in Figure 4.2, processing could involve messaging systems such as AWS SQS (AWS), Kafka[67] (self-hosted Kubernetes), or Azure Event Hubs[68] (Azure). Supporting such diverse components for hybrid deployment can become highly fragmented.

As discussed in section 4.2, FaaS has become well-suited for processing applications due to its flexibility and the rich cloud-native ecosystems that developers can leverage (Baldini et al. 2017, p. 9; Hassan et al. 2021, p. 16; Jonas et al. 2019, p. 7; Eismann et al. 2021b, p. 37). FaaS acts as a “glue” between cloud services, making it an ideal foundation for processing workloads. However, this tight integration with cloud providers presents a challenge for enabling hybrid deployment across the serverful-serverless spectrum, as the rich ecosystems are one of the key advantages that public clouds offer.

Given this, the recommended approach is to adapt to these circumstances by adopting a *serverless-first* strategy for processing applications and aim for *portability*. This approach allows developers to fully leverage the benefits of FaaS and cloud-native ecosystems initially, while leaving room to port applications to serverful deployment targets as needed or back.

Guidelines on Software Development

Requirements **R5** and **R6** are particularly important for hybrid deployment, as they emphasize the need for modularity and separation of concerns: to separate business logic from integration. As the FaaS development model and its integration logic are inherently different, see listing 4 and listing 3 for instance, the two requirements are further emphasized. Therefore, to facilitate the transition between serverless and serverful deployment targets for processing applications, a set of recommended architectural and development guidelines for *serverless-first* processing applications have been conceptualized and worked out. These guidelines are to ensure the goal of easily *portable* processing applications to serverful deployment targets when necessary, or vice versa. They are designed to promote modularity, flexibility, and adaptability, enabling developers to seamlessly transition between serverful and serverless environments. However, much of the discussed aspects are to be recognized from SOA or event-driven microservice architecture (FreeWheel Biz-UI Team 2024; Oliveira Rocha 2022):

Keep events consumable by third-party already surfaced when discussing the LRSS in subsection 4.3.4. This guideline is to ensure first and foremost event processing interoperability across the serverful-serverless spectrum. In that, systems or applications events are accessible by third-party services, especially. For example, as already mentioned, AWS S3 events can only trigger AWS Lambda functions, the system architecture would need to be adapted to also have client SDKs for example process such messages. A solution to this problem would be to facade such S3 events with an AWS SQS queue.

Fan-out instead of multi-responsibility. To maintain separation of concerns, system architecture should adopt a “fan-out” model, where a single responsibility is delegated to each processing application (Taibi, Ioini, et al. 2020, p. 185). Rather than allowing a single processing application to handle multiple tasks, workloads should be divided into smaller, specialized units. This approach would ensure that each processing application remains focused and independent. For example, a message could be broadcasted to multiple processing applications, each processing a distinct aspect of the message, instead of a single application performing multiple processing tasks. Such a use case could be to have a notification, data transform, analysis, and auditing service which each would need to process the same message. This also simplifies troubleshooting, as individual services can be monitored and scaled independently.

[67] https://kafka.apache.org/
[68] https://azure.microsoft.com/en-us/products/event-hubs

Keep processing applications isolated. Systems should be designed to avoid direct communication between processing applications. Instead of tightly coupling the flow between processing units, which increases complexity and reduces flexibility (Taibi, Ioini, et al. 2020, p. 185), it is recommended to use message queues or event brokers (such as AWS SQS, Kafka, or Azure Event Hubs) to decouple components. This allows each processing application to remain isolated and independent of the others, which simplifies scaling, failure management, and deployment (Villamizar et al. 2017, pp. 244, 246). For instance, pre-processing units push their results to a message queue, which is then consumed by downstream processing applications instead of directly triggering them, enabling a more flexible and robust architecture as opposed to a monolith application for instance (Deng et al. 2023, p. 9). In this model, each processing application can process its tasks without being impacted by upstream or downstream dependencies, enabling easier transitions between serverful and serverless environments.

Shift complexity to the infrastructure to reduce the burden on application logic. For example, the complexity of event handling, failover, redundancy, and batching should be managed by the underlying infrastructure. Instead of implementing failover mechanisms within the application code, relying on an infrastructure that provides built-in redundancy and fault tolerance can simplify development and improve reliability (Oliveira Rocha 2022, pp. 30, 31). Similarly, message batching or event processing should be delegated to the infrastructure wherever possible, freeing application developers from having to deal with these issues directly. BaaS offerings such as managed message queues or message brokers can provide this level of abstraction, ensuring more robust and scalable event processing.

Keep processing scope and business logic small and confined. This guideline stems from one interviewee sharing that keeping functions lightweight is key to ensuring that they can be easily ported between serverless and serverful environments (B.6). It also goes hand in hand with the *fan-out instead of multi-responsibility* guideline.

To ensure portability and scalability, processing logic should be kept small, focused, and confined to single responsibilities. This adheres to the principles of microservices and serverless computing, where each function or service is designed to handle a specific task (Deng et al. 2023, p. 9). By limiting the scope of each processing unit, the application would become easier to manage, test, and deploy across different environments (Fan et al. 2020, p. 204). Smaller, modular units of logic are also more portable, enabling smoother transitions between serverless and serverful deployments without requiring significant rewrites or refactoring.

Embrace the serverless development model. When developing processing applications, it is necessary to fully embrace the characteristics of the serverless development model within means of *serverless-first*. These include the already discussed principle of statelessness, where the internal state is either entirely avoided or outsourced to external storage systems like databases or caches. Processing applications should therefore be designed for short-lived, confined computations with ephemeral execution contexts. This would also pay into the account of keeping the processing logic lightweight. It is important to note that the serverless development model is not the same as the FaaS development model, and the guidelines already discussed must be followed to ensure that applications remain portable across different deployment targets.

Moreover, the architecture should be event-driven (Baldini et al. 2017, p. 11), where processing is triggered by events such as API calls, file uploads, message queues, or notifications (Lannurien et al. 2023, p. 288). By adhering to these principles, applications remain lightweight, scalable, and easier to port, also across the serverful-serverless spectrum (Eismann et al. 2021b, p. 34).

Keep business and integration logic decoupled by contract. The final guideline stresses the decoupling of business logic from deployment-specific concerns through clear contracts, i.e., requirements **R5** and **R6**. In this context, contracts refer to well-defined interfaces for event handling that are independent of the underlying event transportation mechanism (e.g., HTTP, SQS, Kafka). The serverless function frameworks presented in subsection 4.3.4 for example exemplify this aspect. By establishing such contracts, the processing logic remains agnostic to the infrastructure it is deployed on. For example, an event processing service should be able to handle events the same way, regardless of whether they are delivered via a serverless function invocation trigger or through a serverful message broker. This ensures that processing logic remains consistent across different deployment targets and that processing applications can be ported easily between environments with minimal changes.

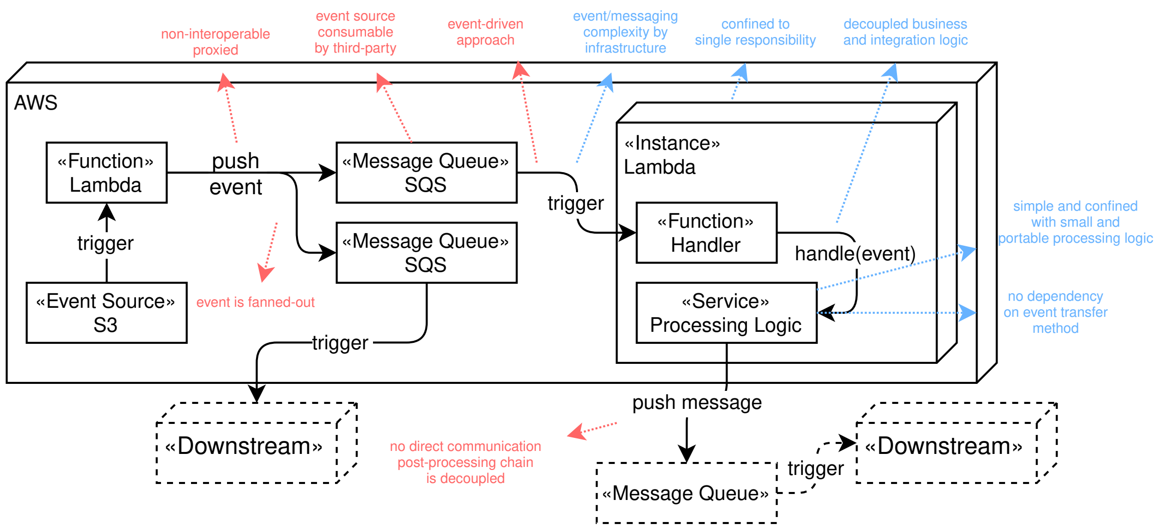

non-interoperable proxied
event source consumable by third-party
event-driven approach
event/messaging complexity by infrastructure
confined to single responsibility
decoupled business and integration logic
AWS
«Function» Lambda
push event
«Message Queue» SQS
«Message Queue» SQS
trigger
«Event Source» S3
event is fanned-out
trigger
trigger
«Instance» Lambda
«Function» Handler
handle(event)
«Service» Processing Logic
simple and confined with small and portable processing logic
no dependency on event transfer method
push message
«Downstream»
no direct communication post-processing chain is decoupled
«Message Queue»
trigger
«Downstream»

Figure 4.4: *The diagram demonstrates different guidelines for architecture and development as an illustration of where these guidelines would apply given a sample processing system architecture. Red-colored dotted arrows indicate guidelines regarding architecture, and blue-colored dotted arrows indicate guidelines regarding development. The system architecture depicts AWS S3 event processing, but with AWS SQS queues to have events consumable by third-party downstream. The AWS Lambda function on the right thereby is demonstrated to be developed according to set development guidelines. Dashed boxes indicate any possible third-party downstream system.*

To recap the guidelines on processing application architecture and development, Figure 4.4 visualizes these guidelines and their applicability to a given sample system architecture.

The GitHub repository linked to this work features an example project based on the sample architecture in Figure 4.2 in AWS, featuring the guidelines worked out prior[69].

How to Use Application Portability

Following the recommended *portable serverles-first* approach, it ensures that processing applications are developed with the flexibility to be deployed across the serverful-serverless spectrum. Given the portability, how to facilitate the shift between serverful and serverless deployment targets remains an open point to be considered in this section.

In subsection 4.3.5, *in-code* and *out-of-code adaptation* strategies were discussed for serving applications. With the flexibility of *in-code adaptation*, this approach can be tuned to fit the context of processing applications. Instead of adapting the underlying framework, the processing

[69] https://github.com/a-grasso/master-thesis-public/tree/main/recommended-processing-approach

logic itself is integrated and adapted to the deployment target. For example, the processing logic is designed — according to the given guidelines — to handle events from any given source, such as HTTP requests, message queues, or MQTT brokers, without requiring changes to the underlying processing logic. Depending on where to deploy to, the service is integrated with the respective event source, and the processing logic is triggered accordingly. For instance, the processing logic can be triggered by an AWS SQS message in a serverless environment, or by an HTTP request in a serverful environment. This approach ensures that the processing logic remains consistent across different deployment targets, and only the integration logic needs to be adjusted or extended. Given in listing 7, the processing logic is kept separate from the integration logic, which allows for easy adaptation of the integration logic to different deployment targets. However, if requirement **R7** is to be satisfied, this would result in the same conditional integration logic as discussed in subsection 4.3.5 and is therefore not recommended, but possible — it is also used within listing 7.

Furthermore, to achieve even greater portability and abstraction for processing applications, standards like Knative[70] and CloudEvents[71] could be leveraged. Knative Eventing provides a streamlined way to route events in a Kubernetes environment, allowing applications to process events across multiple sources — including public cloud events. CloudEvents offers a specification to describe event data in a common format, making it possible to move events across different platforms and clouds. Such standards further enhance portability by supporting the same processing logic to be deployed in different environments, including self-hosted Kubernetes clusters, public cloud functions, or hybrid infrastructures, without significant code changes (Jonas et al. 2019, p. 21). In conjunction with set guidelines, they could help processing applications to remain infrastructure-agnostic by simplifying their adaptation across various cloud and on-premises platforms. By additionally adopting these technologies, the architecture becomes more resilient and adaptable to changes in infrastructure, supporting the recommended serverless-first and portability-focused approach for processing applications.

Limits, Constraints, and Contra-Indications

The guidelines are not a one-size-fits-all solution, but rather a set of best practices and strategies that developers must adapt to their specific use cases. Unlike the *adapted serverful-first* approach for serving applications, the *portable serverless-first* approach for processing applications is more open-ended and requires developers to make decisions based on their specific requirements and constraints.

With that, the biggest drawback of the *portable serverless-first* approach with its guidelines is that it places the responsibility for correct implementation squarely in the hands of the developers. The guidelines are just that, guidelines, and it is up to the developers to ensure that processing applications are developed in a way that conforms to them. While this approach offers greater flexibility and freedom, it also opens the door to potentially damaging choices that can undermine the portability or efficiency of processing applications.

Following that, what contra-indicates the approach is the potential for developers to make decisions that negatively impact the portability of processing applications. For example, failing to adhere to the guidelines could result in processing logic that is tightly coupled to specific deployment targets, making it difficult to transition between serverful and serverless environments or to scale the application effectively. A pitfall in developing portable event-driven systems is the incorrect use of the event-driven paradigm (Taibi, Ioini, et al. 2020, p. 87). Developers might, for example, introduce unnecessary complexity by tightly coupling services through direct communications rather than using a decoupled message broker, which negates the advantages of hybrid deployment and is exactly what the guidelines should make sure to avoid. Another issue is that certain bad practices or antipatterns (Castro, Isahagian, et al. 2022, p. 11) during the

[70] https://knative.dev/docs/eventing/
[71] https://cloudevents.io/

development of FaaS-based applications can compromise portability in the future. For example, tightly coupling processing logic to a specific cloud provider's infrastructure or using provider-specific services without abstraction layers can lead to significant refactoring work when moving to a different environment.

Additionally, while the portability approach simplifies transitions between serverful and serverless environments, it does not eliminate the implementation costs associated with making those transitions. For instance, even though the system can switch easily between deployment targets, developers still need to implement the needed integration logic and the effort required to implement portability features remains highly application-specific.

Based on that, the development or architectural complexity that this approach might introduce for systems or applications of a more modest size might outweigh the benefits in the grand scheme of things. For instance, given one of the go-to examples for serverless — image processing (Castro, Ishakian, et al. 2019, p. 50). In such a case, the guidelines may be too much overhead and a more straightforward approach may be more appropriate. For example, going all in and coupling processing and integration, and only bearing the cost of porting if necessary.

Furthermore, the key constraint in adopting this approach is the dependency on infrastructure capabilities for managing event complexity. Developers would first rely on managed services like message queues or event brokers to handle critical aspects such as failover, redundancy, or batching. This introduces a reliance on cloud providers for those capabilities, which might limit flexibility when shifting to serverful environments that do not offer the same level of built-in infrastructure support.

In terms of requirements, the approach builds off of **R5** and **R6** as mentioned. Requirement **R7** remains similar to that for serving applications, but the *in-code adaptation* strategy is not recommended due to the potential complexity and maintainability issues. However, **R7** cannot be fully satisfied. For **R1**-**R4**, no justifiable quantification is made, and the guidelines do not have a clear negative impact, although each of the four requirements is still recommended.

In terms of non-functional requirements, quantifying performance, scalability, and reliability (**NFR1**-**NFR3**) is difficult because much of the success of this approach depends on developers following the guidelines correctly. While the guidelines themselves are designed to improve scalability and flexibility, poorly implemented or incomplete compliance could lead to challenges such as bottlenecks in message handling or increased complexity in deployment. In particular, scalability could suffer if services are not decoupled as recommended.

For **NFR4**, cost considerations, the most critical factor is development overhead. Following these guidelines could involve a significant up-front investment in time and resources, as discussed, particularly for applications that may not require this level of modularity and portability. Conversely, not following the guidelines could lead to higher costs later on, especially when transitioning between serverful and serverless environments. Developers must strike a balance between initial development costs and the long-term benefits of portability.

In summary, while the guidelines provide a framework for building portable processing applications, the actual costs and benefits will largely depend on the size, complexity, and future needs of the application. However, the approach arguably provides developers with the flexibility they need.

Recap: Hybrid Deployment for Processing Applications

The recommended approach for hybrid deployment of processing applications emphasizes a *serverless-first* strategy with a focus on *portability*. In accordance to requirements **R5** and **R6**, the approach is designed to enable developers to fully exploit the rich cloud-native ecosystems and FaaS capabilities, while ensuring that applications can transition to serverful environments as needed.

