LEVERAGING SERVERLESS CLOUD COMPUTING ARCHITECTURES

Developing a serverless architecture design framework based on best practices utilizing the potential benefits of serverless computing

MASTER THESIS

BOLSCHER, R.T.J. (ROBIN, STUDENT M-BIT)

27 August 2019

UNIVERSITY OF TWENTE.

LEVERAGING SERVERLESS CLOUD COMPUTING ARCHITECTURES

Developing a serverless architecture design framework based on best practices utilizing the potential benefits of serverless computing

Master Thesis

August 2019

Author

Name	R.T.J. Bolscher (Robin)
Programme	MSc Business Information Technology
E-mail	r.t.j.bolscher@student.utwente.nl
Institute	University of Twente PO Box 217 7500AE Enschede The Netherlands

Graduation Committee

Name	Dr. Maya Daneva
E-mail	m.daneva@utwente.nl
Department	Faculty of Electrical Engineering, Mathematics & Computer Science (EWI), Services, Cybersecurity & Safety (SCS)

Name	Prof. Dr. Maria-Eugenia Iacob
E-mail	m.e.iacob@utwente.nl
Department	Faculty of Behavioural, Management and Social Sciences (BMS), Industrial Engineering & Business Information Systems (IEBIS)

Preface

Before you lies the master thesis titled “leveraging serverless cloud computing architectures”. The thesis presents a research effort that mainly consists of an extensive literature analysis. Based on this analysis a framework is constructed that aims to support and improve the design process of serverless cloud computing architectures. It has been written as part of finalizing the Business Information Technology (BIT) master programme at the University of Twente. I have been fully engaged with this research effort from February to August 2019.

Personally, I have always been fond of designing and creating things. Since I was a kid I had the urge to ‘build something’. This finally resulted in me pursuing my bachelor’s degree in Creative Technology. During these studies I discovered a passion for applying software technology to fulfil this urge. Since then I have had the chance to apply this passion in practice as a software engineer in a number of companies, next to pursuing my bachelor’s and master’s degree. This background is what eventually drove me to focus my master thesis research on the topic of serverless cloud computing. I tried to combine my practical experience and academic skills with the goal of creating something useful for software professionals.

I would like to thank my supervisors, Maya and Maria, for their motivating and open-minded guidance during this process. Especially, considering the fact that I was stubborn enough to try and find my own path towards a master thesis project and that this resulted in the direction of my master thesis being somewhat unconventional for a typical BIT student. Additionally, I would like to thank my roommates, friends, family and especially my girlfriend Maaike, who gave me space to focus on my studies and kept supporting me throughout the tough times. Finally, I would like to thank the participants that voluntarily decided to help me evaluate the framework I constructed.

I hope you enjoy reading this thesis.

Enschede, 27 August 2019

Abstract

Serverless computing is a new and interesting cloud computing concept concerned with the deployment of small pieces of software applications and services as serverless functions. Practitioners are already using serverless computing architectures in their products. However, from an academic perspective, to the best of the author's knowledge, a form of coherent accumulation of knowledge around this topic is missing. Serverless computing has a lot to offer to software application architects, developers and owners, however due to the novelty of the technology and the relatively high architectural and economic impact, its adoption is lacking behind or at the very least, is not maximally utilized. To overcome this problem the objective of this thesis is to create a framework for designing serverless architectures which can be used by software architects and developers who want to design a new serverless architecture or migrate an existing application to a serverless architecture. To achieve this objective, an extensive literature review analysis is conducted around various relevant serverless computing topics. This forms the theoretical basis for the framework. The framework consist of four viewpoints (configuration, software design, software architecture and deployment) and 5 cross-viewpoint variables (performance, vendor lock-in, security, costs and serverless suitability), which combined form the proposed framework. The framework is supported by various recommendations and best practices categorized by each viewpoint and cross-viewpoint variable. The framework is evaluated by applying it to a case study and by performing interviews with three domain experts. The goal of the evaluation is to find out if the framework is aligned with the software architecture design and development practice, and if it could leverage the design of serverless architectures. Based on the evaluation can be concluded that the domain experts see added value in the framework and think that it can contribute to creating better serverless architectures more easily, however they struggle with understanding the framework based on the graphical representation and the ambiguousness of the viewpoint definitions. Based on the conclusions of the evaluation the graphical representation of the framework is revised. The goal of the revision is improve the perceived ease of use, and to apply recommendations from the interviewees. The main contributions of this research effort is twofold. First, the proposed serverless architecture design framework (SADF) which has the potential to be useful to practitioners as an up-front design tool as well as in the form of a design and development checklist during the development and implementation process. Second, the accumulation and categorization of relevant knowledge on designing and developing serverless applications and serverless architectures.

Acronyms

API	Application Programming Interface
AWS	Amazon Web Services
BaaS	Backend-as-a-Service
CDN	Content Delivery Network
CRUD	Create Read Update Delete
DDOS	Distributed Denial of Service
DNS	Domain Name System
DSRM	Design Science Research Methodology
FaaS	Function-as-a-Service
I/O	Input/output
IoT	Internet-of-Things
IS	Information Systems
SADF	Serverless Architecture Design Framework
SOA	Service Oriented Architecture
TAM	Technology Acceptance Model
TOSCA	Topology and Orchestration Specification for Cloud Applications
UI	User Interface
VM	Virtual Machine
VPC	Virtual Private Cloud

Table of Figures

Table of Tables

Table of Contents

Chapter 1. Introduction

This chapter will present an introduction to serverless computing (Section 1. 1.), and presents the summary of a literature review on the potential benefits and challenges of the serverless computing technology (Section 1. 2.).

1. 1. Introduction to serverless computing

Serverless computing is a new and interesting cloud computing concept concerned with the deployment of small pieces of software applications and services (Baldini, Castro, et al., 2017). Since 2014 many of the tech giants have invested in this new cloud computing paradigm resulting in various serverless computing cloud services, for example AWS Lambda (Amazon Web Services), AWS Cognito (Amazon Web Services), Google Cloud Functions (Google), Google Firebase (Google), IBM Cloud Functions (IBM), Microsoft Azure Functions (Microsoft), Apache OpenWisk (Apache) and Auth0 (Auth0). Furthermore, Gartner reports that ”the value of [serverless computing] has been clearly demonstrated, maps naturally to microservice software architecture, and is on a trajectory of increased growth and adoption” (Lowery, 2016).

1. 1. 1. Definition of serverless computing

According to Baldini, Castro, et al. (2017) the term ‘*serverless computing’* is coined by the technology industry, therefore it lacks a standardized formal definition. The academic community only recently started giving attention to this topic, a search for “serverless computing” on Scopus shows 63 publications since 2016, with the vast majority in 2017 (20) and 2018 (36). Two definitions of serverless computing, stemming from 2017 and 2018, were found:

1. “[Serverless computing is] a programming model and architecture where small code snippets are executed in the cloud without any control over the resources on which the code runs” (Baldini, Castro, et al., 2017).
2. “Serverless computing is a form of cloud computing that allows users to run event-driven and granularly billed applications, without having to address the operational logic” (Van Eyk et al., 2018).

Both definitions touch upon some of the key features of serverless computing:

1. Allows for granularly billing by cloud service providers.
2. An event-driven programming model and architecture.
3. Handles most, if not all, operational concerns.
4. Removes control over resources.

There are some additional key features not completely covered by the definitions though, for example the following mentioned by Chapin and Roberts (2017):

5. Often short-lived.

6. Auto-scaling and provisioning.
7. Implicit high availability.
8. Implicit fault tolerance.

Both definitions of serverless computing are quite accurate and overlapping, even though they use different terminology. For the purpose of the research in this thesis we will consider the term '*serverless computing'* as indicated below. It is grounded on the definitions of Baldini, Castro, et al. (2017) and Van Eyk et al. (2018):

"A form of cloud computing that removes all operational concerns for users while allowing granular scaling and billing by supporting a programming model and event-driven software architecture that uses small code functions as individual deployment units."

It is important to note that serverless computing is a misnomer in the sense that serverless computing does not actually refer to the absence of servers in cloud computing (Jonas, Pu, Venkataraman, Stoica, & Recht, 2017; Spillner, 2017). From a software developer's perspective the name seems more appropriate; serverless computing abstracts away most resource related concerns (e.g. over-/under provisioning, fault tolerance, scalability, deployment) in such a way that the developers do not have to think about the server to which their code will be deployed (Baldini, Castro, et al., 2017; Hendrickson et al., 2016; Varghese & Buyya, 2018; Villamizar et al., 2017).

Chapin and Roberts (2017) categorize two different types of serverless computing implementations: (1) Backend-as-a-Service (BaaS) and (2) Function-as-a-Service (FaaS).

1. BaaS is concerned with replacing server side components that were usually implemented and deployed by the software application owner through off-the-shelf external third party services. BaaS services expose their functionality, such as a managed database cluster (e.g. Google Firebase (Google)) or authentication components (e.g. Auth0 (Auth0) and Amazon Cognito (Amazon Web Services)), through an API.
2. FaaS is a new method of creating and deploying server-side software applications, based on individual functions as deployment unit. Examples of FaaS providers are AWS Lambda (Amazon Web Services), Google Cloud Functions (Google), IBM Cloud Functions (IBM), Microsoft Azure Functions (Microsoft) and Apache OpenWisk (Apache). The most widely adopted FaaS implementation currently available is AWS Lambda (Amazon Web Services), many people refer to FaaS implementations as serverless computing (Chapin & Roberts, 2017). However, FaaS is just one implementation of the serverless computing concept (Chapin & Roberts, 2017; McGrath, Short, Ennis, Judson, & Brenner, 2016; Varghese & Buyya, 2018; Yan, Castro, Cheng, & Ishakian, 2016).

Initially BaaS and FaaS seem quite different, the reason to group these two types of services together, as stated by Chapin and Roberts (2017), is that "neither require you to manage your own server hosts or server processes". Even though BaaS and FaaS have some common benefits an

explicit distinction will be made in this thesis due to the technical and architectural differences. Therefore, the main subject of this thesis will be based on the serverless computing implementation FaaS, whenever BaaS is discussed this will be explicitly mentioned.

1. 1. 2. Introduction to serverless functions

A serverless function is the basic deployment unit of FaaS. Whereas conventional cloud computing considers large all-encompassing software applications (or services in case of Service Oriented Architectures (SOA) or microservice architectures) as the basic deployment unit, FaaS allows individual deployment of functions or operations (Chapin & Roberts, 2017).

These functions are all separately deployed in lightweight containers which can individually be started (provisioned) the moment the cloud service provider receives an event that wants to use that specific function of the software application (McGrath & Brenner, 2017; Varghese & Buyya, 2018). Containers are the next evolution in lightweight virtualization techniques for cloud deployment (Pahl, Brogi, Soldani, & Jamshidi, 2018), and pose several advantages over traditional Virtual Machine (VM) based deployment such as quick provisioning (server start up time) and less storage and processing overhead (Hendrickson et al., 2016; Tosatto, Ruiu, & Attanasio, 2015).

A simplified example of a serverless function is demonstrated in Figure 1, it shows a simplified function that handles an incoming event and sends a text message using the provided event data via a third-party text message service provider named Twillio (Twillio). Twillio itself can be considered an implementation of serverless computing, since it functions as a BaaS by providing various communication related functionalities through an API on a pay-per-use basis.

```
/**
 * Serverless function exported in sendTextMessage.js.
 * This method takes an event object parameter with text and number properties and
 * passes this on to Twillio (a third-party text message service provider).
 * @param {Object} event
 * @param {string} event.text - The message to send
 * @param {string} event.number - The mobile phone number to send to
 * @param {Object} context
 * @param {Function} callback - Method which sends a (status) response back
 */
module.exports.handler = async (event, context, callback) => {
   if (typeof event.text === 'string' && typeof event.number === 'string') {
      await sendTextMessageViaTwillio({ text: event.text, number: event.number });
      return callback(null, {
         statusCode: 200, // Success
      });
   }

   // Handler was called with invalid event data
   return callback(new Error('Invalid text or number provided'), {
      statusCode: 400, // Bad Request
   });
};
```

Figure 1 Serverless function example

An important difference between conventional long-running software applications (e.g. microservices) is that serverless functions are inherently stateless (Chapin & Roberts, 2017; McGrath et al., 2016). This is enforced by the life cycle of the function instances themselves; they are provisioned on request and deprovisioned whenever possible. Any temporary state stored in memory would be deleted upon deprovisioning. This brings challenges for software developers (forced to use external persistency services), but also opportunities, due to the stateless nature of serverless functions they can be horizontally scaled extremely easy, merely by increasing concurrency (Chapin & Roberts, 2017).

The process of consuming a serverless function is shown in Figure 2 and can be summarized on a high abstraction level as follows:

1. Cloud service provider detects an incoming event.
2. A container will be created running the function.
3. The event will be routed to this container once it has started.
4. The function will process the event.
5. Finally the container will be terminated.

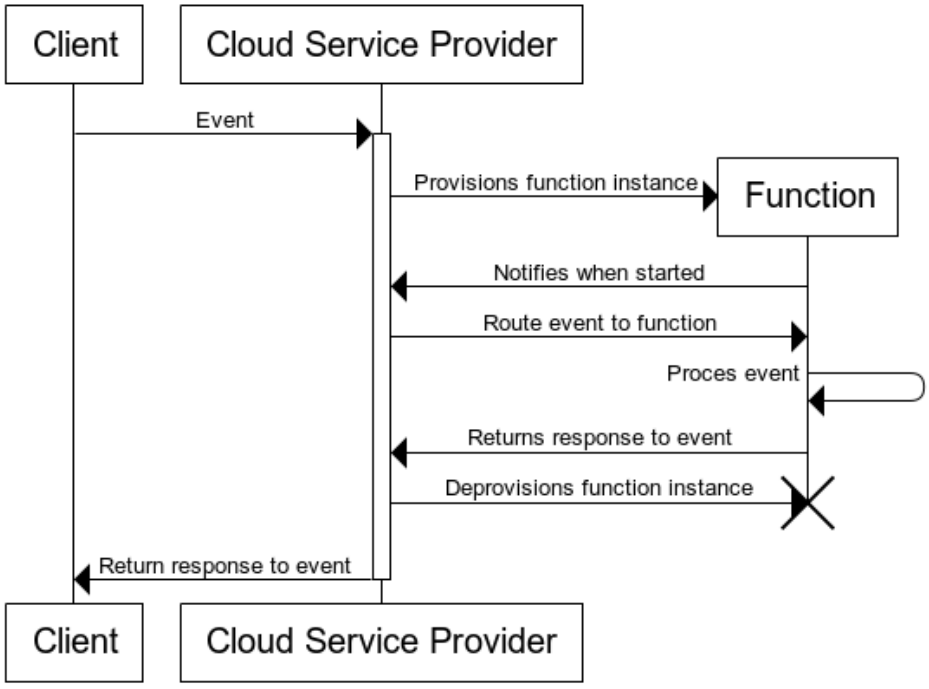


Figure 2 Sequence diagram of basic FaaS consumption

This whole process is executed in a very short time. Although it depends on many different factors such as programming language, dependencies, and required memory. The most time consuming part of this process is the creation of the container running the function (Varghese & Buyya, 2018), this is also called a “cold-start”; the time it takes the cloud service provider to create the container and starting the function that the incoming event needs access to. Most cloud service providers

alleviate this problem by keeping the function "warm" for an arbitrary number of minutes after it has been used; basically the container, including the function, is not directly destroyed after use, but kept idle for some time. This eliminates cold start delay since the container can be re-used.

1. 2. Benefits and challenges of serverless computing

Serverless computing offers several benefits over conventional cloud deployment. This paragraph will shortly address the most prevalent benefits found in recent literature, more detailed analysis can be found in Section 3. 2.

Whereas conventional cloud deployment of monolithic applications poses scaling issues, a microservice or serverless computing architecture allows for far more granular scaling and therefore optimizes resource utilization (Van Eyk et al., 2018; Villamizar et al., 2017). Because of the optimized resource utilization, over and under provisioning is minimized, resulting in reduced operational costs. In contrast to conventional VM based deployments, where the application owner pays for the allocation of the resources whether it is being used or not (Varghese & Buyya, 2018), serverless computing offers a cost model based on execution time rather than resource allocation (Adzic & Chatley, 2017; Eivy, 2017; Villamizar et al., 2017). The latter further aligns the operational costs to actual business demand.

Furthermore, serverless computing, and FaaS in particular, presents novel architectural opportunities for software developers and software architects such as inherent fault tolerance, effortless parallelisation of computing and the composition of various serverless functions to create business value through workflows (Jonas et al., 2017; van Eyk, Iosup, Seif, & Thömmes, 2017; Varghese & Buyya, 2018; Wagner & Sood, 2016; Yan et al., 2016).

Finally, serverless computing potentially improves the development process by reducing lead time (Adzic & Chatley, 2017; Baldini, Castro, et al., 2017; Chapin & Roberts, 2017), reducing risk (Chapin & Roberts, 2017; Ivanov & Smolander, 2018), and allowing software developers to choose the best language, dependencies and tools for a specific serverless function, in contrast to being stuck with the decisions made many years ago in a different context in the case of a monolithic application (Van Eyk et al., 2018).

Serverless computing also poses some severe challenges compared to conventional cloud deployment. This paragraph will shortly address the most prevalent benefits found in recent literature. More detailed analysis can be found in Section 3. 3.

First, there are various performance related challenges to the application of serverless computing in production environments, for example long wait-times due to cold starts (Baldini, Castro, et al., 2017; Chapin & Roberts, 2017; van Eyk et al., 2017), increased complexity in managing performance issues due to the underlying serverless platform which is out of control of the application owner and managed by the cloud provider (Chapin & Roberts, 2017), and more in

general, the unpredictability of FaaS performance makes it difficult to provide Quality-of-Service guarantees which are often desired in business contexts (Yan et al., 2016).

Second, serverless computing brings new or additional development complexities. Developers need to update their knowledge on the concept and learn to work with new tools specific to FaaS development and deployment (Chapin & Roberts, 2017; van Eyk et al., 2017), and monitoring, debugging and versioning becomes more difficult in a serverless context (Baldini, Castro, et al., 2017; McGrath et al., 2016; van Eyk et al., 2017; Villamizar et al., 2017).

Third, FaaS brings the risk of potential vendor lock-in according to various authors (Adzic & Chatley, 2017; Baldini, Castro, et al., 2017; Chapin & Roberts, 2017; Eivy, 2017; Villamizar et al., 2017). Serverless functions are often developed for a specific FaaS platform and might not be deployable to other platforms without necessary changes, besides, FaaS platforms offer a lot of additional services such as platform specific storage adapters, scaling, monitoring and logging tools, authentication components etc., when serverless functions are becoming dependent on these platform provided services it becomes increasingly more difficult to move to another FaaS platform eventually.

Fourth, it is not trivial to decompose existing applications into serverless functions (Hendrickson et al., 2016; McGrath et al., 2016). The ability to do this is necessary to move existing (monolithic) applications towards a microservice or serverless oriented architecture.

Finally, the potential operational cost reductions are not so straightforward as they might appear. According to Eivy (2017) the cost reductions “heavily depend on the execution behaviour and volumes of the application workloads”. Therefore, serverless computing needs to be applied with care to prevent unexpected high costs.

Concluding, serverless computing has the potential to be of great benefit to software developers, software architects and software application owners, by improving the software quality while reducing costs. However, the challenges presented above need careful consideration on a per-case basis in order to ensure the successful application of serverless computing.

1. 3. Problem statement

The problem central to this thesis is based on the novelty of the serverless computing architecture. Due to this, researchers and practitioners have yet to establish a knowledgebase with regard to implementation standards, cost saving techniques, performance optimizations and more general best practices with regard to implementing serverless computing architectures. Practitioners are already using serverless computing architectures in their products. However, from an academic perspective, to the best of the author’s knowledge, a form of coherent accumulation of knowledge around this topic is missing. This claim is substantiated by van Eyk et al. (2017) who propose the interesting research challenge of creating a reference architecture for serverless computing architectures which “would provide developers and researchers with an understanding of the main

components shared by FaaS platforms, facilitating new deployments and enabling comparisons of designs". The proposed reference architecture is just an example of the gap in usable knowledge with regard to serverless architectures. Because of this missing or scattered knowledge, practitioners are, for example, likely to underutilize, misuse, implement anti-patterns, or generate substantially more costs than would be necessary and by doing so forgoing the envisioned benefits of serverless computing.

1. 4. Research motivation

The most desirable benefits of serverless computing are aligning operational costs with actual business demand and cost savings in general, but also more secondary benefits are identified such as improved agility, reduced lead time, et cetera.

Many businesses currently are not using serverless computing technology or are underutilizing it due to the many unknowns such as complex cost structures, performance challenges and uncertainties regarding the implementation of this concept into the current application architecture.

The motivation of this research effort is that serverless computing has a lot to offer to software application architects, developers and owners, however due to the novelty of the technology and the relatively high architectural and economic impact, its adoption is lacking behind or at the very least, is not maximally utilized. Switching to a serverless architecture requires a thoughtful switch, similar to the one architects and developers had to make when microservices where becoming a more common architectural style. It certainly is not a one-size-fits-all kind of approach. A successful serverless computing adoption depends on many factors and requires many important nonarbitrary considerations.

I hope this work can contribute to the successful adoption of serverless computing by providing software architects, developers and owners with the handlebars, insights and references to make informed decisions for their specific applications and to maximize the potential of this interesting new approach to deploying software in the cloud.

1. 5. Research objective

In order to improve the serverless computing adoption, and maximize utilization of the corresponding benefits, the objective of this thesis is *to create a framework for designing serverless architectures which can be used by software architects and developers who want to move an existing application to a serverless architecture or who want to design a new serverless architecture from scratch.* Additionally, this research aims to be interesting for software application owners as well due to the high economic impact of the serverless software architecture.

To achieve this objective a number of consecutive steps have to be taken. First, a large literature review analysis has to be executed. This forms the theoretical basis for the framework that will be developed. The framework will then be evaluated by applying it to a case study and by performing

interviews with a number of domain experts. The goal of the evaluation is to find out if the framework is aligned with the software architecture design and development practice, and if it could leverage the design of serverless architectures.

Chapter 2. Research methodology

This chapter is dedicated to describing the research methodology that is used for this research. It describes the proposed research questions (Section 2. 1.), the research process that will be followed to answer these research questions (Section 2. 2.), the expected contributions of this thesis (Section 2. 3.) and presents the structure of the remainder of this thesis (Section 2. 4.).

2. 1. Research questions

Based on our research objective, a number of core research questions were defined as follows:

RQ1. What are the currently known benefits and challenges of serverless computing?
RQ2. What are the existing approaches or characteristics to determine which parts of an application are suitable to be implemented in a serverless way?
RQ3. How could cloud native architectures be composed and orchestrated to enable the full potential of serverless computing?
RQ4. What are the best practices for creating a serverless application from scratch, and for migrating a non-serverless architecture to a serverless architecture?
RQ5. How can these best practices be applied in a form that supports software architects and developers in the process of designing a serverless application architecture?

In the following paragraphs the reasoning and goals of the proposed research questions are further elaborated.

2. 1. 1. Research question 1

In order to apply serverless computing it is key to know what this technology has to offer in addition to the traditional commonly used cloud computing concepts. Besides, knowing which challenges the technology currently faces creates a new level of insight which is useful when thinking about the questions proposed in RQ4 and RQ5.

2. 1. 2. Research question 2

It is important to be able to understand how applications could be decomposed into functions with the goal of optimizing resource usage in a serverless environment. The goal of this research question is to provide a set of characteristics that can be used in determining whether a certain part of a system, a functionality, or an arbitrary piece of code could benefit from being extracted to a serverless function. This results in knowledge that might be of value for RQ4, since it lays the foundation for a best practice related to migrating to a serverless architecture, and even for creating a new serverless application from the ground up.

2. 1. 3. Research question 3

This question looks at how the granular serverless building blocks can be composed and orchestrated into large cloud native architecture solutions in such a way that the full potential of serverless computing is achieved. Important to consider for this question is the concept of a hybrid cloud architecture.

Hybrid cloud applications (a combination of various types of cloud computing, on-premise, private/public cloud, third-party services, linked together through orchestration) pose a challenge because of the heterogenous nature of the system. It is important to know how to use and develop these different sorts of cloud computing services in conjunction, and to find out how to maximize the benefits of each individual sort of service to create economical, easy-to-develop and highly performant systems. New approaches might be required to assemble serverless components into larger systems that are easier to understand and reason about.

The goal of this research question is to gather knowledge on the various possibilities regarding composability of serverless applications.

2. 1. 4. Research question 4

This research question focusses on defining best practices for creating serverless applications from scratch and the migration from non-serverless architectures towards hybrid serverless architectures. This question heavily relies on RQ2 and RQ3, which already propose some important characteristics of this process.

The goal of this research question is to formalize the body of knowledge concerned with the creation and/or migration of hybrid serverless applications into a set of considerations and choices which can assist software architects and developers when faced with similar challenges.

2. 1. 5. Research question 5

This research question analyses and combines the knowledge gathered by RQ1, RQ2, RQ3 and RQ4. Additionally, it looks at how to create a framework which implements the said knowledge into a form that is usable by software architects and software developers. The goal is to create a serverless architecture design framework which will aid practitioners in the process of designing a serverless architecture while applying best practices found in literature. This framework will be evaluated by applying it to a case study which entails the design of a serverless application architecture and by performing semi-structured interviews with a set of practitioners.

2. 2. Research process

This research is based on the Design Science Research Methodology (DSRM) as described by Peffers, Tuunanen, Rothenberger, and Chatterjee (2007). This research effort can be roughly divided into two parts. The first is the theoretical literature research (steps RMS1 through RMS4)

which will be performed using a structured review approach as proposed by Webster and Watson (2002). The second is concerned with applying this knowledge to design, demonstrate and evaluate a framework (steps RMS5, RMS6 and RMS7).

The DSRM was chosen since it offers a formal method for doing design science research specifically for the Information Systems (IS) domain with a focus on producing an artefact that addresses a predefined problem. The DSRM describes six distinct process steps:

1. Problem identification and motivation
2. Definition of the objectives for a solution
3. Design and development
4. Demonstration
5. Evaluation
6. Communication

The first two DSRM steps are covered by respectively Sections 1. 3. “Problem statement”, 1. 4. “Research motivation”, and 1. 5. “Research objective”. The third, fourth and fifth steps are addressed in the remaining chapters of this thesis. The sixth step is not specifically addressed, however this thesis as a whole can be considered to represent the communication step as it presents and discusses the results of the application of the DSRM. To answer the research questions and reach the research objective a number of consecutive research methodology steps (RMS) have to be taken which are described below.

Part 1 – Background and literature research

RMS1. Literature research on the benefits and challenges of serverless computing.

This step creates the theoretical basis upon which can be built in the next steps by answering RQ1. A large part of this step is already set forth in Chapter 1.

RMS2. Literature research on identifying characteristics that makes chunks of applications suitable to benefit from serverless computing.

The goal of this step is to answer RQ2 by delivering a set of characteristics that can help software developers and architects to determine if a piece of code could/should be deployed serverless or not.

RMS3. Literature research on the composition, orchestration/choreography of (hybrid) serverless applications.

This step will produce techniques and methods on how individual serverless function can be composed into a large coherent software application. Important is to take the hybrid cloud approach into account where serverless functions co-exist and cooperate with other forms of cloud native architectural components.

RMS4. Literature research on best practices for creating a serverless application from scratch and for migrating a non-serverless architecture to a serverless architecture.

The goal of this step is to gain insight into best practices for creating serverless application architectures. This is needed in order to implement this knowledge into a form that can be used by software architects and developers.

Part 2 – Design, demonstration and evaluation of framework

RMS5. Create a serverless architecture design framework.

Deliverable of this step is a serverless architecture design framework created using the gathered knowledge (steps RMS1 through RMS4) which serves as a basis for the next steps of this research. For this step the DSRM is utilized.

RMS6. Apply and evaluate the serverless architecture design framework.

This step entails an application and evaluation of the serverless architecture design framework. It will be applied on a case study which will function as a first evaluation of the framework.

RMS7. Evaluation of the serverless architecture design framework with domain experts.

The final step of this research method is the evaluation of the serverless architecture design framework produced by the DSRM in steps RMS5 and RMS6. This will be done by performing semi-structured interviews with a number of domain experts; preferably three to four. According to Boyce and Neale (2006) "in-depth interviews are useful when you want detailed information about a person's thoughts and behaviours". Since this is exactly the goal of the evaluation, semi-structured interviews are chosen as evaluation method. Moreover, this type of interview was chosen due to the rather broad nature of the problem domain. Additionally, it is hoped that the domain experts will take the opportunity given by semi-structured interviews to divert from the topic where needed to provide valuable insights and knowledge the researchers could not have predicted beforehand. Additionally, the Technology Acceptance Model (TAM) by Davis, Bagozzi, and Warshaw (1989) will be applied to measure some important aspects of the proposed artefacts such as perceived usefulness and perceived ease-of-use.

Figure 3 presents a process model of the proposed research methodology steps. The columns represent the steps described by the DSRM. The first column describes steps in the research process that have already been presented in Chapter 1. The first four RMS are performed serially in order to further build upon the knowledge each step provides. These first four RMS are considered to be the basis of the third DSRM step "design and development". The fifth and sixth step are iterative steps and together with the seventh step cover the rest of the third, fourth and fifth DSRM steps.