By following the recommended guidelines — such as keeping processing logic small and modular, embracing statelessness, and decoupling business logic from infrastructure-specific concerns — developers can build flexible and portable processing applications able to be deployed across the serverful-serverless spectrum. However, this approach is not without limitations as it remains a mere guide for developers, and is highly dependent on the same for the quality of the application and its portability.

4.4 Evaluation

In this section, the previously elaborated recommended approaches for hybrid deployment and the worked-out influencing factors in section 4.2 are evaluated using the methods described in chapter 3.

4.4.1 Views of Practitioners

Practitioner Interviews

Practitioner S was interviewed in section B.5 to get feedback on the recommended *adapted serverful-first* approach for serving applications, expert opinions on its suitability, and open points for evaluation.

The interviewee first highlighted that serverless deployment targets primarily offer the benefit of automated scaling, which was seen as a critical benefit. After being introduced to the recommended hybrid deployment approach for deploying HTTP-serving applications, the interviewee raised several points about potential challenges and areas for evaluation:

The interviewee emphasized that the hybrid approach should be assessed for its ability to maintain scalability, comparable to native serverless functions. Ensuring that the *adapted serverful-first* approach supports parallel processing and seamless scaling — both up and down — was noted as essential. The interviewee suggested real-world benchmarks and load tests provide concrete data on whether this approach matches native serverless scaling. Related to this, they also noted that any increase in cold start time or latency could diminish effectiveness in serving use cases, recommending focused analysis in this area.

The respondent also highlighted potential development scaling complexities, recommending guidelines on best practices for development with the hybrid deployment approach. For instance, with a complex application containing numerous endpoints, they questioned whether the recommended approach could scale and be consistently maintained regarding development complexity.

Overall, the feedback highlights the importance of rigorous testing and performance benchmarks to assess whether the recommended *adapted serverful-first* approach can fully retain the benefits of serverless, particularly regarding automated scaling but also not degrade performance in terms of latency. Its recommendations underscore the need for practical guidelines to address scalability challenges and provide a strong basis for evaluating the applicability of the approach to high-demand, low-latency applications.

Practitioner A was also interviewed for their expert opinion on the recommended *adapted serverful-first* approach to deploying applications, see section B.6 for the full interview. The interviewee emphasized the need for clear communication about the benefits and potential challenges of the hybrid approach, particularly in the context of serverless deployments. They noted that the *in-code* or *out-of-code adaptation* of frameworks could have a negative impact on maintainability and risk to production, suggesting that the recommended approach should be clear about the implications and impact on these areas. Otherwise, although they were skeptical about serverless REST APIs due to past problems with FaaS, they agreed that “if someone was going to do serverless HTTP APIs, they should use [the recommended approach]”.

Practitioner J on the other hand, was interviewed in section B.4 about a particular project migrating from Azure Functions to Azure Container Apps. The interview provides insights into how the guidelines of the recommended *portable serverless-first* approach were applied in practice. Furthermore, what benefits can be expected from such a migration?

The interviewee emphasized the benefits of containerized deployment for processing workloads, citing improved control, lower costs, and better performance with fewer cold starts. They valued local debugging for development efficiency, noting that Container Apps are preferable for complex applications with third-party dependencies. Container Apps also offered flexible pricing and efficient scaling, making them a cost-effective alternative to FaaS for background tasks.

The project to be migrated was a data transformer that processed data from a queue. Their migration from Azure Functions was straightforward and "easy", so noted the interviewee. Due to how the project was structured, the migration required only minor changes to the codebase. Application code changes shown by the interviewee featured minor changes to the integration logic of the processing application — switching from Azure Function to Azure Container Apps — and the addition of a Dockerfile for containerization. The migration took place before this work and knowledge regarding the recommended *portable serverless-first* processing approach, but the interviewee's experience provides valuable insights into the feasibility and benefits of migrating processing workloads to containerized deployment targets. This insight acts as a success story in terms of the portability of *serverless-first* applications, and evaluations of the recommended approach can be drawn from this interview:

As a success story, the interviewee's experience suggests that migrating processing workloads to containerized deployment targets is feasible and can be done with little effort. Benefits of the migration mentioned strengthen points made in section 4.2 regarding serverful deployment targets. The codebase was structured in a way that would facilitate later migration; the interviewee noted that they had used the layered architecture (Richards 2015), although not *par excellence*, but in an applicable way that made migration easy for them. The layered architecture as used by the developers would relate specifically to these guidelines listed in section 4.3.6: *keep applications isolated*; *scopes small and confined*; and *decoupled by contract*. For guidelines *consumable by third-party*, *fan-out instead of multi-responsibility* and *complexity by infrastructure*, no applicable assessment could be made based on the success story. Regarding the guideline *embracing serverless development*, although the migration was successful, the interview gives no specific indication that the project was developed with serverless-first in mind, thus neither falsifying nor confirming the guideline. Although not fully overlapping with all the guidelines of the recommended *portable serverless-first* approach, the interviewee's experience suggests that it can be applied in practice and that the specified guidelines help to support the migration of processing applications to serverful deployment targets, and is a matter of anticipation and design.

Project D Survey Summary

The results in Appendix C capture three developers' perspectives on moving to a framework-based API development model using NestJS adapted for Azure Functions — i.e., the recommended *adapted serverful-first* approach. It should be emphasized that the transition they undertook was at a time before this work, but the developers were using the recommended hybrid deployment approach to deploy applications, specifically *in-code adaptation*, which also inspired the *adapted serverful-first* approach. This survey provides critical insight into the impact of the new approach on the development experience compared to the earlier raw Azure Functions setup similar to *Project X* and *Project B*. Thus, the survey helps to further evaluate the recommended serving approach for hybrid deployment in terms of the requirements **R1**-**R4** listed in section 4.3.3.

The survey responses highlight a generally low satisfaction with the original Azure Functions-based development setup. Developers were particularly dissatisfied with the lack of essential features and tools: request validation, dependency injection, authentication, and automatic documentation generation. This lack of structure forced them to manually implement core HTTP functionalities, adding considerable boilerplate and maintenance burdens. As one respondent noted, without a standardized framework, they were "developing [their] own framework" to handle these basics, which led to duplicate or fragmented code. This experience aligns with the challenges observed in prior evaluations of *Project X* and *Project B* and underscores the necessity of requirements **R1-R4** outlined in section 4.3.3.

Following the migration using the adapted NestJS-based approach, developers reported significantly increased satisfaction with the development experience. The integration of dependency injection, modular code organization, and automatic OpenAPI documentation eliminated much of the previous boilerplate, enabling the team to focus on business logic rather than repetitive code. The survey responses consistently emphasize the benefits of these structural improvements, with developers finding the codebase more maintainable, cohesive, and easier to navigate. Satisfaction scores in this area improved sharply, moving from an average of 2/6 to 5/6, with developers highlighting the adapted mature framework as a "game changer" in reducing development friction and facilitating easier onboarding. The survey also provided insight into deployment experiences. While the migration did not directly change the underlying deployment process, the developers recognized the added flexibility in the potential for containerized deployment — a capability that had not yet been used. However, developers noted that specific Azure Functions configuration issues (e.g., resource scaling, cold starts) remain a concern. The developers highlighted the improvement in terms of being able to launch the application locally and therefore debug locally. While this was possible before the migration, it was made easier by being independent of the Azure Function.

While overall satisfaction improved, respondents highlighted specific challenges related to adapting the serverful-oriented NestJS framework to the serverless environment of Azure Functions. The need to configure request-scoped dependencies for handling Azure-specific context per request posed new architectural considerations and potential performance impacts. Additionally, cold starts were noted as an area of potential concern, though these had not yet been formally tested or shown to impact production performance.

Overall, developers indicate that their chosen approach was the right choice for them and their core principle of using a framework and adapting it to FaaS deployment targets was the more production-ready approach due to the aforementioned aspects. This result strongly endorses the recommended approach for improved development experience in hybrid deployment settings. Developers were able to refocus on business logic and minimize boilerplate coding, significantly raising developer satisfaction levels. While deployment processes were not drastically altered, the additional deployment options available through containerization suggest future flexibility in deployment targets.

However, one developer indicated that their old approach would not lose applicability for "building simple functionality", but for an application scaling in complexity, e.g., more and more sophisticated business logic, their new approach would make more sense.

In sum, this feedback underscores the viability of the *adapted serverful-first* approach as a more production-ready option, especially for teams handling increasingly complex applications. Further, it validates that the approach meets the development-oriented requirements **R1-R4** from section 4.3.3, increasing overall developer satisfaction drastically from an average 1.5/4 to 4/4. In the meantime, developers express that the structural gains achieved by integrating NestJS outweigh the initial adaptation efforts, supporting the recommended approach for teams seeking a hybrid deployment solution for serving applications. The requirements of **R5-R7** cannot be assessed by the survey, as the developers did not migrate to a non-FaaS deployment target. However, the survey results suggest that the *adapted serverful-first* approach is a good choice for the hybrid deployment of serving applications.

4.4.2 Overhead and Performance Benchmarks

As the interview with *Practitioner S* highlighted the importance of performance and scalability, an overhead and performance analysis was carried out.

Using the performance benchmark setup described in section 3.4, the Fibonacci and File services were deployed using the recommended *adapted serverful-first* approach with the Lambda Web Adapter and compared to a native Lambda function integration similar to listing 3 acting as a baseline. The GitHub repository associated with this work contains the code for both services in both integration variants[72].

The analysis focused on evaluating the overhead and performance of the recommended approach compared to a native Lambda function integration. Table F.1 and Table F.3 contain the summary results of the Fibonacci and File services benchmarks, respectively. Results are presented and discussed below:

Overhead Analysis with Short-Duration Requests — Fibonacci Service

In the **Constant Rate** scenario, both configurations maintained a constant request rate of 50 requests per second with no failed requests. The mean latency was similar (20.26ms for Lambda vs. 20.23ms for LWA, a change of -0.1%) and there were slight variations in the percentiles, in particular an increase of +3.1% and +2.6% at P50 and P95, but a decrease of -8.2% at P99. The maximum latency showed a large reduction of -616.8% with LWA, but the percentiles weaken the significance of this result.

Regarding the **Spike** scenario, both variants handled the bursty request rate with no failures either. Mean latency was slightly higher with LWA (+0.9%), while P50 and P95 latencies increased slightly (+2.4% and +1.3% respectively). P99 latency decreased by -6.7% and maximum latency was -26.0% lower with LWA.

In the **Constant VU** scenario, Lambda without LWA had a slightly higher number of successful requests. Mean latency increased by +2.8% with LWA, with small increases in P50 (+2.6%) and P95 (+2.7%). P99 remained almost the same (-0.2% with LWA), although the maximum latency was +92.9% higher with LWA.

Overall, the analysis highlights that while the recommended *adapted serverful-first* approach with the Lambda Web Adapter indeed introduces small latency overheads under certain load scenarios, these differences remain generally acceptable and not significant for client-facing applications. The LWA variant demonstrated greater stability under heavy load conditions, with fewer errors and more consistent request handling, even when latency increased slightly compared to native Lambda functions. In scenarios requiring consistent, high-volume throughput, native Lambda integrations may still offer slight performance advantages. However, the LWA variant effectively maintains comparable performance across varying load profiles, making it a viable and stable option.

Overhead Analysis with Long-Duration Requests — File Service

Considering the **Constant Rate** scenario, both configurations maintained a constant request rate of one request per second with no failed requests. The mean latency was slightly lower with LWA (8247.28ms for Lambda vs. 8134.69ms for LWA, a -1.4% change). The P50, P95, and

[72] `https://github.com/a-grasso/master-thesis-public/tree/main/evaluation-benchmarks`

P99 percentile latencies were also reduced with LWA (approximately -1.4% to -4.6%), while the maximum latency decreased by -9.0%.

Regarding **Constant VU**, both configurations completed the same number of requests, with a slight increase in mean latency for the LWA (+0.1%). P50 and P95 latencies were virtually identical, and P99 showed a -0.7% decrease with LWA. However, the maximum latency was significantly higher for the LWA variant (+26.5%).

In the **Spike** scenario, the LWA variant had slightly fewer successful requests (-3.8%) and more failures (+4.4%), i.e., hitting timeouts. Compared to the native Lambda configuration, the average request rate was slightly lower (-1.5%), while concurrent instances increased by +12.2%. Mean latency dropped by -3.2% with the LWA variant, and P50 latency showed a notable decrease (-16.0%). P95, P99, and max latencies however were nearly unchanged.

Overall, the analysis suggests that the LWA variant provides latency benefits over native Lambda configurations across multiple metrics, with notable improvements in average and high percentile latency values. Furthermore, the LWA variant demonstrated comparable throughput and stability under varying load profiles. Though under the **Spike** scenario, more notable degradation could be observed.

Summary of Overhead Analysis

The performance and overhead benchmark analysis for the recommended *adapted serverful-first* approach with the Lambda Web Adapter provides valuable insight into its suitability for different usage patterns in comparison to its native counterpart. For short-duration request (Fibonacci service) scenarios, the LWA variant demonstrated near parity in latency with native Lambda integration, maintaining similar throughput and low error rate under both constant and spike loads. For long-duration requests (File service), the LWA variant showed slight improvements in latency metrics, particularly in mean and percentile latencies, while maintaining comparable throughput and stability under more constant load profiles, though higher scaling needs and spikes showed more significant degradations. Overall, the results indicate that the *adapted serverful-first* approach, in this case using *out-of-code adaptation* with the Lambda Web Adapter, is on par with the performance of the native Lambda function, and that the approach, i.e., the adaptation, introduces no significant or negligible overhead in terms of performance and latency — invalidating concerns of *Practitioner S* in section B.5. It is therefore a viable option and validates the approach for serving applications in terms of performance requirements, delivering comparable performance and stability to native serverless functions across varying load profiles. Thus, regarding **NFR1-NFR3** listed in section 4.3.3, employing the recommended hybrid deployment approach, specifically the Lambda Web Adapter adaptation, has noticeable but negligible effects, but does not jeopardize further proceedings or contra-indicate the approach.

Cost Comparison and Analysis

To further evaluate the practicality of the recommended *adapted serverful-first* approach of the Lambda Web Adapter, a cost comparison was performed based on the benchmark scenarios. This comparison assesses the relative cost of the native AWS Lambda integration and the LWA variant, focusing on the impact of their architectural differences on the billing, and in particular sheds more light on the **NFR4** listed in section 4.3.3.

AWS Lambda costs are structured based on memory allocation, execution time, and request count, with charges (excluding tiered discounts[73]) as follows: $0.0000166667\frac{\$}{GB\cdot s}$ for memory usage and $0.0000002\frac{\$}{invocation}$ for request count. With this pricing, performance metrics such as execution time and request count have a direct impact on cost. Since both configurations have similar latency and throughput under different loads (see Table F.1 and Table F.3), these

[73]https://aws.amazon.com/de/blogs/compute/introducing-tiered-pricing-for-aws-lambda/

similarities in execution time lead to comparable pricing. However, certain features of LWA introduce potential cost increases. The LWA variant operates as a custom runtime, meaning that AWS charges for both the initialization and execution phases[74]. This design introduces additional costs for the LWA variant, which is particularly noticeable under load patterns with frequent cold starts or intermittent traffic, where the initialization time has a greater impact and therefore affects billing. This inherent increased billing overhead is due to the additional initialization steps required to start the application web server within the LWA adaptation. In high, consistent load scenarios, where cold starts are less frequent, the cost impact of the LWA variant is less pronounced. However, in low, intermittent load scenarios where cold starts are more frequent or per request, the cost impact of LWA is more significant.

Table F.2 summarizes the cost comparison between the native Lambda and LWA variants based on the **Constant Rate** scenario for the short request benchmarks. Notice the inherently higher initialization time of the LWA variant due to the custom runtime, namely $1709.63ms$ compared to $709.96ms$ for the native Lambda variant — an increase of +140.8%. Therefore, the total billed runtime for this benchmark scenario was $474.06GBs$ and $663.6GBs$ respectively[75] — a total increase of +39.9% for the LWA variant. This increase in billed time is directly related to the higher initialization time of the LWA variant, which is fully billed, compared to the native Lambda variant, where initialization time is not billed. Despite the more than doubled initialization time of the LWA variant, the largest part of the total cost remains the number of invocations, which is the same for both configurations, resulting in a total cost of $0.04390 for the native Lambda variant and $0.04706 for the LWA variant — "only" a total increase of +7.2%.