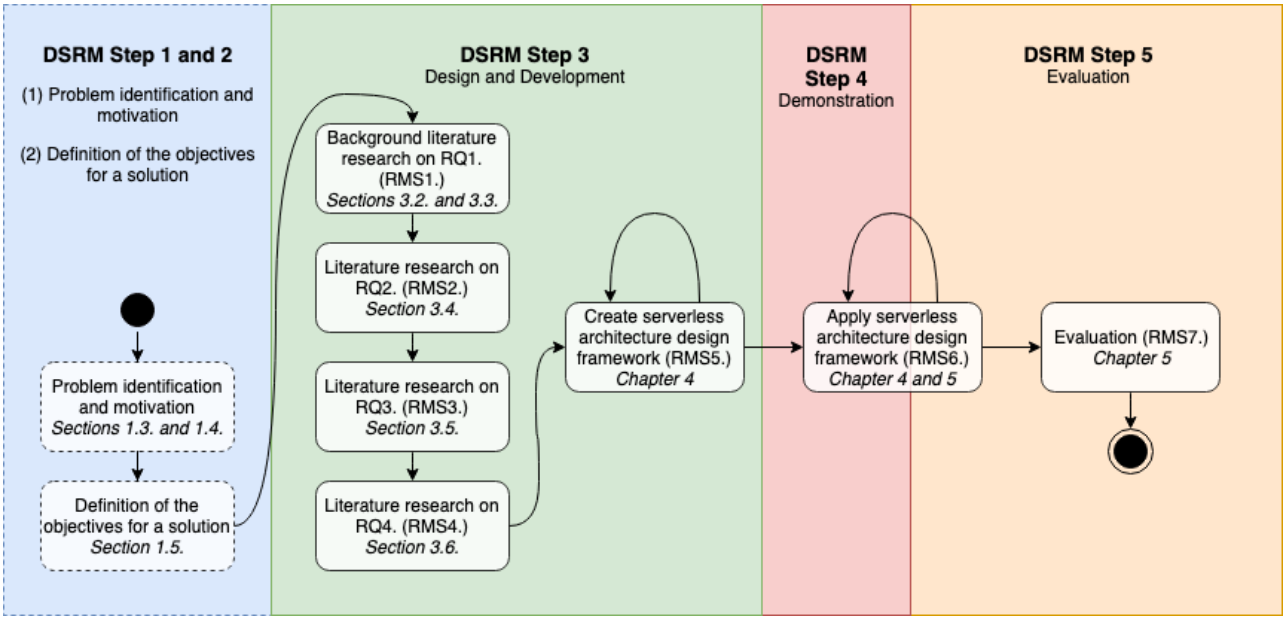


Figure 3 Research methodology process overview

2. 3. Expected contributions

The research efforts in this work are expected to have a twofold contribution to both researchers and practitioners in the area of serverless computing.

First, an extensive literature review will be conducted on the topic of serverless computing and the proposed research questions RQ1, RQ2, RQ3 and RQ4. This is expected to contribute to the state-of-the-art on this topic by synthesizing the relevant literature into a comprehensive overview and making relevant connections between the accumulated knowledge.

Second, the serverless architecture design framework produced by applying the DSRM to RQ5 is expected to be of value to researchers and practitioners. The contribution here is twofold, it describes the creation of the serverless architecture design framework as well as its application onto a case study which demonstrates how a serverless architecture could be designed using this framework. For researchers it demonstrates a more practical approach to the application of serverless computing which might be used as a basis for further research, additionally it points out what research areas of the serverless computing topic need further attention from the research community. For practitioners, such as software developers, software architects and software application owners, it provides valuable insights into the various ways serverless computing could be adopted, what common pitfalls and existing challenges are, and how best practices could be applied.

2. 4. Thesis structure

This thesis is organized as follows. Chapter 3. presents the literature review that serves as the theoretical foundation of this work. Chapter 4. describes the creation of the serverless architecture design framework (SADF). Chapter 5. describes the case that will be used to apply and evaluate the serverless architecture design framework. Chapter 6. presents the demonstration and evaluation of the SADF based on a case study and interviews. Chapter 7. will conclude this work by summarizing the knowledge and experiences gathered during the execution of this research, additionally it will present recommendations for practitioners and researchers. Finally, Section 7. 6. will discuss some of the most important limitations to this research effort.

Table 1 presents a mapping of the research methodology steps (RMS) as presented in Section 2. 2. onto the research questions and the DSRM steps. This shows the direct relation between the actions that will be performed during this research effort and the research questions they try to address. Additionally, the final column describes in what specific section of this thesis the research question and corresponding RMSs will be addressed.

Research question	**Research methodology step**	**DSRM Step**	**Addressed in section**
RQ1	RMS1	3 - Design and development	Sections 3. 2. and 3. 3.
RQ2	RMS2	3 - Design and development	Section 3. 4.
RQ3	RMS3	3 - Design and development	Section 3. 5.
RQ4	RMS4	3 - Design and development	Section 3. 6.
RQ5	RMS5	3 - Design and development	Chapter 4.
	RMS6	4/5 - Demonstration and evaluation	Chapter 6.
	RMS7	5 - Evaluation	Chapter 6.

Table 1 Mapping of research methodology steps onto research questions

Chapter 3. Literature review

This chapter presents the main theoretical contribution of this thesis which consists of several separate literature reviews on the subjects of: benefits and challenges of serverless computing (resp. Section 3. 2. and Section 3. 3.), characteristics of serverless suitability (Section 3. 4.), composition of serverless applications (Section 3. 5.) and finally best practices for creating and migrating towards a serverless architecture (Section 3. 6.). This chapter will conclude with the implications of the found literature for this work by answering the research questions as presented in Section 2. 1.

3. 1. Literature review process

There are various separately executed literature reviews presented in this chapter. Even though they were executed separately they followed the same review process: a structured review approach as proposed by Webster and Watson (2002). The literature reviews presented here are not exhaustive and are merely structured (and not systematic) in order to quickly create a broad base of information which in turn can be built upon later on in this research effort. Roughly the following steps were executed for each literature review:

1. Establish search keywords
2. Execute search on Scopus
3. Select relevant and recent papers by reading titles and abstracts
4. Select useful papers by reading them thoroughly
5. Based on the resulting papers execute additional explorative literature research

The main scientific literature source used is Scopus, however, in some cases (especially the explorative review part) Google Scholar was used as additional source.

3. 1. 1. Literature review process RQ1

This section shortly addresses the structured literature review for RQ1 which tries to answer the question of "What are the currently known benefits and challenges of serverless computing?". This literature review is addressed in both Section 3. 2. and Section 3. 3. This literature research was very much explorative, as it served as the initial deep dive into the subject matter. Notwithstanding, the following search queries were used as basis for this explorative research:

1. "serverless AND benefits"
2. "serverless AND advantages"
3. "serverless AND challenges"
4. "serverless AND issues"

3. 1. 2. Literature review process RQ2

This section describes the literature review executed for the purpose of answering RQ2 "What are the existing approaches or characteristics to determine which parts of an application are suitable to be implemented in a serverless way?". The result of this review is described in Section 3. 4. The following search queries were executed:

1. "serverless AND computing"
2. "serverless AND code"
3. "serverless AND computing AND code AND characteristics"
4. "serverless AND suitable"
5. "serverless AND code AND suitability"

3. 1. 3. Literature review process RQ3

This section presents the structured literature review executed in order to answer RQ3 "How could cloud native architectures be composed and orchestrated to enable the full potential of serverless computing?". The results are presented in Section 3. 5. The following search keywords were used:

1. "serverless AND composition"
2. "serverless AND architecture"
3. "serverless AND orchestration"

This resulted in respectively 6 papers (1st query), 107 papers (2nd query) and 9 papers (3rd query). Due to the large number of results for the second search query, the fact that the papers were spread over a large time period (2009 to 2019), and the large concentration of publications in 2017 (22) and 2018 (36), was decided to only include papers for this query from 2017 through 2019.

After filtering all publications based on their abstracts (and duplicate results) there were 2 papers (1st query), 19 papers (2nd query) and 5 papers (3rd query) left. Next, the papers were read thoroughly and selected based on their relevance towards to topic of state-of-the-art serverless composition. This resulted in a final 13 papers; 2 papers (1st query), 9 papers (2nd query) and 2 papers (3rd query).

Some additional papers were found through explorative literature research or originate from earlier stages of this research, these papers are incorporated into the text where applicable.

3. 1. 4. Literature review process RQ4

This section presents the structured literature review executed in order to answer RQ4 "What are the best practices for creating a serverless application from scratch, and for migrating a non-serverless architecture to a serverless architecture?". The results are presented in Section 3. 5. The following search keywords were used:

1. "serverless AND best-practices"

2. “serverless AND best practices”
3. “serverless AND guidelines”
4. “serverless AND migration”

This resulted in respectively 2 papers (1st query), 6 papers (2nd query), 1 paper (3rd query) and 12 papers (4th query). After reading the papers’ title and abstract there was respectively 1 paper (2nd query) and 2 papers (4th query) left that were expected to be relevant to this literature review. Reading the papers thoroughly resulted in 2 papers that were included in this review.

Since the academic literature yielded very little results, also (exploratively found) grey literature is included for this review (e.g. blog posts by practitioners, company presentations and white papers). Additionally, since a lot of literature research on the topic of serverless computing has preceded this review, a lot of relevant information was extract from papers that were found through previous reviews related to research questions RQ1, RQ2 and RQ3. The grey literature and the previously found literature complementary make up for the lack of paper results for this specific research question.

3. 2. Benefits of serverless computing

This section is concerned with answering the first part of RQ1 “What are the currently known benefits and challenges of serverless computing?”. Serverless computing offers several benefits over conventional cloud deployment; granular scaling ability, optimisation of resource utilization, reduction of operational costs, architectural opportunities and improvements to the development process. This paragraph presents a non-exhaustive summary of the most important benefits found in literature.

3. 2. 1. Granular scaling opportunities

The nature of serverless computing allows for very granular scaling compared to more conventional types of cloud computing. As noted by Villamizar et al. (2017) and Van Eyk et al. (2018) scaling a monolithic application is difficult since it comprises of many different services, one service might be in high demand and therefore unnecessarily degrading the performance of all constituent services. The only solution is to scale the whole monolithic application, even though some services might be sitting idle (Van Eyk et al., 2018; Villamizar et al., 2017).

Microservices, as described by (Lewis & Fowler, 2014), offer a solution to the aforementioned scaling issues of monolithic applications. Basically, the monolith is decomposed into the various services it consists of, these services are then deployed separately, and can therefore be scaled separately as well (Villamizar et al., 2017). Even though this is a big improvement over monolithic applications, there are still some benefits to be gained; microservice cloud architectures require quite some effort when it comes to deploying, operating and scaling (Villamizar et al., 2017). Ideally, from the perspective of software application owners, these activities would be handled by the cloud service provider as well.

This is why companies like Amazon started creating products like AWS Lambda (Amazon Web Services) to fill that gap. These serverless computing services enable creating microservices, without the need to manage servers, that can easily be deployed and are automatically scaled (Villamizar et al., 2017).

Another benefit stemming from the granular scaling capabilities is the ability to parallelize heavy workloads almost instantly as demonstrated by McGrath et al. (2016) and their Trek10 media management service application (MMS) example. The Trek10 MMS is able to re-size several hundred images in about the same time it would take to re-size just one image. This has been realized by applying a fan-out approach; a "master" function takes the image resize request and invokes a new "worker" function to perform the CPU intensive task of resizing the image. This parallelization reduces the total execution time of resizing hundreds of images as each image is being resized by a different serverless computing instance.

An important note to make here is that scaling serverless computing functions is done automatically by the cloud service provider, without any required configuration (Chapin & Roberts, 2017). Depending on the cloud service provider there are some scaling configuration options, however the gist is that for each incoming request a new function instance will be created, running on its own resources, hence the automatic and configuration-less scaling. Because of its automatic scaling capability, serverless computing is especially useful in contexts where demand is hard to predict and experiences large spikes and drops.

3. 2. 2. Optimizing resource utilization

The most obvious benefit of serverless computing, and actually a result of the granular scaling opportunities, is the improved alignment of resource usage to actual demand. In conventional VM based deployments a server has to be continuously running the deployed software application, moreover, the resources available to the VM must be of such performant level in order to be able to handle peak demand at all times which results in continuous high costs and unnecessary over-provisioning (Chapin & Roberts, 2017).

There are some solutions that allow for better aligning the available computing resources with actual demand such as Amazon EC2 Auto Scaling (EC2 is an elastic on-demand cloud computing service based on VMs) (Amazon Web Services). EC2 Auto Scaling allows users to set certain resource or usage metric related thresholds (e.g. more than 80% CPU used, more than 1 million requests per minute per instance) which will trigger deployment of more instances when crossed. Unfortunately there are various downsides to this approach, for example deploying extra instances can take (more than) several minutes (Van Eyk et al., 2018), which might result in severe performance issues even before new instances could be deployed. Even worse, from a cost perspective, would be when new instances are only finished deploying after the peak load has already began to drop; incurring costs but no performance improvement.

Serverless computing optimizes resource utilization by being extremely granularly scalable (Van Eyk et al., 2018). For each incoming event a new function instance is provisioned (if there are no free warm instances available at that moment), meaning that serverless computing scales horizontally almost instantly compared to demand (Chapin & Roberts, 2017). This results in basically zero under-provisioning, and due to the fact that instances are automatically deprovisioned when not used, there is also zero over-provisioning. The optimization of resource utilization of serverless computing compared to conventional VMs is demonstrated in Figure 4 as presented by Van Eyk et al. (2018).

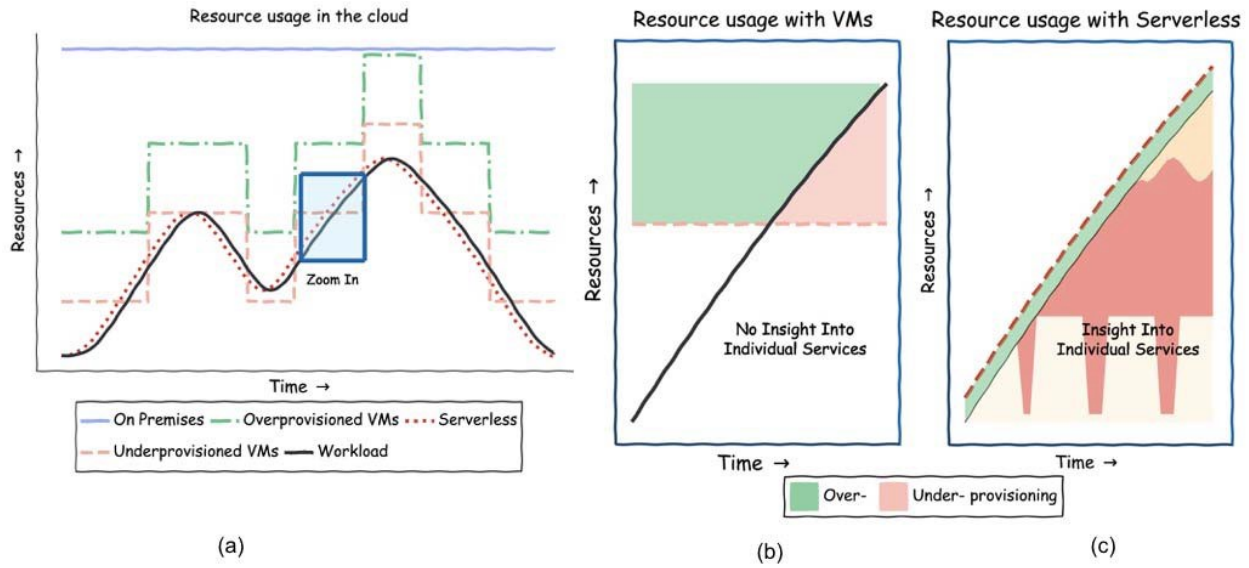


Figure 4 Optimization of resource utilization by serverless computing compared to conventional VMs deployment. (a) Resource utilization in the cloud. (b) Zoom-in of resource utilization in the cloud with VMs. (c) Zoom-in of resource utilization in the cloud with serverless © 2018 IEEE

Another benefit of serverless computing is the implicit built-in fault tolerance. If an error occurs within a serverless computing instance when processing an event, the instance is restarted and provided the same input in order to retry (Jonas et al., 2017). This is related to optimizing resource utilization since the conventional approach to handle fault tolerance is to deploy fail-over instances, which doubles the operational costs and is part of over-provisioning (Singh, Singh, & Chhabra, 2012).

3. 2. 3. Reducing operational costs

Conventional cloud computing requires a software application to be hosted on a VM or inside a container (Varghese & Buyya, 2018). The software application owner pays for the whole time the VM or container is running the software application, regardless of whether it is being used or not; a cost model of 'per VM per hour', ignoring idle time (Varghese & Buyya, 2018).

When the software application would be deployed serverless, the owner would pay very granularly based on actual usage of the software application. Serverless computing even allows 'scaling to

zero' (Baldini, Castro, et al., 2017), meaning that cloud service providers can destroy all deployed instances resulting in zero costs for the software application owner as long as it is not being used. The cost model for FaaS is based on execution time rather than resource allocation (Adzic & Chatley, 2017; Eivy, 2017; Villamizar et al., 2017). Different cloud service providers use different billing schemes, an example taken from AWS (Amazon Web Services) shows that depending on the required RAM (ranging from 128MB to 3008MB) a software application owner would pay from US$ 0.000000208 to US$ 0.000004897 per 100ms of execution time. The cloud service provider can offer these low prices due to the fact that there is not one large VM continuously running the entire software application, rather, the software application is divided into many small serverless functions.

Serverless deployment is not necessarily equally beneficial in all contexts. Especially (parts of) software applications that are not highly utilized benefit from cost reductions due to serverless deployment (Warzon, 2016). Furthermore, software applications that experience inconsistent, burst and compute intensive workloads are expected to benefit from cost reductions when deployed serverless (Baldini, Castro, et al., 2017; Chapin & Roberts, 2017).

It is nonarbitrary to determine how low-utilized a software application has to be in order for it to be cheaper when deployed serverless, to this end Warzon (2016) performed a breakeven analysis between AWS Lambda (Amazon Web Services) and AWS EC2 (Amazon Web Services) which at least gives an idea on when serverless deployment would be cheaper than conventional deployment. The breakeven analysis has no singular answer, since there are various configuration options for both AWS Lambda (e.g. RAM and CPU) and AWS EC2 (different instance types). The smallest AWS Lambda configuration (128MB RAM) with a function execution time of 100ms is cheaper than a AWS EC2 m4.large instance up till 295,000 requests/hour. More frequent requests and an m4.large instance becomes cheaper to run continuously. When execution time and/or required RAM rises for AWS Lambda functions, the breakeven point lowers quickly; 200ms and 512MB breaks even at 64,000 requests/hour, 200ms and 1GB at 34,000 requests/hour and 1sec and 1GB at 7,100 requests/hour.

A cost comparison performed in laboratory setting with comparable loads between AWS Lambda, microservices and conventional monolithic deployment shows the potential of cost savings of 50% up to 62% compared to microservice deployment and 55% up to 77% compared to monolithic deployment (Villamizar et al., 2017). Another cost comparison performed on two real software applications MindMup and Yubl which were migrated to a serverless deployment architecture results in comparable cost reductions; respectively 66% and 95% (Adzic & Chatley, 2017). MindMup benefited most from moving file uploads and export conversion function to AWS Lambda, these are relatively infrequently used and are CPU intensive tasks well-suited for FaaS. In the case of Yubl (a social network), the benefits were mostly caused by the ability to quickly scale to spike demands, they used to be over-provisioning a lot to prevent performance issues when such a spike would hit. Another example is presented by Chapin and Roberts (2017), the authors compare the serverless and non-serverless implementation of a small turn-by-turn mobile game.

Their findings are that both the labour cost of developing the game and the resource costs of running the necessary cloud infrastructure are reduced, additionally the authors mention a risk reduction due to inherent FaaS features such as high availability, scaling and security.

Another benefit mentioned earlier is also responsible for cost reductions: implicit failover. Conventional failover implementations require an additional instance to be running all the time to pick up the pieces once another instance has failed. This results in double the costs of running the software application without failover. FaaS has inherent failover, whenever a function execution fails it will be retried on a newly provisioned function, hence making conventional costly failover implementations superfluous (Jonas et al., 2017; Wagner & Sood, 2016).

Cost reductions are a huge intrinsic benefit. Moreover the lower costs create new opportunities that used to be too costly. The shift from capital to operational expenses is one of them, cloud computing in general removes the necessity of purchasing expensive hardware before a software application can be deployed. Serverless computing further improves this by enabling companies to almost perfectly align the costs of hosting a software application to actual business processes due to its 'pay-as-you-go' cost model, additionally it allows companies to monitor costs more granularly on a per business process basis (van Eyk et al., 2017). Removing barriers to versioning is another accessory opportunity, each serverless function deployment receives a unique numeric identifier, and multiple versions of a serverless function can be deployed simultaneously, this empowers developers in a way that was never possible with conventional cloud deployment since it would cost twice as much to have two versions of the same software application deployed at once (Adzic & Chatley, 2017; Van Eyk et al., 2018).

3. 2. 4. Architectural opportunities

Serverless computing, and FaaS especially, presents new architectural opportunities to software application developers and architects. One of these opportunities is touched on before, inherent fault tolerance provided by FaaS (Jonas et al., 2017; Wagner & Sood, 2016). This removes the need for developing and managing fault tolerance from the start, these saved efforts could therefore be spend more effectively.

Another opportunity is the effortless parallelisation of computing. Due to the automatic horizontal scaling of serverless functions it is very easy to distribute and parallelize computing when desired (Varghese & Buyya, 2018). An example is the ExCamera application described by Van Eyk et al. (2018) where the authors used FaaS for improving the performance of editing, transforming and encoding videos by parallelizing these tasks over many concurrent serverless functions. McGrath and Brenner (2017) describe similar performance improvements through parallelisation; a media management service application that parallelised image resizing making the execution time independent of how many resizes are being executed simultaneously.

Serverless functions are not necessarily designed for the purpose of chaining multiple functions together to form specific workflows although they do lend themselves very well for that purpose

due to the provided data flow and event-based abstractions (Yan et al., 2016). This is further substantiated by van Eyk et al. (2017) who envision a 'workflow' like composition of serverless functions in order to facilitate development. Amazon recognized this need and developed the AWS Step Functions (Amazon Web Services) service which allows "[the coordination of] multiple AWS services into serverless workflows", or in other words, it enables the chaining or composing of serverless functions.

3. 2. 5. Improvements to the development process

A different category of benefits is that of development process improvements. For example, the independent serverless functions allow developers to choose the best language, dependencies and tools for that specific use case (Van Eyk et al., 2018). This way they are not bound to choices made in the past which may not work for their current challenge.

Reducing lead time is another major potential benefit to the development process, not only does it save labour costs, it also empowers developers to more quickly try and test potential solutions which enables faster feedback and improves rapid iterations (Adzic & Chatley, 2017; Baldini, Castro, et al., 2017; Chapin & Roberts, 2017). As a consequence of reducing lead time due to migration from a monolithic architecture to a serverless driven architecture Adzic and Chatley (2017) note that the release frequency for a real software application (Yubl) could be greatly increased, from 6 releases a month to more than 80. This reduction is mainly caused by the fact that serverless functions remove the need to manage infrastructure and other operational aspects for developers, which in turn allows them to focus their time and effort on business logic (Adzic & Chatley, 2017; Baldini, Castro, et al., 2017).

Finally, reduction of risk is mentioned as a result of adopting serverless computing (Chapin & Roberts, 2017). While Chapin and Roberts (2017) do not mention any further specifics, Ivanov and Smolander (2018) note that serverless can reduce risk by providing easy deployment and rollbacks due to the atomic nature of serverless functions.

3. 3. Challenges of serverless computing

This section is concerned with answering the second part of RQ1 "What are the currently known benefits and challenges of serverless computing?". Serverless computing poses some severe challenges compared to conventional cloud deployment; operational cost reductions are not straightforward, performance issues, development complexities, migration and decomposition challenges, increased monitoring difficulty, potential threat of vendor lock-in and security concerns. This paragraph presents a non-exhaustive summarization of the most common challenges found in literature.

3. 3. 1. Performance challenges

Various authors describe performance as one of the main challenges FaaS is currently facing (Baldini, Castro, et al., 2017; Chapin & Roberts, 2017; van Eyk et al., 2017; Van Eyk et al., 2018). Examples of these performance issues are relatively long wait-times due to cold starts (Baldini, Castro, et al., 2017; Chapin & Roberts, 2017; van Eyk et al., 2017) and high latency due to the HTTP transport protocol (Chapin & Roberts, 2017) but also currently very basic scaling and scheduling policies (Baldini, Castro, et al., 2017; van Eyk et al., 2017).

Cold starts are especially challenging since the nature of this challenge is inherent to FaaS, the deprovisioning of functions when not in use. Depending on the load on a specific serverless function it might or might not cause performance issues, for example, a function that is being used at least once a second only experiences cold starts once every 10000 executions, or 0,01% of the time (Chapin & Roberts, 2017). As Chapin and Roberts (2017) mention, the cold start problem makes predicting FaaS performance especially difficult, but the authors feel that this problem might be minimized or addressed in the future.

Another interesting performance challenge is that not only on application level performance issues might occur, but also on the underlying serverless platform managed by the cloud provider, this makes it much more difficult to predict or optimize code performance (Chapin & Roberts, 2017).

The unpredictability of FaaS performance makes it hard, if not impossible, to provide Quality-of-Service guarantees (Yan et al., 2016).

3. 3. 2. Development and operations complexities

Since serverless computing is a new architectural concept, it requires developers to change their modus operandi in some aspects. Let's consider state, most cloud software developers are accustomed to developing stateful microservices or monolithic applications, e.g. user sessions, caching, et cetera, due to the nature of serverless functions this is not possible (Baldini, Castro, et al., 2017; Chapin & Roberts, 2017). Therefore, developers are forced to use external stateful components, even persisting the tiniest amount of state. This might increase complexity, but surely changes how software developers create software.

Another aspect regarding the development process that poses additional challenges when adopting FaaS is integration testing. Not all cloud service providers offer the tools to test serverless functions locally, without additional costs of running in the cloud (van Eyk et al., 2017). This brings extra complexity, as suddenly running tests becomes costly (since the cloud provider charges for tests equally as any other user would use the serverless function), requires deployment (next to already deployed serverless functions), and increases complexity due to its distributed nature (Chapin & Roberts, 2017; van Eyk et al., 2017).

Versioning is another development aspect that suffers from additional complexity according to various authors (McGrath et al., 2016; van Eyk et al., 2017; Villamizar et al., 2017). McGrath et

al. (2016) explain this as due to the individual deployability of serverless functions that can occur across arbitrary parts of a versioned software application. This is in line with the findings of Villamizar et al. (2017) who noted during the application of a serverless architecture in a laboratory setting that due to the many separately versioned parts of such an architecture it is very important to maintain a proper service versioning by properly defining an upgrade process and increasing coordination among teams.

Vendor-interoperability is another challenge for developers. Each serverless computing vendor requires different implementations of serverless functions, hence there is a need for a vender-agnostic definition or platform which enables developers to move their functions without adaptation between various cloud service providers (van Eyk et al., 2017). Besides, each platform might behave different from another; scaling, resource usage, performance, all might be subject to change between various cloud service providers. Therefore it is important for developers to really understand the specific platform to which the application is (going to be) deployed (Baldini, Castro, et al., 2017).

Finally, specific tooling for serverless function development and deployment is required for various reasons. For example to enable developers in testing and running their code locally (van Eyk et al., 2017) and to allow distributed monitoring (Chapin & Roberts, 2017). Besides that, deploying large-scale serverless applications quickly becomes unmanageable, serverless deployment tools should assist developers in easy-deployment (Chapin & Roberts, 2017), preferably across multiple cloud service providers as for example the Serverless Framework (Serverless Inc., 2019) currently does.

Monitoring and debugging becomes especially hard in a serverless context (Baldini, Castro, et al., 2017). Functions only live for a short amount of time and are completely destroyed afterwards, removing any trace of their execution (Baldini, Castro, et al., 2017), therefore it is not possible to attach a remote debugger (Chapin & Roberts, 2017). Besides, distributed monitoring and debugging is still an area of improvement, it should enable developers or operations personnel to trace a business request through all the individual components that are processing the request and form a response or action (Chapin & Roberts, 2017). In conventional cloud computing the concept of a stack trace often presents developers with the necessary information to track down bugs, such a concept does not exist for (distributed) serverless computing (Yan et al., 2016).

Another point to note is that even though serverless functions brings many benefits for software developers, it is not to be forgotten that it also brings additional overhead on a per-function basis. For example logging, monitoring, versioning/updating, documentation, all these things that used to be handled on a microservice or monolithic codebase level now have to be handled for each function separately, at least to some extend (Villamizar et al., 2017).

3. 3. 3. Serverless composition

Serverless function composition is another area that requires attention. Composition allows software developers to create patterns or workflows by calling serverless functions from other serverless functions resulting in a business function. Another aspect of this challenge is that of the hybrid cloud, there are more than one cloud platforms provided by various cloud service providers, according to Baldini, Castro, et al. (2017) it is unlikely that one platform will have all the functionality a developer could need and will work for every use case, therefore it is important to consider that composition should be possible between different cloud platforms.

An interesting research challenge that is posed by van Eyk et al. (2017) concerns the currently missing reference architecture for serverless cloud computing applications. Such a reference architecture would guide software architects and developers by providing examples and best practices with regard to implementing (hybrid) serverless cloud applications. According to van Eyk et al. (2017) a reference architecture "would provide developers and researchers with an understanding of the main components shared by FaaS platforms, facilitating new deployments and enabling comparisons of designs".

3. 3. 4. Increased monitoring complexity

Monitoring becomes more complex as the software to be monitored becomes more complex, this also holds for serverless functions due to its distributed nature. Villamizar et al. (2017) and Chapin and Roberts (2017) identify the challenge of tracing the flow of a request made by an end-user through the various cloud infrastructure components and software services (distributed monitoring), something that used to be trivial when there was only a single software application deployed. Not only is monitoring more complex in a serverless context, it is also more important since the execution of the functions is directly and linearly related to the operational costs (Yan et al., 2016). Therefore, monitoring needs to be used to see for example if cost savings can be made, or if a bug is causing growing costs.