In summary, the cost comparison reveals a nuanced financial impact, driven primarily by architectural differences under the hood. While both configurations show similar execution performance metrics under varying loads as shown in section 4.4.2, the LWA variant incurs additional costs due to its custom runtime setup, specifically in terms of initialization time. This needs to be taken into account to avoid unexpected costs, but is specific to Lambda and the Lambda Web Adapter and may not apply to other serverless platforms or adaptation methods. In addition, the cost comparison is based on Lambda pricing and may vary for other cloud providers or serverless platforms.

Nevertheless, this analysis has shown a longer initialization time for the LWA variant due to the custom runtime, and apart from taking this into account for billing purposes as just discussed, the fact that the LWA variant has a longer initialization time can also be seen as a potential drawback in terms of latency. This would be particularly true for high-load scenarios with frequent cold starts, where the LWA variant may not be as suitable due to the increased initialization time and therefore increased latency.

While the results presented here are primarily specific to the Lambda Web Adapter and Lambda, the cost comparison and implications of the *adapted serverful-first* approach, particularly the details of the adaptation, in this case, the custom runtime and the LWA, can be seen as a general implication for other adaptation methods and serverless platforms as well — or as remarks for considerations and further evaluation of approaches.

4.4.3 Field Test: *Project K*

As mentioned in section 3.2, additionally, the recommended *adapted serverful-first* approach was evaluated in a field test with a practitioner on a real-world project, *Project K*.

Project K and its migration discussed in detail in Appendix D, was initially deployed on Kubernetes, with a total cost for the API services — including the Kubernetes cluster — of approximately €100 per month.

[74] AWS describes this here (docs.aws.amazon.com)
[75] Extracted manually from AWS CloudWatch

This expense, along with the fluctuating API usage patterns captured in the survey, drove the decision to migrate to a serverless platform, specifically AWS Lambda, to capitalize on the *assumed* cost-efficiency of pay-per-use pricing — further strengthening points made in section 4.2. The interviewee included a graph (see Figure D.1) showcasing typical usage over two days, from Sunday noon to Tuesday morning. It highlights periods of low or no traffic, interspersed with bursts of activity, typically during certain hours each day, which presented an opportunity to save costs by paying only for active usage. The traffic patterns illustrate that demand is concentrated at certain times, especially in the evenings. Given these non-uniform usage times, the switch to Lambda aimed to align the project's expenses with actual demand, avoiding the constant cost of a Kubernetes cluster for sporadic traffic.

Difficulties during Migration

The survey in Appendix D detailed several challenges faced by the interviewee during the migration from Kubernetes to Lambda. The switch was rated 5/10 for ease, as the application was already stateless, making the transition manageable. Adapting to the serverless environment, however, revealed critical areas requiring modification:

- Environment-specific variables and secrets had to be adapted for Lambda, as they differed from Kubernetes configurations.
- Internal dependencies tied to Kubernetes were removed, and non-stateless components like internal event handling and scheduling were decoupled.
- Managing Postgres connections posed issues, as serverless scaling could lead to unmanageable client connections, risking runtime failures.
- Cold starts in Lambda introduced latency concerns, as each cold start extended response time.
- Best practices for serverless environments had to be learned and applied, particularly around monitoring and observability.

Despite these roadblocks, most issues were either resolved or worked around. For instance, the Lambda Web Adapter proved straightforward to integrate with existing Docker images, and several strategies from the survey — such as refactoring certain components to run separately in Kubernetes — helped address compatibility issues. However, areas like connection limits and monitoring remain open challenges for future optimization.

Post-Migration

After migration, several aspects of development and operations saw improvements, though limitations remained.

The interviewee aimed to keep latency below one second, ideally around 200ms, and noted that the migration had a noticeable impact on latency. Most of the obstacles were either solved or worked around by splitting application components that couldn't be moved to a serverless environment, and running them on Kubernetes instead. This strategy was made possible by the existing modular design of the *Project K* codebase, which facilitated the separation of stateful and stateless components.

Cold starts in Lambda were reduced from around ten seconds to three seconds by optimizing the size of the container image and removing excessive dependencies. Despite these improvements, antipatterns such as keeping Lambda functions warm have been used to minimize cold starts.

Certain limitations persisted:

- The Postgres connection limitation remains a concern for large-scale load spikes, although the current load is manageable with a dedicated Postgres connection pool.
- Monitoring limitations continued, as tools like Prometheus[76] could not be used directly with Lambda. The reliance on AWS CloudWatch for logs brought potential vendor lock-in and limited observability for certain metrics.
- The deployment strategy had to be refined to align with best practices for a serverless environment, e.g., deviating from Kubernetes-based init container practices[77].

The post-migration cost analysis demonstrated a substantial reduction, bringing expenses down to around €10 per month — *a tenth of the previous cost.*

Conclusion

The interviewee states that the migration was successful overall and relatively smooth, with manageable adjustments to the serverless environment. While challenges arose in adapting the application to fit Lambda's constraints, the process validated serverless as a cost-effective option, thus the recommended hybrid deployment approach for **NFR4** — especially facing similar usage patterns as such presented in Appendix D and taking advantage of them. Also, it shows that the latency requirement **NFR2** was severely affected, although it was able to be kept within acceptable limits. **NFR1** and **NFR3** were neither jeopardized nor affected — the interviewee notes that no incidents occurred regarding these requirements. Discussed migration aspects align with the usage patterns observed in Figure D.1 regarding serverless potential for significant cost savings, and notions regarding the need for careful planning and consideration of constraints when migrating to serverless environments.

Regarding the Lambda Web Adapter, its applicability to adapt a *serverful-first* applications for serverless deployment targets was validated, as it allowed for the straightforward integration of existing web service container images into Lambda — aligning with **R1**-**R4** listed section 4.3.3. The survey also showed the potential for further optimization, as the interviewee noted that the LWA could be improved to for example provide more detailed monitoring and observability. This however is beyond the scope of this work, but would be possible, constituting open source contributions to the LWA.

All in all, the migration to a serverless deployment target and environment was successful, demonstrating the potential for cost savings by shifting deployment targets. The interviewee indicated being very happy with the migration and the cost savings achieved, despite technical challenges and learning opportunities that arose during or after the migration.

4.4.4 Benchmark Extensions: AWS Compute and Virtual Machine

The in subsection 4.4.2 presented overhead and performance analysis of the *adapted serverful-first* approach with the Lambda Web Adapter was extended to include additional deployment targets, namely AWS compute offerings such as AWS ECS based on Fargate and the AWS App Runner, as well as a traditional serverful deployment on a Hetzner VM.

Extending the benchmarks to these additional deployment targets evaluates worked-out influencing factors by providing further insights into the effects of the requirements influencing deployment target decision, as discussed in section 4.2. For instance, the benchmarks provide insights into analyzing when to choose which of the deployment targets based on usage patterns, scalability, concurrency, latency, throughput, and cost.

The benchmarks also serve to validate the adaptability of the recommended *adapted serverful-first* approach across the serverful-serverless spectrum, demonstrating its use across various

[76] https://prometheus.io/
[77] https://kubernetes.io/docs/concepts/workloads/pods/init-containers/

serverful and serverless deployment targets. This further facilitates hybrid deployment in support of the utilized requirement **R7** listed in section 4.3.3.

The results of the extended benchmarks contained in Table G.1 and Table G.2 for the Fibonacci and File services respectively are presented and discussed below:

Latency and Throughput Analysis: Short-Duration Requests — Fibonacci Service

Constant Rate: Fargate consistently outperformed the App Runner in terms of latency, with a mean latency of 12.61ms compared to the App Runner's 13.79ms. Fargate also demonstrated tighter latency bounds across percentiles, with a P99 latency of 16.86ms compared to App Runner's 21.88ms. The Hetzner VM, meanwhile, held steady with a mean latency of 14.73ms and maintained tight latency bounds across percentiles (P99 percentile at 17.45ms).

Compared to the Lambda benchmarks, the non-FaaS targets showed lower latencies and more stable performance across percentiles — with the Lambdas showing higher mean latencies of 20.26ms and 20.23ms for the native and LWA variants, respectively, and less stable latency stability across percentiles.

Spike: The Hetzner VM demonstrated the best performance with the lowest mean latency of 15.72ms. Though the P99 latency was drastically higher at 98.93ms (+529% compared to mean), the Hetzner VM maintained a comparatively very stable performance profile under the bursty load, with a maximum latency of 230.34ms. Fargate and the App Runner on the other hand showed higher mean latencies of 28.78ms (+83.1%) and 131.24ms (+734.9%), respectively, with Fargate demonstrating a more stable performance across percentiles than the App Runner did. It is noted, that the App Runner at a certain point could not handle the load and failed requests. Considering P50, the non-FaaS deployment targets outshine both Lambdas (18.31ms native and 18.75ms with LWA), with the lowest P50 latency of 11.47ms achieved by Fargate, 13.71ms by the Hetzner VM and 14.53ms by the App Runner.

The Lambdas followed closer compared to the Hetzner VM, with the LWA variant showing a mean latency of 19.33ms (+23.0%), while the native Lambda variant achieved 19.15ms (+21.8%). Though both Lambdas showed a more stable performance profile across percentiles overall than the other deployment targets — P99 percentile deviation of mean latencies of +75.7% for native and +63.0% for the LWA variant. While the P50 percentiles for the non-FaaS deployment targets were consistently lower than both Lambdas, the Lambdas showed better latency performance in higher percentiles.

Constant VU: Fargate demonstrated the best performance with a mean latency of 12.23ms, closely followed by the App Runner and the Hetzner VM with 13.59ms and 14.37ms, respectively. Mean latency performance in comparison to P95 for all three was within 15.0%, while the App Runner showcased a noticeable higher P99 latency in comparison: 19.78ms (+45.5%) vs 16.19ms (Fargate, +32.4% to mean) and 16.53ms (Hetzner VM, +15.0% to mean).

The Lambdas on the other hand reached mean latencies of 20.61ms for the LWA and 20.04ms native variant, while deviations from percentiles for both Lambdas were more noticeable — P99 percentile deviation of mean latencies of +55.3% for native and +50.8% for the LWA variant.

Latency and Throughput Analysis: Long-Duration Requests — File Service

Constant Rate: Most notably, the App Runner had 34 failed requests, while the other configurations had none. The Hetzner VM and Fargate both managed to achieve drastically lower latencies compared to the App Runner or both Lambdas: 2264.44ms and 2558.71ms; −259.2% and −217.9% lower mean latencies compared to the Lambda variant with LWA, respectively. The App Runner had the highest mean latency at 10 431.73ms, which is +28.2% higher than the Lambda with LWA. P50, P95, and P99 latencies were therefore generally the lowest for the Hetzner VM followed by Fargate, while the App Runner had the highest latencies overall. Other than the App Runner, all others did have stable performance across percentiles with lower deviation between mean and higher percentile latencies.

The drastic lower latency for Fargate and the Hetzner VM compared to both Lambdas could be explained in that Lambda instances were configured with one vCPU while the others with four vCPUs, which could have affected the performance. However, the App Runner was configured with four vCPUs as well, but still had the highest latencies and did not show the same performance as the other non-FaaS deployment targets. In an experiment with a similar resource configuration for both Lambdas, i.e., four vCPUs, the performance could not be reproduced either. No clear reason for this was found other than the differences in vCPUs, but it can be assumed that the underlying infrastructure did have its effect here in terms of performance. Additionally, tracing the File services during benchmarking could be validated that it must indeed be due to differences in compute and not due to network latency or the like.

Constant VU: Again, the drastic lower latencies for the Hetzner VM and Fargate compared to both Lambdas and the App Runner could be observed: −263.6% and −242.8% lower mean latencies compared to the Lambda variant with LWA, respectively. The Hetzner VM and Fargate completed more requests (767 and 724, respectively) than both Lambdas (213, both), with increases of +260.0% and +239.9%, respectively. The App Runner managed 237 requests (+11.3%, LWA Lambda variant) with overall lower latencies compared to both Lambdas. Performance across percentiles was stable across deployment targets; though the Lambda variant with LWA showcases a much higher max latency compared to the other deployment targets.

Spike: Notably, this scenario is not featured in Table G.2 besides for the Lambda variants. During the development and testing of this scenario, issues arose in terms of achieving comparable performance results that could not be resolved. Benchmark runs were not stable enough for comparison, and scaling of deployment targets could not be configured to solve this issue. Deployment targets and respective scaling configurations were in particular challenging to set up, especially with Fargate and the App Runner, to have comparable performance to Lambda and the Hetzner VM.

Conclusions of the Extended Benchmark Analysis — Latency and Throughput

First and foremost, the benchmarks support that the recommended hybrid deployment approach for serving applications, i.e., the recommended *adapted serverful-first* approach, can indeed be successfully deployed across the serverful-serverless spectrum, validating the approach in terms of **R5** and **R6**, and in particular **R7** listed in section 4.3.3.

In addition, the results of the extended benchmark provide insights into the selection of deployment targets based on the specific requirements discussed in section 4.2. They highlight and validate how characteristics of deployment targets — whether serverful or serverless — affect performance under different scenarios and loads, combined with how requirements influence how practitioners change deployment targets.

In terms of latency, the Hetzner VM and Fargate consistently delivered a lower latency with more stable performance compared to the App Runner and Lambda, particularly in constant load scenarios. For short-duration requests with steady and predictable usage patterns, such as the **Constant Rate** and **Constant VU**, serverful deployment targets such as the Hetzner VM and Fargate — the latter still part of the serverless spectrum — outperformed FaaS deployment targets, the epitome of serverless, by delivering a lower and more stable latency. Both Fargate and the Hetzner VM outperformed Lambda in terms of latency due to their ability to handle concurrent traffic without requiring scaling inherent in FaaS solutions in these scenarios. The inherent scaling of FaaS proved detrimental in terms of latency in these scenarios — as did the notion of cold starts affecting latency. For example, in constant rate scenarios, the Hetzner

VM and Fargate maintained lower mean latencies and tighter latency bounds than Lambda. This suggests that for applications with known, stable loads, non-FaaS deployment options offer better latency and throughput by eliminating the scaling lag that affects FaaS deployment targets in these conditions. It also suggests and combines that the usage pattern aspect discussed in subsection 4.2.3 influences the choice of deployment target during operation, in that the latency requirement discussed in subsection 4.2.1 during planning facing such usage patterns would indeed lead developers to move to non-FaaS deployment targets.

Conversely, when dealing with bursty, unpredictable traffic, as in the **Spike** scenario, the fast, on-demand scalability of FaaS becomes advantageous. Both Lambdas achieved a more stable performance profile, benefiting from the scaling capabilities of their platforms and proving their efficiency in maintaining more consistent latencies in the face of such usage patterns compared to their non-FaaS target counterparts. Fargate and the App Runner were not particularly well suited to keeping up with the demands of rapid scaling and therefore showed greater performance degradation. This suggests that for workloads with high concurrency instability and fluctuating load patterns, the more serverless nature of Lambda can more effectively support scalability requirements and provide the stability that serverful deployments may struggle to achieve without pre-scaling configurations in anticipation of surges in demand. It also reinforces the aspect of usage patterns influencing the suitability of deployment targets, also discussed above; where latency would have influenced the deployment target decision in favor of serverful, such scalability requirements would then reverse that decision. However, the Hetzner VM demonstrated exceptional resilience, maintaining low and stable latency while effectively handling surges, in contrast to the App Runner, which experienced degraded performance and even occasional failed requests under higher load. This particular aspect reinforces the above arguments that for usage patterns and loads where it is known that a deployment target can handle such loads without the need for scaling, serverful deployment targets will have the upper hand over their serverless counterparts, in this case, the Hetzner VM over Lambda, due to their superior performance in handling concurrent requests. It also reinforces that such a rationale would need to be present during planning, as discussed in subsection 4.2.1. It may also be influenced by expertise, whether practitioners are confident about the expected load, and therefore the expected usage patterns and assumptions to be made about them. However, it also reinforces and validates arguments that in anticipation of large volumes of traffic, serverful deployment targets would need to be pre-scaled — vertically or horizontally — which could then lead to potentially higher and wasted costs, as discussed in subsection 4.2.1.

In the **Spike** scenario for longer requests, achieving effective scaling configurations proved challenging. While the results were not stable enough for comparison, the difficulty of configuring scaling for Fargate or the App Runner to achieve such stable and comparable results suggests that serverful deployment targets may offer a simpler approach to dealing with sudden load spikes, but specifically on the premise that they are already capable enough to handle such spikes without the need to scale in the first place — avoiding these issues. Furthermore, this observation also underlines the serverless experience and principles discussed in section 4.1. It reinforces the points made about the challenges of scaling and the need for practitioners to think about and configure scaling configurations to achieve stable performance under such conditions — the very complexity that serverless aims to abstract, reinforcing its conceptual appeal in managing these aspects automatically.

Overall, these benchmarks confirm the importance of selecting deployment targets based on workload characteristics and usage patterns. Serverful deployment targets such as the Hetzner VM and Fargate, although the latter is still part of the serverless spectrum, were ideal for stable, predictable loads, providing better performance without the need for frequent scaling. However, for applications with more volatile usage patterns, FaaS deployment targets can meet these demands more dynamically through their inherent scaling capabilities. Furthermore, these benchmarks validate the benefits of the recommended *adapted serverful-first* hybrid deployment approach, which was able to leverage both serverful and serverless deployment targets, facilitating the shift of deployment targets in the face of steady or predictable workloads, or highlighting their adaptability to be deployed on FaaS for rapidly scaling scenarios. It therefore validates the hybrid deployment strategy that can adapt to workload requirements and facilitate the shift of deployment targets in favor of performance or cost in different operational scenarios — thus in line with the **NFR1-NFR4** listed in section 4.3.3.

Cost Comparisons

As also discussed in subsection 4.4.2, a cost comparison serves to provide a more comprehensive validation of the cost factor examined in section 4.2. Table G.3 presents the cost comparison calculations for the extended benchmarks of both services under the **Constant Rate** scenario.

The Fibonacci service shows a wide range of costs across the different deployment targets. With a minimum cost of \$0.04390 per hour for the native Lambda and the highest cost of \$0.07800 per hour for the App Runner, there is a clear variation in economics between deployment targets. Fargate, although more expensive than Lambda, is positioned as an intermediate option at \$0.04937 per hour. Notably, the Hetzner VM is competitively priced at \$0.04660, rivaling Lambda. Indeed, at first glance, Lambda appears to be the more cost-effective option for the Fibonacci service, offering one of the lowest costs per hour. However, the results also highlight the potential for cost savings with the Hetzner VM or comparable costs with Fargate, both of which would offer better performance than Lambda, as shown in section 4.4.4. Looking at the CPU usage of the non-FaaS targets, it is clear that all three were significantly underutilized for this scenario. The Hetzner VM, for example, had a CPU usage of less than 15%; from this example, cost extrapolations can be made.

While the native Lambda could handle 50 requests per second at a cost of \$0.04390 per hour, the Hetzner VM, due to its underutilization, could have handled up to six times as many requests, i.e., up to 300 requests per second, at the same cost of \$0.04660 per hour. In comparison, Lambda, due to its pay-per-use model, would have incurred much higher costs to handle the same load of 300 requests per second: $\$0.04390 \times 6 = \$0.2634$. While this is an oversimplification, it would suggest that for such workloads, the Hetzner VM would be the more cost-effective option, given its underutilization and the potential to handle more requests for the same cost — the same could be said for Fargate or the Hetzner VM. However, as this assumes a constant load, it ignores Lambda's scale-to-zero capability. So where Lambda would catch up in terms of cost efficiency is if the load were to fluctuate and have periods of no load as given in *Project K* in subsection 4.4.3 for example, where the non-FaaS deployment targets would remain at the same cost, while Lambda would scale down to zero incurring no further cost.

The File service, on the other hand, incurs higher costs due to its higher computational and concurrent instance requirements. The native Lambda is priced at \$0.84125 per hour, while Lambda with the LWA has a slightly lower cost of \$0.83035 per hour. The App Runner has the highest cost at \$1.87200 per hour. Fargate and the Hetzner VM are priced at \$0.19748 and \$0.04660 per hour respectively, due to their exceptional performance as mentioned in section 4.4.4. As this performance discrepancy could not be fully explained, cost comparisons are skewed. Nevertheless, the Hetzner VM and Fargate would outperform both Lambdas and the App Runner by a wide margin in terms of cost and performance, with the Hetzner VM being the most cost-effective option for this scenario. Nonetheless, the results suggest that both Lambdas offer a more cost-effective solution compared to the App Runner. Again, usage patterns would need to be taken into account, as the scale-to-zero capability of the lambda would again be a factor in cost efficiency, as discussed above.

In conclusion, the cost comparisons for the extended benchmarks confirm the significant cost differences between deployment targets, validating discussions about the cost requirement in section 4.2 and how cost efficiency is influenced in particular by usage patterns. The benchmarks show that the cost of deployment targets can vary significantly, and that cost efficiency depends on the deployment target itself and how it is billed, while also being influenced by usage patterns that would indicate the most cost-effective deployment target. Depending on the expected load, the scale-to-zero capability of serverless deployment targets may make them the more cost-effective option, as they would scale to zero and incur no further costs during off-hours. On the other hand, the more continuous or constant the traffic becomes, the more cost-effective serverful deployment targets are. It is also shown that scalability is detrimental to cost, as the inherent scaling to achieve concurrent requests for FaaS affects cost efficiency, whereas the non-FaaS targets could handle concurrent loads to a certain point without having to scale, making them more cost-effective for such workloads. For example, by staying below the scaling limits, a Hetzner VM instance will be more cost-effective in terms of throughput than a Lambda instance because it can handle more requests for the same cost. However, when faced with traffic that requires greater scalability, such as a fluctuating or bursty load, this cost-effectiveness is reversed and the scalability of serverless deployment targets would make them the more cost-effective option.

Chapter 5

Conclusion

5.1 Summary and Contributions

Chapter 1 introduces the diverse deployment landscape for applications, with a wide variety of deployment targets, ranging from traditional on-premises servers to serverless functions in the cloud. The choice of deployment target can have a significant impact on the development and operations process. Navigating this landscape can be challenging for practitioners, as each deployment target has different requirements, constraints, and trade-offs (Yussupov, Soldani, et al. 2021). However, the choice of deployment target is not always clear-cut and may change over time. Practitioners may need to change deployment targets due to changing requirements, cost considerations, or other factors. This can be challenging and costly, as different deployment targets have different requirements, implications, and constraints, and if not facilitated in advance, can lead to a significant amount of rework (Baldini et al. 2017; Eivy et al. 2017).

As such, this work focused on enabling practitioners to *decide about deployment later* to promote flexibility and adaptability and to provide insight into the factors that influence the choice of deployment target for applications. The following research questions were therefore addressed:

RQ1 What are the key factors, and how do they influence the choice of deployment target for cloud-native applications?

RQ2 How can application components be enabled to shift within the spectrum of serverful and serverless deployment targets?

Using the methodology described in chapter 3, answers to the research questions were explored through a literature review, interviews and surveys, technical research and experimentation, and evaluated through interviews, a field test and performance benchmarks. The main findings and contributions of this work are as follows:

- A streamlined definition and broader view of serverless that encompasses the spectrum of serverful and serverless deployment targets
- Synthesis of why practitioners choose a deployment target and what constitutes a shift from it, distilling the key factors that influence the deployment target decision
- Requirements for and insights into hybrid deployment approaches
- Two recommended approaches towards hybrid deployment, covering serving and processing applications
- Insights into the viability of the recommended approaches, including performance benchmarks and qualitative evaluation

Serverless, previously commonly associated with FaaS (Baldini et al. 2017, p. 18; Jonas et al. 2019, p. 4; Hassan et al. 2021, p. 2), is now defined by three principles: **SP1)** *costs nothing when not in use*; **SP2)** *no responsibility for infrastructure management* and **SP3)** *no responsibility for scaling configuration.* This allows the term serverless to be applied to a wider range of deployment targets, including traditional on-premises servers, virtual machines, containers, and serverless functions for applications and components. The serverless experience can be achieved on any of these deployment targets as long as the three principles are met, allowing the deployment target to be considered part of the serverless spectrum. Therefore, the serverless experience itself can be present regardless of the deployment target. Section 4.1 condenses the above to:

> In terms of application components or deployment targets, where
> **(i)** the practitioner has no responsibility for managing the infrastructure or scaling configurations, and where
> **(ii)** costs are incurred if and only if the component is in use,
> such a component or deployment target is considered to be serverless;
> and in cases where these criteria are not fully satisfied, it can still be considered as part of the serverless spectrum.

The key factors influencing the choice of deployment target for cloud-native applications, distilled in section 4.2 from the literature review, interviews, and surveys, characterize the decision-making process of practitioners: how they choose a deployment target and what constitutes a shift from it. While the factors identified in the planning process are diverse, they have been grouped into four categories: *non-functional requirements* alongside *top-down constraints*, *know-how* and *use cases, workloads & architecture* (subsection 4.2.1). Non-functional requirements include *ownership & control*, *vendor lock-in & independence*, *scalability & elasticity*, *cost*, *latency* and *time-to-market.*

Assumptions made at the planning stage about the above factors may change over time, leading to increasing operational costs or poor development experience. Shifting the deployment target could mitigate such issues, but can be challenging and costly, as different deployment targets have different requirements, implications, and constraints. If practitioners spend more time fighting the deployment target than developing the application, then rethinking and shifting the deployment target during development will be necessary (subsection 4.2.2). Depending on the usage patterns and characteristics of the actual load during operation, requirements such as *cost*, *latency* or *scalability & elasticity* constitute a shift of the deployment target (subsection 4.2.3).

Based on the optimal experience and requirements identified in subsection 4.3.3, existing solutions such as Infrastructure from Code, serverless function frameworks, service description, and code generation, and runtime substitution sidecars were evaluated and found to have potential. However, they were not promising enough to be fully recommended for hybrid deployment or the basis for further work (subsection 4.3.4).

Therefore, two recommended approaches for hybrid deployment have been developed: *Adapted serverful-first* builds on mature and established frameworks that address the issues faced when developing serving applications with FaaS, while adapting them for FaaS deployment targets to unlock the full breadth of the serverful-serverless spectrum — with either *in-code* or *out-of-code adaptation* (subsection 4.3.5). However, given the heterogeneity of processing applications across deployment targets, *portable serverless-first* instead focuses on guidelines and principles for processing applications to achieve the same result: enabling processing logic to be shifted between serverful and serverless deployment targets (subsection 4.3.6). The guidelines provide a structured approach to keeping processing logic portable across deployment targets by separating the processing logic from the deployment target-specific integration logic. Both recommended approaches enable practitioners to *decide about deployment later* — either during development or even during operation — promoting flexibility and adaptability in navigating the deployment landscape.

The practicality of the recommended approaches was evaluated using expert interviews and surveys (subsection 4.4.1), overhead and performance benchmarks (subsection 4.4.2 and subsection 4.4.4), and a field test (subsection 4.4.3). Focusing on aspects such as *cost*, *scalability & elasticity* and *latency*, among those listed in subsection 4.3.3, the evaluation confirmed that the recommended approaches can be applied in practice without significant overhead while providing the flexibility and adaptability needed to navigate the landscape between serverful and serverless deployment targets. Conducting further benchmarks (subsection 4.4.4) with additional AWS compute services and a rented virtual machine further validated the previously identified factors and how they influence the choice of deployment target — particularly in the context of usage patterns and characteristics of expected traffic.

5.2 Discussion

Practitioners in subsection 4.4.1 emphasized that the *adapted serverful-first* approach significantly improved the development experience by using mature frameworks, which helped to overcome the limitations of FaaS for more complex applications. The *portable serverless-first* approach for processing applications was praised for its flexibility in migrating workloads to containerized environments, which improved control, performance, and cost efficiency for the particular success story. The feedback confirms that the recommended approaches provide practical solutions for maintaining flexibility in deployment targets, but also highlights areas such as latency and scaling complexity that warrant further evaluation in specific use cases.

In subsection 4.4.3, these findings were further validated through a field test with *Project K*, a real-world project initially deployed on Kubernetes with high fixed costs and fluctuating API demand. Migrating *Project K* to AWS Lambda resulted in significant cost savings — reducing expenses from approximately €100 to now approximately €10 per month (−90%) — by more closely aligning costs with actual usage patterns. While the migration did present challenges, such as limited database connections affecting scalability or cold-start latency, most issues were mitigated. Overall, the migration confirmed that the recommended adapted serverful-first approach effectively balances cost efficiency with operational flexibility, supporting the practical benefits of the approach for projects with variable demand to navigate the deployment landscape as needed.

The results in subsection 4.4.2 further reinforced the viability of the recommended *adapted serverful-first* approach and allayed the concerns of practitioners in interviews about cold-start latency and scalability compared to native FaaS performance. While some expected overhead was observed, it was minimal and did not outweigh the practical benefits of the approach. Benchmarking showed that the approach introduced little latency compared to native serverless options, with scalability remaining largely unaffected. Overall, the evaluation confirmed that this approach is feasible in practice and provides capabilities to navigate the spectrum between serverful and serverless deployments without significant performance trade-offs.

The recommended hybrid deployment approaches for serving and processing applications offer flexibility and adaptability, but come with notable limitations and contraindications, as discussed in section 4.3.5 and section 4.3.6. The *adapted serverful-first* approach requires applications to comply with both serverless and serverful constraints, such as statelessness, as persistent storage or extended background tasks can lead to complete failure in serverless environments. Supporting multiple FaaS targets can introduce complex conditional logic, which impacts maintainability. In addition, the performance overhead of adaptation can reduce suitability for strictly latency-sensitive applications. The *portable serverless-first* approach provides portability but places great responsibility on developers to follow the guidelines carefully, as deviations may reduce portability and efficiency. Applications may rely on cloud-managed services for critical tasks, which can complicate transitions to serverful deployment targets without equivalent support. For smaller applications, the complexity introduced by these guidelines may outweigh their benefits, as simpler solutions may suffice. While both approaches allow for flexible deployments, they require careful consideration of application needs, as over-complication or misuse may hinder rather than help.

The evaluation in subsection 4.4.4 further validated the factors influencing the choice of deployment target, particularly in the context of usage patterns and expected traffic characteristics. The extended benchmarks confirmed that the choice of deployment target has a significant impact on cost, scalability, and latency, with serverless deployment targets offering scalability benefits and cost savings for spiky, bursty workloads or when facing unpredictable traffic patterns with long idle times. However, serverful deployment targets offer more predictable costs and lower latency for constant, continuous, or predictable workloads. The results highlight the importance of considering these factors when choosing a deployment target, as they can have a significant impact on application performance and operational costs. These findings are in accordance with comparable tests conducted by (Fan et al. 2020).

Answers to the Research Questions

The first research question, *"what are the key factors and how do they influence the choice of deployment target for cloud-native applications?"* is answered in section 4.2 and evaluated through interviews and benchmarks in subsection 4.4.1 and subsection 4.4.4 respectively. Practitioners are helped to detail the factors that influence the choice of deployment target and when to look out for them:

Discussed top-down constraints influence the decision more directly, as they may require or deny a particular deployment target, without giving practitioners any leeway to shift the deployment target.

Know-how also plays a role in directly influencing the decision process, as one may not have the necessary skills or knowledge to operate a specific deployment target beforehand. However, know-how also plays a role in indirectly influencing all the other factors and therefore has a significant impact on the development and operational process later on.

Use cases, workloads & architecture facilitate the decision in one or the other direction of the spectrum, as they may be more suitable for serverful or serverless deployment targets, depending on the specifics. Microservice architecture or specific use cases do not consequent a final decision, as both extremes could facilitate. However, event-driven architectures are more suited to serverless, while the need for stateful management is more suited to serverful deployment targets. The type of workload can significantly influence the choice of deployment target, as spiky, inconsistent workloads are more suited to serverless, while more constant and predictable workloads are more suited to serverful deployment targets.

In terms of non-functional requirements, *ownership & control* or *vendor lock-in & independence* make practitioners tend towards more serverful deployment targets, while *scalability & elasticity*, *cost* or *time-to-market* are reasons for more serverless deployment targets. In particular, influencing factors such as *cost*, *scalability & elasticity* or *latency* are susceptible to the above-detailed usage patterns and type of workload traffic.

The second research question, *"how can application components be enabled to shift within the spectrum of serverless and serverful deployment targets?"* is answered by exploring existing solutions and approaches in subsection 4.3.4 and the recommended approaches in subsection 4.3.5 and subsection 4.3.6 for serving and processing applications respectively. Both recommended approaches enable practitioners to *decide about deployment later* — either during development or even during operations — promoting flexibility and adaptability in navigating the deployment landscape.

With adaptation, the *adapted serverful-first* approach for serving applications enables practitioners to develop applications with mature serverful-oriented frameworks and then adapt them for serverless deployment targets, unlocking the full breadth of the serverful-serverless spectrum.

By prioritizing portability in development, the *portable serverless-first* approach for processing applications provides guidelines and principles for maintaining the portability of processing logic across different deployment targets. This is achieved by decoupling the processing logic from the integration logic specific to each deployment target.
Both of these recommended approaches are the preferred solutions, although there are alternatives such as infrastructure from code or service description and code generation. However, these approaches are not currently *yet* recommended. The Lambda runtime substitute sidecar approach is recommended as a fallback solution specific to AWS Lambda, while serverless function frameworks can be used for either serving or processing applications for a limited range of use cases, trading flexibility and features for broader applicability.