3. 3. 5. Migration and decomposition efforts

It is not trivial to decompose an existing application, whether that is a monolithic application or a number of microservices, into various serverless functions. This might be desired for existing applications that want to take benefit of serverless computing for parts of their system. Serverless functions are designed to respond to events and perform certain operations which collectively can be seen as a full-fledged web application, however McGrath et al. (2016) mention that a traditional application is not easily split into separate serverless function endpoints. This is supported by Hendrickson et al. (2016) who note that decomposing monolithic web applications into serverless functions presents similar challenges as decomposition practices applied to operating systems, web browsers, web servers and other applications. Adding to the decomposition challenge is the fact that not all operations are well suited for migration to serverless functions; I/O operations are

costlier and less effective than compute operations in a serverless function context (Baldini, Castro, et al., 2017).

3. 3. 6. Threat of vendor lock-in

Another important issue is the threat of vendor lock-in. Many authors mention that there are various reasons for FaaS users to be subject to vendor lock-in by the cloud service provider (Adzic & Chatley, 2017; Baldini, Castro, et al., 2017; Chapin & Roberts, 2017; Eivy, 2017; Villamizar et al., 2017). Serverless functions are often developed for a specific FaaS platform such as AWS Lambda, which brings inherent vendor lock-in potential (Villamizar et al., 2017) since it might take additional effort, or is just not possible, to move the function to a different cloud platform. Besides, many FaaS platforms offer additional services such as platform specific storage adapters, scaling, logging and monitoring, configuration management, authentication services and more (Adzic & Chatley, 2017). These services seem rather convenient at first, but might make it impossible to move to a different FaaS vendor when desired (Baldini, Castro, et al., 2017).

Adzic and Chatley (2017) make another important observation; serverless architectures enable client applications to connect directly to storage and authentication services, this results in more tightly coupled software which in turn requires a lot of rewriting and refactoring when one would move from one cloud platform to another.

3. 3. 7. Security concerns

Some authors argue that serverless functions bring additional security challenges. Yan et al. (2016) describe it as “understanding the attack surface of a serverless composition is an on-going issue”, whereas van Eyk et al. (2017) state that centralizing operational logic in the infrastructure reduces the attack surface, but also that the new security issues introduced by FaaS (such as resource sharing) are not fully understood yet. The risk of resource sharing is supported by Wagner and Sood (2016), they argue that if an attacker would be able to break out of the serverless function container it would have access to all the containers running code on that specific machine.

3. 3. 8. Reducing operational costs

As discussed before in Section 3. 2. operational cost reduction is one of the main benefits of adopting serverless cloud computing. Achieving these cost reductions however is not as straightforward as it might appear, the potential savings of serverless computing “heavily depend on the execution behaviour and volumes of the application workloads” (Eivy, 2017).

The same author also notes that cost prediction is very difficult, and can only really be estimated by running the code in the deployment environment (on the FaaS platform of the cloud service provider) and testing it under expected loads. However, this can become costly so it should be done with care.

Another aspect that should not be forgotten when thinking about cost reductions through serverless computing is that of the additional service usage. One of the advantages of FaaS is that the cloud service provider offers many services that connect directly to the serverless functions and offer functionalities such as authentication, storage, monitoring etc. FaaS is a pay-per-use service, and so are the additional services. It is very likely that a FaaS implementation will use an API Gateway, external storage (Amazon S3, DynamoDB) and will have data egress (data leaving the local network to an external location), the usage costs of these services potentially add up, especially relatively to the low execution cost of serverless functions (Eivy, 2017).

Eivy (2017) also describes a real-world case study where a cost analysis was performed for a to-be-developed public facing API endpoint. The usage was projected at 150 executions per second initially, but could grow to 30,000 executions per second over the next year. Initially FaaS looked like a perfect fit for this situation, however, when looking closer at the expected growth of the endpoint, FaaS appeared to become three times more expensive than a conventional reserved compute instances approach.

3. 4. Characteristics of serverless suitability

This section is concerned with providing an answer to RQ2 “What are the existing approaches or characteristics to determine which parts of an application are suitable to be implemented in a serverless way?”. To answer this question a literature review is conducted aimed at finding software related characteristics that help identify the fit- or unfitness of a part of a software application to be implemented and deployed in a serverless fashion.

Serverless has been growing in popularity for quite some time, however it is still not entirely clear in which situations to use this architecture, and just as important, in which situations to not use this architecture (Grumuldis, 2019). Use cases, often provided by cloud platform providers, demonstrate a situation in which serverless can be beneficial, however they do not demonstrate in which situations it is not (Grumuldis, 2019). Therefore, it is important to have the knowledge to make these decisions as a software developer, software architect or software application owner.

After analysing the literature that was found by executing the search queries defined for this review (as described in Section 3. 1. 2.) a categorization of serverless suitability characteristics is created with the purpose of grouping similar characteristics together: response time, invocation patterns, types of operations, data limits, vendor dependence and runtime restrictions. This categorization is used in the following sections to group and present the findings of this literature review.

3. 4. 1. Response time

If the software is highly performant and has to fulfil certain response time requirements serverless might not be the best choice. Although it can perform at acceptable levels, it is difficult to guarantee due the unpredictable nature of FaaS performance (see Section 3. 3. 1. Performance challenges). It depends on the specifics of the performance requirements whether the response time

(accumulation of resource provisioning and execution) can be considered acceptable (Völker, 2018).

On the other hand, serverless is a good fit for performing background tasks which do not directly respond to users (Völker, 2018), or which continue running in the background without keeping the user waiting.

Another aspect to consider related to the response time is the expected size of the serverless function deployment package. Besides the limitations set by various cloud platform providers (e.g. AWS limits the package size to 50MB, and the extracted package size to 250MB), the package size relates to the cold-start time (Grumuldis, 2019). Larger packages, with a lot of dependencies for example, take longer to initialize which in turn degrades the response time of that specific function (Abad, Boza, & Eyk, 2018).

3. 4. 2. Invocation patterns

This characteristic is highly connected to the operational costs of a serverless function. It depends on the invocation frequency and invocation density whether a piece of code can be cost efficiently deployed in a serverless fashion or not. This is supported by Eivy (2017) who states that cost reductions "heavily depend on the execution behaviour and volumes of the application workloads".

In general can be concluded that software applications which experience either low utilization or dynamic and irregular workloads are a good fit for serverless architectures (Grumuldis, 2019; Völker, 2018). This can be explained in twofold. First, serverless functions can scale to zero, meaning there will be no costs incurred when the function is not used. Second, due to the fact that serverless functions scale automatically and very quickly there is extremely little over and under provisioning while still keeping up with demand.

Software applications that experience a moderate and especially consistent load are often better suited when deployed on a light always-on dedicated VM as there is no need to scale for peak loads and the advantage of scaling to zero of serverless functions would not be used due to the consistent load.

A more general note of consideration, while serverless functions scale effortlessly, it is important to make sure that the services being called by these functions scale as well. Otherwise applying a serverless architecture will only move the bottleneck to less flexible parts of the system (Grumuldis, 2019).

3. 4. 3. Type of operation

The type of operation that is executed determines if it would be a good fit to be deployed serverless. For this characteristic the distinction between I/O and CPU bound operations is made. The former is concerned with read/write operations such as writing an image or large document to a database,

but also heavy network communication like downloading a data set. The latter is concerned with CPU intensive calculations, such as image editing and mathematic calculations.

Code that is CPU bound is in general a better fit for serverless deployment than code that is I/O bound (Grumuldis, 2019; Völker, 2018). This has multiple reasons. First, I/O operations can be relatively slow, even if they are quite fast, the serverless function execution has to wait for it to finish, and considering the execution time based cost model of serverless computing this is wasted money (Grumuldis, 2019). Second, considering the stateless nature of serverless functions, it is rather inefficient to perform I/O operations since each invocation has to create a new connection with the I/O service, which results in lower performance when compared with a dedicated VM that caches these connections (Völker, 2018).

Additionally, it is important to consider that due to the implicit fault tolerance of serverless functions, the operations these functions execute should be idempotent (Grumuldis, 2019). Meaning that the function should produce identical output and side effects for two invocations with identical input. Since the implicit fault tolerance will retry an invocation in case it failed, the code should be able to handle the expected idempotency.

3. 4. 4. Data limits

Serverless computing comes with various data limitations, for example on the allocated memory (e.g. AWS Lambda limit 3008MB RAM), available temporary disk space (e.g. AWS Lambda limit 512MB), deployment package size (e.g. AWS Lambda limit 50MB compressed and 250MB uncompressed) and request payload size (e.g. AWS Lambda limit for synchronous execution is 6MB) (Grumuldis, 2019). While it is difficult to generalize on this characteristic due to the differences between various FaaS platforms, it is important to note that these limits are inherent to the serverless technology and might be a constraining factor in the implementation of a serverless application. Therefore, it is paramount to consider this characteristic when deciding whether a piece of code is suitable for serverless deployment or not.

3. 4. 5. Vendor dependence

Another consideration when looking at the serverless suitability of a software application is whether or not it is acceptable to become, to a certain extent, dependent on the cloud platform vendor.

The risks of vendor lock-in have been elaborately discussed in 3. 3. 6. These risks might or might not be worth the benefits of serverless computing, and not all risks are equally dangerous or impactful. Anyhow, vendor lock-in is a relevant threat and should be considered while choosing whether or not to go serverless, or even which FaaS provider to commit to (Grumuldis, 2019).

Moreover, most vendors put a relatively high price on data egress (data leaving the cloud vendor's network to an external location) which might limit a hybrid cloud approach (a distributed software

architecture deployed to different cloud platforms) and enforce the vendor lock-in even further, forcing the use of platform specific services (Grumuldis, 2019).

3. 4. 6. Runtime restrictions

Serverless computing adds an additional layer of abstraction onto the stack, therefore the software that is deployed in a serverless fashion needs to be hardware agnostic (Grumuldis, 2019). Besides, the FaaS environment often provides only a few programming language runtimes which can be used in the serverless function. Additionally, the software should be able to handle the short-lived runtime environments and not depend on making local changes due to the automatic deprovisioning of containers (Grumuldis, 2019).

3. 5. Composition of serverless applications

This section is concerned with RQ3 ”How could cloud native architectures be composed and orchestrated to enable the full potential of serverless computing?”[1]. In order to answer this question a structured literature review is executed with the goal of assembling a state-of-the-art with regard to serverless computing application architectures, focus will lie on the relevance of serverless composition, the challenges of serverless composition, currently known methods, (modelling) tools and architectures related to serverless composition, and finally it will zoom in on a particular issue; hybrid serverless composition. Before continuing, it is important to clarify some terms.

First, cloud native applications, as explained by Gannon, Barga, and Sundaresan (2017), are often considered to have the following characteristics:

1. Operates at global scale.
2. Designed to scale to many concurrent users.
3. Assumes fluid and failure prone infrastructure.
4. Testing and updating applications should not disrupt production.
5. Security is considered early in development.

Second, hybrid composition (often also referred to as multi-cloud) is very basically the composition of services across various cloud platforms, e.g. an application that is deployed partly on the AWS infrastructure, partly on the Google platform, and possibly even partly on a private cloud. It is very likely that a cloud platform does not offer all the functionality a developer needs (not even considering pricing differences), besides there are many scenarios where an application cannot be built completely serverless or needs to connect with legacy systems, therefore it is important that software developers and software architects are not confined to a specific cloud vendor but that there are composition techniques that allow for (relatively) easy cross-cloud

[1] This literature review has been executed within the Capita Selecta Software Technology (201400171) course under supervision of an EWI teacher and has been approved for reuse in this thesis.

platform compositions (Baldini, Cheng, et al., 2017; Wurster, Breitenbucher, Kepes, Leymann, & Yussupov, 2019) .

3. 5. 1. Relevance and challenge of serverless composition

First it is important to consider the question of why serverless composition is a relevant issue. There are multiple reasons for this. The serverless functions concept is rather young and, according to Garcia Lopez et al. (2019), lacks adequate coordination mechanisms between functions. Currently, it is difficult and requires quite some effort to orchestrate a large set of serverless functions to create a complex application (Garcia Lopez et al., 2019). Furthermore, Baldini, Cheng, et al. (2017) state that serverless function composition is lagging behind the state-of-the-art.

Serverless (and microservice) architectures are not arbitrarily derived from or created upon conventional application architectures since they use a less centralized and more distributed approach, hence requiring a cloud application architecture redesign (Kratzke, 2018). Besides, as stated before in Section 3. 3. 3. , serverless computing enables the creation of complicated execution patterns (Baldini, Castro, et al., 2017).

Due to this growing complexity and novel architecture composition approach it is important to look at what composition models are suitable for serverless architectures and what are "ways to express (…) compositions of functions, and (…) hybrid-cloud deployment" (van Eyk et al., 2017). This is supported by Baldini, Castro, et al. (2017) who set forth the goal of developing tools that support the creation of compositions and their maintenance.

Vaquero et al. (2019) also underline the current challenge of orchestrating serverless functions, although the authors are more focussed on serverless computing in the context of edge computing (executing code geographically closer to the consumer) which brings additional but comparable issues.

Serverless functions are not necessarily conceived with the idea of complex compositions in mind, however there are many authors who envision such complex compositions in the form of 'workflows' (van Eyk et al., 2017; Yan et al., 2016). Not only academics saw this opportunity, Amazon created their own workflow composition service that allows developers to chain AWS Lambda functions together to create more complex behaviours, it is called AWS Step Functions. This service is of course limited to the AWS platform and comes with a pricing scheme, but it demonstrates the recognized need for serverless composition solutions.

Baldini, Cheng, et al. (2017) present the serverless trilemma which demonstrates the inherent challenges of serverless composition. Loosely translated the trilemma consists of three competing constraints for serverless function composition (Garcia Lopez et al., 2019; Soltani, Ghenai, & Zeghib, 2018):

1. Functions should operate as black boxes.
2. A composition of functions should be a function.

3. Invocations should not be double-billed.

What this trilemma means for serverless composition is that there are some inherent limitations to serverless composition which cannot (yet) be overcome. A composition will always suffer from one or more of the three constraints. A valid question arises; why is this trilemma useful? At the very least it provides practitioners with insight into the limitations of serverless function composition and allows reasoning on which constraint would be acceptable in a certain situation and which not. Furthermore, it shows what directions of serverless composition could benefit from further research.

3. 5. 2. Serverless composition state-of-the-art

According to Garcia Lopez et al. (2019) there are two types of serverless composition patterns:

1. Functions that orchestrate other functions.
2. External 'client schedulers'.

The first type can be seen as an 'entry' function that will delegate some of the work to other functions and finally return once all the results are gathered. For example, a function called 'calculate total costs' is the orchestrator, in order to actually determine the total costs it needs to gather costs for several products, so it will execute other functions 'calculate costs product A' and 'calculate costs product B' in parallel and add the results and respond to the client that invoked the 'calculate total costs' function.

The second type is based on an external application or service which will orchestrate several functions into a workflow. An example of such a service is the previously discussed AWS Step Functions. The services exposes an API which when called will forward the request to the corresponding function which would be the next step in the workflow.

Both approaches have their advantages and drawbacks. In light of the serverless trilemma (Baldini, Cheng, et al., 2017) the first approach violates the 'double billing' (3) principle, whereas the second approach violates the 'composition of functions should be a function' principle.

3. 5. 2. 1. Client focussed composition

With regard to the first type of serverless composition patterns, Kratzke (2018) demonstrates the concept of client focussed composition. This entails moving the composition from the cloud to the client application. For example, a smartphone application that handles the orchestration of many serverless functions itself instead of relying on a microservice or AWS Step Functions-like service handling the orchestration. Obviously this moves the complexity from the cloud to the client, which might or might not be desired. Additionally, it is interesting to note that this shifts the resource provisioning for composition from the cloud to the client and prevents violation of the 'double billing' principle. Figure 5 presents a very basic example of how a client focussed composition pattern could be implemented and shows the growing complexity of the composition

client-side, it coordinates API requests to three separate serverless functions and has to manage the (asynchronous) results.

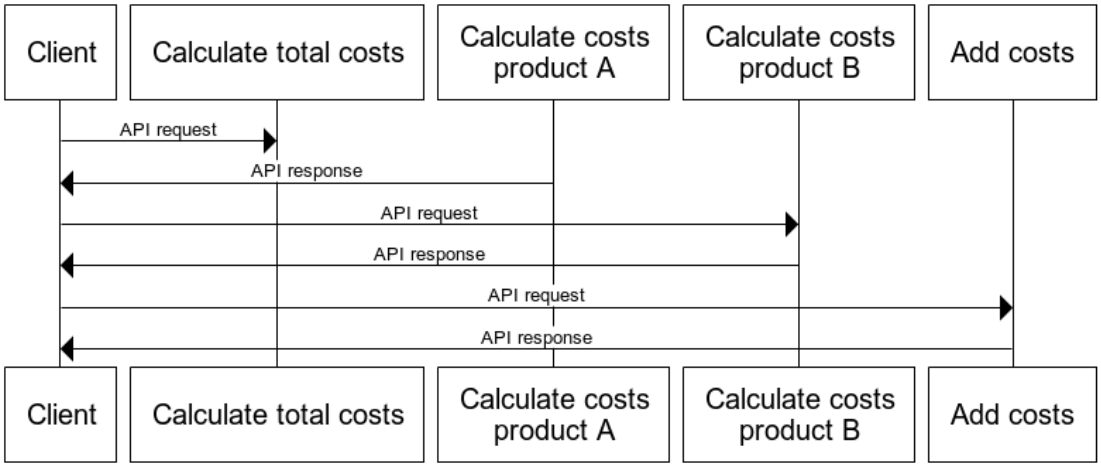


Figure 5 Sequence diagram for client focused composition pattern

3. 5. 2. 2. Serverless composition services

With regard to the second type of serverless composition patterns, there are quite some companies that jumped on this opportunity around 2016/2017; Azure Durable Functions, IBM Composer, and the previously discussed AWS Step Functions. In order to create an entire process out of various separate tasks (implemented as functions) a software developer needs to write code that orchestrates these tasks into a process, this can become very complex and difficult to debug. Serverless composition services such as AWS Step Functions provide a visual tool that enables the creation and debugging of complex workflows consisting of serverless functions. To give an idea of how this would work, Figure 6 shows a very basic example of serverless service based composition. The benefit compared to the client focussed composition pattern is that there is only one API request to make client-side. All necessary consecutive steps are coordinated and executed by the serverless composition service.

There are some downsides to serverless composition through these services, most dominantly performance. This is not unknown to serverless computing, however it should be considered. All mentioned services experience performance overheads of 0.3 to 10 seconds on sequential executions, and 18 to 32 seconds on parallel executions (Garcia Lopez et al., 2019).

Garcia Lopez et al. (2019) propose an evaluation framework for serverless composition services based on a number of metrics; serverless trilemma safeness, programming model, parallel execution support, state management, software packaging and repositories, architecture, overhead and billing model. Based on this framework the authors execute an evaluation of the three major services (as described above). In general they conclude that the performance overheads are too high on all services, furthermore none of the services is prepared for parallel programming. While

all evaluated serverless composition services are considered young and experimental, AWS Step Functions is currently considered the most mature and performant (Garcia Lopez et al., 2019).

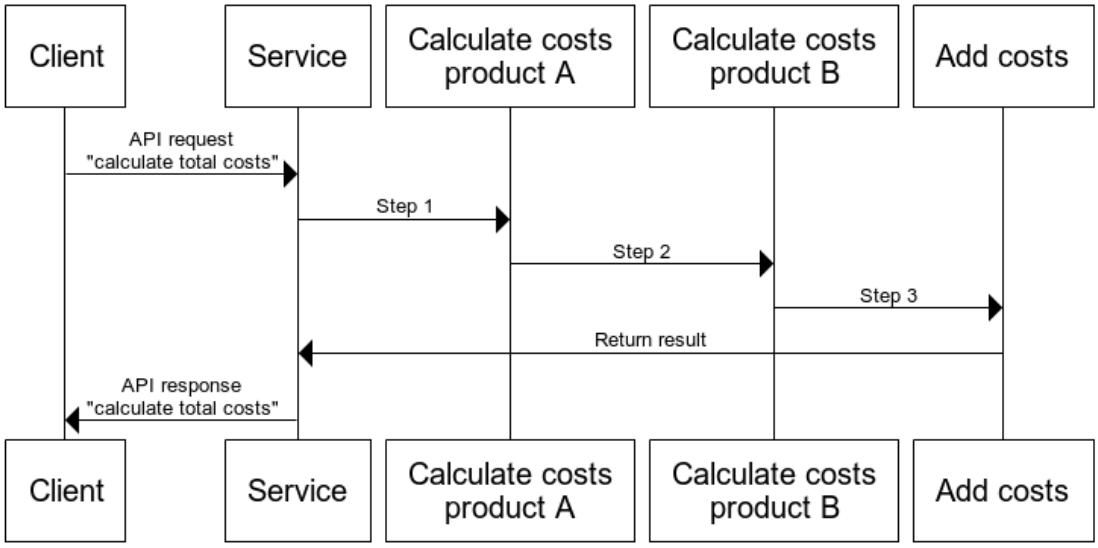


Figure 6 Basic example of a serverless composition service implementation

3. 5. 2. 3. Modelling serverless composition

There were a few serverless composition modelling approaches found in the literature which will be described here. The first and most common modelling approach is TOSCA (Topology and Orchestration Specification for Cloud Applications), according to Vaquero et al. (2019) it is becoming the de facto standard for modelling service orchestration. TOSCA is very useful for defining services and their building blocks, and requirements and capabilities, however it does not address serverless composition specific issues such as orchestration of small functions in a (hybrid) cloud environment at scale (Vaquero et al., 2019).

To that end, Wurster et al. (2019) describe how TOSCA can be used to model serverless architectures, they explain how to represent functions, events and event sources using the constructs provided by TOSCA. Wurster et al. (2019) point out that there are some limitations to this approach; TOSCA does not support function chaining and it is focussed on being an executable model from which code could be generated.

Perera and Perera (2018) present TheArchitect, which is a tool that can generate a very high level microservice and/or serverless architecture for a given application. This approach looks at the problem of serverless composition from another angle but unfortunately does not provide any insights on the applied composition patterns.

3. 5. 2. 4. Tools and methods

Some methods and tools with regard to serverless composition were found in literature and are presented here. The first is Serverless Container-aware ARchitectures (SCAR) created by Pérez,

Moltó, Caballer, and Calatrava (2018). The SCAR framework enables the development of highly-parallel event-driven serverless applications based on a custom runtime environment. This custom runtime environment is created by deploying Docker images on top of AWS Lambda. It greatly helps overcoming the restricted runtimes that are by default provided by cloud vendors by facilitating many other programming languages through the Docker platform (Soltani et al., 2018). After evaluation it seems to not necessarily be a tool that could contribute to (improving) serverless composition.

Soltani et al. (2018) summarize some other noteworthy tools; PyWren (Jonas et al., 2017), Podilizer (Spillner & Dorodko, 2017) and Snafu (Spillner, 2017). The first is an opensource project allows running python code and its dependencies on various serverless computing platforms. The second is similar in that it tries to automatically convert Java code into small pieces of code that are ready to be deployed serverless. Snafu on the other hand is put forth as a new design for FaaS hosts, so this is one abstraction layer deeper than this thesis is focussing on.

Another interesting result is the Cloud Application Maturity Model as presented by Kratzke (2018). It defines four levels of maturity and corresponding criteria, from low to high maturity; cloud ready, cloud friendly, cloud resilient, cloud native. The criteria can be used to measure the maturity of a given cloud application, and it would be interesting to apply this to serverless architectures as well.

Additionally, Kratzke (2018) presents a reference model for cloud-native applications consisting of 4 viewpoints (layers); infrastructure provisioning, clustered elastic platforms, service composing and application. It is interesting to see this reference model as it provides insight into how a cloud-native stack is constructed, unfortunately it does not focus on the actual serverless function composition itself which would be positioned in the top two layers; service composing and application.

3. 5. 2. 5. Hybrid cloud and multi-cloud composition

The introduction of Section 3. 5. explains what is understood by hybrid cloud composition and why it is relevant. Multi-cloud composition is the concept of have a single software system deployed to different cloud platform providers. From literature it appears that there is a special focus on hybrid and multi-cloud composition, therefore this section is created separately from serverless composition.

Hybrid cloud composition presents the challenge of integrating private and public clouds, furthermore hybrid cloud deployment becomes even more complex when conventional deployment technologies are combined with serverless architectures (Wurster et al., 2019). Vaquero et al. (2019) highlight the concept of an orchestration broker to overcome these challenges. Broker models are not a new concept, and in fact have been around the area of distributed computing for the last 20 years (Vaquero et al., 2019). Therefore, it might be a mature approach to tackling hybrid and multi-cloud composition. Additionally, Vaquero et al. (2019)

discuss the possibility of developing adapters and brokering layers for each available cloud vendor (as there are currently only a few) in order to homogenise access to these different platforms.

In their paper Soltani et al. (2018) try to extend the serverless architecture to a hybrid cloud context using a container cluster manager technology, this should distribute the serverless architecture on different cloud platforms while applying the serverless principles in a larger context. To achieve this, the authors propose a generic distributed peer-to-peer system that wraps serverless application code in container runtimes. This system can be considered as a first step in enabling hybrid serverless compositions and should allow the client to benefit both from hybrid cloud and serverless computing advantages.

3. 5. 2. 6. Architecture use-cases and demonstrations

Various papers reported on the development of a serverless architecture, these might be interesting to look at since they might provide valuable information with regard to best practices of serverless composition.

Wurster et al. (2019) show two very basic high-level serverless architectures, the first demonstrates an implementation of a client focussed composition pattern and the second approach concerns a hybrid cloud serverless architecture where part of the system is run on a private cloud.

Kratzke (2018) presents a reference architecture for a serverless platform which is focussed on the workings of a FaaS platform itself, it shows how HTTP requests are routed through various components and are eventually consumed by a worker function. Again, this approach does not look at serverless composition itself, although it does create insight into the structure of a FaaS application.

A notable fact that seemed to be recurring in the literature is the application of serverless computing in the context of scientific workflows (Malawski, Gajek, Zima, Balis, & Figiela, 2017). Apparently, the pay-as-you-go model of serverless computing is attractive for academics as is the instant parallelisation of computation.

3. 5. 3. Improving serverless composition

Literature proposed some recommendations or ideas for future improvements of serverless composition. The first is a rather interesting approach to serverless composition while respecting the serverless trilemma. Garcia Lopez et al. (2019) present the idea of suspending serverless functions while making asynchronous calls to other serverless functions. This would resolve the issue of double billing where a function has to wait for another function to execute, hence two functions are running simultaneously resulting in double billing. Furthermore, it respects the other two items of the trilemma; substitution principle (compositions are functions) and composed functions can be seen as black boxes. More concrete, the authors propose to expose a method to the serverless runtime environments which when called halts the execution (and billing) of the

function until a certain event is received. This enables developers to create compositions of serverless functions that respect the serverless trilemma.

A recommendation from Garcia Lopez et al. (2019) is that "the best of orchestration might be having no orchestration at all". They argue for very simple orchestration and composition techniques, as it enables software developers and operators to quickly create and debug.

Finally, the problem of discovering and enabling the re-use of many small functions is put forth, this is not necessarily a serverless composition specific issue, however it does get worse due to the very granular and plentiful nature of serverless functions (Garcia Lopez et al., 2019). The absence of well-defined discovery practices might be addressed by building on classic software engineering practices such as creating libraries with functions of similar nature, according to Garcia Lopez et al. (2019).

3. 6. Best practices for creating and migrating towards a serverless architecture

This section is concerned with RQ4 "What are the best practices for creating a serverless application from scratch, and for migrating a non-serverless architecture to a serverless architecture?". To the purpose of answering this question a structured literature review is executed. The goal of this section is to describe what has been found in literature with regard to established best practices for creating, or migrating to, a serverless architecture. This information will be used to further build upon in later sections. This section will be structured as follows: first a few of the problem areas of serverless computing are addressed (e.g. costs, performance, vendor lock-in). Second, best practices related to serverless architecture in general (and migrating towards serverless architecture) will be discussed.

3. 6. 1. Performance

Bardsley, Ryan, and Howard (2018) argue to keep the serverless function call stack short. This improves performance since there are less "links in the chain" which have potentially high latencies or suffer from cold-start times. The same authors make a case for wisely choosing the programming language which will be deployed as a serverless function. In general interpreted languages such as Node.js and Python have faster initialization times, on the other hand, compiled languages such as Java and .NET have a higher maximum performance. Depending on the desired performance either interpreted or compiled languages may be the way to go. It is important to note that performance of a specific runtime may vary substantially between cloud platforms, e.g. Node.js has an acceptable cold start time on AWS, but rather poor performance on Azure Functions (and vice versa for .NET C#) (Jackson & Clynch, 2018). The takeaway advice with regard to language performance is to check the language performance on the cloud platform you are deploying to, it might dramatically improve your application's performance and simultaneously save costs.

3. 6. 1. 1. Dealing with cold-start time

Quite a few authors discuss mitigating, overcoming or coping with performance issues that are known to come with serverless architectures. One of the largest performance issues is the cold start problem (as described in Section 3. 3.).

To overcome this, Lloyd, Vu, Zhang, David, and Leavesley (2018) present "Keep-Alive" workloads. This approach is based on the re-use of existing infrastructure by preventing serverless function instances from being killed completely by 'pinging' them on a set interval. By pinging these functions the cloud platform provider will never fully destroy the serverless function instances and hence removing the cold start problem. This might be a good solution for some, however it must be noted that this smells like an anti-pattern: serverless functions are designed to be ephemeral which allows this type of cloud deployment to exist at all (using spare computing resources temporarily). Using Keep-Alive incurs costs for pinging the serverless functions, so careful consideration is required to determine if these costs (plus regular serverless computing costs) are acceptable or that a regular continuously running VM or container service might be more economical.

Grumuldis (2019) presents other best practices for dealing with cold-start performance issues. First, reducing the overall size of the serverless function (especially considering libraries) helps to reduce the cold-start time by speeding up the function initialization time. Second, if possible, choose to increase the amount of RAM allocated to the function instance, this means almost always proportionally more CPU power which in turn improves initialization speed of serverless functions (especially CPU heavy runtimes such as Java). Third, it is advised to reduce the execution time of the serverless functions. Not only because this might save costs due to the pay-per-execution-time cost model, but due to the fact that cold-starts occur less often when more function instances are available to process new requests.

3. 6. 1. 2. I/O bound operations

Serverless functions are best suited for CPU bound operations, I/O bound operations are supported as well, however the latter is often associated with longer execution times due to dependencies on local or external file systems. Grumuldis (2019) discusses a way to circumvent this (mostly cost related) performance issue: using (event based) async operations. This means that a serverless function can start a write/read operation and then stops, then another serverless function is triggered by the event that the write/read operation has completed and will continue where the previous function has stopped. This comes very close to the proposal of Garcia Lopez et al. (2019) suspending serverless functions when possible.

In order to improve the performance of a serverless function dealing with database connections it is argued to reuse database connections, this is also known as 'connection pooling' and is a common practice on monolithic and micro-service based applications (Grumuldis, 2019). For serverless functions this is a bit more difficult due to the ephemeral and stateless nature of such a

function. The functions cannot be stateful (i.e. having a connection pool in memory), besides the pool of connections would be destroyed instantly after execution, requiring each invocation to open a new connection (with additional overhead). However, there is a way to overcome this, which only works on warm executions, each function instance can use some memory which can be used to store a database connection until the instance is destroyed, therefore each subsequent warm invocation can use the database connection the previous invocation established, resulting in a performance improvement (Grumuldis, 2019). This implementation is exemplified in Figure 7.

```
// Declare database connection outside of function scope
let databaseConnection = null;

/**
 * Serverless function example re-using database connection.
 * This method wants to read from a database and tries to re-use
 * an existing database connection.
 * @param {Object} event
 * @param {Object} context
 * @param {Function} callback – Method which sends a (status) response back
 */
module.exports.handler = async (event, context, callback) => {
  // Check if db connection available, else create it
  if (!databaseConnection) {
    databaseConnection = createDatabaseConnection();
  }

  // Return database product data
  return callback(null, databaseConnection.getProducts());
};
```

Figure 7 Example of database connection pooling in a serverless function

3. 6. 2. Security

There is not much to be found with regard to serverless security, various authors mention that serverless applications have a larger attack surface due to the many separate function configurations. Therefore, Völker (2018) makes a case for applying proper access management, which basically means that a serverless function is by default not permitted to do anything in the larger cloud context, only access to resources that is really needed is white listed. This reduces the impact of a possible breach.

Additionally, PureSec[2] describes in a white paper the top 10 serverless security risks:

1. Function event data injection;
2. Broken authentication;
3. Insecure serverless deployment configuration;

[2] PureSec, 'PureSec Launches the First and Most Comprehensive Guide to Shed Light on Security Risks Related to Serverless Architectures', Tel Aviv Israel, *PureSec*, 2019, https://www.puresec.io/press_releases/sas_top_10_2018_released (accessed 11 June 2019).

4. Over-privileged function permissions and roles;
5. Inadequate function monitoring and logging;
6. Insecure 3rd party dependencies;
7. Insecure application secrets storage;
8. Denial of service and financial resource exhaustion;
9. Serverless function execution flow manipulation;
10. Improper exception handling and verbose error messages.

Some of these are general security risks (e.g. insecure application secrets storage) but become more risky due to the distributed serverless environment. Their white paper also provides best practices for each security risk respectively which is not repeated here in order to retain our scope.

3. 6. 3. Vendor lock-in

One of the biggest issues of serverless computing, as described in Section 3. 3. , is the risk of vendor lock-in. There are a few strategies found in literature to mitigate this risk. In a blog post by the company Vacation Tracker is described that vendor lock-in is not necessarily the right name for this issue, the author proposes 'switching cost'[3]. Since that is actually what is holding you back from moving your code from one cloud provider to another. In order to keep switching costs low it is important to keep your application architecture agile to allow for easy migration. The author argues for a 'ports and adapter' based architecture to support this. The best way to describe such an architecture is through Figure 8. The adapters are abstractions between the business logic and external components. This allows for quick switching of adapters in case a different database or other external service needs to be used.

According to Grumuldis (2019) vendor lock-in should not necessarily be seen as a limitation of serverless but more as a tradeoff: there are various platforms but all offer similar features, albeit in a different form or implementation. The author also proposes products as Serverless Framework to help reduce the switching costs and allow for easy migration between various platforms. Kritikos and Skrzypek (2018) also describe the use of serverless frameworks (e.g. the Serverless Framework) to combat vendor lock-in by allowing for easy switching between cloud platforms.

[3] Vacation Tracker, 'Fighting vendor lock-in and designing testable serverless apps using hexagonal architecture', *Vacation Tracker*, 2019, https://vacationtracker.io/blog/big-bad-serverless-vendor-lock-in/, (accessed 11 June 2019).

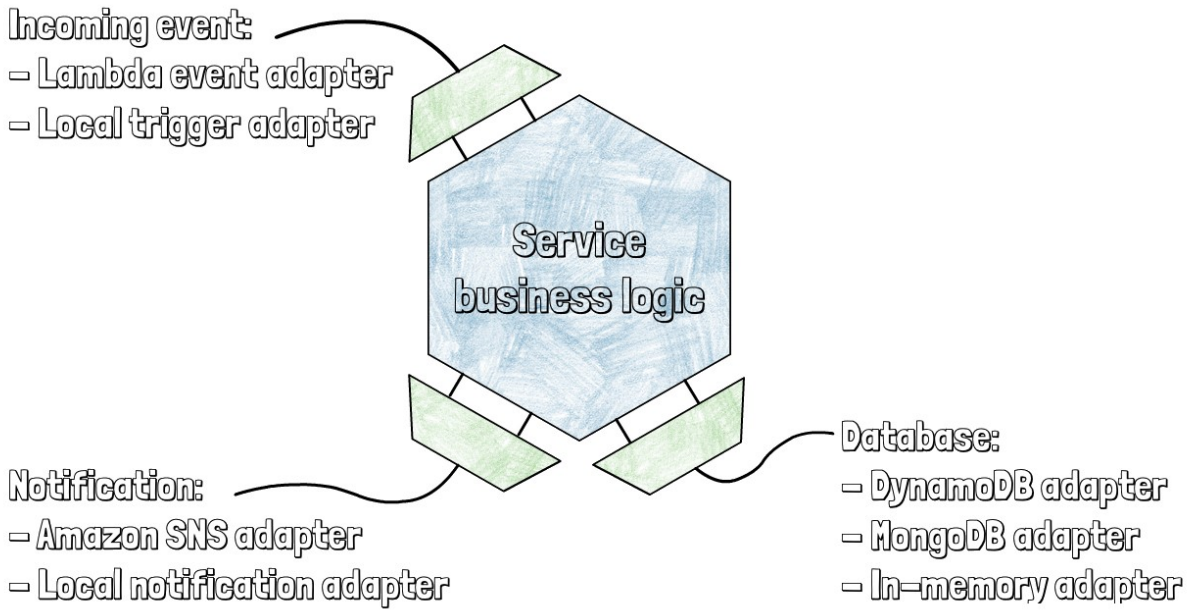


Figure 8 Example of 'ports and adapter' based architecture © 2019 Vacation Tracker

3. 6. 4. Costs

There are some authors that are trying to create cost simulators for FaaS (Manner, 2019), or algorithmically optimize the costs of a serverless application (Elgamal, 2018). However, there are not many cost related best practices mentioned in literature. One comment by Bardsley et al. (2018) to keep the function chain short to improve performance might have its reflection on cost savings as well, but is not its primary goal. In general can be said that proper monitoring is necessary to keep an eye on operational costs and proper security measures need to be taken so that DDOS (or similar) attacks do not result in untenable costs.

Additionally, it is important to consider the fact that serverless is not necessarily cheaper than conventional deployment in all cases. It has the potential to reduce costs if applied properly. We could fairly confidently state, as a rule of thumb, if your application is experiencing consistently spread load it is cheaper to go with a dedicated instance, if the load is fluctuating heavily, or just very low, serverless could result in a lower cloud computing bill (Baldini, Castro, et al., 2017; Chapin & Roberts, 2017; Warzon, 2016).

3. 6. 5. Monitoring and debugging

Live debugging is not supported by most serverless cloud providers, therefore development and operations personnel is dependent on posteriori log-based debugging (Manner, Kolb, & Wirtz, 2018). A plug and play solution for this problem could be AWS CloudWatch, which is a monitoring service that automatically collects a lot of important information on applications running in the cloud (Tran, 2017). Important to note is that since this is an Amazon product this

will add to the vendor lock-in and will only work as expected for other cloud services running on AWS. There are similar tools available (such as Graphite) but none come close to the features and integration CloudWatch has to offer. The important features are being able to monitor performance, set custom alerts, and gain insights into request and response combinations in order to debug issues after they have occurred.

That being said, it seems that monitoring and debugging is a very cloud platform specific problem and is most easily solved by using the monitoring tools provided by the cloud platform itself. Therefore, this can be considered another important aspect when choosing a cloud platform.

3. 6. 6. Architecture and migration

Grumuldis (2019) discusses three code patterns for serverless applications. The author is considering these code patterns in the context of the Serverless Framework which is a software tool that allows developers to write serverless functions once and then deploy it to all major cloud platforms without additional changes. The first is the monolithic pattern (see Figure 9): one serverless function that serves as the single entry point for the application and executes different logic based on the payload. This pattern is especially useful when migrating legacy code to a serverless application for the first time, since it minimizes the need for rewriting the code. There are also some downsides e.g. hard to monitor and debug, increased package size and therefore lower performance. The second is the service pattern (see Figure 10): in this pattern each serverless function is handling all request for a specific (business) domain. This already improves separation of concerns, simplifies monitoring and debugging, reduces package size and can be managed by different teams if desired. The third is the nano-service pattern (see Figure 12): this is probably the most recognizable, each serverless function has a very specific responsibility, for example performing CRUD operations. This makes monitoring and debugging easier since the functions are completely separated, allows for better resource allocation due to spread load on the functions, and reduces impact possible bugs have on the overall serverless application. Downside is that it is possible that due to infrequent execution some functions might suffer more from cold-starts.

```
/**
 * Monolithic serverless function example.
 * This method takes an event object parameter with certain properties
 * and determines which sub-function to execute accordingly.
 * @param {Object} event
 * @param {Object} context
 * @param {Function} callback — Method which sends a (status) response back
 */
module.exports.handler = async (event, context, callback) => {
  if (typeof event.type === 'string') {
    switch (event.type) {
      case 'add_product':
        addProduct(event.product); // Execute sub-function
        break;
      case 'delete_product':
        deleteProduct(event.product); // Execute sub-function
        break;
      case 'purchase_product':
        purchaseProduct(event.product, event.user); // Execute sub-function
        break;
    }
  }

  // Handler was called with invalid event data
  return callback(new Error('Invalid event type provided'), {
    statusCode: 400, // Bad Request
  });
};
```

Figure 9 Example of monolithic serverless function code pattern

File 1: product.js

```
/**
 * This handler reacts on https://example.api.com/product requests, and handles all
 * requests related to product.
 * @type {*}
 */
const productHandler = require('productsHandler');
module.exports.handler = async (event, context, callback) => {
  return productHandler(event, context, callback);
};
```

File 2: payment.js

```
/**
 * This handler reacts on https://example.api.com/payment requests, and handles all
 * requests related to payments.
 * @type {*}
 */
const paymentHandler = require('paymentHandler');
module.exports.handler = async (event, context, callback) => {
  return paymentHandler(event, context, callback);
};
```

Figure 10 Example of serverless function service code pattern

```
/**
 * Serverless nano-service pattern example.
 * Handler is registered on https://example.api.com/product and responds
 * to GET requests only.
 * @param {Object} event
 * @param {Object} event.filter
 * @param {Object} context
 * @param {Function} callback – Method which sends a (status) response back
 */
module.exports.handler = async (event, context, callback) => {
  return callback(null, getFilteredProductsList(event.filter));
};

/**
 * Serverless nano-service pattern example.
 * Handler is registered on https://example.api.com/product and responds
 * to PUT requests only.
 * @param {Object} event
 * @param {Object} event.product
 * @param {Object} context
 * @param {Function} callback – Method which sends a (status) response back
 */
module.exports.handler = async (event, context, callback) => {
  return callback(null, addProductToDatabase(event.product));
};
```

Figure 12 Example of serverless function nano-service code pattern

```
// Import business logic from outside function handler
const BusinessLogic = require('./BusinessLogic');

/**
 * Serverless function example.
 * @param {Object} event
 * @param {Object} context
 * @param {Function} callback – Method which sends a (status) response back
 */
module.exports.handler = async (event, context, callback) => {
  BusinessLogic.productHandler();
  return callback(null, true);
};
```

Figure 11 Example serverless function with separated core logic

A presentation by AWS[4] describes some best practices and architecture patterns for serverless computing (especially on AWS). Separate the serverless function handler from core logic, see the example in Figure 11. Minimize package size of the function to be deployed, this minimizes the initialization time. Use environment variables to modify operational behaviour, the idea is that the environment variables can be changed 'on the fly' while the code is running in production. Contain

[4] Gupta, A. and Hornsby, A, 'Serverless Architecture Patterns And Best Practices', AWS, 2017, https://www.jfokus.se/jfokus18/preso/Serverless-Architecture-Patterns-and-Best-Practices.pdf, (accessed 12 June 2019).

dependencies in the function packages, do not rely on external (CDN) dependencies, they can become unavailable or unresponsive. Finally the author argues to use the AWS products X-Ray (for monitoring) and Step Functions (for composition), it is indeed best to use platform provided products since they work together very well, however keep in mind the vendor dependency and cross-platform restrictions.

As mentioned by Jambunathan and Yoganathan (2018) the best way to start converting a monolith to serverless functions is by first moving towards a micro-service based architecture. This brings amongst other benefits, speed, agility and scalability. After this migration step it is easier to identify which parts of the application are suitable for serverless deployment.

More importantly, a completely serverless architecture is not always a desired goal. Micro-services and serverless functions are complementary technologies which are optimally used in conjunction (Jambunathan & Yoganathan, 2018). The authors give some recommendations for serverless and micro-service architectures: serverless functions are for stateless short running tasks in contrast to micro-service which can be stateful and are long running (1), functions scale very precisely, more so than micro-services (2), use serverless functions when it is really needed or when it has a substantial improvement (3). The third recommendation is because of the increase complexity serverless brings to development and operations teams, it should not be considered a default choice.

Amazon presents the Well-Architected Framework[5], which is a whitepaper focussed on five pillars of design principles and best practices. These pillars are: operational excellence (1), security (2), reliability (3), performance efficiency (4) and cost optimization (5). For each pillar a couple of design principles and best practices are defined.

1. Perform operations as code, annotate documentation, make frequent, small, reversible changes.
2. Implement a strong identity foundation, enable traceability, apply security at all layers, automate security best practices, protect data in transit and at rest, prepare for security events.
3. Test recovery procedures, automatically recover from failure, scale horizontally to increase aggregate system availability, stop guessing capacity, manage change in automation.
4. Democratize advanced technologies, go global in minutes, use serverless architectures, experiment more often, mechanical sympathy.

[5] Belt, D. 'The 5 Pillars of the AWS Well-Architected Framework', AWS, 2018, https://aws.amazon.com/blogs/apn/the-5-pillars-of-the-aws-well-architected-framework/, (accessed 12 June 2019).

5. Adopt a consumption model, measure overall efficiency, stop spending money on data centre operations, analyse and attribute expenditure, use managed services to reduce cost of ownership.

Although these recommendations are somewhat AWS biased and not specifically for serverless architecture, they are good guidelines to use in the construction of cloud applications.

Already addressed in this section is the 'ports and adapter' architecture style, which allows for quick migrations by switching the ports and adapters (see Figure 8). This architecture style is not only valuable for combating vendor lock-in, it also serves as a way to easily migrate code from monolithic or micro-services to serverless. The ports and adapters can be switched by ones that are connection to external serverless function for example.

A general recommendation is to make use of the available serverless frameworks (Kritikos & Skrzypek, 2018). There are quite a few available, however the Serverless Framework seems to be leading at the moment according to the serverless framework evaluation by Kritikos and Skrzypek (2018). These frameworks help with portability of the serverless functions on to various cloud platforms by offering standardisation and abstracting away from the technical specificities of a cloud platform (Kritikos & Skrzypek, 2018).

3. 7. Conclusions

This section will present the conclusion for each research question addressed in this chapter. It will discuss the main findings and tries to synthesize the scattered knowledge on each respective topic.

3. 7. 1. Research question 1

This section concludes RQ1 "What are the currently known benefits and challenges of serverless computing?".

There seems to be a lot of potential for serverless computing, looking at the identified benefits in recent literature; granular scaling opportunities, optimizing resource utilization, reducing operational costs, improvements to the development process and various architectural opportunities.

These benefits will be considered as goals in this work for applying a serverless architecture to a software application. Additionally, they can be used to discuss to (un-) successfulness of the application of serverless computing and act as guidance during the various DSRM iterations on the artefacts which are to be created in this work.

Considering the benefits presented in Section 3. 2. it was to be expected that serverless computing also comes with a set of challenges. This section described the most prevalent challenges found in recent literature; performance challenges, complexities related to the development and operations processes, composition issues, migration and de-composition challenges, the threat of vendor lock-in, security considerations and non-arbitrary operational cost reductions. This is by no means an

exhaustive list of challenges, however it does addresses issues that are common to most, if not all, serverless computing implementations.

The challenges presented in this section are important for this work. Having a clear picture of the challenges serverless implementations face helps to identify and create artefacts incorporating work-arounds or solutions using the DSRM.

The serverless computing related benefits and challenges that are described in Section 3. 2. and Section 3. 3. are analysed and graphically presented in Figure 13. The red components are challenges, whereas the green are benefits. Some benefits can be grouped together with challenges based on the specific area of the serverless computing topic they are concerned with. The figure also demonstrates how granular scaling and optimising resource utilisation eventually contribute to the benefit of reducing operational costs. The benefits and challenges that are grouped together

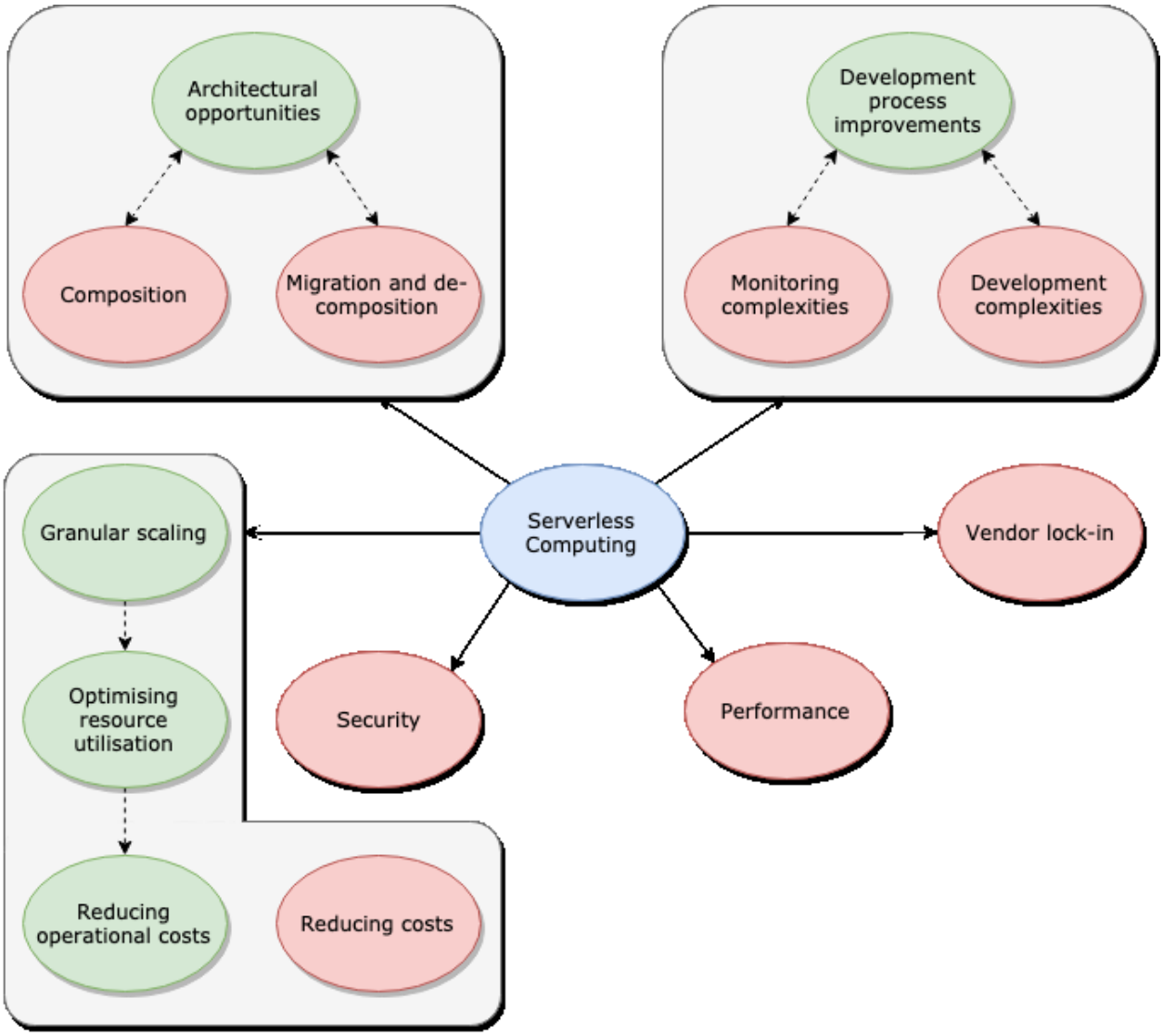


Figure 13 Serverless computing benefits and challenges map

could potentially be considered as related concepts when looking at solutions for the challenges or best practices for the benefits.

3. 7. 2. Research question 2

This section concludes RQ2 “What are the existing approaches or characteristics to determine which parts of an application are suitable to be implemented in a serverless way?”.

After performing a structured literature review on the topic of characteristics of serverless suitability it appears that there are very few literature sources addressing this specific issue. And what these sources present has not been peer-reviewed, nor empirically evaluated in real-word contexts. In fact, to the best of the author’s knowledge there were only two works that were found to be very useful; the master thesis of Grumuldis (2019) and the one of Völker (2018). Drawing on the works of these authors a set of characteristics was identified that concern serverless suitability; response time, invocation patterns, type of operation, data limits, vendor dependence and runtime restrictions. Based on these characteristics a flow chart was created (see Figure 14) which further summarizes and clarifies the body of literature into a practical and concise model. The contribution of this section to this thesis as a whole is the centralized knowledge around the characteristics of serverless suitability and the actionable flow chart that can help guide practitioners in making the choice to go serverless or not.

3. 7. 3. Research question 3

This section concludes RQ3 “How could cloud native architectures be composed and orchestrated to enable the full potential of serverless computing?”.

Literature showed there is quite some knowledge with regard to serverless composition, however mostly on either a low or high level; looking at either the FaaS platform architecture or the high-level application components. A more practical approach with regard to the composition of serverless functions in a possibly hybrid environment seems to be less represented.

This claim seems to be supported by a couple of authors. Kratzke (2018) argues that existing cloud standards only focus on a specific cloud service category (mostly on the infrastructure level) and pass over a more integrated view point, the author proposes that the development of an integrated reference model which includes best practices of practitioners could contribute to filling this gap. Furthermore, van Eyk et al. (2017) identify the research challenge of a currently missing reference architecture for serverless cloud computing applications. They state that such a reference architecture could assist software developers and software architects by providing examples and best practices related to the implementation of hybrid serverless cloud applications.

Serverless composition services offer an architecture pattern where a client invokes a single remote function which starts a cloud-based workflow of multiple serverless function invocations created using the serverless composition service. This moves the complexity to the cloud and offers better insights and graphical user interfaces to create workflows. Client focused compositions are an

architecture pattern that moves the orchestration of remote API calls to the client. Suspending serverless functions while making asynchronous calls to other serverless functions is suggested as an improvement of serverless composition as it would respect the serverless trilemma. Using a brokering layer in a hybrid serverless architecture context is suggested to manage the complexity of connecting private and public clouds. This is a well-known concept in computer science and might therefore be a good solution to this context.

Two modelling approaches were identified that could be applied in a serverless context, however both have quite a high barrier-to-use. TOSCA is focused on being an executable model which brings unnecessary complexities with regard to the modeling process. TheArchitect automatically generates a high level micro-services based or serverless architecture, but requires extensive input for the tool to work. The Cloud Application Maturity Model (CAMM) presents four levels of maturity which can be used to measure a cloud-native application's maturity. It was not designed for serverless architectures specifically, but could be applied regardless since a serverless architecture is often a cloud-native application as well. Additionally a reference model for cloud-native applications consisting of 4 viewpoints was found. This can be applied to serverless as well for the same reason as specified for the CAMM. The 4 viewpoints can be used as guidelines when modelling a serverless architecture.

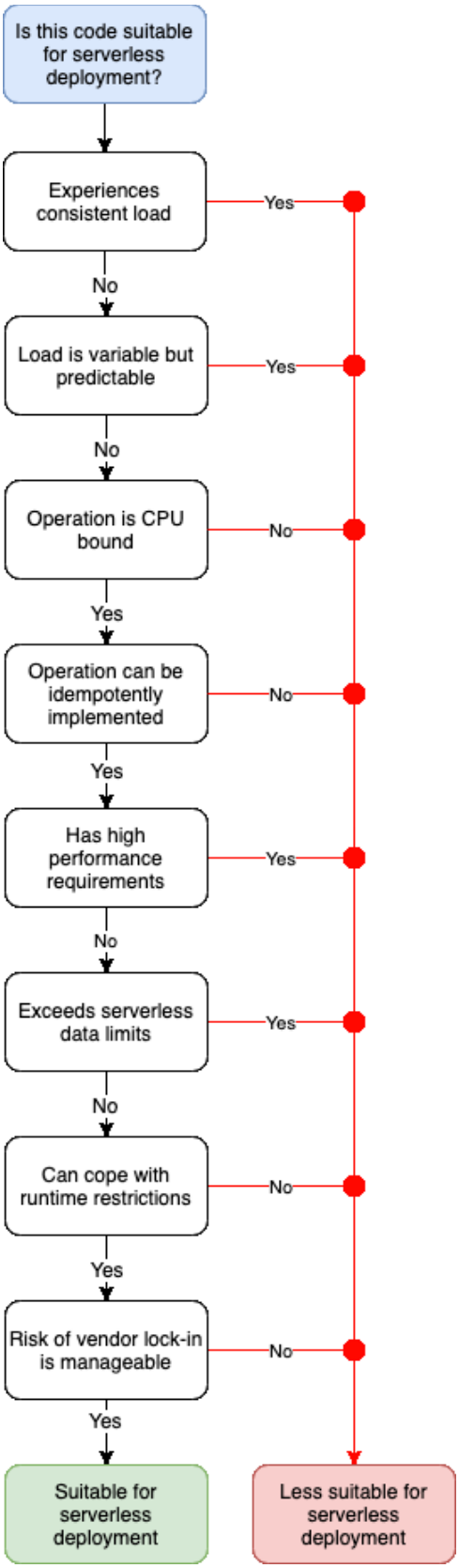


Figure 14 Flow chart to determine serverless suitability

3. 7. 4. Research question 4

This section concludes RQ4 “What are the best practices for creating a serverless application from scratch, and for migrating a non-serverless architecture to a serverless architecture?”

Initially the literature review yielded few to none usable results, however based on literature that was found during previous literature reviews in this thesis quite some usable knowledge could be gathered. Additionally, some explorative searching resulted in information valuable to the research question. Grey literature was added from a few sources such as practitioners and companies active in the serverless computing field. Some of the most notable results are repeated here.

Performance is one of the fields of serverless computing that is well represented in literature, many sources report on performance comparisons for example. Hence, there are quite a few strategies to optimize the latency of serverless applications. Regardless, it is important to note that serverless is best suited for background tasks which do not directly require interaction with users. A way to separate the user from the serverless execution is making use of an event-driven architecture to create a non-blocking UI.

Security, remains an issue, as it does for all (cloud) computing projects. It is an ongoing battle to keep your applications safe. Serverless might have a larger attack surface, but when applying proper access management (as for every piece of cloud software) this risk can be mitigated properly.

Vendor lock-in should not be seen as a limitation of serverless but rather as a trade-off. The knife cuts both ways; serverless cloud platforms provide many platform specific services with features that cannot be realized by external services, on the other side it tightens the vendor lock-in even more. As long as the software architect and developer keep an eye on the switching costs of their applications this is not necessarily a big problem. There are architecture patterns that allow for quicker (and therefore cheaper) switching, for example the ‘ports and adapter’ pattern. Also, tools such as the Serverless Framework help to reduce switching costs by acting as an abstraction layer between the serverless function and the various cloud platforms.

With regard to migration strategies two ideas were found: use a ‘ports and adapter’ architecture (1) and first migrate to micro-services architecture then slowly move to serverless if required (2). The first is of course a prerequisite, if your current architecture does not implement this other strategies need to be found or the architecture needs to be adapted to the ‘ports and adapter’ architecture first. It is important that an architecture is prepared for migration to micro-services or serverless.

Generally, it is important to conclude that serverless is not always the way to go. There are many situations where conventional VM or container based deployment is more economical and performant. Software architects and developers need to be aware of this and recognize situations where serverless is suitable and not. A rule of thumb could be: if your application is experiencing

consistently spread load it is cheaper to go with a dedicated instance, if the load is fluctuating heavily, or just very low, serverless could result in a lower cloud computing bill.