Implications and Consequences

By answering research questions **RQ1** and **RQ2**, this work provides practitioners with actionable strategies for achieving flexibility in deployment target decisions, enabling them to *decide about deployment later* and dynamically navigate the spectrum between serverful and serverless deployment targets. By addressing the core challenges of shifting deployment targets, this work lays the foundation for adaptive deployment strategies that more closely align with evolving application deployment needs, usage patterns, and cost considerations.

The recommended hybrid deployment approaches — *adapted serverful-first* for serving applications and *portable serverless-first* for processing applications — provide practitioners with practical options for deployment flexibility. These approaches allow applications to start with the simplicity and cost-effectiveness of serverless during early development, and as traffic or complexity increases, seamlessly move to more serverful deployment targets when greater control or optimization is required.

As shown, the flexibility achieved through these approaches can lead to significant cost savings without sacrificing performance. By adapting mature frameworks or decoupling processing logic from integration, practitioners can maintain a high degree of adaptability, enabling application components to better respond to changing operational requirements.

However, adopting these approaches requires careful planning. Practitioners must weigh the implications of managing potentially complex adaptation layers and adhering to best practices, as these can introduce significant overheads in terms of latency or deployment complexity. The guidelines provided in this research help to mitigate these challenges, but successful implementation depends on a thorough understanding of the trade-offs inherent in the chosen deployment path.

5.3 Limitations

The answers to the research questions and the recommended approaches for hybrid deployment have limitations.

Section 1.1 outlines the focus of this work and defines the scope of the answers to the research questions. The primary focus has been on compute, i.e., business logic application code. The serverless definition, experience, and principles can also be evaluated for storage or networking. However, the answers to **RQ1** and **RQ2** explored are tied to compute components and their options for deployment targets across the serverful-serverless spectrum. Storage and networking components were excluded and could therefore be revisited in further research and evaluation regarding hybrid deployment approaches and whether or how such would be possible. For example, infrastructure related to message queues may remain inaccessible within the cloud when processing logic is moved to serverful deployment targets within a private cloud — how to deal with such scenarios in terms of hybrid deployment approaches was beyond the scope of this work.

Furthermore, the recommended *adapted serverful-first* approach is limited to HTTP-based serving applications. Other protocols, such as gRPC[1] for synchronous request-response communication, are not addressed in this approach. While HTTP remains the most widely used protocol for serving applications (FreeWheel Biz-UI Team 2024, pp. 37, 38), exploring methods to extend support for other protocols could increase flexibility and broaden the applicability of hybrid deployments for serving applications.

In addition, evaluation in real-world scenarios is limited to a single real-world project, and lessons learned may not be applicable to other projects. Further real-world evaluation is required to validate the practicality and effectiveness of the recommended approaches across a wider range of use cases and deployment scenarios. Also, the real-world scenario covered in *Project K* represented a shift from a serverful deployment target to a serverless deployment target, the other way around could not be covered, so further evaluation in real-world scenarios is needed to assess the generalizability and effectiveness of the recommended approaches.

Furthermore, the evaluation of the recommended *portable serverless-first* approach to application processing is limited to a single success story. Furthermore, the interview itself represented a retrospective view of the migration, the migration itself was not followed and tracked within this work, potentially missing out on more in-depth insights and evaluation. As a result, the evaluation of the approach is limited to the insights gained from the interview and the success story, and further qualitative evaluation is needed to assess whether the recommended guidelines apply to a wide range of processing applications and are effective in maintaining portability across different deployment targets.

5.4 Open Points for Further Work

This work presents hybrid deployment approaches and actionable insights to improve decision-making about deployment targets for cloud-native applications. However, several areas merit further exploration to deepen and extend these findings. Addressing these points may increase the flexibility and general applicability of the proposed approaches, or provide new insights into the challenges and opportunities of hybrid deployment strategies. The following points outline potential directions for further research and development:

Automated tools and decision frameworks could be a promising area of research to assist practitioners in selecting and employing deployment targets. These tools could evaluate related application metrics (e.g., traffic or usage patterns, latency or cost requirements), suggest optimal deployment configurations within the serverful-serverless spectrum, and carry out the shift to the optimal deployment target. Automation, especially when integrated with CI/CD pipelines, could reduce the complexity of making dynamic deployment decisions and align deployments more closely with real-time usage patterns.

Integrate the interface for deployment targets into subsection 4.1.7 through a framework, allowing practitioners to deploy to different deployment targets well within the serverless spectrum using a standardized framework. This could also facilitate the automatic switching between deployment targets discussed earlier.

[1] https://grpc.io/

Assessing the cost implications of self-hosted OpenFaaS within organizations could be essential to understanding its impact on individual project costs and broader organizational budgets — further deepening the cost requirements and implications for hybrid deployments and decision-making strategies for deployment targets. Section 4.1.6 discusses the issue of self-hosted OpenFaaS in relation to the serverless definition and evaluates its principles for it. Implementing OpenFaaS in a self-hosted environment though costly (FreeWheel Biz-UI Team 2024, p. 192), may offer potential cost savings compared to managed serverless solutions, particularly for organizations with high workloads and predictable demand. However, these savings must be balanced against infrastructure and maintenance costs, as self-hosting requires dedicated resources for setup, management, and ongoing operational support. Further work could assess the economics and sustainability of self-hosted OpenFaaS in different organizational contexts, providing deeper insights into its cost implications, particularly for large-scale or resource-intensive applications, also in comparison to FaaS offerings of public cloud providers.

Achieving a serverless experience on predictably priced, in-house hardware can represent a transformative paradigm shift for organizations seeking the scalability and simplicity of serverless computing without the variable costs associated with cloud-based solutions. By leveraging owned or on-premises hardware, organizations can achieve predictable pricing structures that would facilitate budget planning, while still benefiting from serverless-like deployment, management, and scaling. This approach would enable tailored resource management, where workloads can be optimized to take advantage of fixed hardware investments and reduce dependence on external providers. However, achieving this balance may require effective orchestration tools and in-house expertise to replicate the serverless experience in a self-hosted environment, presenting both opportunities for cost savings and challenges in building the infrastructure. This may be related to the self-hosted OpenFaaS research discussed earlier and could be a potential area for further exploration towards this open point.

Extending hybrid deployment to other cloud components could provide valuable insights. While this work primarily addresses compute resources, further research could explore the applicability of serverless principles to storage and networking components. In particular, exploring how storage or networking components could adhere to serverless principles (such as scalability, cost efficiency, and managed services) could lead to more holistic hybrid deployment strategies. For example, message queues or databases that remain exclusively in the cloud could complicate a move to private cloud environments. Research into approaches to dealing with such scenarios — particularly in terms of data integrity, performance, and access — would complement the recommended hybrid deployment approaches presented in this work.

Implementing hybrid deployment interfacing with external systems and infrastructure components, could counter hybrid deployment issues when moving from fully cloud-hosted deployment targets to private or serverful deployment targets. Related to the previous point, hybrid deployment with proprietary infrastructure components, such as message queues or other event-driven components, could be a significant challenge when moving between serverful and serverless deployment targets. A different approach might be to use a consistent interface layer, such as DAPR, also mentioned in section 4.3.4, which abstracts the interfacing with external systems. Consistent interface layers could provide a solution by abstracting communication and state management, allowing applications to remain agnostic to the underlying infrastructure. Such approaches need to be evaluated and tested in real-world scenarios to assess their effectiveness and practicality for hybrid deployments. In addition, for platform services beyond compute, such as messaging and data storage, implementing such abstractions ensures that consumer applications can operate independently of specific cloud services, providing deployment flexibility while maintaining essential functionality.

Diversity in application delivery protocols, beyond HTTP, could be further explored. The recommended *adapted serverful-first* approach is based on HTTP-based applications, leaving the potential for further work to extend hybrid deployment strategies to other protocols, such as gRPC. An adaptation approach that supports these communication standards could greatly enhance the flexibility of application deployment, and exploring the potential of this could broaden the applicability of application deployment across the serverful-serverless spectrum.

Broader real-world evaluations and use cases could help validate the proposed approaches. As discussed in section 5.3, the real-world application tested, *Project K*, was used to validate the transition from a serverful to a serverless deployment target, but additional case studies could provide broader insights. In particular, observing shifts in the opposite direction — from serverless to serverful — would help to demonstrate whether the approaches presented here are equally viable in different operational scenarios. Furthermore, exploring additional domains beyond web-based or API-driven applications would clarify the broader applicability and limitations of hybrid deployments, helping to refine these approaches for a wider range of industries and application types.

A broader evaluation of the *portable serverless first* approach would be beneficial. The *portable serverless-first* approach to application processing and its evaluation is limited. Future studies could conduct a broader evaluation across different industries, application types, and deployment targets to assess whether these guidelines are broadly effective in ensuring portability and flexibility in deployment.

Longitudinal analysis of cost and performance would provide valuable insights into hybrid deployments over time. This work provides insights into cost, scalability, and performance considerations based on initial benchmarks and field test observations. However, further research could include longitudinal studies to assess how these factors evolve, particularly as application requirements or operational complexity increase. For example, observing how costs and latency change over time as applications scale could provide valuable data for organizations planning long-term hybrid deployments. In addition, this could provide further insight into the threshold at which a shift between serverful and serverless becomes most cost or performance-effective, based on more mature, stable, and real-world data.

Extending the shortcomings of existing approaches in subsection 4.3.4 to promote maturity for hybrid deployment. Such further work would extend the options for hybrid deployment beyond the two recommended approaches, and practitioners would gain more flexibility and adaptability in development and deployment. For example, OpenAPI could be extended with integration logic generators to facilitate the generation of integration logic for different deployment targets based on the common service description. Infrastructure from Code could be extended to enable hybrid deployment and facilitate a wide range of deployment targets across the serverful-serverless spectrum.

5.5 Concluding Remarks

This work has presented and discussed the factors that influence practitioners in their choice of deployment target for cloud-native applications, and how applications are enabled to move within the spectrum of serverful and serverless deployment targets. The methods described in chapter 3 were used to explore the answers to the research questions and to evaluate the recommended approaches. The evaluation shows that the recommended approaches can be applied in practice and provide the flexibility and adaptability needed to navigate the landscape between serverful and serverless, albeit with limitations and constraints. The work provides a foundation for future research and valuable insights for practitioners, demonstrating that the chosen approaches and methods effectively addressed the defined research questions within the specified scope.

List of Listings

List of Figures

List of Tables

Bibliography

Andrikopoulos, Vasilios et al. (2014). "CloudDSF – The Cloud Decision Support Framework for Application Migration". In: *Service-Oriented and Cloud Computing*. Ed. by Massimo Villari, Wolf Zimmermann, and Kung-Kiu Lau. Berlin, Heidelberg: Springer, pp. 1–16. ISBN: 978-3-662-44879-3. DOI: 10.1007/978-3-662-44879-3_1.

Aviv, Itzhak et al. (Jan. 2023). "Infrastructure From Code: The Next Generation of Cloud Lifecycle Automation". In: *IEEE Software* 40.1, pp. 42–49. ISSN: 1937-4194. DOI: 10.1109/MS.2022.3209958. URL: https://ieeexplore.ieee.org/abstract/document/9908145 (visited on 10/15/2024).

Baldini, Ioana et al. (2017). "Serverless Computing: Current Trends and Open Problems". In: *Research Advances in Cloud Computing*. Ed. by Sanjay Chaudhary, Gaurav Somani, and Rajkumar Buyya. Singapore: Springer, pp. 1–20. ISBN: 978-981-10-5026-8. DOI: 10.1007/978-981-10-5026-8_1. URL: https://doi.org/10.1007/978-981-10-5026-8_1 (visited on 05/07/2024).

Bermbach, David, Erik Wittern, and Stefan Tai (2017). *Cloud Service Benchmarking*. Cham: Springer International Publishing. ISBN: 978-3-319-55482-2 978-3-319-55483-9. DOI: 10.1007/978-3-319-55483-9. URL: http://link.springer.com/10.1007/978-3-319-55483-9 (visited on 08/27/2024).

Castro, Paul, Vatche Isahagian, et al. (Sept. 14, 2022). *Hybrid Serverless Computing: Opportunities and Challenges*. arXiv: 2208.04213 [cs]. URL: http://arxiv.org/abs/2208.04213 (visited on 04/16/2024). Pre-published.

Castro, Paul, Vatche Ishakian, et al. (Nov. 21, 2019). "The Rise of Serverless Computing". In: *Communications of the ACM* 62.12, pp. 44–54. ISSN: 0001-0782. DOI: 10.1145/3368454. URL: https://dl.acm.org/doi/10.1145/3368454 (visited on 05/22/2024).

Cooper, Harris, Larry V. Hedges, and Jeffrey C. Valentine, red. (2009). *The Handbook of Research Synthesis and Meta-Analysis, 2nd Ed.* The Handbook of Research Synthesis and Meta-Analysis, 2nd Ed. New York, NY, US: Russell Sage Foundation, pp. xvi, 615. xvi, 615. ISBN: 978-0-87154-163-5.

Deng, Shuiguang et al. (June 25, 2023). *Cloud-Native Computing: A Survey from the Perspective of Services*. DOI: 10.48550/arXiv.2306.14402. arXiv: 2306.14402 [cs]. URL: http://arxiv.org/abs/2306.14402 (visited on 09/10/2024). Pre-published.

Di Lauro, Fabio (2024). "GitHub-Sourced Web API Evolution: A Large-Scale OpenAPI Dataset". In: *Web Engineering*. Ed. by Kostas Stefanidis et al. Cham: Springer Nature Switzerland, pp. 360–368. ISBN: 978-3-031-62362-2. DOI: 10.1007/978-3-031-62362-2_26.

Eismann, Simon et al. (Jan. 28, 2021a). *A Review of Serverless Use Cases and Their Characteristics*. DOI: 10.48550/arXiv.2008.11110. arXiv: 2008.11110 [cs]. URL: http://arxiv.org/abs/2008.11110 (visited on 06/17/2024). Pre-published.

— (Jan. 2021b). "Serverless Applications: Why, When, and How?" In: *IEEE Software* 38.1, pp. 32–39. ISSN: 1937-4194. DOI: 10.1109/MS.2020.3023302. URL: https://ieeexplore-ieee-org.ezproxy.bib.hm.edu/document/9190031 (visited on 06/17/2024).

Eivy, Adam and Joe Weinman (Mar. 2017). "Be Wary of the Economics of "Serverless" Cloud Computing". In: *IEEE Cloud Computing* 4.2, pp. 6–12. ISSN: 2325-6095, 2372-2568. DOI:

10.1109/MCC.2017.32. URL: https://ieeexplore.ieee.org/document/7912239/ (visited on 06/18/2024).

Fan, Chen-Fu, Anshul Jindal, and Michael Gerndt (2020). "Microservices vs Serverless: A Performance Comparison on a Cloud-native Web Application:" in: *Proceedings of the 10th International Conference on Cloud Computing and Services Science*. 10th International Conference on Cloud Computing and Services Science. Prague, Czech Republic: SCITEPRESS - Science and Technology Publications, pp. 204–215. ISBN: 978-989-758-424-4. DOI: 10.5220/0009792702040215. URL: http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0009792702040215 (visited on 05/21/2024).

FreeWheel Biz-UI Team (2024). *Cloud-Native Application Architecture: Microservice Development Best Practice*. Singapore: Springer Nature. ISBN: 978-981-19978-1-5 978-981-19978-2-2. DOI: 10.1007/978-981-19-9782-2. URL: https://link.springer.com/10.1007/978-981-19-9782-2 (visited on 05/11/2024).

Gatev, Radoslav (2021). *Introducing Distributed Application Runtime (Dapr): Simplifying Microservices Applications Development Through Proven and Reusable Patterns and Practices*. Berkeley, CA: Apress. ISBN: 978-1-4842-6997-8 978-1-4842-6998-5. DOI: 10.1007/978-1-4842-6998-5. URL: https://link.springer.com/10.1007/978-1-4842-6998-5 (visited on 11/05/2024).

Hassan, Hassan B., Saman A. Barakat, and Qusay I. Sarhan (July 12, 2021). "Survey on Serverless Computing". In: *Journal of Cloud Computing* 10.1, p. 39. ISSN: 2192-113X. DOI: 10.1186/s13677-021-00253-7. URL: https://doi.org/10.1186/s13677-021-00253-7 (visited on 05/06/2024).

Jonas, Eric et al. (Feb. 9, 2019). *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. DOI: 10.48550/arXiv.1902.03383. arXiv: 1902.03383 [cs]. URL: http://arxiv.org/abs/1902.03383 (visited on 04/30/2024). Pre-published.