Chapter 4. Artefact design

This chapter is concerned with designing and presenting the artefacts for this thesis. This concerns the third DSRM step "design and development" (as described in Section 2. 2. "Research process"). Section 4. 1. presents the specific design process applied in this chapter. Section 4. 2. describes how the literature reviews presented in Chapter 3. were analysed and applied to the development of the framework. Section 4. 3. presents the actual design of the serverless architecture design framework (SADF). Finally, Section 4. 4. will conclude this chapter by summarizing the design of the SADF.

The SADF will be created based on the literature reviews on the topics of; benefits and challenges of serverless computing, suitability of serverless architectures, composition of serverless architecture and best practices of designing a serverless architecture. This framework will guide the process of implementing a serverless architecture while applying the gathered knowledge on serverless computing. The goal of this framework is to be used as a guideline for software practitioners to help them make the most of their architecture design.

In order to position the SADF and clarify its intended purpose, the taxonomy of theory types in information systems research is applied (Gregor, 2006). Based on the extensive literature analysis in Chapter 3. the framework tries to present a description and analysis of the topics relevant to the design process of a serverless architecture. In that regard the framework can be classified as a Type 1 theory (analysis). Additionally, in some cases the framework goes a step further and tries to explain how to do something, for example in the form of examples, techniques or methods. Therefore, the framework also has some elements of a Type 5 theory (design and action). Initially this might seem like an uncommon combination (Type 1 and Type 5), but actually it is described by Gregor (2006) as a possible interrelationship among theory types.

4. 1. Artefact design process

The artefact design process is further divided into the following steps: identify relevant viewpoints as basis for the framework (1), identify important cross-viewpoint variables to consider in the framework (2), construct a framework based on the first two steps (3), use this framework to design a serverless architecture based on the case (4), evaluate the framework by performing interviews with domain experts (5), and iterate on the framework by implementing the lessons learned during the case application and the interview evaluation (6). This sequence of steps (graphically represented in Figure 15) is chosen since it provides the opportunity to implement the gathered knowledge into a framework which will be tested and evaluated by applying it to a case study. Step 4, 5, and 6 are addressed in Chapter 6. where the evaluation and corresponding conclusions and recommendations are presented.

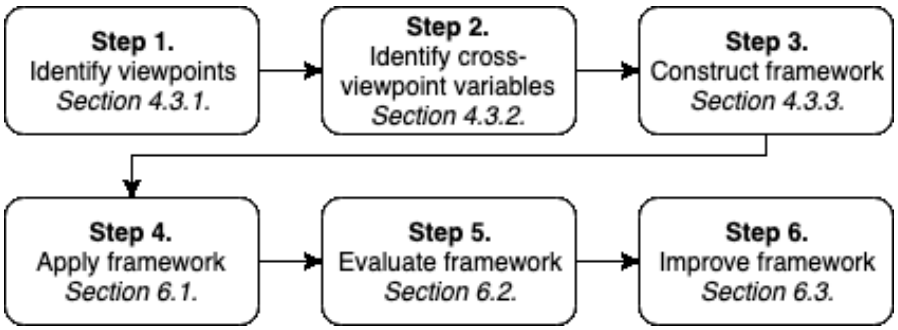


Figure 15 Artefact design process overview

4. 2. Literature analysis description

The structure of the SADF on the one hand, and the best practices, guidelines and knowledge supporting the framework on the other hand, are both based on extensive literature reviews as presented in Chapter 3. These literature reviews are focussed on a number of core topics related to serverless computing; challenges and benefits of serverless computing, characteristics of serverless suitability, composition of serverless architectures and in general best practices related to serverless (application) architecture design. This section shortly describes how the literature from these literature reviews was analysed with the goal of creating the SADF.

Initially, the structured literature reviews with regard to RQ1 (Sections 3. 2. and 3. 3.), RQ2 (Section 3. 4.), RQ3 (Section 3. 5.) and RQ4 (Section 3. 6.) were performed with the goal of answering the research question. Then, the literature related to each respective research question was analysed to discover general trends, important topics, often addressed problems and opportunities. The goal was to create a clear image of the most relevant topics that would affect a serverless architecture implementation. This resulted in a categorization of topics that are considered relevant for the serverless (application) architecture design process. By categorizing these topics it was possible to combine knowledge from various literature sources into a coherent and structured framework.

After the extensive literature analysis presented in Chapter 3. additional analysis was performed on the topic categorization itself. The topics were reviewed and categorized again on a meta-level by grouping similar topics and relating various concepts where possible. This higher level perspective was created to serve as the foundation of the SADF. The following section describes the creation of the SADF based on the knowledge sourced from the extensive literature reviews and analysis.

4. 3. Design of the serverless architecture design framework

This section presents the implementation of a serverless architecture design framework. First, based on the literature analysis, as discusses in Section 4. 2. , a number of viewpoints and cross-viewpoint variables are identified (Sections 4. 3. 1. and 4. 3. 2.). Second, the viewpoints and cross-

viewpoint variables are combined into a framework definition and are presented visually (Section 4. 3. 3.). Finally, each viewpoint and cross-viewpoint variable is discussed in-depth; common pitfalls, best practices and other relevant information with regard to each viewpoint and cross-viewpoint variable is presented (Sections 4. 3. 4. through 4. 3. 12.).

4. 3. 1. Identify viewpoints

Viewpoints are a classification of a complex problem domain into more comprehensible sub-domains. These viewpoints can be considered as different perspectives on a certain topic, in this case serverless architecture design. In order to identify viewpoints that are properly connected to the problem domain and provide a usable classification the literature presented in Chapter 3. is analysed. After analysing the literature for the first four research questions of this thesis a couple of important and recurring topics with respect to serverless architecture design start to appear. These are listed below and will be considered the four corner viewpoints of the serverless architecture design framework.

VP1. Configuration
VP2. Software design
VP3. Software architecture
VP4. Deployment

Configuration (VP1) is a perspective that is important to serverless applications; in order to deploy and maintain a serverless application many different components require configuration. This might seem like an obvious or irrelevant viewpoint, however, it is becoming increasingly complex to manage these configurations in a large cloud-native application. Especially when going all-in on serverless computing, since it brings many additional services (e.g. message queues, workflow composition services) that need to be configured and connected to each other. Moreover, the serverless functions themselves are also configurable in many ways (e.g. CPU, RAM). This is a viewpoint that is crucial to a successful serverless application and is therefore included in the serverless architecture design framework.

Software design (VP2) is about the code level design of a software application[6]. It is not directly tied to the architecture of an application, but serverless architectures are very 'bare bones', meaning: the code is extremely closely connected to the actual (cloud-native) architecture. These two concepts cannot be viewed separately in the context of serverless computing.

[6] Codeburst.io, 'Software Architecture – The Difference Between Architecture and Design', *Mohamed Aladdin*, 2018, https://codeburst.io/software-architecture-the-difference-between-architecture-and-design-7936abdd5830 (accessed 29 July 2019).

Software architecture (VP3) is about the higher level structure of a software application[7]. It offers a perspective on the composition of serverless functions within a larger application (either solely serverless, or a mixed application architecture with various cloud architecture patterns).

Deployment (VP4), is an important viewpoint as inherent to cloud-native software is that it has to be deployed to the cloud. Specifically in the case of serverless this is somewhat different from conventional deployment due to its granularity. The granularity of the deployment artefacts allows for equally granular scaling and operational cost reduction. Depending on each cloud platform the format of the serverless function has to be changed, each cloud platform uses a different abstraction for serverless functions. There are many differences between cloud platforms which can be very relevant when considering where to deploy a serverless application (e.g. performance and costs). Therefore, deployment is considered the fourth and final viewpoint.

4. 3. 2. Identify cross-viewpoint variables

Cross-viewpoint variables are considered to be characteristics of a serverless architecture that depend in one way or another on the architecture or application itself. These variables are important to consider during the entire architecture design process and are affected or addressed by more than one viewpoint. These variables are identified after analysing the literature in Chapter 3. and are listed below.

VA1. Performance
VA2. Vendor lock-in
VA3. Security
VA4. Costs
VA5. Serverless suitability

Performance (VA1) is a recurring characteristic of a serverless architecture (Baldini, Castro, et al., 2017; Bardsley et al., 2018; Chapin & Roberts, 2017; Garcia Lopez et al., 2019; Grumuldis, 2019; Jackson & Clynch, 2018; Lloyd et al., 2018; van Eyk et al., 2017; Van Eyk et al., 2018; Yan et al., 2016). It can be considered from the configuration, software design, and software architecture viewpoint and is affected by many aspects of the serverless architecture.

Vendor lock-in (VA2) is an often discussed aspect of serverless applications (Adzic & Chatley, 2017; Baldini, Castro, et al., 2017; Chapin & Roberts, 2017; Eivy, 2017; Grumuldis, 2019; Villamizar et al., 2017). It can become a constraining factor when the architecture is not designed

[7] Codeburst.io, 'Software Architecture – The Difference Between Architecture and Design', *Mohamed Aladdin*, 2018, https://codeburst.io/software-architecture-the-difference-between-architecture-and-design-7936abdd5830 (accessed 29 July 2019).

to allow easy (cheap) switching between cloud platform providers. This variable is most relevant to the software architecture and software design viewpoints.

Security (VA3) is an important characteristic of all software applications. For serverless architectures there are a few things that are somewhat different from conventional cloud applications (van Eyk et al., 2017; Völker, 2018; Wagner & Sood, 2016; Yan et al., 2016). This variable should be considered from two viewpoints: configuration and software architecture.

Costs (VA4) are often the driver for implementing a serverless architecture in the first place, therefore this is a rather important variable (Adzic & Chatley, 2017; Baldini, Castro, et al., 2017; Eivy, 2017; Villamizar et al., 2017). It is not trivial to achieve operational cost reductions through a serverless architecture (Baldini, Castro, et al., 2017; Chapin & Roberts, 2017; Warzon, 2016). It is primarily affected from two viewpoints: configuration and software architecture.

Serverless suitability (VA5) is considered a cross-viewpoint variable, as it is important to always determine whether the goal you are trying to achieve, the part of a software system you are implementing or changing, is actually suitable for a serverless approach. Serverless is not a one-size-fits-all kind of technology, it is very specific in what it can achieve and when it should be applied to reap the most benefit (Grumuldis, 2019). There are various considerations to be made within all four viewpoints that affect the serverless suitability, therefore it is important to keep track of the variable across all viewpoints (Abad et al., 2018; Grumuldis, 2019; Völker, 2018).

4. 3. 3. Serverless architecture design framework definition and visualization

This section combines the identified viewpoints and cross-viewpoint variables into a graphical representation of the serverless architecture design framework. Table 2 presents a mapping of the cross-viewpoint variables onto the viewpoints. This shows the implied connection and overlap between the cross-viewpoint variables and the viewpoints.

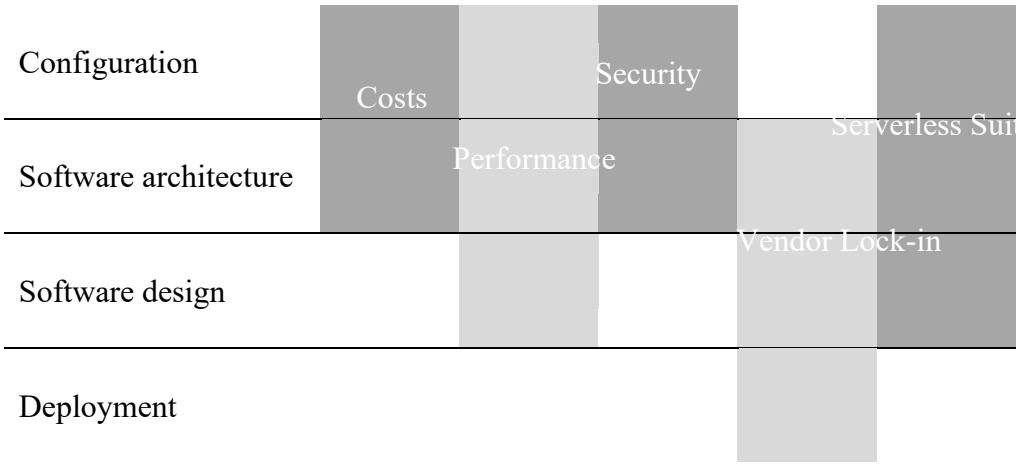


Table 2 Mapping cross-viewpoint variables onto viewpoints

In order to make the framework more easily comprehensible a graphical representation is created in the form of Figure 16. This section will shortly explain how this figure is constructed and how it should be interpreted.

The four large coloured circles represent the four identified viewpoints; software architecture, software design, configuration and deployment. The circles are positioned in such a way that the overlap indicates where the viewpoints touch upon each other. The viewpoints have fuzzy boundaries and (especially in a serverless context) are somewhat entangled. The viewpoints should be considered as the main perspectives one could take on a serverless software architecture.

The five cross-viewpoint variables are placed within the viewpoints; vendor lock-in, performance, serverless suitability, security and costs. The position indicates to what viewpoint the cross-viewpoint variables are most relevant. These are always multiple viewpoints, therefore they are positioned on the overlapping sections of the viewpoint circles. For example, the cross-viewpoint variables security and costs are positioned on the overlapping section of the software architecture

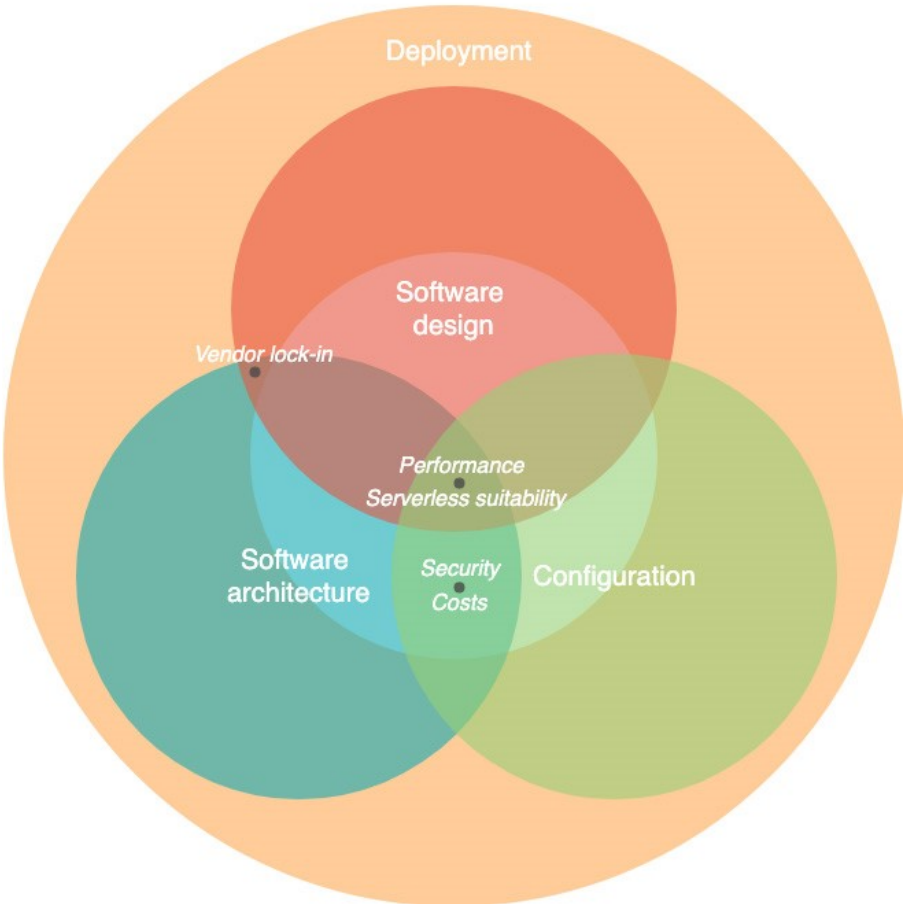


Figure 16 Graphical representation of serverless architecture design framework

and configuration viewpoint since they are considered to be most relevant to the respective viewpoints in a serverless context.

The framework should be interpreted as a conceptual model of the serverless architecture design problem domain. It describes the largest and most relevant concepts identified from literature; the viewpoints. Additionally, it describes non-functional software (architecture) characteristics which are identified from literature to be relevant to serverless architecture design. The framework can be applied either during application development process, or up-front to guide the initial architecture design. The goal of the framework is to provide handlebars for rationalizing about the serverless architecture problem domain. It presents the non-functional characteristics that require special attention in a serverless context. Additionally, it presents the four most important viewpoints which enforces reasoning about the problem domain from a specific point of view which should improve the understanding of the problem as well as provide a frame of reference. The framework is substantiated by many best practices and guidelines for designing and implementing serverless software architectures. These best practices and guidelines will be presented in the following sections, categorized by the defined viewpoints and cross-viewpoint variables.

It is completely up to the practitioner how to apply the framework, it does not require a specific methodology. As mentioned before, the framework can be applied either during the application development phase or up-front as a tool to apply while designing the initial software architecture. An example of how the framework could be used is the following. During the initial architecture design process the application requirements are analysed from the perspective of each of the viewpoints and cross-viewpoint variables. This will enforce thinking about the serverless specific topics and helps to earlier detect potential architectural issues. Additionally, it will ensure that all the differentiating aspects of serverless are at least discussed once. This makes sure that software architects or developers do not omit important considerations, especially the ones new to the serverless computing technology. The guidelines and best practices constituting the framework can be consulted by the practitioner when needed. They provide guidance in the context of the viewpoints and cross-viewpoint variables but also present deeper insights into potential problems, solutions and other relevant concepts.

4. 3. 4. Variable 1 – Performance

The performance variable is influenced by many factors. This section will shortly address the various factors and to which viewpoint they belong. For more elaborate discussion see the respective viewpoints.

From the software design viewpoint (VP2) a number of factors can be considered. Cold-starts (the time it takes for a serverless function instance to be initialized) can become quite slow depending on the configuration of the function (e.g. RAM/CPU and run time language) (Baldini, Castro, et al., 2017; Chapin & Roberts, 2017; van Eyk et al., 2017). This is also relevant in light of the configuration viewpoint (VP1). Cold-starts can drastically affect performance in some cases. Then

we also need to take into account that the performance of various run times can differ substantially between cloud platforms (Jackson & Clynch, 2018). I/O bound operations can slow down the function execution when waiting for I/O operations to complete, consider adopting an event-driven design pattern to overcome this (Grumuldis, 2019). In context of the software architecture viewpoint (VP3) it is advised to keep function chains (in compositions) short, this prevents accumulating latencies and cold-start delays (Bardsley et al., 2018).

4. 3. 5. Variable 2 – Vendor lock-in

Vendor lock-in is important to consider throughout the design process, since (if not accounted for) it can become quite an issue at a later moment when the application needs to be moved to another cloud platform (Adzic & Chatley, 2017; Baldini, Castro, et al., 2017; Chapin & Roberts, 2017; Eivy, 2017; Villamizar et al., 2017). Vendor lock-in is inherent to serverless computing and cannot be completely overcome, however there are ways to reduce the so-called 'switching costs'[8]. If the switching costs are kept low moving from one platform to another is oversee able. It also supports the decision process when considering to migrate to another platform. Vendor lock-in is a consistent trade-off that should be given the necessary attention in the design process (Grumuldis, 2019). Most notably in the context of the software design and architecture design viewpoints (resp. VP2 and VP3).

4. 3. 6. Variable 3 – Security

Some argue that serverless suffers from a larger attack surface due to the many endpoints (Yan et al., 2016). While this is true, it also depends heavily on the cloud infrastructure. To reduce attack surface consider running the serverless functions inside a VPC which is shielded from the outside network, or a similar construction (van Eyk et al., 2017). Above all, as with any software application, it is imperative that proper access management policies are in place (Völker, 2018). Make sure that the functions only have access to the resources they really need, this reduces the possible impact of a potential breach.

Finally, dealing with DDOS attacks. Since serverless functions scale very rapidly based on demand, and due to the 'pay-per-execution' cost model a DDOS attack quickly becomes very costly if not intervened. There are several strategies that can help in situations like this. First, set up proper budget alarms in the cloud platform's cost management interface. This makes sure you are notified when the costs grow above what is expected at a certain moment. Additionally, API throttling could be applied in the case that the maximum possible load of the application can be

[8] Vacation Tracker, 'Fighting vendor lock-in and designing testable serverless apps using hexagonal architecture', *Vacation Tracker*, 2019, https://vacationtracker.io/blog/big-bad-serverless-vendor-lock-in/, (accessed 11 June 2019).

estimated. This means that after a certain amount of function invocations all additional invocations will be blocked eliminating the possibility of unlimited growing costs due to a DDOS attack.

Security, as described above is mostly a configuration (VP1) effort, but it is important that the architecture is designed from the ground up with security in mind (VP3). Obviously, the code itself needs to be secure as well, however serverless functions are often placed behind various other cloud components which implement firewalls or authentication schemes. Therefore, they are not directly accessible through the internet which improves security.

4. 3. 7. Variable 4 – Costs

Reducing operational costs is often one of the reasons of adopting serverless technologies. This is not a trivial effort though, it depends on many factor such as applications workloads (Eivy, 2017). To make sure this goal is not lost during the design process it is defined as a cross-viewpoint variable. The viewpoints configuration (VP1) and software architecture (VP3) are most relevant to this variable. The configuration of the serverless function (e.g. RAM/CPU and language run time) determine the costs per execution and have a determining factor in the performance (and therefore duration) of the execution itself. The software architecture determines how often serverless functions are being invoked, and how composition of functions is handled. Workflow services provided by the cloud platform or by third-parties will induce extra costs.

4. 3. 8. Variable 5 – Serverless suitability

This variable is treated a bit more extensive here since it is applicable to the configuration viewpoint VP1, the software design viewpoint VP2, and the software architecture viewpoint VP3 and will not be repeated in the respective viewpoints sections. The information presented in this section can be used to determine during the design of the architecture whether certain parts of the system are suitable for a serverless implementation or not.

It is important to get a clear picture of the advantages a serverless architecture has to offer, and which disadvantages are inherently attached to that. This helps to clarify the potential problems that need to be overcome during the architecture design process.

Table 3 describes an overview of the benefits and challenges of serverless computing (Baldini, Castro, et al., 2017; Chapin & Roberts, 2017; Jonas et al., 2017; McGrath et al., 2016; Van Eyk et al., 2018; Varghese & Buyya, 2018; Villamizar et al., 2017; Wagner & Sood, 2016). These can help to determine whether a serverless approach could be taken and what problems need to be addressed. This is not an absolute or definitive list, it merely points out areas of interest that should be considered during this phase.

Benefits of serverless computing	Challenges of serverless computing

- Architectural opportunities
- Development process improvements
- Granular scaling
  o Optimising resource utilisation
  o Reducing operational costs
	- Performance
	- Vendor lock-in
	- Security
	- Non-trivial cost reduction
	- Composition complexity
	- Monitoring complexity
	- Development complexity

Table 3 Serverless benefits and challenges

The following list of characteristics (see Table 4) can be used to determine whether part of a software application or architecture is suitable for a serverless implementation or might not be as suitable (Abad et al., 2018; Eivy, 2017; Grumuldis, 2019; Völker, 2018). If the application matches more with the right column than the left it should be reconsidered if a serverless architecture is the way to go. None of the characteristics are deal breakers, most can be considered as trade-offs or are to be overcome if proper counter measures are applied. Table 4 is merely meant as an additional way to identify problem areas in the architecture and steer the choice for a serverless or conventional architecture.

Suitable for serverless	Less suitable for serverless
… experiences highly variable load	… experiences consistent load
… operations are CPU bound	… operations are IO bound
… operations can be idempotent	… operations cannot be idempotent
… is low to moderately performant	… is high performant
… does not exceed serverless data limits	… transfers high amounts of data
… can cope with runtime restrictions	… cannot cope with runtime restrictions
… vendor lock-in is manageable	… vendor lock-in is not acceptable

Table 4 Serverless suitability characteristics

4. 3. 9. Viewpoint 1 – Configuration

Choosing the right programming language for a serverless function is important. Serverless enables the use of different programming languages on a per function basis, this allows developers to apply the language that is best suited for solving the problem the serverless function is addressing.

If possible compare the performance of a specific language on the cloud platform you want to deploy to, it has been shown that performance differs substantially between the various available cloud platforms (Jackson & Clynch, 2018). In general it seems compiled languages (e.g. Java and .NET) perform faster once initialized but interpreted languages (e.g. Node.js and Python) have shorter initialization (and cold-start) times.

When dealing with relatively slow function executions consider upgrading the assigned RAM and CPU (Grumuldis, 2019). More RAM and CPU translates into higher execution costs per 100ms, but if the function is CPU bound it might become substantially faster and by doing so compensate for the higher execution costs. This could result in faster performance and lower or equal costs.

4. 3. 10. Viewpoint 2 – Software design

Here are code quality principles described that can be applied to the serverless code base to improve it in various ways.

4. 3. 10. 1. Design for performance

Since performance can be a bottle neck in a serverless architecture it is important to identify ways to optimize or work around it (Baldini, Castro, et al., 2017; Chapin & Roberts, 2017; van Eyk et al., 2017; Van Eyk et al., 2018).

Depending on the load of the application a percentage of the requests will experience cold-starts (most cloud platforms keep a function instance warm from 5 to 45 minutes). If the application has consistent load cold-starts will occur less frequent compared to an application that has fluctuating loads (Chapin & Roberts, 2017). In order to keep cold-starts as short as possible, a few things should be considered. Reduce the package size of the function (including possible dependencies), a smaller package will be quicker initialized (Grumuldis, 2019). Choose the right language and RAM/CPU, as explained above, interpreted languages (e.g. Node.js and Python) initialize quicker than compiled languages (e.g. Java) (Jackson & Clynch, 2018). More RAM and CPU positively influences the initialization time as well (Grumuldis, 2019).

Another strategy is a bit of an anti-pattern. There are many variants of this approach available, but they all boil down to the following: have a cloud component keep a set a functions warm at all times (Lloyd et al., 2018). Such a cloud component can be a micro-service or AWS CloudWatch for example, as long as it can make recurrent ‘keep-alive’ calls to functions to make sure the instances are not killed. This approach can be extended by a load prediction algorithm which increases and decreases the pool of warm functions depending on expected load. Consider if the additional costs of consistently making keep-alive calls to the functions are worth the performance optimization, maybe a dedicated micro-service instance might be a better candidate if cold-starts are problematic (especially if the load can be predicted to a certain degree).

Already discussed above, but worthy of repeating: keep the function chains short. Regardless of the composition style applied, it is often good to try to keep the chains short to reduce latencies and prevent accumulating cold-start delays (Bardsley et al., 2018).

Already mentioned above, but since this is a significant performance determining factor it is repeated here. The different cloud platforms have varying performances for the same programming language, check which platform might be best suited for the language you would like to use or determine the language based on the performance of a specific cloud platform.

I/O bound operations are operations that depend on reading or writing to some kind of persistent storage (e.g. local file system, remote database). An I/O bound operation can become quite expensive and low performant in a serverless context as the function is sitting idle while waiting for the I/O operation to complete (Grumuldis, 2019). Applying an event based code pattern can be a workaround: dispatch an event that initiates the I/O operation on the remote component (e.g. reading an image from an AWS S3 bucket) and have another function listening on the event that is dispatched when the I/O operation is complete, this function will then continue processing the result from the I/O operation (Grumuldis, 2019).

4. 3. 10. 2. Function re-use and grouping

When a serverless application grows the many functions become an issue to manage. This is not necessarily a serverless issue only, however it does get worse due to the granularity and plentiful nature of these functions. It is suggested to apply classis software engineering practices to manage the serverless functions. For example, re-use of functions can reduce the number of distinct functions in production systems. In order to support re-use serverless functions can be grouped together (using the Serverless Framework for example) into libraries of similar functionality, this improves discoverability of existing functions and their re-use as well. Another method to improve re-usability of serverless code is to decouple the core logic from the function handler (see Figure 17) [9]. By doing so, the core logic can be included as a library and referenced from the function handler. This makes the core logic re-usable and decoupled.

[9] Gupta, A. and Hornsby, A, 'Serverless Architecture Patterns And Best Practices', AWS, 2017, https://www.jfokus.se/jfokus18/preso/Serverless-Architecture-Patterns-and-Best-Practices.pdf, (accessed 12 June 2019).

```
// Import business logic from outside function handler
const BusinessLogic = require('./BusinessLogic');

/**
 * Serverless function example.
 * @param {Object} event
 * @param {Object} context
 * @param {Function} callback – Method which sends a (status) response back
 */
module.exports.handler = async (event, context, callback) => {
  BusinessLogic.productHandler();
  return callback(null, true);
};
```

Figure 17 Decoupled core logic and function handler

4. 3. 10. 3. Consider desired granularity of code patterns

The granularity of the applied code patterns is something to consider as well. It needs to fit the serverless application: it does not make sense to have an extremely granular set of serverless functions for a small serverless application, whereas a large application might benefit from a more granular approach due to improved separation of concerns. There are a few levels of granularity defined as a reference (Grumuldis, 2019).

A *monolithic* serverless function is a single function handler (or API endpoint) that handles several separate operations for various business domains (see Figure 18). Which operation will be executed is based on the payload received by the handler. This is a very low granular approach and is best fit for very small serverless applications.

A *service* based serverless function is similar to a monolithic serverless function only different in that it only handles operations for a specific business domain (see Figure 19). The operation that will be executed is again based on the payload received by the handler. This is a medium granular approach best fit for small to medium sized applications.

A *nano-service* based serverless function is the most granular approach (see Figure 20). It handles only one specific operation for one specific business domain. This is most common in large serverless applications where a high level of decoupling and separation of concerns is desired.