Kaushik, Vishal, Prajwal Bhardwaj, and Kaustubh Lohani (2022). "Layered Service Model Architecture for Cloud Computing". In: *Cloud Computing*. Ed. by Mohammad R. Khosravi, Qiang He, and Haipeng Dai. Cham: Springer International Publishing, pp. 91–106. ISBN: 978-3-030-99191-3. DOI: 10.1007/978-3-030-99191-3_8.

Lannurien, Vincent et al. (2023). "Serverless Cloud Computing: State of the Art and Challenges". In: *Serverless Computing: Principles and Paradigms*. Ed. by Rajalakshmi Krishnamurthi et al. Cham: Springer International Publishing, pp. 275–316. ISBN: 978-3-031-26633-1. DOI: 10.1007/978-3-031-26633-1_11. URL: https://doi.org/10.1007/978-3-031-26633-1_11 (visited on 04/22/2024).

Lau, Kung-Kiu and Zheng Wang (Oct. 2007). "Software Component Models". In: *IEEE Transactions on Software Engineering* 33.10, pp. 709–724. ISSN: 1939-3520. DOI: 10.1109/TSE.2007.70726. URL: https://ieeexplore.ieee.org/document/4302781 (visited on 09/10/2024).

Leitner, Philipp et al. (Mar. 1, 2019). "A Mixed-Method Empirical Study of Function-as-a-Service Software Development in Industrial Practice". In: *Journal of Systems and Software* 149, pp. 340–359. ISSN: 0164-1212. DOI: 10.1016/j.jss.2018.12.013. URL: https://www.sciencedirect.com/science/article/pii/S0164121218302735 (visited on 04/16/2024).

Li, Yongkang et al. (Mar. 2023). "Serverless Computing: State-of-the-Art, Challenges and Opportunities". In: *IEEE Transactions on Services Computing* 16.2, pp. 1522–1539. ISSN: 1939-1374. DOI: 10.1109/TSC.2022.3166553. URL: https://ieeexplore.ieee.org/abstract/document/9756233 (visited on 11/07/2024).

Lloyd, Wes et al. (Dec. 2018). "Improving Application Migration to Serverless Computing Platforms: Latency Mitigation with Keep-Alive Workloads". In: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), pp. 195–200. DOI: 10.1109/UCC-Companion.2018.00056. URL: https://ieeexplore.ieee.org/document/8605779 (visited on 05/07/2024).

Mell, Peter and Tim Grance (Sept. 28, 2011). *The NIST Definition of Cloud Computing*. NIST Special Publication (SP) 800-145. National Institute of Standards and Technology. DOI: 10.6028/NIST.SP.800-145. URL: https://csrc.nist.gov/pubs/sp/800/145/final (visited on 09/24/2024).

Messerschmitt, David G. (July 2007). "Rethinking Components: From Hardware and Software to Systems". In: *Proceedings of the IEEE* 95.7, pp. 1473–1496. ISSN: 0018-9219. DOI: 10.1109/JPROC.2007.898862. URL: http://ieeexplore.ieee.org/document/4287202/ (visited on 09/10/2024).

Mohammed, Chnar Mustafa and Subhi R. M. Zeebaree (2021). "Sufficient Comparison Among Cloud Computing Services: IaaS, PaaS, and SaaS: A Review". In: *International Journal of Science and Business* 5.2, pp. 17–30. URL: https://ideas.repec.org//a/aif/journl/v5y2021i2p17-30.html (visited on 09/11/2024).

Oliveira Rocha, Hugo Filipe (2022). *Practical Event-Driven Microservices Architecture: Building Sustainable and Highly Scalable Event-Driven Microservices*. Berkeley, CA: Apress. ISBN: 978-1-4842-7467-5 978-1-4842-7468-2. DOI: 10.1007/978-1-4842-7468-2. URL: https://link.springer.com/10.1007/978-1-4842-7468-2 (visited on 10/21/2024).

Opara-Martins, Justice, Reza Sahandi, and Feng Tian (Apr. 15, 2016). "Critical Analysis of Vendor Lock-in and Its Impact on Cloud Computing Migration: A Business Perspective". In: *Journal of Cloud Computing* 5.1, p. 4. ISSN: 2192-113X. DOI: 10.1186/s13677-016-0054-z. URL: https://doi.org/10.1186/s13677-016-0054-z (visited on 05/08/2024).

Papazoglou, M.P. (Dec. 2003). "Service-Oriented Computing: Concepts, Characteristics and Directions". In: *Proceedings of the Fourth International Conference on Web Information Systems Engineering, 2003. WISE 2003.* Proceedings of the Fourth International Conference on Web Information Systems Engineering, 2003. WISE 2003. Pp. 3–12. DOI: 10.1109/WISE.2003.1254461. URL: https://ieeexplore.ieee.org/document/1254461/?arnumber=1254461 (visited on 09/10/2024).

Rajan, Arokia Paul (Feb. 1, 2020). "A Review on Serverless Architectures - Function as a Service (FaaS) in Cloud Computing". In: *TELKOMNIKA (Telecommunication Computing Electronics and Control)* 18.1 (1), pp. 530–537. ISSN: 2302-9293. DOI: 10.12928/telkomnika.v18i1.12169. URL: http://telkomnika.uad.ac.id/index.php/TELKOMNIKA/article/view/12169 (visited on 05/07/2024).

Reuter, Anja, Timon Back, and Vasilios Andrikopoulos (Aug. 2020). "Cost Efficiency under Mixed Serverless and Serverful Deployments". In: *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 242–245. DOI: 10.1109/SEAA51224.2020.00049. URL: https://ieeexplore.ieee.org/document/9226321 (visited on 05/18/2024).

Richards, Mark (2015). *Software Architecture Patterns*. O'Reilly Media, Inc. ISBN: 978-1-4919-2540-9.

Roberts, Mike and John Chapin (May 24, 2017). *What Is Serverless? [Book]*. URL: https://www.oreilly.com/library/view/what-is-serverless/9781491984178/ (visited on 05/18/2024).

Sfondrini, Nicola, Gianmario Motta, and Antonella Longo (July 2018). "Public Cloud Adoption in Multinational Companies: A Survey". In: *2018 IEEE International Conference on Services Computing (SCC)*. 2018 IEEE International Conference on Services Computing (SCC), pp. 177–184. DOI: 10.1109/SCC.2018.00030. URL: https://ieeexplore.ieee.org/document/8456416 (visited on 09/10/2024).

Shafiei, Hossein, Ahmad Khonsari, and Payam Mousavi (June 4, 2021). *Serverless Computing: A Survey of Opportunities, Challenges and Applications*. DOI: 10.48550/arXiv.1911.01296. arXiv: 1911.01296 [cs]. URL: http://arxiv.org/abs/1911.01296 (visited on 09/21/2024). Pre-published.

Shrestha, Raju and Beebu Nisha (Dec. 2022). “Microservices vs Serverless Deployment in AWS: A Case Study with an Image Processing Application”. In: *2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC)*. 2022 IEEE/ACM 15th International Conference on Utility and Cloud Computing (UCC). Vancouver, WA, USA: IEEE, pp. 183–184. ISBN: 978-1-66546-087-3. DOI: 10.1109/UCC56403.2022.00033. URL: https://ieeexplore.ieee.org/document/10061802/ (visited on 05/21/2024).

Taibi, Davide, Nabil El Ioini, et al. (2020). “Patterns for Serverless Functions (Function-as-a-Service): A Multivocal Literature Review”. In: *CLOSER 2020 - Proceedings of the 10th International Conference on Cloud Computing and Services Science*. International Conference on Cloud Computing and Services Science. Science and Technology Publications (SciTePress), pp. 181–192. DOI: 10.5220/0009578501810192. URL: https://researchportal.tuni.fi/en/publications/patterns-for-serverless-functions-function-as-a-service-a-multivo (visited on 05/21/2024).

Taibi, Davide, Ben Kehoe, and Danilo Poccia (Aug. 2022). “Serverless: From Bad Practices to Good Solutions”. In: *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. 2022 IEEE International Conference on Service-Oriented System Engineering (SOSE). Newark, CA, USA: IEEE, pp. 85–92. ISBN: 978-1-66547-534-1. DOI: 10.1109/SOSE55356.2022.00016. URL: https://ieeexplore.ieee.org/document/9912641/ (visited on 05/21/2024).

Villamizar, Mario et al. (June 2017). “Cost Comparison of Running Web Applications in the Cloud Using Monolithic, Microservice, and AWS Lambda Architectures”. In: *Service Oriented Computing and Applications* 11.2, pp. 233–247. ISSN: 1863-2386, 1863-2394. DOI: 10.1007/s11761-017-0208-y. URL: http://link.springer.com/10.1007/s11761-017-0208-y (visited on 05/21/2024).

Wohlin, Claes (May 13, 2014). “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*. EASE ’14: 18th International Conference on Evaluation and Assessment in Software Engineering. London England United Kingdom: ACM, pp. 1–10. ISBN: 978-1-4503-2476-2. DOI: 10.1145/2601248.2601268. URL: https://dl.acm.org/doi/10.1145/2601248.2601268 (visited on 09/03/2024).

Yussupov, Vladimir, Uwe Breitenbücher, Antonio Brogi, et al. (2022). “Serverless or Serverful? A Pattern-Based Approach for Exploring Hosting Alternatives”. In: *Service-Oriented Computing*. Ed. by Johanna Barzen, Frank Leymann, and Schahram Dustdar. Cham: Springer International Publishing, pp. 45–67. ISBN: 978-3-031-18304-1. DOI: 10.1007/978-3-031-18304-1_3.

Yussupov, Vladimir, Uwe Breitenbücher, Frank Leymann, et al. (Dec. 2, 2019). “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’19. New York, NY, USA: Association for Computing Machinery, pp. 273–283. ISBN: 978-1-4503-6894-0. DOI: 10.1145/3344341.3368813. URL: https://dl.acm.org/doi/10.1145/3344341.3368813 (visited on 05/07/2024).

Yussupov, Vladimir, Jacopo Soldani, et al. (2021). “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*. 11th International Conference on Cloud Computing and Services Science. Online Streaming, — Select a Country —: SCITEPRESS - Science and Technology Publications, pp. 268–279. ISBN: 978-989-758-510-4. DOI: 10.5220/0010481002680279. URL: https://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0010481002680279 (visited on 04/22/2024).

Appendix A

Deployment Target Examples

Deployment Target/Scenario	Category	**SP1**	**SP2**	**SP3**	Serverless
Basement Server	Self-Host				
AWS EC2	IaaS				
Azure VMs	IaaS				
AWS VPC	IaaS				
Kubernetes on Hetzner	CaaS				
Kubernetes on Hetzner w/ OpenFaaS	FaaS			✓	
Managed Kubernetes (AKS/AWS EKS)	CaaS				
Managed Kubernetes w/ Knative	CaaS		✓		
Managed Kubernetes w/ OpenFaaS	FaaS		✓	✓	
AWS EKS w/ Fargate	CaaS		✓	✓	
Azure Container Instances	CaaS		✓		
AWS RDS	PaaS		✓		
AWS ECS w/ EC2	CaaS			✓	
AWS Beanstalk	PaaS		✓	✓	
AWS Aurora Serverless	PaaS		✓	✓	
AWS Route53	PaaS		✓	✓	
Managed Kubernetes w/ Knative managed by IT	CaaS	✓	✓	✓	✓
Basement Server w/ OpenWhisk managed by IT	FaaS	✓	✓	✓	✓
Managed Kubernetes w/ OpenFaaS managed by IT	FaaS	✓	✓	✓	✓
AWS ECS w/ Fargate	CaaS	(✓)	✓	✓	✓
Google Cloud Run	CaaS	✓	✓	✓	✓
Azure Container Apps	CaaS	✓	✓	✓	✓
AWS AppRunner	CaaS	(✓)	✓	✓	✓
AWS Lambda w/ Provisioned Concurrency > 2	FaaS		✓	✓	✓
AWS Lambda	FaaS	✓	✓	✓	✓
Azure Functions	FaaS	✓	✓	✓	✓
AWS SQS	BaaS	✓	✓	✓	✓
Azure Event Hubs	BaaS	✓	✓	✓	✓
AWS S3	BaaS	✓	✓	✓	✓
AWS Cognito	BaaS	✓	✓	✓	✓
AWS API Gateway	BaaS	✓	✓	✓	✓

Table A.1: *Selected deployment targets and how they adhere to the principles of serverless, vertical positioning gives a sense of position on the spectrum but is not to be taken too strictly;* ***SP1****: costs nothing when not in use,* ***SP2****: no responsibility on infrastructure management,* ***SP3****: no responsibility on scaling configuration; ✓: true, (✓): partially true.*

Appendix B

Interview Transcripts

Notes on the Interviews:

- Interviews were conducted with practitioners within Device Insight (DI).
- The interviews were held in German and have been translated into English by the author of this work.
- Where necessary, information was pseudonymized to protect the identity of the interviewee or company internal information.
- The following is a representation of notes taken during the interviews.
- To ensure the accuracy of the information, all transcribed, translated and pseudonymized interview notes presented here were submitted to the interviewees for approval.

B.1 Interview - Cloud Acceleration Team (CAT)

Interviewee Details:

- Position: *Head of Solution Engineering*
- Years of Experience: 10 years

Interview Goals:

- Insights into how projects get started.
- Architectures and decisions.
- Under which aspects decisions are weighed up.
- Customers' wishes, ideas, specifications, and requirements.
- What is important and what is irrelevant?
- What does not matter, or where, and to what extent do we have free choice in decisions?

Interview Transcript:

Q: What is your state of business — i.e., how does CAT realize projects?

- We develop 95% serverless.
- serverful only where needed:
  - We once had a Proof of Concept (PoC) with a managed Kubernetes cluster for a customer.

  - Was a PoC to test our company product in Azure Cloud.
  - Explicitly asked by the customer.
  - Reasoning: The customer wanted to also manage operations and not only have the business value.

Q: Why serverless?

- "CAT can focus on the essential" → business value for customers
- Suitable for IoT:
  - Scaling requirements.
  - Small but numerous data events.
  - Event-driven processing of data.
- Sales pitch: scalability upwards
- Technical Pre-Sales PoC: scalability downwards (not a Kubernetes cluster!!).
- Customers not only want to be cloud-native but Azure-native.
- Business strategies: serverless
- Customers want their business value at a low price.

Q: What are costs?

- Infrastructure (Lambdas, EC2, Azure Event hubs, AKS, etc.).
- Development.
- Operations.
- Personnel: monitoring or maintenance

Q: When to switch to serverful?

- "If services run 24/7".
- High load on the system.
- Look out for cost parameters of infrastructure: messages, IOPS, storage.
- IT strategy: no dependence long-term and their ecosystem, i.e., avoid vendor lock-in (e.g., Microsoft Time Series Insights).

Q: Is it always about cost?

- Other aspects can be important too:
  - Know-how: to handle operation and maintenance, personnel that needs to be able to handle a Kubernetes or Kafka cluster.
  - This aspect becomes irrelevant with serverless.
  - Personnel shortages.
  - Availability, SLAs, geo-redundancy, disaster scenarios.
  - This is why we choose cloud/serverless.

Q: Where no serverless?

- Azure Functions: problems for backend services that communicate with frontend.
- Cold-starts.
- Limitations with the development stack.
- Difficulties getting the architecture productive.
- From a certain complexity of the application logic on, you benefit from the flexibility of serverful as you have more control.

Q: Thoughts on vendor lock-in?

- Commercial: prices of the offerings in the hands of the provider
- Technical: bound to what the provider makes available

Q: Why the cloud?

- Would Kubernetes and OpenFaaS not be sufficient?
- We have occasionally thought about this.
- Biggest drawbacks mentioned would not be solved by this.
- Other motives could be regulatory: the private cloud is necessary because of data protection, for example.

B.2 Interview - *Project X*

Interviewee Details:

- Interviewee 1:
  - Position: *Software Engineer and Delivery Team Lead*
  - Years of Experience: 13 years
- Interviewee 2:
  - Position: *Software Engineer*
  - Years of Experience: 17 years
- The transcript summarizes the interview and both interviewees' responses.

Interview Goals:

- Information about the project requirements and architecture.
- Reasoning why serverless was chosen.
- Insights about what went wrong with serverless.

Interview Transcript:

Q: What requirements were faced?

- Operational cost budget of €1 per day per connected machine.
  - It was calculated as resources + time.
  - However, it was not an issue *operation-wise.
  - But the development cost was too high.
- Geo-location: the system had to run completely inside of the EU.
- Microservice architecture.
- Azure and therefore public cloud was customer-given.

Q: What architecture did the project employ?

- Authentication purposes outsourced to BaaS.
- FaaS as the main component of compute.
- Additional Azure offerings (e.g., Event Hub or ADX) which FaaS should "glue" together.
- ⇒ *typical serverless architecture: frontend-backend application with FaaS + BaaS*

Q: What were the reasons for deciding to go with serverless and what deployment target was chosen?

- They had an ADR about it, and went with Azure Functions.
- Azure Function with a keep-alive mechanism to keep functions warm and prevent cold starts.
- Azure Container Instances were considered too but would have been too expensive.

Q: What went bad in the project?

- ⇒ *unguided experience*
- Missing know-how on
  - Best practices.
  - No experience with FaaS and using it properly.
- FaaS was getting more difficult the more complexity and features were introduced into the application/architecture.
  - The more functions, the more complex was development.
  - Function composition and complex workflows made development tedious.
- Complexity increased due to Azure constraints and the chosen development stack.
- Difficult to work with unknown defaults in Azure environment.
- OpenAPI and Azure were not compatible, though it was an afterthought in the project.
- Custom implementation to imitate common framework benefits: request validation, documentation generation, feature and size code-scalability.

Q: Specifically, what were the problems with serverless/FaaS?

- Runtime/Environment constraints
  - Hard resource and runtime limits.
  - (ML-) jobs that had to be run, introduced code complexity to fit into short-running FaaS-suitable workloads.
- Cold-start problems.

Q: What are the core takeaways from this experience?

- Serverless is less suitable for REST APIs.
  - Frameworks and good tooling are essential for a good development experience.
  - Rather should have gone with containerized solutions.
- Main pain points were missing know-how with Azure and the FaaS development stack.

B.3 Interview - *Project B*

Interviewee Details:

- Position: *Software Engineer, IoT Architecture Consultant*
- Position within Project specifically: 'Technical Lead / Backend Developer'
- Years of Experience: 16 years

Interview Goals:

- Find out about requirements and architecture.
- Gain insights if serverless was successful here?

Q: What requirements were faced?

- Customer had a strict cost sensitivity that had to be catered to.

Q: What architecture contained the project?

- Synchronous Rest API service:
  - Spring Boot project w/ Maven Azure Function adapter library.
  - Custom implemented request validation.
- Asynchronous processing of events.
  - Mainly Azure Functions that handled processing flows.

Q: Was serverless successful in this project?

- "For HTTP REST APIs, it was not".
- It was a mistake to use Azure Functions for their HTTP Rest API → "never again".
  - Latency concerns due to cold-start issues.
  - There was no direct latency requirement from the customer, but they were too big nevertheless.
- Opinion: serverless for something quick and small → processing.
- Opinion: no serverless for synchronous HTTP REST APIs in production.

Q: What is a preferred architecture as opposed to the one here discussed?

- Favored architecture from a different project was referenced:
  - HTTP APIs: Kubernetes services (CaaS).
  - Processing events: Azure Functions (FaaS) as IotHub and EventHub consumers.

Q: If your preferred architecture involves Azure Functions for processing events, would there be a need in this case for switching?

- The cloud ecosystem is too strong to be able to foster swift switching.
- Azure Functions are integrated too well into Azure's ecosystem.
- "If you keep your functions 'pfiffig', portability is not that much of an issue".

B.4 Interview - *Practitioner J*

Interviewee Details:

- Position: *Software Engineer*
- Years of Experience: 6 years

Interview Goals:

- Gain insights into experience migrating a processing application to a specific project; from Azure Functions to Azure Container Apps.
- Gain insights into why Container Apps are allegedly superior to Azure Functions.

Q: What makes one choose Container Apps over Azure Functions for processing applications?

- Operation-wise, no bigger overhead in comparison to AFs.
- Lower cost possible as more control about execution environment: e.g., resource restrictions.
- Migration to, e.g., Kubernetes possible, no vendor lock-in into effects.
- Better performance in terms of latency because no cold-starts: *they switched a project because of this*
- Local debugging and development is important: one can expect compromises if fully leaning into Azure.
- One would be slowed in development if too reliant on the cloud.
- Azure functions good for prototyping, for almost time-to-value, but as soon as dependencies to third-party systems (e.g., Event Hubs, Databases) are introduced, pivot to Container Apps.

Q: In the mentioned project, how come you switched from Azure Functions to Azure Container Apps, and how did this go?

- All around better suitability for the requirements of the projects ⇒ "generally, no reason for Azure Functions anymore".
- Cold-starts were not important for processing workloads, though.
- Better development experience:
  - Local debugging is important.
  - Faster development because of no reliance on the cloud.
- Background tasks are also possible for services.
- "Switch was easy": *the interviewee proceeded to refer to the exact merge request for details about the switch: following is a description of the MR, added after the interview*
  - +208/-256 LOC.
  - Azure Function-specific configuration was deleted.
  - The Azure Function integration wrapper around the data transform service was deleted.
  - A Spring Kafka listener was added that calls the data transformer service.

- ⇒ *they kept processing logic and integration strictly separate as possible — layered architecture*
- ⇒ *they could do development-wise and did it easily effort-wise, how the recommended processing approach aims to enable it, without knowing about it*

Q: From your experience, what is important when working with Azure Container Apps?

- Layer of abstraction important to change underlying services: Cloud ⇔ Local, e.g., Azure CosmosDB and its MongoDB compatibility layer.
- Background tasks are available.
- Pricing differences for idle and non-idle periods: min/max potential possible.
- Container App jobs and auto-scaler give a great combo for quick and cheap services: an alternative for FaaS.

B.5 Interview - *Practitioner S*

Interviewee Details:

- Position: *Director of Cloud Architecture & Innovation*
- Years of Experience: 15 year

Interview Goals:

- Introduce and present the recommended serving approach.
- Get insights and opinions about suitability and viability.

Q: What is the most important benefit of serverless?

- The automated scaling.

Q: Having you presented the recommended approach for hybrid deployment of HTTP serving applications in AWS Lambda, i.e., LWA, what do you think of it? In your experience, what negative effects could this have? What would I need to evaluate?

- Scaling in development: the more complex the application gets, how does it scale with it?
- Scaling in deployment: can it handle the same load the same way as *raw functions*?
  - Parallel computing, automated scale-down and scale-up assured?
- $\Rightarrow$ *Guidelines about how to work with scaling applications and the concept would be good*
- Related to the previous point, if the benefits of automated scaling are still given with the new approach?
  - $\Rightarrow$ *Benchmarks and load tests are important here*
- Is latency/cold-start an issue?
  - Most important for serving applications.
  - For processing, on the other hand, is somewhat important but to some degree acceptable.

B.6 Interview - *Practitioner A*

Interviewee Details:

- Position: *Software Engineer, IoT Architecture Consultant*
- Years of Experience: 16 years

Interview Goals:

- Present the recommended hybrid deployment approach for serving applications, LWA specifically.
- Feedback and opinions about concerns of the approach.

Q: What are your concerns regarding the presented approach, specifically the Lambda Web Adapter?

- Most important: scope has to be communicated very clearly — what can the approach do and what can it not do.
- Regarding the LWA Dockerfile example: should not contain AWS-specific extension if it is not used, e.g., when deployed on Kubernetes.
  - Can negatively influence maintainability.
  - Potential risk factor.
- *The interviewee was skeptical about the approach, partially because of bad experiences with FaaS REST APIs in projects*
- *The interviewee agrees though to: "if someone would do serverless HTTP APIs, one should use it"*

Appendix C

Project D Survey

Notes on the Surveys:

- **Three** developers were asked to fill out the survey, amongst them the 'Technical Lead' of the project.
- Where necessary, information was pseudonymized to protect the identity of the interviewee or the company's internal information.
- The following is a representation of the survey results.

Survey Questions

General Questions

Q: How satisfied were you with the old approach of the API? *from 1-4, 1 being very not satisfied and 4 being very much satisfied*
One developer answered to be *very not satisfied (1)*;
the other two answered to be *not satisfied (2).*

Q: How satisfied are you with the new approach of the API? *from 1-4, 1 being very not satisfied and 4 being very much satisfied*
All three developers have answered to be *very much satisfied (4).*

Q: What pain points led to the decision to switch to the new approach?

- *No automatic OpenAPI Spec, Manual work in i.e., request validation, hard to add new end points (always a new function.json file), Spring like decorators are hard to maintain and are already deprecated.*
- *I guess mainly the development part.*
- *There was no framework in place for common tasks like validation, authentication, error responses, parsing, etc. Lack of dependency injection caused a lot of boilerplate code for instantiating common classes. The team tried to add convenient helpers to abstract away such things, but this often resulted in multiple similar implementations.*

Q: Did the new approach solve any lingering issues or circumvent the constraints of the old approach?

- *Yes, automatic OpenAPI generation, and it is now possibly to easily host the API in a dedicated way more reliable than starting a functions host.*

- *No.*
- *OpenAPI spec is generated automatically. Validation of request objects and parameters is automatic. The codebase looks more coherent in general. The use of modules and dependency injection makes boundaries more clear and allows replacing services more easily. It's easy to find solutions to common problems on the internet and in documentation without having to re-invent the wheel.*

Q: Since switching to the new approach, did you encounter any *new constraints* or *pain points*?

- *Mentioned small issues such as headers missing.*
- *No.*
- *Using NestJS in a serverless environment has its quirks which you need to be aware of. For example, the IoC container by default instantiates objects only once on startup, which is good for performance. In a serverless environment, we need to configure some parts of our dependencies to be "request-scoped" — such objects are instantiated on every single request. This needs to be done because we need to access Azure Context information per request. This has some impact on the overall architecture and maybe also on performance (not measured yet). On the other hand, cold starts are not the fastest and they take some seconds. We haven't measured this either and don't have an issue with this (yet).*

Numerical Questions

Question	1	2	3	4	5	6	Average	Median
a) How satisfied were you with the **old development experience** of the API?		3					2	2
b) How satisfied were you with the **old deployment experience** of the API?				1	2		4.667	5
c) How satisfied were you with the **old experience of dealing with any kinds of problems, bugs, or issues** of the API?		1		1	1		3.667	4
d) How satisfied are you with the **current development experience** of the API?					3		5	5
e) How satisfied are you with the **current deployment experience** of the API?				2	1		4.333	4
f) How satisfied are you with the **current experience of dealing with any kinds of problems, bugs, or issues** of the API?		1			2		4	5

The participants were also asked to provide comments if possible on their assessment for the above questions *a* to *f*:

a: *a.1: The "Spring like" decorators made it a lot better, but the raw azure functions experience was horrible. Had to re-invent the wheel with like request validation and error handling. Plus, always maintaining the OpenAPI doc felt like a hassle.*

a.2: Azure functions lack any structures, so we had to develop our own like framework.

a.3: A lot of basic http functionality had to be implemented manually all across the codebase. Things like auth, parsing requests and formatting responses. We were implementing our own framework to try and deduplicate such logic. This had the effect of frequent large refactoring or deprecation of internal code. Since the "framework" doesn't automatically get documented, it is not easy to onboard new developers, as a lot of things are known by convention to experienced developers. Another big disadvantage was no automatic OpenAPI integration — An OpenAPI spec was maintained manually to reflect the code.

b: *b.1: Not really a problem, however it was / is unsure when & how azure functions are restarted after an update.*

b.2: Deployment was pretty nice, building sources and then uploading them with one command.

b.3: Deployment was relatively straight forward to azure functions. My minus points are attributed to general dissatisfaction with Azure Function deployments and the relative complexity of the configuration of such deployments in terms of resource allocation (outgoing connections, scaling, . . .).

c: *c.1: Problems are hard to investigate if functions crash and don't send any log output.*

c.2: There weren't many issues that come to my mind that were attributed to the "old" codebase. Many of the more difficult problems had something to do with resource allocation in the Azure Functions environment or the lack of a good observability configuration.

d: *d.1: Development is more standardized due to popular framework, however due to the "hacky wiring" some initial work had to be done, as for example request headers were missing.*

d.2: Nest is very nice. There is a solution for every problem in API development. Also, dependency injection is a game changer.

d.3: We're using NestJS and benefit from core functionality for the pain points listed in the previous questions. The code feels more maintainable and easier to navigate due to opinionated structures and defaults.

e: *e.1: The deployment with pulumi in typescript is nice.*

e.2: When looking at the old vs. new business logic codebase, not much has changed in terms of deployment, so the previous issues are still there when deploying to azure functions. There are some additional benefits since we're deploying the functions with Pulumi now, but that is a separate topic. There's also the added benefit of being able to deploy the same codebase to a container, but since we haven't actually taken advantage of this, it doesn't really count in practice yet.

f: *f.1: Easily startable locally without functions host.*

f.2: Nothing has changed here, since azure functions are still used under the hood.

f.3: There seems to be an improvement to local development/debugging as well as integration testing of the code base as there are common patterns for doing integration tests with NestJS.

Appendix D

Project K Survey

Notes on the Survey:

- The survey was conducted after the projects' migration using the recommended hybrid deployment approach for serving applications of *Project K*.
- Where necessary, information was pseudonymized to protect the identity of the interviewee or internal information.
- The following is a representation of the survey results.

Survey Questions

Q: Do you know your services' usage patterns? how do they look like?

- *There are more requests during the day and very few during the night. On weekends and during vacation times, there are also fewer requests. (see attachment in chat)*

The respondent also provided a graph showing the usage patterns of the service over 2 days — Sunday noon to Tuesday morning:

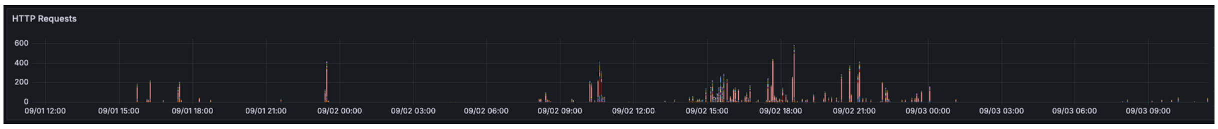


Figure D.1: *This image showcases example usage patterns for* Project K*$\dot{T}$he graph shows the number of HTTP requests per API path over a period of three days.*

Q: How easy was it to switch?

- *5/10. Not too difficult as the existing application was already quite stateless in nature. The lambda web adapter was easy to integrate with the existing docker images. The deployment had to be adjusted to push the new image to lambda.*

Q: Did you encounter roadblocks on the way?

- *Environment variables and secrets are different in Lambda from Kubernetes*
- *Internal service dependencies running in Kubernetes had to be removed*
- *NestJS internals that were not 100% stateless had to be removed (internal eventing, scheduling)*

- *Postgres connection limits are less controllable in serverless environment due to potentially unbound scaling of instances/client connections*
- *Database migration best practices*
- *Monitoring best practices*
- *Long cold starts*

Q: Could you solve them and if so how?

- *Lambda environment variable size limit could be circumvented by embedding a "static" CA certificate inside the docker image*
- *Kubernetes secret env vars were simply put into lambda environment variables without using AWS Secret Management, as this would require adjustments to the source code. The secrets are still encrypted at rest inside AWS but visible in the AWS* management console.
- *Internal event handlers and scheduling could be removed from the lambda application. There's a separate deployment in the existing Kubernetes environment that contains the scheduling functionality. This can be migrated to separate scheduled lambda deployments in the future. This was only possible because the existing* Project K *codebase already had the capability to run with different profiles to allow a "scheduling-only" deployment which can continue to run in the Kubernetes environment.*
- *Cold starts could be reduced from around 10s to around 3s by reducing the docker image size by making use of Webpack bundling. Also, a dedicated NestJS module was introduced with fewer dependencies to further reduce the bundle size.*

Q: If, what roadblocks could not be solved (yet)?

- *Postgres connection limits are expected to be hit if the Lambda load increases drastically. Currently, the average load on the application is not as high, so it's possible to use a dedicated Postgres connection pool with no issues. If there is a sudden burst of load that creates parallel Lambda instances, too many connections will be opened against the Postgres, which will result in failure at runtime. The simplest solution for this is to either scale up the Postgres instance to allow for more concurrent connections (high cost, does not scale with load) or to use a "serverless" Postgres offering like CockroachDB or Neon which advertises more dynamic connection management.*
- *Kubernetes init container migration best practices were used prior. Now the migration scripts will have to be moved to a "deployment" script. Best practices need to be researched.*
- *Monitoring was based on Prometheus/Grafana. Since Lambdas can not be scraped by Prometheus as they do not persist over time, a different solution will need to be found. Probably, heavier vendor lock-in will happen when using integrated CloudWatch tooling. Currently, there is less observability of the API Gateway + Lambda integration. Logs need to be monitored to find possible errors.*

Q: Did you notice any different behavior in your system?