Note that the more granular you go, the more overhead is generated by having to deploy, monitor and develop each single serverless function. It can be argued that it is best to start at a low level of granularity, and slowly migrate to more granularity as needed. Another important thing to consider is that the more granular you go, the more likely it is that cold-starts will start to occur as there are more individual functions with a life cycle of their own.

```
/**
 * Monolithic serverless function example.
 * This method takes an event object parameter with certain properties
 * and determines which sub-function to execute accordingly.
 * @param {Object} event
 * @param {Object} context
 * @param {Function} callback — Method which sends a (status) response back
 */
module.exports.handler = async (event, context, callback) => {
  if (typeof event.type === 'string') {
    switch (event.type) {
      case 'add_product':
        addProduct(event.product); // Execute sub-function
        break;
      case 'delete_product':
        deleteProduct(event.product); // Execute sub-function
        break;
      case 'purchase_product':
        purchaseProduct(event.product, event.user); // Execute sub-function
        break;
    }
  }

  // Handler was called with invalid event data
  return callback(new Error('Invalid event type provided'), {
    statusCode: 400, // Bad Request
  });
};
```

Figure 18 Monolithic serverless function example

File 1: product.js

```
/**
 * This handler reacts on https://example.api.com/product requests, and handles all
 * requests related to product.
 * @type {*}
 */
const productHandler = require('productsHandler');
module.exports.handler = async (event, context, callback) => {
  return productHandler(event, context, callback);
};
```

File 2: payment.js

```
/**
 * This handler reacts on https://example.api.com/payment requests, and handles all
 * requests related to payments.
 * @type {*}
 */
const paymentHandler = require('paymentHandler');
module.exports.handler = async (event, context, callback) => {
  return paymentHandler(event, context, callback);
};
```

Figure 19 Service serverless function example

```
/**
 * Serverless nano-service pattern example.
 * Handler is registered on https://example.api.com/product and responds
 * to GET requests only.
 * @param {Object} event
 * @param {Object} event.filter
 * @param {Object} context
 * @param {Function} callback – Method which sends a (status) response back
 */
module.exports.handler = async (event, context, callback) => {
  return callback(null, getFilteredProductsList(event.filter));
};

/**
 * Serverless nano-service pattern example.
 * Handler is registered on https://example.api.com/product and responds
 * to PUT requests only.
 * @param {Object} event
 * @param {Object} event.product
 * @param {Object} context
 * @param {Function} callback – Method which sends a (status) response back
 */
module.exports.handler = async (event, context, callback) => {
  return callback(null, addProductToDatabase(event.product));
};
```

Figure 20 Nano-service serverless function example

4. 3. 11. Viewpoint 3 – Software architecture

Here are architectural quality principles described that can be applied to the architecture to improve it in various ways.

4. 3. 11. 1. Serverless trilemma

The serverless trilemma demonstrates the inherent challenges of serverless composition (Baldini, Cheng, et al., 2017). Summarized, the trilemma consists of three competing constraints: functions should operate as black boxes (1), a composition of functions should be a function (2), invocations should not be double-billed (3). This trilemma helps when thinking about serverless architecture composition patterns by providing insight into the trade-offs that are to be made.

4. 3. 11. 2. Composition styles

There are other ways to manage function chaining. Different composition styles have different goals and advantages. The most obvious composition style is chaining serverless functions by calling a function from within another function. This way a client using the ‘entry’ serverless function only has to call and await a single function call as the ‘entry’ function is handling coordination of further function executions. There are various issues to this approach that you should be aware of. First, this approach is hard to scale well. It becomes unmanageable to keep track of which function depends on which function, creating a web of dependencies that quickly becomes an operational issue. Second, it violates the third serverless trilemma principle (Baldini, Cheng, et al., 2017). As the ‘entry’ function is calling other functions it has to wait for them to

complete before it can return a result to the client. This waiting results in double billing, once for the function that is waiting and once for the function that is working.

Client focused composition is comparable to a 'thick client' where all the composition and coordination logic is embedded in the client itself (Kratzke, 2018). Instead of calling one serverless function that coordinates or relies on other functions, the client calls and coordinates these other functions itself. Although this style does not result in double billing, it must be questioned whether the client is suited to handle this, keeping in mind that the composition might evolve to become very extensive and complex. Therefore, in some cases external composition might be desired.

Composition can also be coordinated in the cloud by using a serverless composition service (e.g. AWS Step Functions or Azure Durable Functions). These services can only be used by serverless functions deployed on the respective platform. These composition services allow the creation of workflows by chaining serverless functions together through a visual tool provided by the cloud platform. The benefit compared to client focused composition is that it reduces complexity on the client side and allows for easier development of new, and changes to existing, workflows. These composition services do not violate the double billing principle, however these services charge similarly to serverless functions, based on execution. Consider if the costs are worth it for your specific application. Additionally, note that these serverless composition services experience additional performance overheads ranging from 0.3 to 32 seconds depending on the execution type (Garcia Lopez et al., 2019).

Especially in the context of hybrid or multi-cloud architectures, but also in general in order to prepare for vendor lock-in, it can be beneficial to adopt a loosely coupled or agile architecture. This means adopting architecture patterns that allow relatively easy refactoring. For example, being able to switch in and out certain components (e.g. queue services, authentication services, storage solutions) gives the architecture agility by reducing required development efforts. This also addresses vendor lock-in by keeping the switching costs low. An architecture pattern that serves this purpose is 'adapters and ports' also known as brokering layers (Vaquero et al., 2019). The concept introduces small interchangeable components called adapters (or ports) which has a middleware functionality for connecting various services and components. If desired, an adapter can be added, removed, or changed to accommodate for a change in the architecture.

4. 3. 11. 3. Keep function chains short

With regard to serverless architecture composition it is generally considered best to keep function chains short (regardless of the chosen composition style) (Bardsley et al., 2018). This has two reasons: it improves performance since each additional function in the chain adds latency and potential cold-start delay (1), and it decreases the architecture complexity and helps reducing tight coupling (2).

4. 3. 12. Viewpoint 4 – Deployment

4. 3. 12. 1. Cloud platforms and their differences

Each cloud platform has different characteristics, services, features and more. Therefore, it is important to properly investigate which cloud platform suits the serverless application best. There have been performance tests which indicate significant performance differences between cloud platforms for the same programming language: Node.js seems to perform better on AWS Lambda than on Azure Functions and for .NET C# vice versa (Jackson & Clynch, 2018). Additionally, consider the vast eco-system (existing of tools, services, documentation, APIs and more) that come with a cloud platform. In general AWS Lambda is considered the most mature, but depending on the serverless applications others might be more suited. It is worth the initial investment, as it can become costly to switch later on in the process.

4. 3. 12. 2. Hybrid and multi-cloud

Hybrid cloud composition is the concept of deploying a software system partially on a private cloud and partially on a public cloud. Multi-cloud composition is the concept of deploying a software system across multiple different cloud platforms. These two concepts are often interchangeably used, but are in fact different in nature. A proposed solution for managing these two composition challenges addressed by the software architecture viewpoint (VP4): applying adapters and brokering layers for each cloud platform to connect either with a private cloud or with another public cloud platform (Vaquero et al., 2019). The use of adapters and brokering layers also reduces the vendor lock-in since it allows faster (and therefore lower cost) switching between cloud platforms[10].

4. 3. 12. 3. Consider using a serverless framework

Serverless frameworks (such as Serverless Framework, Apex or Standard Library) are designed to simplify development of serverless functions, while simultaneously offering standardization and abstracting away from the technical specificities of a cloud platform by acting as an adapter for the various FaaS providers (Kritikos & Skrzypek, 2018). The Serverless Framework[11] in particular can deploy serverless functions to most (if not all) of the large FaaS providers without having to make changes to the code itself. There are many benefits connected to this ability, but the most notable is that it combats vendor lock-in. At time of writing, the Serverless Framework is considered the leading serverless framework and is applied by quite a few Fortune 500 companies.

---

[10] Vacation Tracker, 'Fighting vendor lock-in and designing testable serverless apps using hexagonal architecture', *Vacation Tracker*, 2019, https://vacationtracker.io/blog/big-bad-serverless-vendor-lock-in/, (accessed 11 June 2019).
[11] Serverless Framework, 'Build apps with radically less overhead and cost', *Serverless*, 2019, https://serverless.com/, (accessed 30 July 2019).

4. 4. Conclusion

The five cross-viewpoint variables (performance, vendor lock-in, security, costs and serverless suitability) and four viewpoints (configuration, software design, software architecture and deployment) give a structure to the problem domain of serverless architecture design. The cross-viewpoint variables and viewpoints allow for discussing the most relevant and common issues, opportunities and best practices from a couple of different perspectives. This should contribute to a better overall understanding of the problem domain and provide guidelines and relevant information to software architects and practitioners. The identified viewpoints and cross-viewpoint variables are summarized and presented in Table 2. It also demonstrates the overlap of the cross-viewpoint variables with the respective viewpoints.

The framework should not be considered to be exhaustive or complete in any form, it is solely based on what was discovered through the various literature reviews on the subject. These reviews were not systematic and therefore might miss some relevant literature. The framework should be viewed as a mental model of the problem domain categorized by the four most relevant perspectives, with important characteristics described as variables which are considered throughout the various viewpoints.

The framework as described above (including the guidelines and best practices) is rather extensive and complicated. In order to make the framework more comprehensible a graphical representation of the viewpoints and variables has been made based on Table 2 and is presented in Figure 16. The viewpoints are organized so that the overlap indicates where the different perspectives overlap as well. It can be concluded that all viewpoints have a certain amount of overlap with all other viewpoints. The deployment viewpoint is presented as embodying circle around the other three viewpoints due to it being the corner stone of serverless architectures and its impact on the remaining viewpoints. The variables are positioned in such a way that the overlap indicates to what viewpoint they are considered to be most relevant.

Chapter 5. Case analysis

This chapter presents a fictitious case description based on which a serverless architecture will be created. This chapter forms the basis for the fourth and fifth step of the DSRM "demonstration and evaluation" (as described in Section 2. 2. "Research process"). Based on the case analysis presented in this chapter a serverless architecture implementation will be created while applying the SADF. The goal of this implementation (as presented in Section 6. 1.) is to demonstrate and evaluate the proposed SADF.

5. 1. Case description

This section presents the case that will be used for the development of the serverless architecture and proposed serverless architecture design framework. It entails a couple of key processes of a fictitious IoT-enabled traffic control system. The nature of the system itself is not relevant to the development of the architecture and framework, it only serves as an example to build upon. First, the general structure of the system will be described. The system consists out of several components:

4. A Software-as-a-Service based application for traffic monitoring and control
    a. Client software for road supervisors
    b. Cloud-native backend system serving the clients
5. Various IoT nodes in the real world
    a. Road sensors which count traffic
    b. Traffic lights which can be controlled
    c. Hub nodes which are co-located to the various sensors and actuators
    d. Vehicles on the road

Some requirements are created to guide the creation of the serverless architecture:

4. Historic traffic data must be stored persistently
5. Live traffic data must be accessible by road supervisors
6. Road supervisors must be able to manually control the IoT actuators (e.g. traffic lights)

A few key processes of the traffic control system are described below. These are selected to be implemented in the serverless architecture.

1. Detect traffic through road sensors and display in client software
2. Cloud-native backend processes live traffic data and controls actuators
3. Cloud-native backend detects anomalies and alerts road supervisors
4. Road supervisors manually override actuators
5. Report historic traffic data via e-mail

5. 2. Process analysis

This section discusses the processes proposed in the previous section. These processes will be modelled so that they are correctly understood and can be translated into a serverless architecture.

5. 2. 1. Process 1 – Traffic detection and presentation

The first process is a basic one: it entails the connection between the sensors in the road which detect vehicles and, eventually, the client software used by the road supervisors to monitor traffic flows.

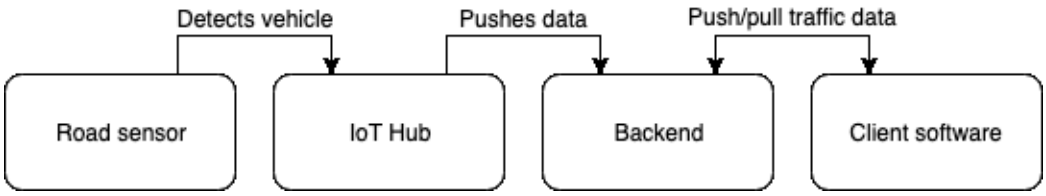


Figure 21 Process description of traffic detection and presentation

5. 2. 2. Process 2 – Controlling actuators based on live traffic data

The second process is similar to the first process in that it also starts with the vehicle detection by sensors in the road. It differs, however, in what happens with these detection events. The goal of this process is to establish the link between the road sensors, the traffic flow algorithms running in the cloud, and the traffic light actuators which eventually control traffic flows.

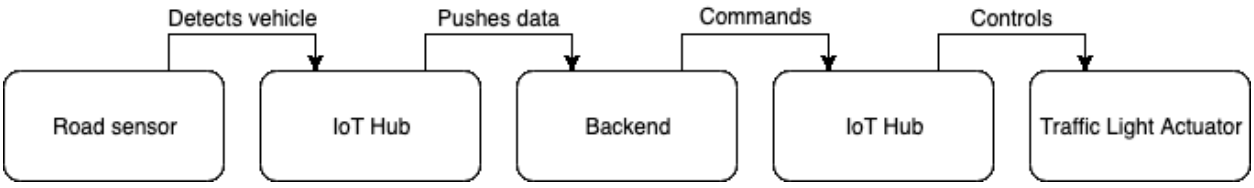


Figure 22 Process description of controlling actuators based on live traffic data

5. 2. 3. Process 3 – Anomaly detection and alerting

The third process is somewhat more complex. Again it starts with the traffic detecting road sensors which report their events to the cloud where the traffic flow algorithms run. The cloud processes these events and runs anomaly detection on the traffic flows, such anomalies could be indicators of a large traffic incident, road blockage, or anything else that would need attention of the road supervisors. The cloud then needs to alert the road supervisors in various ways; by showing the anomaly in the client software, by sending out e-mails, making automated phone calls and triggering visual and acoustic alarms inside the road supervisors office.

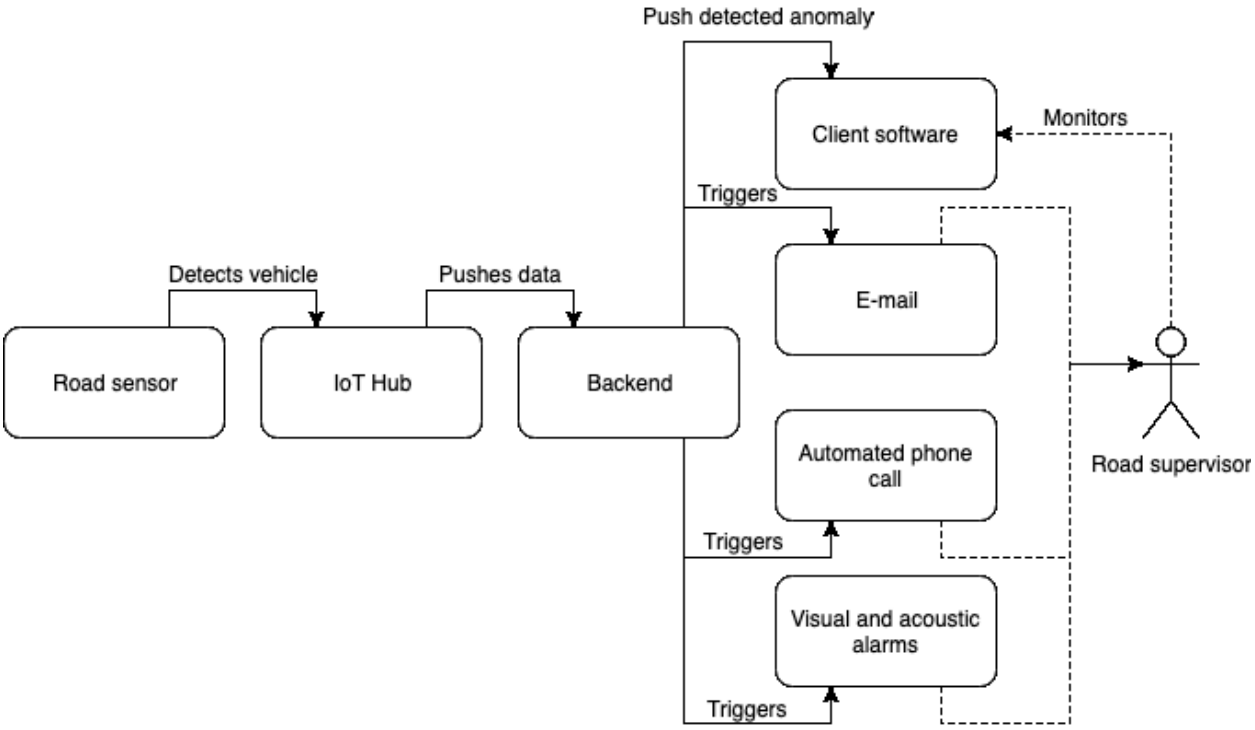


Figure 23 Process description of anomaly detection and alerting

5. 2. 4. Process 4 – Manual control by road supervisors

The fourth process is related to the manual control of traffic light actuators by road supervisors. Normally, the system runs on its own and optimizes traffic flow where possible, however, in the case of a detected anomaly road supervisors need to be able to manually override traffic lights to manage the chaos. Additionally, in cases where oversized road transport or emergency vehicles need to be guided through a city it is important that the traffic lights they run into can be manually overridden to give them a quick pass.

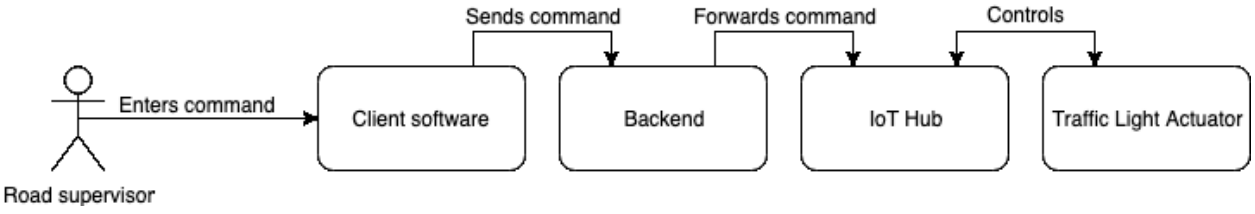


Figure 24 Process description of manual control by road supervisors

5. 2. 5. Process 5 – Generate historic traffic data report

The fifth and final process is about generating a e-mail report based on historic traffic data. A road supervisors should be able to trigger the generation of such a report from the client software. Given a few parameters (such as time span) the system should fetch historic data and process it in such a way that it is usable for the road supervisor as is delivered in his e-mail box.

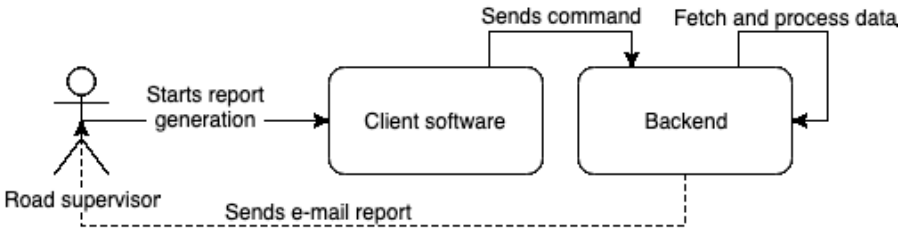


Figure 25 Process description of historic traffic data report generation

5. 3. Conclusion

The five processes described in this chapter are central to the serverless architecture that will be designed in the following chapter. It is not necessarily relevant what these processes entail, however they need to be interesting and complex enough so that they will be proper candidates for demonstrating the benefits, challenges and best practices of applying a serverless architecture.

Chapter 6. Evaluation

This chapter presents the evaluation of the SADF. This concerns the fourth and fifth DSRM step "demonstration and evaluation" (as described in Section 2. 2. "Research process"). The framework is evaluated in two steps: (1) a serverless architecture design is created by the author based on a case analysis (presented in Chapter 5.) by applying the SADF (presented in Chapter 4.) and (2) semi-structured interviews are performed with three domain experts (either software architects or experienced developers). The goal of the first step is twofold: by applying the framework it is demonstrated how it could be used in practice, additionally it acts as evaluation by looking at how the framework aligns with the practice of architecture design. The goal of the second step is to improve the framework by gathering opinions and experiences from domain experts. This chapter is divided into two parts. Section 6. 1. presents the case study evaluation and has the dual-purpose of addressing both DSRM step four "demonstration" and DSRM step five "evaluation". Section 6. 2. presents the structured interview evaluation and addresses the fifth DSRM step "evaluation" as well. Finally in Section 6. 3. the chapter will be concluded and recommendations for the SADF will be presented. Additionally, a revised version of the graphical representation of the SADF will be presented.

6. 1. Case study

The objective of this section is to create a serverless architecture design by applying the SADF presented in Chapter 4. The architecture will be based on the case analysis as presented in Chapter 5. Note that it is not the specific design choices that are central to this evaluation, but more so a practical demonstration of the adoption of the SADF.

Initially, the viewpoints, as defined by the framework, will be discussed in the context of the case analysis. This will provide the chance to reason about the architecture to-be from a number of relevant perspectives. Following, the cross-viewpoint variables, as specified by the framework, will create insight into how the various variables should be treated in the design process. Combined, these two steps will form the foundation for the serverless architecture design.

6. 1. 1. Viewpoint analysis

6. 1. 1. 1. Configuration

Since there are no requirements regarding the choice of a runtime environment, Node.js can be considered a good choice. It is available on most, if not all, cloud platforms and offers a large eco-system with examples, tools, libraries and more. Moreover, it is an interpreted language which initializes relatively quickly compared to compiled languages. Which is in this case more valuable than high performant execution once initialized. The performance of the Node.js runtime does differ between cloud platforms, but is considered acceptable for this purpose. The exact

configuration of the CPU and RAM of the serverless functions is to be considered during development.

6. 1. 1. 2. Software design

This viewpoint is concerned with the actual code of the application and is most relevant during the implementation of the architecture. Even though, we will present some (probable) choices that have to be made during the implementation process.

Since performance is not extremely important for the serverless parts of the system this variable is not addressed in detail. For the highly performant parts of the system a micro-service approach will be applied. Various performance optimisation techniques are discussed in the framework and can be applied, but are not required for this architecture. The I/O bound operations will be moved to the micro-service based part of the system to optimize performance of reading and writing live traffic data.

Function re-use will be encouraged by decoupling core logic from function handlers and grouping the core logic into libraries. This offers maintainability advantages in the long run. The granularity of code patterns is difficult to determine upfront, since it depends on the size and maturity of the system. Therefore, the objective is to start at a service level based granularity, this offers good separation of concerns while still being maintainable in the early stages of development. Eventually, the granularity might move towards a nano-service level if desired.

6. 1. 1. 3. Software architecture

This viewpoint focusses on the higher level application architecture. There are various composition styles to apply. In general keeping function composition chains short is desired, therefore this will be pursued in this specific architecture as well. The architecture needs to be designed loosely coupled and remain agile. This is needed due to the multi-cloud requirement, the system needs to be able to move between cloud platforms without high switching costs. In order to achieve such an agile architecture, the architecture style “adapter and ports” will be applied. This results in an architecture that can be connected and disconnected to various cloud components just by switching or designing a new adapter or port. These adapters and ports will be small software components that act as a middleware between the core system logic and the cloud components (such as different FaaS platforms, message queues, data stores etc).

6. 1. 1. 4. Deployment

This viewpoint is concerned with the deployment aspect of a serverless application. We have already specified the multi-cloud requirement and hence the application should be able to support simultaneous deployment of various cloud platforms. This removes the need for choosing a specific cloud platform based on its price, performance, and various other features.

In order to achieve multi-cloud deployment in an efficient way, the only practical solution known at the moment is adopting the Serverless Framework. This is a development framework that allows

developers to write function handlers once, and based on the specification deploy it to a number of different FaaS providers. The Serverless Framework can be viewed of as an implementation of the adapter and ports architecture style discussed in the software architecture viewpoint.

6. 1. 2. Cross-viewpoint variable analysis

6. 1. 2. 1. Performance

For the IoT-enabled traffic control system performance is not a major requirement in general. Most parts of the system can deal with minor delays, for example the storing of historic traffic data, and alerting road supervisors of anomalies. Of course, the latter is preferably fast, however a few (milli-) seconds are not going to make the difference in this context.

A few parts of the system are considered to be high performant: controlling actuators (e.g. traffic lights) and the anomaly detection based on live traffic data. If needed road supervisors need to be able to control actuators without delay. Additionally, anomaly detection algorithms should be able to keep up with the live traffic data flow.

6. 1. 2. 2. Vendor lock-in

Since this is an infrastructure critical system it is of utmost importance that it is highly available. Most cloud platform providers offer at least a 99.95% monthly uptime percentage, but to increase the availability in the rare case of a provider wide outage it is desired to deploy the system to multiple cloud platforms (e.g. AWS and Azure). This requires a multi-cloud approach. The system should be able to function with at least one of the cloud platforms being unavailable. By applying a multi-cloud approach vendor lock-in is considerable less problematic as the architecture will be designed to move over various cloud platforms.

6. 1. 2. 3. Security

Since this system is going to be able to control traffic actuators and possibly communicate with vehicles on the road it is critical to be well protected. Controlling actuators should only be possible by road supervisors and the system needs to be protected against false inputs (e.g. DDOS attacks). The remaining components, such as storing historic traffic data and offering live traffic data insights should be reasonably protected from outside interference.

6. 1. 2. 4. Costs

Costs are needed to be kept low as possible, since the system will be created on a limited budget. Also it is important that the costs will not grow exponentially when usage of the system increases.

6. 1. 2. 5. Serverless suitability

For this variable Table 4 is consulted and applied to the case context. Based on that can be concluded that certain parts of the system might not be best suited for serverless implementation. As described under the performance variable, certain parts of the system have high performance

requirements (e.g. anomaly detection and actuator controlling). Additionally, the anomaly detection based on live traffic data is highly I/O bound, and requires reading (and writing) high amounts of data. Due to these two factors it is not desired to design these parts of the system in a serverless way, it would be better to deploy these parts to dedicated instances which are continuously running and can be both horizontally as well as vertically scaled for optimal performance.

The other parts of the system are better suited for a serverless architecture, since performance is less of an issue, there are no initial runtime restrictions and vendor lock-in acceptable if a multi-cloud-ready architecture design is implemented.

6. 1. 3. Serverless architecture design

After analysing the viewpoints and variables of the SADF a coherent picture was created of how the architecture could be designed. It gave direction to the design process and the resulting architecture. The architecture is created using a modelling tool specifically aimed at AWS cloud architectures[12]. Therefore, the cloud components used in the design are AWS specific. However, the concepts of the design are widely generalizable and the basic cloud components are available across cloud platforms. For the used cloud components and their description see the legend in "Appendix 1. Serverless architecture design legend". In general the architecture is designed from a relatively high level perspective. This has two reasons: for brevity the designs are kept compact and to the point, additionally the designs are merely a means to an end. The goal of this chapter is demonstrating the process of applying the SADF more than designing a complete and highly detailed cloud-native software architecture.

The architecture is divided into two separate views, this was done to be able to present the design on A4 paper format. If desired the two views can be merged to generate a complete overview.

The first view presented in Figure 26 entails the architecture with respect to the following case analysis processes:

- Process 1 – Traffic detection and presentation
- Process 2 – Controlling actuators based on live traffic data
- Process 4 – Manual control by road supervisors

The second view presented in Figure 27 entails the architecture with respect to the following case analysis processes:

- Process 3 – Anomaly detection and alerting

[12] Cloudcraft, 'Visualize your cloud architecture like a pro – Create smart AWS diagrams', New York, USA, *Cloudcraft*, 2019, https://cloudcraft.co/ (accessed 2 July 2019).

- Process 5 – Generate historic traffic data report

6. 1. 3. 1. Description of view 1

This section will address the different architecture components of view 1 (see Figure 26). This view entails all the components needed for case analysis processes 1, 2 and 4. At the bottom of the architecture the two types of IoT nodes can be found: road sensors and traffic light actuators.

The road sensors report to a nearby IoT hub whenever traffic is detected. The data from these IoT hubs are send to a AWS Kinesis real-time data stream. This is a serverless cloud service offered by AWS which ingests large amounts of data in real-time and does some initial processing simultaneously, additionally it handles storing the raw data. After the data is stored it is processed by the micro-service responsible for traffic data processing. This results in processed and aggregated data that will be stored long term in a relational database.

The traffic light actuators are eventually controlled by the road supervisors through a nearby IoT hub from the web-client interface. The interface allows the road supervisors to make an API call to the micro-service responsible for communicating to IoT nodes. These API calls are routed through ELBs (elastic load balancers) which direct and distribute traffic to available micro-services. The micro-service then directly targets an IoT hub with the command to change a traffic light.

The micro-services discussed before are all EC2 instances, which are AWS dedicated elastic cloud compute instances. There are a few different micro-services in this architecture with different responsibilities: traffic data processing and anomaly detection (1), and traffic light controlling (2). These micro-services are all configured to be auto-scaling, meaning that if the load on the micro-services of a specific type becomes larger than a preconfigured threshold it will start a new instance to compensate for the increase in load and maintain optimal performance.

The web-client is used by road supervisors to monitor real-time traffic and control traffic light actuators. The web-client is a web application statically hosted through an AWS S3 bucket (cheap and simple object storage service) and Route 53 DNS service. It fetches traffic data through API calls to the micro-service routed via the load balancer. The micro-service retrieves the data from the relational database. The web-client can register itself to receive updates from the micro-service as new data comes in, this is done through a publisher/subscriber pattern [13].

This view of the architecture demonstrates the parts of the system that were not entirely serverless suitable. Therefore, a micro-service based implementation was implemented. The AWS Kinesis service is an implementation of serverless computing. The micro-services could be replaced by

[13] Microsoft Azure, 'Publisher-Subscriber pattern', *Microsoft Azure*, 2019, https://docs.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber (accessed 2 July 2019).

serverless functions that react on events by the Kinesis data stream. However, due to the performance requirement this was not desired. In similar situations where minor delays are acceptable serverless functions might be a suitable replacement for always-running EC2 instances (depending on the costs).

6. 1. 3. 2. Description of view 2

This section will address the different architecture components of view 2 (see Figure 27). This view entails all the components needed for case analysis processes 3 and 5. The lower half of the architecture is cut off but is identical to the lower half of the architecture presented in view 1.

The first three Lambdas (AWS FaaS instances) are triggered by the micro-service responsible for traffic data processing and anomaly detection. The micro-service has certain algorithms that detect abnormal traffic flows and when this is detected three Lambdas are invoked. One which sends an e-mail to the road supervisors with information on the anomaly that is detected. One that triggers a visual and acoustic alarm in the road supervisors' office through a local IoT device. And finally one that makes an automated phone call to a few of responsible road supervisors. For e-mail and automated phone calls (third-party) serverless services are used. AWS offers a Simple E-mail Service (SES) which is invoked from the Lambda. There are various third party serverless services such as Twillio[14] which offer automated phone calls through a single API invocation with a pay-by-execution cost model. These Lambdas could also be merged, however, keeping them separate increases agility of the architecture and enables very easy adding, removing and changing the way road supervisors are notified of anomalies.

Additionally, road supervisors are able to generate historic traffic reports on demand. Through the web-client a Lambda is invoked which starts a data retrieval process on the relational database. This can potentially concern large amounts of data that needs to be aggregated by the database, therefore the Lambda is killed after starting the data retrieval process and another Lambda is waiting for the event that the data retrieval and aggregation has finished. This second Lambda then receives the data and continues processing it into an e-mail for the road supervisors. Note that for the first Lambda one of the micro-services could also be used, this might be necessary if the data size becomes too big for Lambdas to handle. This is something that should be tested and experienced in practice. If it is needed to let a micro-service do the data fetching and aggregation it can still trigger the second lambda which parses the data and triggers the SES to send an e-mail. This again contributes to the overall agility of the architecture. This is an example of an event based architecture pattern which helps to overcome the unsuitability of I/O bound operations by serverless functions. This pattern also respects the third aspect of the serverless trilemma: "invocations should not be double-billed".

[14] Twillio, 2019, https://www.twilio.com (accessed 2 July 2019).

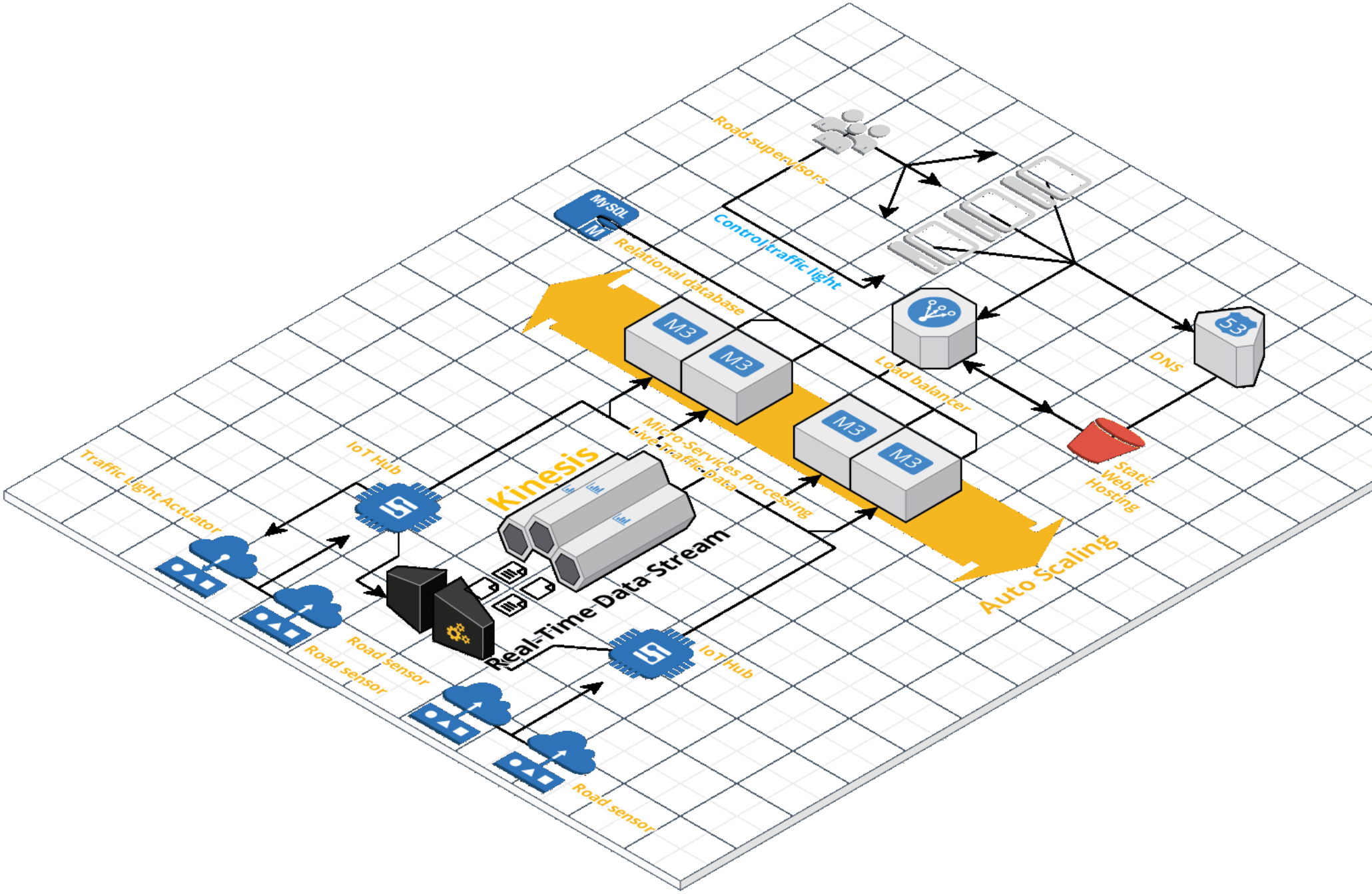


Figure 26 Architecture design view 1 (entails case processes 1,2 and 4)

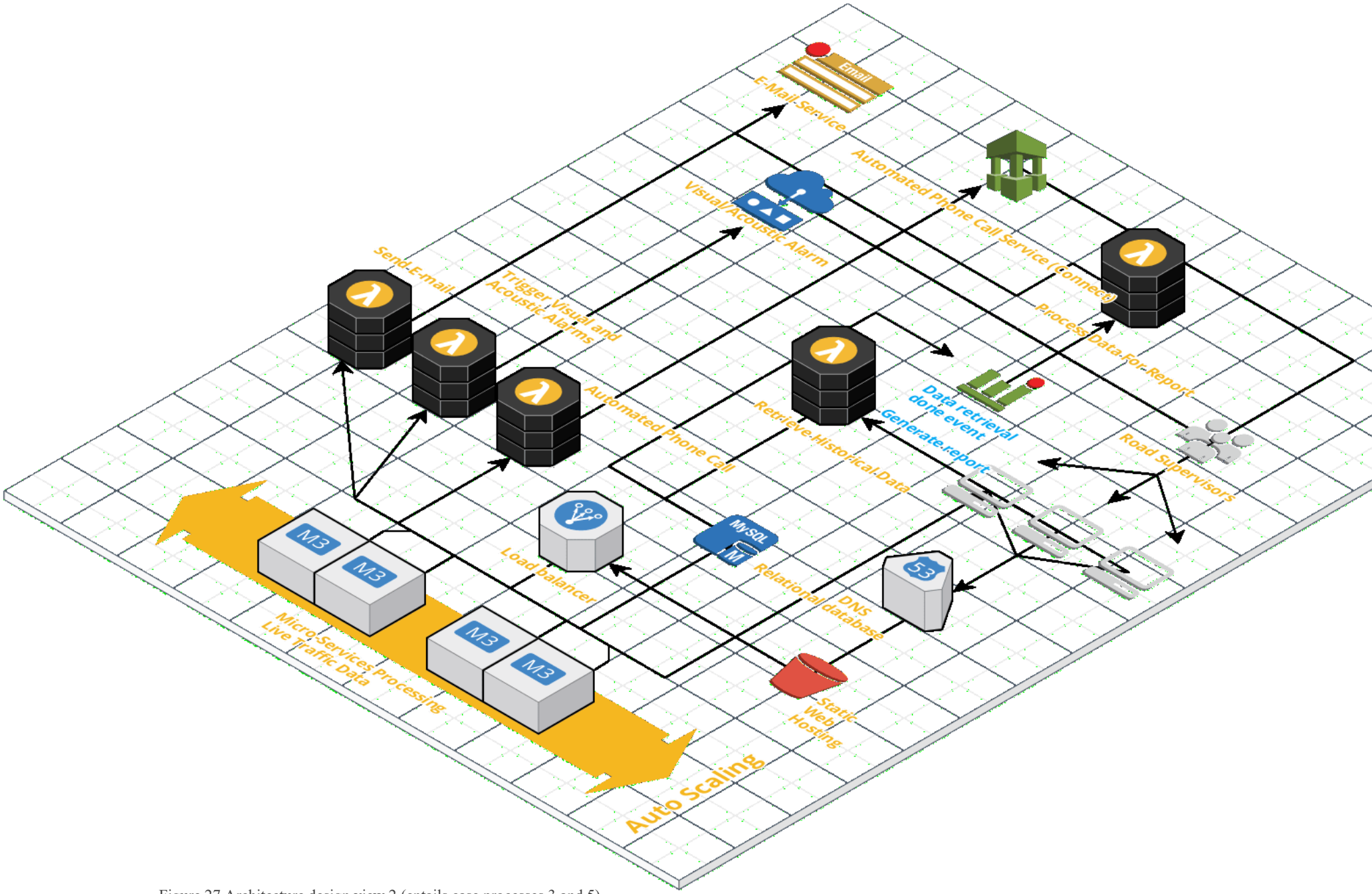


Figure 27 Architecture design view 2 (entails case processes 3 and 5)

6. 1. 4. Case study results

This section will discuss the findings after applying the SADF onto the case study. Section 6. 1. presents two contributions: a textual analysis of the case description based on the four SADF viewpoints and five SADF cross-viewpoint variables (1), and a serverless architecture design which is created based on the SADF analysis of the case study (2).

The first thing that was noted during the application of the SADF is that it is focussed solely on the serverless aspects of a software application. This was an intentional choice, creating a coherent architecture design framework encompassing all possible (cloud-native) software architecture styles would be too complex and extensive for the scope of this thesis. Nevertheless, this is a limitation that should be noted. Due to this fact, a practitioner applying the SADF must have at least basic knowledge of general software architecture design in order to apply the serverless specific principles.

Connected to the limitation discussed above, the SADF misses connection with general cloud computing concepts and components. The list of available hardware components, software components, serverless services and tools is ever growing and difficult to keep up with. In order to create a state-of-the-art serverless software architecture it might be needed to get familiar with the current state-of-the-art of the cloud-native ecosystems. This can be considered a limitation of the SADF, on the other hand, there is room in the framework (especially in the software design and software architecture viewpoints) to focus on this aspect if desired.

Another observation is that the software design viewpoint is not very well represented by the textual analysis and the architecture designs. This is caused by the fact that software design is relatively low level, and the architecture designs were intentionally designed from a higher level of perspective. Additionally, software design can become extremely granular. Often the software design of an application is determined during the implementation of the code itself, especially in development teams adopting an agile development methodology. Therefore, it might be an improvement to stress the difference between software design and architecture design with respect to at what point in the software system development and design phase these viewpoints are relevant.

Because the software design viewpoint is rather difficult to represent (as discussed above) the two cross-viewpoint variables configuration, performance and vendor lock-in might be under represented in the SADF based analysis. On the other hand, the SADF explicitly presents the variables that require (cross-viewpoint) focus in a serverless architecture design. It is then up to the practitioner(s) to make sure it is given the needed attention in the design and development process.

During the evaluation it was noted that the cross-viewpoint variable serverless suitability is more relevant to the software architecture, software design and deployment viewpoints. Its relevance with regard to the configuration viewpoint can be neglected.

In general only a few recommendations resulted from applying the SADF onto the case analysis. This is mostly caused due to the focus being to a higher degree on the demonstration of applying the SADF as opposed to a formal evaluation.

6. 2. Semi-structured interviews

In order to evaluate the proposed framework with respect to alignment with practice, usability and usefulness three semi-structured interviews are conducted. The process for conducting the structured interviews is based on the process for conducting in-depth interviews as presented by Boyce and Neale (2006). They describe six steps:

1. Plan (Section 2. 2.)
2. Develop instruments (Section 6. 2. 1. and Appendix 2. Interviews)
3. Train data collectors (-)
4. Collect data (-)
5. Analyse data (Section 6. 2. 2.)
6. Disseminate findings (Section 6. 3. 2.)

Steps 3 and 4 are not explicitly described in this chapter, the other are addressed in their respective sections. Training data collectors is not necessary, all interviews are conducted by the author. Collecting data has been performed in practice, but is considered superfluous to describe separately here.

The goal of the interviews is to get a sense of the usefulness and usability. Additionally, the alignment with practice and the question whether the framework would be beneficial in the architecture design process are central to the interviews. To evaluate the usefulness and usability of the proposed framework the Technology Acceptance Model (TAM) by Davis et al. (1989) is partially applied. The two variables: perceived usefulness (PU) and perceived ease of use (PEOU) are to be determined by a set of statements which will be rated by the interviewee on a range from ‘very false’ to ‘very true’. The two variables PU and PEOU are considered “the main variables required to determine the variance in a user’s intention behaviour” (Davis et al., 1989).

6. 2. 1. Interview structure

The interview is divided into four steps which will be shortly discussed below. Throughout the interview a semi-structured approach is applied. In practice this means that when necessary the discussion is diverted from the questions in order to go deeper into the subject. For reference, the complete interview in both English and Dutch can be found in “Appendix 2. Interview”.

6. 2. 1. 1. Interview step 1

The interview starts with three questions which are aimed at getting to know the experience of the interviewee with respect to the topic of software architectures and more specifically serverless architectures. These questions are stated below.

1. What is your experience in software architecture?
2. Are you into serverless computing technology?
3. What do you do as part of your job in regard to serverless computing technology and software architectures?

6. 2. 1. 2. Interview step 2

In the second part of the interview, the interviewer provides explanation about the SADF and how it came to be. The viewpoints and variables are presented and shortly discussed without trying to form the opinion of the interviewee too much. This is done using the graphical representation of the framework as shown in Figure 16. This step also allows for additional questions by the interviewee concerning the SADF.

6. 2. 1. 3. Interview step 3

The third part of the interview is the core of the semi-structured interview. Four questions are asked with a few distinct goals. The first question aims at getting the interviewee's opinion with regard to the alignment of the viewpoints with respect to the interviewee's professional experience:

4. To what extent do you think the viewpoints align with the practice that you observe in the projects that you work on?

The second question is of similar nature but concerns the alignment of the cross-viewpoint variables:

5. To what extent do you think the cross-viewpoint variables align with practice that you observe in the projects that you work on??

The third question asks whether the interviewee expects the framework could contribute to designing better serverless architectures when applied in practice:

6. Do you think applying the framework would result in better serverless software architectures?

The fourth question asks for a recommendation from the interviewee based on his professional experience with regard to improving the framework:

7. Thinking from your professional experience, would you add any specific aspect to improve the framework?

6. 2. 1. 4. Interview step 4

The fourth and final part of the interview consists of 6 statements. The interviewee is asked to rate these statements with respect to the degree of agreeableness. The rating scale used is as follows:

Very true – True – Neutral – False – Very false

Three statements are focussed at assessing the interviewee's perceived ease of use (PEOU) with respect to the framework:

8. Learning to use the framework for designing serverless applications would be easy for me.
9. It would be easy for me to become skilful at applying the framework.
10. It is easy for me to understand the framework based on the graphical representation.

Three statements are focussed at assessing the interviewee's perceived usefulness (PU) with respect to the framework:

11. Using the framework would enable me to more easily create serverless software architectures.
12. Using the framework would enable me to create better serverless software architectures.
13. Using the framework would help me to create a more structured understanding of the problem domain.

6. 2. 2. Semi-structured interviews results

This section will present the results of the semi-structured interviews. First, each interviewee will be described based on the background information questions. Second, each interview question will be discussed and the respective answers of the interviewees will be summarized and presented. Third, the results of rating the statements will be presented.

6. 2. 2. 1. Interviewees

Interviewee 1 (I1) has 4 to 5 years of professional experience with large cloud architectures. These architectures consist of more than 30 distributed micro-services, connect with tens of thousands of physical devices and contain about 15 applications of serverless computing. The interviewee is most experienced with AWS cloud infrastructure and the corresponding FaaS AWS Lambda.

Interviewee 2 (I2) has over 12 years of professional experience with software architecture and software development. In particular the interviewee mentions to have experience with managing physical and virtual servers and the corresponding (cloud-based) infrastructure, embedded software, mobile app development and cloud-native applications. The interviewee has been working with serverless technologies since its inception around 5 years ago (2014).

Interviewee 3 (I3) has about 7 years of academic and practical experience in the field of IT and software architectures. Additionally, the interviewee mentions to have experience with cloud-native technologies such as containers and FaaS, in particular on the AWS cloud platform.

6. 2. 2. 2. Answers to interview questions

4. *"To what extend do you think the viewpoints align with the practice that you observe in the projects that you work on?"*

I1: The deployment viewpoint is not considered to be an important viewpoint as it is presented currently. Additionally, it is considered too broad, since it can entail various management related aspects as well. The boundaries between software design and software architecture are not directly clear. In general, the viewpoints are considered to be somewhat ambiguous and subjective. Of all viewpoints, configuration seems to be the most clear. Interviewee argues that the (definition of the) viewpoints can be changed due to their ambiguity, but the variables remain consistent. Interviewee stresses that based on the graphical representation the viewpoints are not well-defined enough, and proposes to add small examples for each viewpoint to make them clearer and more distinct.

I2: Interviewee remarks that based on the graphical representation it is expected that the size of the viewpoint circles represent their importance. Interviewee additionally mentions that the importance of these viewpoints significantly differ between specific software systems. Besides, the size of the overlap of the circles are expected to have similar meaning. Interviewee notes that the boundary between software design and software architecture is not very explicit and depends on interpretation and personal experience. The difference between configuration and software architecture seemed clearer to the interviewee. Software design and configuration on the other hand are considered quite overlapping by the interviewee. With regard to the deployment viewpoint the interviewee mentions that this viewpoint indeed has overlap with all other viewpoints in practice. In general the interviewee notes that the boundaries between the viewpoints are not that well defined nor easily interpretable.

I3: The deployment viewpoint is considered rather relevant to the interviewee as this is a substantial part of the daily job activities. The same holds for the configuration viewpoint. With regard to the software architecture viewpoint the interviewee states that in practice this is often something that arises automatically when applying a bottom-up approach. In that scenario, the viewpoint would make less sense. However, the interviewee mentions that it is better to have an up-front architecture design to guide the application implementation. Therefore, the viewpoint would be quite useful to increase the architecture quality. In general, the interviewee argues that with regard to the viewpoint nothing seems to be missing and the defined viewpoints seem to cover the serverless architecture problem domain quite well in practice.

5. *"To what extend do you think the cross-viewpoint variables align with practice that you observe in the projects that you work on?"*

I1: The variables initially seem to be more relevant and interesting to the interviewee than the viewpoints. Vendor lock-in is considered very relevant and important to the interviewee. A number of arguments are given to support this (e.g. dependence, unpredictable, moral argument: code should be portable). Interviewee argues that the security variable should also be connected to the software design viewpoint and gives the example of defining which components are allowed to invoke other functions or components. After discussing that this is considered by the author to

belong to the configuration viewpoint the interviewee agrees that it is indeed more connected to configuration than software design.

I2: Interviewee discusses that vendor lock-in can occur on various levels, and therefore within other viewpoints as well. For example, choosing a third-party library can form a vendor lock-in which is related to the software design viewpoint. After explaining that this interview focusses on the cloud-native aspects of a software architecture the interviewee agrees with the vendor lock-in variable placement between software architecture, software design and deployment. Also the interviewee agrees with the relevance of the vendor lock-in variable to practice. In general interviewee comments that the variables addressed by the framework are considered relevant and important. Interviewee states that all variables have a relation with all viewpoints depending on the interpretation and the way you interpret the terms. Suggestion by interviewee, place variables in the centre of all viewpoints, since the boundaries of the viewpoints are ambiguous the variables cannot be properly confined by this categorization. Interviewee notes that some variables might only be relevant during specific phases of the design and development phase of a software project. The framework does not consider these phases.

I3: Interviewee mentions that the cross-viewpoint variables seem to be properly aligned with practice. Vendor lock-in is especially important since it is inherent to the serverless technology according to the interviewee. Initially, the interviewee argued that costs should also be affected by software design. After discussing this cross-viewpoint variable in the context of serverless computing the interviewee agreed that software design might not be especially relevant in this context. At least less relevant than the other two viewpoints. Finally, the interviewee mentions that security might also be relevant to the software design viewpoint. This observation indicates that this interviewee also has some issues with the ambiguity of the viewpoint boundaries.

6. *"Do you think applying the framework would result in better serverless software architectures?"*

I1: Interviewee agrees that compared to not using any framework the serverless software architecture would definitely benefit from applying the framework. The reason the interviewee gives for this is that the framework aids the practitioner by acting as a checklist. This makes sure that all important aspects are covered during the design process. According to the interviewee a big problem many people have is determining when to go serverless and when not. The serverless suitability variable of the framework would be of help in these situations. The interviewee also sees value in the framework from the perspective of evaluating an (existing) architecture using the cross-viewpoint variables. This would give insight into the quality of the architecture with respect to its serverless characteristics. On a more critical note, the interviewee does not see the added value of the connections between the variables and the viewpoints. It might make more sense to separate these two concepts since the variables are to be considered from the perspective of all viewpoints (depending on interpretation), according to the interviewee.

I2: It is not directly clear to the interviewee how to apply the framework, whether to use it as a checklist or as a pre-design phase reference. After explaining that every viewpoint and variable is supported by various best practices and recommendations the interviewee thinks that the framework will definitely help in creating better serverless architectures. Apparently, this was not directly clear from the graphical representation of the framework. Interviewee stresses that the framework would be most beneficial in the form of a checklist. Additionally, the interviewee notes that with respect to cloud-native architectures the framework structure makes more sense since it implies a structured mental model of the design to-be. This would also help thinking about the boundaries of the various viewpoints and how, when and where to address the variable concepts according to the interviewee.

I3: The interviewee argues that by applying the framework the software (architecture) quality would increase. According to the interviewee this is due to the fact that the important (differentiating) factors of serverless computing (in contrast to regular software architectures) are listed and are discussed within the proper context. Combined with the guidelines and best practices this is considered quite useful to the interviewee. The interviewee states that the viewpoints and cross-viewpoint variables provide extra focus to the important aspects of serverless architectures. This would help practitioners to improve the decision making process and ample considerations to be made during the design and implementation of a serverless architecture.

7. *"Thinking from your professional experience, would you add any specific aspect to improve the framework?"*

I1: The balance between the various trade-offs concerning serverless computing could be more emphasized. For example the trade-off between costs and performance, or load and serverless suitability. In addition, the interviewee states that adding examples to each viewpoint definition would help to define the boundaries between the various viewpoints and contribute to a better understanding of the viewpoints.

I2: The interviewee argues that the choice of communication protocols are very important in the architecture design process. An example given is the choice for using DynamoDB (AWS NoSQL database service) or MongoDB (third-party NoSQL database system). This sort of choice is not directly represented by the framework according to the interviewee. The interviewee thinks the framework is well suited for designing new architectures from scratch, but not so much for migrating or adapting a currently existing architecture to a more serverless architecture style.

I3: The interviewee approaches the question more from a developer/management kind of perspective. The interviewee discusses the potential knowledge gap for developers that are not familiar with serverless computing. Especially in organizations where distinct development and operations departments are in place, it becomes more complex to manage the deployment and maintenance of complex serverless applications which exist of many 'moving parts'. Additionally, the development workflow would need to be adapted to accommodate for the serverless parts of the applications since it requires a different way of developing, testing and deploying. Finally, the

interviewee states that the impact of the additional complexity due to the serverless granularity on the development and operations workflow should not be underestimated.

6. 2. 2. 3. Statement rating results

8. Learning to use the framework for designing serverless applications would be easy for me.
    - I1: Neutral
    - I2: True
    - I3: True
9. It would be easy for me to become skilful at applying the framework.
    - I1: True
    - I2: True
    - I3: True
10. It is easy for me to understand the framework based on the graphical representation.
    - I1: Not true
    - I2: Neutral
    - I3: Very true
11. Using the framework would enable me to more easily create serverless software architectures.
    - I1: True
    - I2: True
    - I3: Neutral
12. Using the framework would enable me to create better serverless software architectures.
    - I1: True
    - I2: True
    - I3: True
13. Using the framework would help me to create a more structured understanding of the problem domain.
    - I1: Neutral
    - I2: Neutral
    - I3: Very true

6. 2. 2. 4. General comments during the interviews

I1: It was not directly clear to the interviewee what the framework exactly entails. The interviewee asked if it was merely the graphical representation. After explaining the best practices and recommendations that are categorized and ordered by the framework the interviewee mentioned that this was very interesting and potentially useful. Due to the extensiveness of the theory behind the framework this was not discussed during the interview any further. Besides, with regard to the rating of the statements, the interviewee mentioned to have some difficulties since the complete framework and underlying theory was not shown.

I2: It was not clear to the interviewee with respect to statement 4 and 5 whether it should be considered from the perspective of designing new architectures or migrating existing ones. The author mentioned that it is mostly focussed on the former, but was also interested in the interviewee's opinion with regard to the latter. The interviewee mentions with respect to statement 5 that the interviewee usually employs a bottom-up approach when creating software application (architectures), this means that the up-front design phase is not as dominant as presumed by the author. Due to this bottom-up approach the framework does not directly fit the practice of the interviewee since it is considered more as an up-front tool. Additionally, the interview stresses that the framework seems most useful functioning as a checklist, where the concepts presented by the framework are more or less reminders for a well-designed architecture.

I3: The interviewee had little general comments during the interview that fell outside the scope of the questions. The only noteworthy comment is on the dominating position of AWS. The interviewee argues that with respect to FaaS a vendor lock-in is unavoidable due to the 'monopoly' of AWS in this domain. The author somewhat disagrees here, Google, Microsoft, IBM and others currently have similar, albeit less mature, products. Therefore, substitution products are available and will continue to be developed further.

6. 3. Conclusions and recommendations

This section presents the conclusions and recommendations based on the two applied evaluation methods; case study and semi-structured interviews. The limitations to both these evaluations will be discussed in Chapter 7. Additionally, in this section a revised version of the SADF's graphical representation will be presented. The revision is based on the conclusions and recommendations stemming from the case study and interview-based evaluations.

6. 3. 1. Case study

This section discusses the main findings from the case study. First a general limitation must be noted which was noticed during the case study. The framework is specifically designed to address the serverless specific parts of a software architecture. Therefore, the framework does not address general software architecture aspects. This is intentional, creating an all-encompassing software design framework is considered out of scope for this thesis.

The software design viewpoint is not very well represented by the framework. Although it is part of the framework, the way the case study is analysed and the way the software architecture is designed did not allow for going in depth with regard to the software design. The software design is often not an upfront exercise but part of the actual software development phase(s). It still deserves a place in the framework due to the importance of it to the overall serverless application, however it could not be evaluated properly by applying the framework to this case study. A recommendation would be to evaluate the framework on a real-world software project where the software design viewpoint will be covered more extensively.

The fact that the software design viewpoint could not be evaluated properly by the case study shows a larger underlying issue. The framework does not address in which phases of a software project the various viewpoints or variables should be considered or applied. This can be seen as a shortcoming of the framework. On the other hand, the framework is not designed to dictate the process of designing software applications, but should be considered more as a supporting tool that can be applied throughout the whole process. A possible future improvement to the framework could be connecting the framework to well-known software design methodologies.

In general must be concluded that the evaluation results from the case study are not substantial. Some recommendations could be made, but overall the case study functioned more as a demonstration of the application of the framework than a true evaluation. The added value of the case study must therefore be found in an example application of the framework, showing how to analyse a serverless architecture design problem based on the viewpoints and variables as defined by the framework.

6. 3. 2. Semi-structured interviews

This section discusses the main findings resulting from the interviews. The interviewees agree that the framework seems useful for creating better serverless software architectures. Additionally, two interviewees agree that it is expected that the framework helps to create serverless software architectures more easily by guiding the practitioner, one interviewee is neutral. These findings result from the semi-structured part of the interview as well as the rating of statements. The perceived usefulness (PU) is considered quite high. However, two interviewees report that they don't expect the framework to help create a more structured understanding of the problem domain (both rated this statement with neutral). In contrast, one interviewee is very confident that the framework would help to create a more structured understanding. Regarding the perceived ease of use (PEOU), two interviewees have some difficulties interpreting the framework based on the graphical representation. The two interviewees think that learning the framework is not very easy nor very difficult. The third interviewee seems to be more positive towards easiness of learning the framework. Additionally, understanding the framework based on the graphical representation is not considered very easy by the first two interviewees, the third states to have no problems understanding the framework as it is presented. On the other hand, all three interviewees expect to easily become skilful at applying the framework. A general note, all interviewees argue that they see added value in the framework in the context of using it as a checklist during the design and development phases of a serverless architecture.