- *Some cold start delay can be observed when using the web application, as the NestJS cold start invocation takes around three seconds. CloudWatch metrics also show the cold starts quite well.*
- *From user perspective, no further issues were seen.*
- *From operation perspective, no incidents happened so far.*

Q: How important is latency for your service, and did you notice any losses in latency after switching?

- *There are no hard requirements for latency or SLA/SLOs*
- *There's a personal goal of staying below 1s for every request, ideally around 200ms per request.*
- *Previously slow requests were caused mostly by unoptimized database queries, which could be improved. Currently, the slow cold starts can occur on any request.*

Q: Did you look into alternatives other than LWA?

- *I considered deploying the NestJS/Node application as a "normal" zip deployment. My initial attempts failed because the resulting package exceeded the 50 MB.*
- *According to my latest knowledge this should work if using my latest Webpack configuration where the package size seems to be around 2 MB.*

Q: How happy are you with the switch?

- *4/6.*
- *Judgement of the switch is currently only based on the fact that the switch was possible and seems promising in terms of costs. Since I haven't run this for long enough in production, I will update my answer later.*
- *EDIT: I have now run this in production for at least a month, cost was reduced to a tenth of the previous cost — around ten euros now.*
- *EDIT: I am very happy with the switch, technically, some stuff was to be done and learned, some bugs during migration that needed to be fixed due to moving to serverless — though nothing critical. The ten times saving is nice. Cold-starts are a topic — I use antipatterns with keeping the Lambda functions warm, to work around this.*

Appendix E

Processing Applications — Multi-Integration Example

```go
// File: src/main.go
...

type Payload struct {
  Name string `json:"message"`
}

type Service struct{
}

func (s *Service) Process(payload Payload) {
  fmt.Printf("Hello %s!", payload.Name)
}

func lambdaHandler(ctx context.Context, sqsEvent events.SQSEvent) (error) {
  // parse the SQS event
  payload := ...

  service := Service{}
  service.Process(payload)

  return nil
}

func httpHandler(w http.ResponseWriter, r *http.Request) {
  // parse the HTTP request
  payload := ...

  service := Service{}
  service.Process(payload)
}

func mqttHandler(client mqtt.Client, msg mqtt.Message) {
  // parse the MQTT message
  payload := ...

  service := Service{}
  service.Process(payload)
}

func inLambdaEnv() bool { ... }
func inHTTPEnv() bool { ... }
func inMQTTEnv() bool { ... }

func main(){
  // conditional integration logic:
  if (inLambdaEnv()){
    lambda.Start(lambdaHandler)
  } else if (inHTTPEnv()){
    http.HandleFunc("/", httpHandler)
    http.ListenAndServe(":8080", nil)
  } else if (inMQTTEnv()){
    client := mqtt.NewClient(...)
    client.Subscribe("topic", 0, mqttHandler)
  }
}
```

Listing 7: *Example of a processing application in Go that is adapted for different deployment targets and event sources — HTTP, SQS with Lambda, and consuming a MQTT topic. The processing logic is kept separate from the integration logic, which allows for easy adaptation of the integration logic to different deployment targets. Conditional integration logic is utilized to determine the deployment target and trigger the appropriate integration.*

Appendix F

Overhead and Performance Benchmarks — Lambda Web Adapter

Short Requests — Go Fibonacci service

Scenario	Category	Lambda	Lambda w/LWA	Increase (%)
Constant Rate $50\frac{req}{s}$	Duration	1h00m00s	1h00m00s	-
	Succ. Reqs.	179 999	180 000	-
	Failed Reqs.	0	0	-
	Avg. Req. Rate	50.0	50.0	-
	Avg. Conc. Inst.	7	7	-
	Mean (ms)	20.26	20.23	-0.1%
	P50 (ms)	18.84	19.44	$+3.1\%$
	P95 (ms)	23.11	23.71	$+2.6\%$
	P99 (ms)	37.77	34.91	-8.2%
	Max (ms)	3300.60	460.48	-616.8%
Spike	Duration	0h03m00s	0h03m00s	-
	Succ. Reqs.	179 994	179 994	-
	Failed Reqs.	0	0	-
	Avg. Req. Rate	999.96	999.96	-
	Avg. Conc. Inst.	*108**	*39**	$+176.9\%$
	Mean (ms)	19.15	19.33	$+0.9\%$
	P50 (ms)	18.31	18.75	$+2.4\%$
	P95 (ms)	23.63	23.94	$+1.3\%$
	P99 (ms)	33.64	31.52	-6.7%
	Max (ms)	442.64	351.38	-26.0%
Constant VU	Duration	0h30m00s	0h30m00s	-
	Succ. Reqs.	88 467	86 091	-2.8%
	Failed Reqs.	0	0	-
	Avg. Req. Rate	49.15	47.83	-2.8%
	Avg. Conc. Inst.	1	1	-
	Mean (ms)	20.04	20.61	$+2.8\%$
	P50 (ms)	19.51	20.02	$+2.6\%$
	P95 (ms)	23.50	24.14	$+2.7\%$
	P99 (ms)	31.13	31.08	-0.2%
	Max (ms)	408.33	787.78	$+92.9\%$

Table F.1: *Performance benchmark results of a simple Go Fibonacci service deployed to AWS Lambda with and without the Lambda Web Adapter (LWA) for different load scenarios. Metrics provided are: duration of the benchmark; successful and failed requests, average request rate of the client; average number of concurrent instances; mean, p50, p95, p99 and max latencies in milliseconds. *NOTE: the average number of instances for the spike scenario results from a single data point available — this should be considered for further interpretation*

Billing and Cost Comparison — Constant Rate Scenario

Category	**Lambda**	**Lambda w/LWA**	**Increase (%)**
Total Invocations	179 999	180 000	-
Initialization Duration (ms)	709.96	1709.63	+140.8%
Total Billed Duration (GBs)	474.06	663.60	+39.9%
Invocation Cost ($)	0.03599	0.03600	-
Total Cost ($)	0.04390	0.04706	+7.2%

Table F.2: *Billing and cost comparison of a simple Go Fibonacci service deployed to AWS Lambda with and without the Lambda Web Adapter (LWA) for a constant rate of 50 requests per second over one hour. Metrics provided are: total invocations, initialization duration in milliseconds, total billed duration in GBs; invocation cost and total cost in USD. Initialization duration and total billed duration were extracted manually from AWS CloudWatch during the benchmark run. Total cost was calculated (excluding AWS tiered pricing for Lambda, see here) with pricing:* $0.0000166667\frac{\$}{GBs}; 0.0000002\frac{\$}{invocation}$, *see here.*

Long Requests — Go File service

Scenario	Category	Lambda	Lambda w/LWA	Increase (%)
Constant Rate $1\frac{req}{s}$	Duration	1h00m00s	1h00m00s	-
	Succ. Reqs.	3601	3600	-
	Failed Reqs.	0	0	-
	Avg. Req. Rate	1.0	1.0	-
	Avg. Conc. Inst.	9	9	-
	Mean (ms)	8247.28	8134.69	-1.4%
	P50 (ms)	8265.20	8161.22	-1.4%
	P95 (ms)	8756.88	8369.35	-4.6%
	P99 (ms)	8830.71	8454.33	-4.5%
	Max (ms)	9825.65	9011.45	-9.0%
Spike	Duration	0h03m00s	0h03m00s	-
	Succ. Reqs.	772	744	-3.8%
	Failed Reqs.	318	332	$+4.4\%$
	Avg. Req. Rate	5.59	5.51	-1.5%
	Avg. Conc. Inst.	90	101	$+12.2\%$
	Mean (ms)	31 068.81	30 091.59	-3.2%
	P50 (ms)	31 990.87	27 576.14	-16.0%
	P95 (ms)	57 682.76	57 933.76	$+0.4\%$
	P99 (ms)	59 628.76	59 654.27	$+/-0.0\%$
	Max (ms)	59 957.10	59 889.66	-0.1%
Constant VU	Duration	0h30m00s	0h30m00s	-
	Succ. Reqs.	213	213	-
	Failed Reqs.	0	0	-
	Avg. Req. Rate	0.12	0.12	-
	Avg. Conc. Inst.	1	1	-
	Mean (ms)	8452.72	8457.69	$+0.1\%$
	P50 (ms)	8478.00	8477.55	$+/-0.0\%$
	P95 (ms)	8682.93	8673.05	-0.1%
	P99 (ms)	8853.88	8791.33	-0.7%
	Max (ms)	8982.20	11 362.45	$+26.5\%$

Table F.3: *Performance benchmark results of a compute-heavy Go File service that hashes, encrypts, and stores files to S3 deployed to AWS Lambda with and without the Lambda Web Adapter (LWA) for different load scenarios. Metrics provided are: duration of the benchmark; successful and failed requests, average request rate of the client; average number of concurrent instances; mean, p50, p95, p99 and max latencies in milliseconds.*

Appendix G

Benchmark Extensions — AWS Compute and Virtual Machine

Short Requests — Go Fibonacci service

This table is an extension from the previous table Table F.1:

Scenario	**Category**	**Lambda**	**Lambda w/LWA**	**Fargate**	**App Runner**	**Hetzner VM**
Constant Rate $50\frac{req}{s}$	Duration	1h00m00s	1h00m00s	1h00m00s	1h00m00s	1h00m00s
	Succ. Reqs.	179 999	180 000	180 000	179 999	180 000
	Failed Reqs.	0	0	0	0	0
	Avg. Req. Rate	50.0	50.0	50.0	50.0	50.0
	Avg. Conc. Inst.	7	7	1	1	1
	Mean (ms)	20.26	20.23	12.61	13.79	14.73
	P50 (ms)	18.84	19.44	12.13	13.10	14.37
	P95 (ms)	23.11	23.71	13.73	15.65	15.33
	P99 (ms)	37.77	34.91	16.86	21.88	17.45
	Max (ms)	3300.60	460.48	244.95	1018.02	250.45
Spike	Duration	0h03m00s	0h03m00s	0h03m00s	0h03m00s	0h03m00s
	Succ. Reqs.	179 994	179 994	179 986	125 456	179 994
	Failed Reqs.	0	0	0	54 456	0
	Avg. Req. Rate	999.96	999.96	999.97	999.96	999.96
	Avg. Conc. Inst.	*108**	*39**	10	*12**	1
	Mean (ms)	19.15	19.33	28.78	131.24	15.72
	P50 (ms)	18.31	18.75	11.47	14.53	13.71
	P95 (ms)	23.63	23.94	22.80	457.20	15.93
	P99 (ms)	33.64	31.52	771.03	2598.91	98.93
	Max (ms)	442.64	351.38	1755.89	59 250.60	230.34
Constant VU	Duration	0h30m00s	0h30m00s	0h30m00s	0h30m00s	0h30m00s
	Succ. Reqs.	88 467	86 091	143 875	129 169	122 922
	Failed Reqs.	0	0	0	0	0
	Avg. Req. Rate	49.15	47.83	78.93	71.76	68.29
	Avg. Conc. Inst.	1	1	1	1	1
	Mean (ms)	20.04	20.61	12.23	13.59	14.37
	P50 (ms)	19.51	20.02	11.96	13.14	14.19
	P95 (ms)	23.50	24.14	13.73	15.40	15.18
	P99 (ms)	31.13	31.08	16.19	19.78	16.53
	Max (ms)	408.33	787.78	228.85	350.79	566.59

Table G.1: *Performance benchmark results of a simple Go Fibonacci service deployed to AWS Lambda with and without the Lambda Web Adapter (LWA), AWS Fargate, AWS App Runner, and a Hetzner VM for different load scenarios. Metrics provided are: duration of the benchmark; successful and failed requests, average request rate of the client; average number of concurrent instances; mean, p50, p95, p99 and max latencies in milliseconds. * NOTE: the average number of instances for the spike scenario results from a single data point available — this should be considered in further interpretation.*

Long Requests — Go File service

This table is an extension from the previous table Table F.3:

Scenario	Category	Lambda	Lambda w/LWA	Fargate	App Runner	Hetzner VM
Constant Rate $1\frac{req}{s}$	Duration	1h00m00s	1h00m00s	1h00m00s	1h00m00s	1h00m00s
	Succ. Reqs.	3601	3600	3600	3566	3601
	Failed Reqs.	0	0	0	34	0
	Avg. Req. Rate	1.0	1.0	1.0	1.0	1.0
	Avg. Conc. Inst.	9	9	1	6	1
	Mean (ms)	8247.28	8134.69	2558.71	10 431.73	2264.44
	P50 (ms)	8265.20	8161.22	2536.18	9736.36	2222.72
	P95 (ms)	8756.88	8369.35	2777.33	17.170.01	2477.17
	P99 (ms)	8830.71	8454.33	2906.62	20 405.80	2642.75
	Max (ms)	9825.65	9011.45	3043.20	22 084.07	5430.82
Spike*	Duration	0h03m00s	0h03m00s	N/A	N/A	N/A
	Succ. Reqs.	772	744	N/A	N/A	N/A
	Failed Reqs.	318	332	N/A	N/A	N/A
	Avg. Req. Rate	5.59	5.51	N/A	N/A	N/A
	Avg. Conc. Inst.	90	101	N/A	N/A	N/A
	Mean (ms)	31 068.81	30 091.59	N/A	N/A	N/A
	P50 (ms)	31 990.87	27 576.14	N/A	N/A	N/A
	P95 (ms)	57 682.76	57 933.76	N/A	N/A	N/A
	P99 (ms)	59 628.76	59 654.27	N/A	N/A	N/A
	Max (ms)	59 957.10	59 889.66	N/A	N/A	N/A
Constant VU	Duration	0h30m00s	0h30m00s	0h30m00s	0h30m00s	0h30m00s
	Succ. Reqs.	213	213	724	237	767
	Failed Reqs.	0	0	0	0	0
	Avg. Req. Rate	0.12	0.12	0.40	0.13	0.43
	Avg. Conc. Inst.	1	1	1	1	1
	Mean (ms)	8452.72	8457.69	2467.18	7576.48	2326.03
	P50 (ms)	8478.00	8477.55	2438,88	7556.64	2335.10
	P95 (ms)	8682.93	8673.05	2643.66	7750.36	2435.53
	P99 (ms)	8853.88	8791.33	2745.67	7775.01	2644.67
	Max (ms)	8982.20	11 362.45	2950.43	7865.19	2952.47

Table G.2: *Performance benchmark results of a compute-heavy Go File service that hashes, encrypts, and stores files to S3 deployed to AWS Lambda with and without the Lambda Web Adapter (LWA), AWS Fargate, AWS App Runner and a Hetzner VM for different load scenarios. Metrics provided are: duration of the benchmark; successful and failed requests, average request rate of the client; average number of concurrent instances; mean, p50, p95, p99 and max latencies in milliseconds. *NOTE: data for the spike scenario is not available for AWS Fargate, AWS App Runner, and the Hetzner VM due to challenges in achieving comparable results — nonetheless included for completeness for both Lambdas, but no conclusions should be drawn from it.*

Cost and Resource Comparison — Constant Rate Scenario

Pricing for the Hetzner VM was calculated based on information available on the Hetzner Cloud pricing page, see here *for dedicated vCPUs.* For cost calculations of AWS services, the following AWS pricing information for region *us-east-1* was utilized:

- **Lambda**: $0.0000166667\frac{\$}{GBs}; 0.0000002\frac{\$}{invocation}$, see here
- **Fargate**: $0.004445\frac{\$}{GBh}; 0.04048\frac{\$}{vCPUh}$, see here
- **App Runner**: $0.007\frac{\$}{GBh}; 0.064\frac{\$}{vCPUh}$, see here

Service	**Category**	**Lambda**	**Lambda w/LWA**	**Fargate**	**App Runner**	**Hetzner VM**
Fibonacci	Tot. Cost ($\frac{\$}{h}$)	0.04390	0.04706	0.04937	0.07800	0.04660
	Avg. Conc. Inst.	7	7	1	1	1
	Avg. CPU Usage (%)	N/A	N/A	< 4	< 6	< 15
File	Tot. Cost ($\frac{\$}{h}$)	0.84125	0.83035	0.19748	1.87200	0.04660
	Avg. Conc. Inst.	9	9	1	6	1
	Avg. CPU Usage (%)	N/A	N/A	< 60	< 80	< 50

Table G.3: *This table features calculated costs for the Go Fibonacci and File service for the* ***Constant Rate*** *scenario. Metrics provided are: total cost, average CPU resource utilization, and average number of concurrent instances. Total cost was calculated using the above-listed pricing information and based on the total duration of the benchmark and the respective cost per hour for each service. The average CPU usage cap was extracted during benchmarks from AWS's available metrics or the Hetzner Web UI.*