Some critical conclusions can be drawn as well. The first two interviewees consider the viewpoint definitions too ambiguous, the third seems to have some issues with it, however not as prevalent as the first two. The terms that are used (e.g. software design) are not very well defined and therefore leave room for interpretation. This could potentially be solved by improving the way the viewpoints are presented. A recommendation by one of the interviewees is to add examples to each viewpoint to guide the interpretation of these terms. Another critical note, all interviewees discuss

that the relation between the variables and the viewpoints is not so straightforward. Depending on the interpretation (and application) the variables could potentially be relevant to all viewpoints. A recommendation to overcome this is to decouple the viewpoints and the variables in the graphical representation. The viewpoints and variables could still complement each other but do no longer have strict interrelations graphically presented. In general, the first two interviewees have some difficulties interpreting the framework based on the graphical representation. This affects the PEOU considerably. This might be a result from the previously mentioned viewpoint ambiguity and the non-trivial relations between variables and viewpoints. Interestingly, one interviewee thought the size of the viewpoint circles indicated its importance. This was not clear to the author and might need re-evaluation since this is not the case.

6. 3. 3. Revised SADF

Based on the conclusions and recommendations resulting from the case study and the interview evaluation some changes to the SADF can be made. The changes are almost exclusively limited to the graphical representation of the framework. This is logical since this graphical representation has been the sole subject of evaluation.

The most significant recommended changes are related to the ambiguity of the viewpoint definitions as well as the relation of the cross-viewpoint variables to these viewpoints. The interviewees had different views on which cross-viewpoint variable should relate to which viewpoint. Additionally, it has been noted that all cross-viewpoint variables are related to all viewpoints, depending on interpretation of the practitioner. Therefore, it is recommended to separate the cross-viewpoint variables from the viewpoints. This will make sure no ambiguous relations between the viewpoints and cross-viewpoint variables are prescribed by the framework.

The graphical representation is adapted to not describe the presumed relations between the cross-viewpoint variables and the viewpoints. This leaves room for interpretation by the users and is, according to the interviewees, likely to align better with practice.

Furthermore, one of the interviewees argued that in order to create more clarity with regard to the viewpoint definitions and boundaries, examples could be added to guide the user with the interpretation. The graphical representation is adapted to describe a few examples for each viewpoint, this might clarify the viewpoint definitions when looking at the graphical representation.

Another minor revision is the name of one of the cross-viewpoint variables; vendor lock-in. One of the interviewees correctly noted that vendor lock-in can occur on many levels of the software architecture. For example, by choosing a third-party library to include in your application. To remove this ambiguousness the cross-viewpoint variable should be named more specifically, for this ‘cloud platform vendor lock-in’ is suggested.

All the aforementioned recommendations and proposed changes are implemented into a revised version of the SADF's graphical representation. The revised graphical representation is presented in Figure 28.

Figure 28 Revised graphical representation of SADF

Chapter 7. Conclusions and recommendations

This chapter is concerned with presenting the general conclusions and recommendations of this thesis. The conclusions and recommendations will be presented according to the research questions as defined in Chapter 2.

7. 1. Benefits and challenges of serverless computing

This section concludes the answer to the first research question:

"What are the currently known benefits and challenges of serverless computing?"

After performing a structured literature review a number of serverless computing specific benefits and challenges have been identified. The following challenges were identified:

- Composition
- Migration and de-composition
- Monitoring complexities
- Development complexities
- Vendor lock-in
- Performance
- Security
- Reducing costs

The following benefits were identified:

- Architectural opportunities
- Development process improvements
- Granular scaling
- Optimising resource utilisation
- Reducing operation costs

These benefits and challenges have been analysed and are graphically presented in Figure 13. Additionally, they have been incorporated into the SADF in the form of Table 3 and as part of the cross-viewpoint variables.

7. 2. Serverless suitability characteristics

This section concludes the answer to the second research question:

"What are the existing approaches or characteristics to determine which parts of an application are suitable to be implemented in a serverless way?"

After performing a structured literature review it can be concluded that there is very little scientific literature on this subject. Only two academic sources were found to be useful. Based on these

sources a set of characteristics has been identified that affect the serverless suitability of a software component:

- Response time
- Invocation patterns
- Type of operation
- Data limits
- Vendor dependence
- Runtime restrictions

Based on these characteristics a flow chart was created that can be used to more easily determine the serverless suitability of a specific software component. This flow chart is presented in Figure 14. Additionally, these characteristics are implemented into the SADF in the form of Table 4. This table can be consulted by practitioners during the design of a serverless architecture to check whether the software component is suitable for a serverless implementation or if a more conventional approach based on an always-on compute instance should be considered.

7. 3. Serverless composition

This section concludes the answer to the third research question:

"How could cloud native architectures be composed and orchestrated to enable the full potential of serverless computing?"

After performing a structured literature review on the topic of serverless composition a number of different areas of interest with respect to serverless composition were identified:

- Tools and methods
- Architecture use-cases and demonstrations
- Hybrid cloud composition
- Modelling serverless composition
- Recommendations for improving serverless composition techniques
- Function chaining
- Serverless composition services
- Client focussed composition

Obviously, the most straightforward approach to serverless composition is to chain functions by invoking one from within another. This approach has a number of drawbacks, of which possibly the biggest: double-billing due to multiple functions running simultaneously. Other styles of serverless composition were found in literature. Literature discerned between two different types of composition techniques specific to serverless computing. First, client focussed composition moves the orchestration and composition of serverless functions to the client, resulting in a 'thick client'. Second, serverless composition services are third-party or cloud platform specific

workflow composition services. They allow connecting multiple serverless functions together through a GUI to form a more complex workflow.

In general can be concluded that composition of serverless applications is rather under-developed from an academic point of view due to the lack of relevant literature with regard to serverless composition styles and techniques.

7. 4. Serverless specific best practices

This section concludes the answer to the fourth research question:

"What are the best practices for creating a serverless application from scratch, and for migrating a non-serverless architecture to a serverless architecture?"

The structured literature review for this research questions yielded zero to none usable scientific sources. To accommodate for this, a lot of literature sources that were found by performing the first three structured literature reviews were analysed with respect to this particular research question. Many of these literature sources were found to be very relevant to the topic of best practices for serverless application and architecture creation. Additionally, an explorative search strategy was employed to try and find more relevant literature. Grey literature, such as blog posts and white papers by practitioners and organizations, was included to answer this research question as well. A number of relevant and distinct topics were categorized based on analysis of the literature:

- Performance
- Security
- Vendor lock-in
- Costs
- Monitoring
- Architecture

For each of these topics a number of recommendations and best practices were found that are extensively covered in Section 3. 6. Based on this theory, and in combination with the knowledge from the first three research questions, the serverless architecture design framework (SADF) was created. It includes most of the topics and contains many of the recommendations and best practices, that were identified through answering this research question.

7. 5. Serverless architecture design framework

This section concludes the answer to the fifth and final research question:

"How can these best practices be applied in a form that supports software architects and developers in the process of designing a serverless application architecture?*"*

This research question is focussed on applying the knowledge that was gathered through the first four literature review based research questions. The goal is to present this information in such a way that it becomes usable for practitioners and helps them to create better serverless software architectures more easily.

To this end the DSRM was applied in order to eventually arrive at an artefact that would serve the goal of this research question. After analysing the problem domain and the respective literature a framework was constructed which consists of four viewpoints (configuration, software design, software architecture and deployment) and five cross-viewpoint variables (performance, vendor lock-in, security, costs and serverless suitability). Based on these viewpoints and cross-viewpoint variables a best practices guide was written which is a textual representation of the most common recommendations found in literature. Additionally, a graphical representation of the framework was created and is shown in Figure 16.

Following, the framework was evaluated via two methods. First, the framework was applied on a case study which simulated how practitioners would apply the framework in a real-world software development project. It demonstrates how the framework could be applied and serves as a first evaluation step by the author. Second, semi-structured interviews were conducted with three domain experts in order to assess whether the framework aligns with practice and is actually deemed useful by practitioners.

The most notable conclusion from the case study evaluation is that the software design viewpoint could not be fully evaluated due to the limited case study. The architecture was not actually implemented and therefore the software design viewpoint could not be fully assed by the case study. The added value of the case study evaluation should be considered to be the demonstration of the application of the framework, since it shows how the framework supports the analysis of the architecture design process.

In order to evaluate the, quite textually extensive, framework the graphical representation (as presented in Figure 16) was used to explain the framework to the interviewees. Based on the semi-structured interviews a number of additional conclusions can be drawn. Summarized, the interviewees agree that the framework seems useful for creating better serverless software architectures more easily: the perceived usefulness is considered quite high. The interviewees especially mention the expected usefulness of the framework as a checklist reference during the design and development of a serverless application. According to the interviewees, the framework seems to be somewhat difficult to understand at first (based on the graphical representation), however easy master eventually.

The perceived ease of use is less positively evaluated. The interviewees argue that the graphical representation is confusing for various reasons. Most notably, the viewpoints are considered to be too ambiguous and not defined well enough. Additionally, according to the interviewees the cross-viewpoint variables are not confined to the viewpoints they were matched with. The author positioned the cross-viewpoint variables in the graphical representation as such that they

correspond to viewpoints to which they are ought to be most relevant. The interviewees argued that all the cross-viewpoint variables can be relevant to all viewpoints, it largely depends on the interpretation of the terms, the boundaries of the definitions and the application context.

Based on the conclusions from the interview evaluations the SADF is revised to improve the perceived ease of use, and to implement various other recommendations that surfaced during the interviews. The most significant change is the decoupling of the cross-viewpoint variables and the viewpoints, in the initial version a relation between these two was presented. After feedback from the interviewees it was decided that this pre-scribed relation should be removed. Additionally, the viewpoint definitions are improved in the graphical representation of the SADF. Interviewees argued that the viewpoints seemed ambiguous at first. To overcome this, short examples related to each viewpoint were added to the graphical representation in order to provide more context. The revised graphical representation of the SAF is presented in Figure 28.

Concluding, the artefact resulting from answering this research question, and also the main contribution of this thesis, is the SADF. It has the potential to be useful to practitioners by providing best practices and recommendations within a structured framework as well as serving as a checklist reference of important aspects and considerations during the architecture design process. Although, there is still room for improvement with regard to the content and completeness of the framework. A secondary contribution of this thesis is the accumulation and categorization of relevant information on designing and developing serverless applications and serverless architectures.

7. 6. Limitations

This section summarizes and presents the limitations of this research effort.

7. 6. 1. Structured literature reviews

The literature reviews were not systematic or exhaustive. Therefore, it should be noted that the presented literature analysis might be incomplete. Resulting from this, the SADF that is created based on the literature analysis might suffer from this incompleteness as well. Additionally, the analysis of the literature might be biased, or otherwise limited by the author's choices and interpretation. This is hardly avoidable. However, since the author has both academic and practical experience with software (architecture) design the impact of this limitation might be minimal.

7. 6. 2. Case study

The case study evaluation is limited in a couple of ways. The application of the framework based on the case study is limited due to the fact that the case study is relatively small and undetailed compared to real-world software projects. This was done in order to keep the architecture design and evaluation manageable within the time constraints of this research effort. A side effect of this is that the generalizability of the results of the case study is limited. Additionally, the case study

only considers the architecture design part of a software project. It ignores the actual software implementation. Since the thesis focusses on architecture design this is understandable. However, since parts of the framework relate to the actual implementation of the software (e.g. software design and configuration) these parts might be under evaluated due to this shortcoming. Finally, the case study is solely focussed on the serverless aspects of a software application architecture. This was an intentional choice to manage the scope of this research effort and to create a framework specifically for serverless architectures.

7. 6. 3. Interviews

The second part of the evaluation, the structured interviews, are conducted on a limited number of interviewees. This qualitative analysis is therefore limited to the knowledge, experiences and opinions of these interviewees. Performing the same interviews with a larger group of interviewees might yield different or additional results. For the scope of this thesis the current number of interviewees (three) is considered sufficient as it is expected to cover the most important aspects of the evaluation of the framework. This is supported by Boyce and Neale (2006) who note that when similar answers are being observed across the interviewees a sufficient sample size has been reached. This is, to some extent, the case with the results from our interviews.

A noteworthy limitation with respect to the interview-based evaluation is that the framework is evaluated by using a graphical representation of the framework and some additional explanation. This might affect the interviewees' view of the framework, since there is more to the framework (the textual recommendations and best practices) than the graphical representation only.

An additional limitation is the authors interpretation of the answers given by the interviewees. This is partly covered by the statements that the interviewees have to rate. This statement rating part of the interview acts as an additional validation of the answers given by the interviewees to the structured interview questions. This lowers part of the interpretation bias.

Another bias that should be noted is the one the interviewees might have; a bias towards answering what is socially accepted. Meaning, the interviewees might, subconsciously or not, give more positively tuned answers in order to please the interviewer since they are aware that they are reviewing something the interviewer has created.

The generalizability of the conclusions based on the interviews is another limitation. The conclusions are based on a small number of interviewees which are not randomly selected, therefore the results are in most cases not fully generalizable (Boyce & Neale, 2006). Although this is the case, the interviews are still considered to be quite useful as they provided insights into the expected usefulness and usability. As noted before, performing the interview evaluation on a larger sample size would improve the generalizability and might yield different or additional results. According to Seddon and Scheepers (2012) a key characteristic of sound generalizations is the representativeness of the interviewees. As described in Section 6. 2. 2. the interviewees are

quite representative for the software practitioners domain, they have ample experience with the subject and varying years of experience in software development in general.

7. 6. 4. Serverless architecture design framework

The SADF is designed to focus on the serverless aspects of a software architecture only. This choice was made to keep a manageable scope for this research effort. However, it should be noted that the framework relies on conventional software architecture knowledge, especially for the non-serverless parts of a software architecture. This is not considered a big issue since the practitioners that would apply the SADF are quite certainly familiar with designing conventional software architectures.

The SADF has a limited applicability, since it is designed specifically for designing cloud-native serverless software architectures. The framework can also be applied on conventional cloud-native software architectures if the goal is to start with the implementation of serverless components. In contexts other than these, the framework is most likely of little added value since it was not designed for this. The generalizability of this research effort is therefore rather limited and confined to the problem domain of designing cloud-native serverless software architectures. Due to this, not all software practitioners might benefit from the framework. Only, cloud (software) engineers, and especially the ones working with serverless technologies. It should be noted that to apply and understand the framework substantial knowledge of software engineering and software architecture is required. The framework cannot exist on its own, and depends on practitioners having conventional software development knowledge.

Finally, the SADF is created based on academic literature, although it is aimed at helping practitioners. There might be a difference between the topics academic literature addresses and what practitioners experience in practice. This is partly covered by the interviews, by validating that the framework aligns with practice. However, it must be noted that the SADF was created based on academic theory instead of practical experiences by software architects and developers from real-world software projects. This might result in an inherent mismatch between the framework and practice.

7. 7. Recommendations and future work

Some general recommendations for future work: evaluating the framework with a larger group of test subjects. This will result in more varied insights into the quality of the framework. This way the gaps in, and flaws of the framework can be more thoroughly identified and improved upon. Performing systematic, or at least more extensive, literature reviews might be considered in order to make sure that all available knowledge is gathered, analysed and incorporated into the framework. Additionally, the framework could be evaluated within the context of a real software project instead of the fictitious case analysis context used in this thesis. This will address the aforementioned limitations of the minimal evaluation coverage of viewpoints such as configuration and software design.

Based on the initial evaluation by means of the case study and interviews the SADFs graphical representation was revised. This revision has not been re-evaluated in this work. It would be interesting to see how the improvements would be perceived by practitioners. Therefore, a re-evaluation based on the revised SADFs graphical representation is a recommendation for future work.

Another more foundational recommendation for future work is related to the how the SADF came to be. It is based on extensive scientific literature research, although its goal is to aid practitioners with the design and development of serverless architectures. Therefore, there might be a mismatch, or bias, due to the academic theoretical basis. It would be interesting to develop a similar framework from a practical perspective. Not from literature, but from actual software development projects and by interviewing experienced (serverless) software practitioners. This might result in different perspectives on the SADF and might contribute to completing it. The evaluation showed that domain experts recognize the potential added valued of the SADF. Therefore, the framework can be extended and improved upon in the future to make it more complete and clear.

References

Abad, C. L., Boza, E. F., & Eyk, E. V. (2018). *Package-aware scheduling of FaaS functions*.

Adzic, G., & Chatley, R. (2017). *Serverless computing: economic and architectural impact*. Paper presented at the Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering.

Amazon Web Services. Amazon Cognito - Simple and Secure User Sign Up & Sign In | Amazon Web Services (AWS). Retrieved from https://aws.amazon.com/cognito/

Amazon Web Services. Amazon EC2 Auto Scaling. Retrieved from https://aws.amazon.com/ec2/autoscaling/

Amazon Web Services. AWS Lambda - Serverless Compute - Amazon Web Services. Retrieved from https://aws.amazon.com/lambda/

Amazon Web Services. AWS Step Functions. Retrieved from https://aws.amazon.com/step-functions/

Apache. Apache OpenWhisk is a serverless, open source cloud platform. Retrieved from https://openwhisk.apache.org/

Autho. Never Compromise on Identity. Retrieved from https://autho.com/

Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., . . . Slominski, A. (2017). Serverless computing: Current trends and open problems *Research Advances in Cloud Computing* (pp. 1-20): Springer.

Baldini, I., Cheng, P., Fink, S. J., Mitchell, N., Muthusamy, V., Rabbah, R., . . . Tardieu, O. (2017). *The serverless trilemma: Function composition for serverless computing*.

Bardsley, D., Ryan, L., & Howard, J. (2018). *Serverless performance and optimization strategies*.

Boyce, C., & Neale, P. (2006). Conducting in-depth interviews: A guide for designing and conducting in-depth interviews for evaluation input.

Chapin, J., & Roberts, M. (2017). *What is Serverless?* : O'Reilly Media, Inc.

Davis, F. D., Bagozzi, R. P., & Warshaw, P. R. (1989). User acceptance of computer technology: a comparison of two theoretical models. *Management science, 35*(8), 982-1003.

Eivy, A. (2017). Be Wary of the Economics of" Serverless" Cloud Computing. *IEEE Cloud Computing, 4*(2), 6-12.

Elgamal, T. (2018). *Costless: Optimizing cost of serverless computing through function fusion and placement*. Paper presented at the 2018 IEEE/ACM Symposium on Edge Computing (SEC).

Gannon, D., Barga, R., & Sundaresan, N. (2017). Cloud-Native Applications. *IEEE Cloud Computing, 4*(5), 16-21. doi:10.1109/MCC.2017.4250939

Garcia Lopez, P., Sanchez-Artigas, M., Paris, G., Barcelona Pons, D., Ruiz Ollobarren, A., & Arroyo Pinto, D. (2019). *Comparison of FaaS orchestration systems*.

Google. Cloud Functions - Event-driven Serverless Computing | Cloud Functions | Google Cloud. Retrieved from https://cloud.google.com/functions/

Google. Firebase. Retrieved from https://firebase.google.com/

Gregor, S. (2006). The nature of theory in information systems. *MIS quarterly*, 611-642.

Grumuldis, A. (2019). Evaluation of "Serverless" Application Programming Model: How and when to start Serverles.

Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A. C., & Arpaci-Dusseau, R. H. (2016). Serverless computation with openlambda. *Elastic, 60*, 80.

IBM. Cloud Functions - Overview | IBM. Retrieved from https://www.ibm.com/cloud/functions

Ivanov, V., & Smolander, K. (2018). *Implementation of a DevOps Pipeline for Serverless Applications.* Paper presented at the International Conference on Product-Focused Software Process Improvement.

Jackson, D., & Clynch, G. (2018). *An Investigation of the Impact of Language Runtime on the Performance and Cost of Serverless Functions.* Paper presented at the 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion).

Jambunathan, B., & Yoganathan, K. (2018). *Architecture Decision on using Microservices or Serverless Functions with Containers.* Paper presented at the 2018 International Conference on Current Trends towards Converging Technologies (ICCTCT).

Jonas, E., Pu, Q., Venkataraman, S., Stoica, I., & Recht, B. (2017). *Occupy the cloud: Distributed computing for the 99%.*

Kratzke, N. (2018). A brief history of cloud application architectures. *Applied Sciences (Switzerland), 8*(8). doi:10.3390/app8081368

Kritikos, K., & Skrzypek, P. (2018). *A Review of Serverless Frameworks.* Paper presented at the 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion).

Lewis, J., & Fowler, M. (2014). Microservices. Retrieved from https://martinfowler.com/articles/microservices.html

Lloyd, W., Vu, M., Zhang, B., David, O., & Leavesley, G. (2018). *Improving Application Migration to Serverless Computing Platforms: Latency Mitigation with Keep-Alive Workloads.* Paper presented at the 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion).

Lowery, C. (2016). Emerging technology analysis: Serverless computing and function platform as a service. *Gartner, Tech. Rep.*

Malawski, M., Gajek, A., Zima, A., Balis, B., & Figiela, K. (2017). Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions. *Future Generation Computer Systems*. doi:10.1016/j.future.2017.10.029

Manner, J. (2019). Towards Performance and Cost Simulation in Function as a Service.

Manner, J., Kolb, S., & Wirtz, G. (2018). Troubleshooting Serverless functions: a combined monitoring and debugging approach. *SICS Software-Intensive Cyber-Physical Systems*, 1-6.

McGrath, G., & Brenner, P. R. (2017). *Serverless computing: Design, implementation, and performance.* Paper presented at the Distributed Computing Systems Workshops (ICDCSW), 2017 IEEE 37th International Conference on.

McGrath, G., Short, J., Ennis, S., Judson, B., & Brenner, P. (2016). *Cloud event programming paradigms: Applications and analysis.* Paper presented at the Cloud Computing (CLOUD), 2016 IEEE 9th International Conference on.

Microsoft. Azure Functions – Serverless architecture | Microsoft Azure. Retrieved from https://azure.microsoft.com/en-gb/services/functions/

Pahl, C., Brogi, A., Soldani, J., & Jamshidi, P. (2018). Cloud Container Technologies: a State-of-the-Art Review. *IEEE Transactions on Cloud Computing*, 1-1. doi:10.1109/TCC.2017.2702586

Peffers, K., Tuunanen, T., Rothenberger, M. A., & Chatterjee, S. (2007). A design science research methodology for information systems research. *Journal of management information systems, 24*(3), 45-77.

Perera, K. J. P. G., & Perera, I. (2018). *TheArchitect: A Serverless-Microservices Based High-level Architecture Generation Tool*.

Pérez, A., Moltó, G., Caballer, M., & Calatrava, A. (2018). Serverless computing for container-based architectures. *Future Generation Computer Systems, 83*, 50-59. doi:10.1016/j.future.2018.01.022

Seddon, P. B., & Scheepers, R. (2012). Towards the improved treatment of generalization of knowledge claims in IS research: drawing general conclusions from samples. *European journal of information systems, 21*(1), 6-21.

Serverless Inc. (2019). Serverless - The Serverless Application Framework powered by AWS Lambda, API Gateway, and more. Retrieved from https://serverless.com/

Singh, D., Singh, J., & Chhabra, A. (2012). *High availability of clouds: Failover strategies for cloud computing using integrated checkpointing algorithms.* Paper presented at the 2012 International Conference on Communication Systems and Network Technologies.

Soltani, B., Ghenai, A., & Zeghib, N. (2018). *Towards Distributed Containerized Serverless Architecture in Multi Cloud Environment*.

Spillner, J. (2017). Snafu: Function-as-a-service (faas) runtime design and implementation. *arXiv preprint arXiv:1703.07562*.

Spillner, J., & Dorodko, S. (2017). Java code analysis and transformation into AWS lambda functions. *arXiv preprint arXiv:1702.05510*.

Tosatto, A., Ruiu, P., & Attanasio, A. (2015, 8-10 July 2015). *Container-Based Orchestration in Cloud: State of the Art and Challenges.* Paper presented at the 2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems.

Tran, T. H. (2017). Developing web services with serverless architecture.

Twillio. SMS, Short Message Service | Text Messaging for Mobile & Web Apps. Retrieved from https://www.twilio.com/sms

van Eyk, E., Iosup, A., Seif, S., & Thömmes, M. (2017). *The SPEC cloud group's research vision on FaaS and serverless architectures.* Paper presented at the Proceedings of the 2nd International Workshop on Serverless Computing.

Van Eyk, E., Toader, L., Talluri, S., Versluis, L., Uță, A., & Iosup, A. (2018). Serverless is More: From PaaS to Present Cloud Computing. *IEEE Internet Computing, 22*(5), 8-17.

Vaquero, L. M., Cuadrado, F., Elkhatib, Y., Bernal-Bernabe, J., Srirama, S. N., & Zhani, M. F. (2019). Research challenges in nextgen service orchestration. *Future Generation Computer Systems, 90*, 20-38. doi:10.1016/j.future.2018.07.039

Varghese, B., & Buyya, R. (2018). Next generation cloud computing: New trends and research directions. *Future Generation Computer Systems, 79*, 849-861.

Villamizar, M., Garcés, O., Ochoa, L., Castro, H., Salamanca, L., Verano, M., . . . Zambrano, A. (2017). Cost comparison of running web applications in the cloud using monolithic, microservice, and AWS Lambda architectures. *Service Oriented Computing and Applications, 11*(2), 233-247.

Völker, C. (2018). *Suitability of serverless computing approaches.*

Wagner, B., & Sood, A. (2016). *Economics of resilient cloud services.* Paper presented at the 2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C).

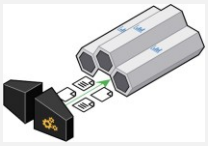
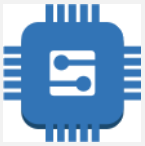
Warzon, A. (2016). AWS Lambda pricing in context: A comparison to EC2. Retrieved from https://www.trek10.com/blog/lambda-cost/

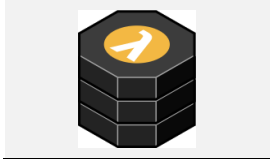
Webster, J., & Watson, R. T. (2002). Analyzing the past to prepare for the future: Writing a literature review. *MIS quarterly*, xiii-xxiii.

Wurster, M., Breitenbucher, U., Kepes, K., Leymann, F., & Yussupov, V. (2019). *Modeling and Automated Deployment of Serverless Applications Using TOSCA*.

Yan, M., Castro, P., Cheng, P., & Ishakian, V. (2016). *Building a chatbot with serverless computing.* Paper presented at the Proceedings of the 1st International Workshop on Mashups of Things and APIs.

Appendix 1. Serverless architecture design legend

	Elastic Load Balancer (ELB)
	AWS Kinesis real-time data stream
	IoT hub – small hardware device physically near IoT nodes
	Programmatic event
	IoT road sensor
	IoT traffic light actuator
	AWS Route 53 DNS
	AWS S3 Bucket – simple object storage service
	Relational database instance
	EC2 instance – elastic compute cloud – dedicated virtual compute hardware



AWS Lambda – serverless function instance

Appendix 2. Interviews

English

Step 1. Background questions

1. What is your experience in software architecture?
2. Are you into serverless computing technology?
3. What do you do as part of your job in regard to serverless computing technology and software architectures?

Step 2. Show the graphical representations of the SADF and shortly discuss how it came to be.

Step 3. Structured interview on the SADF

4. To what extent do you think the viewpoints align with the practice that you observe in the projects that you work on?
5. To what extent do you think the cross-viewpoint variables align with practice that you observe in the projects that you work on?
6. Do you think applying the framework would result in better serverless software architectures?
    a. If yes, why do you think the framework would improve serverless software architectures?
    b. If no, why do you think the framework does not contribute to better serverless software architectures?
7. Thinking from your professional experience, would you add any specific aspect to improve the framework?

Step 4. Rating statements from 'very true' to 'very false'

Options: *Very true – True – Neutral – False – Very false*

8. Learning to use the framework for designing serverless applications would be easy for me.
9. It would be easy for me to become skilful at applying the framework.
10. It is easy for me to understand the framework based on the graphical representation.
11. Using the framework would enable me to more easily create serverless software architectures.
12. Using the framework would enable me to create better serverless software architectures.
13. Using the framework would help me to create a more structured understanding of the problem domain.

Dutch

Stap 1. Achtergrond informatie

1. Wat is jouw ervaring met software architecture?
2. In hoeverre ben je betrokken bij serverless computing technologie?

3. In welke mate ben je op professionele basis bezig met serverless computing technologie en software architecture?

Stap 2. Laat de grafische representatie van de SADF zien en bespreek kort hoe deze tot stand kwam.

Stap 3. Gestructureerd interview over de SADF

4. In hoeverre denk je dat de viewpoints aansluiten op de praktijk die je ervaart bij de projecten waar je aan werkt?
5. In hoeverre denk je dat de cross-viewpoint variables aansluiten op de praktijk die je ervaart bij de projecten waar je aan werkt?
6. Denk je dat het toepassen van het framework zal resulteren in betere serverless software architectures?
    a. Zo ja, waarom denk je dat het framework zal bijdragen aan betere serverless software architectures?
    b. Zo nee, waarom denk je dat het framework niet zal bijdragen aan betere serverless software architectures?
7. Redenerend vanuit jouw professionele ervaring, zijn er bepaalde aspecten die je zou willen toevoegen aan het framework om het te verbeteren?

Stap 4. Geef op een schaal van 'helemaal waar' tot 'helemaal niet waar' aan in hoeverre je het eens bent met de volgende uitspraken

Options: *Helemaal waar – Waar – Neutraal – Niet waar – Helemaal niet waar*

8. Het zal erg makkelijk zijn voor mij om te leren hoe ik het framework kan gebruiken voor het ontwerpen van serverless applicaties.
9. Het zal erg makkelijk zijn voor mij om bekwaam te raken in het toepassen van het framework.
10. Het is makkelijk voor mij om het framework te begrijpen op basis van de grafische representatie.
11. Door het gebruik van het framework wordt ik in staat gesteld om makkelijker serverless software architectures te ontwerpen.
12. Door het gebruik van het framework wordt ik in staat gesteld om kwalitatief betere serverless software architectures te ontwerpen.
13. Het gebruik van het framework zou mij helpen bij het gestructureerd begrijpen van de probleem context.