Institute of Software Engineering
Software Quality and Architecture

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master's Thesis

Monitoring Serverless Applications: An SLO-based Approach

Raoul Ghit

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Steffen Becker
Supervisor:	Sandro Speth, M.Sc. Dr. Christian von der Ehe (Vector Informatik GmbH)
Commenced:	February 26, 2021
Completed:	August 26, 2021

Abstract

The serverless paradigm in cloud computing is definitely here to stay as it offers cloud service customers the possibility to focus even more on business and forget about operational and infrastructural concerns, which are handled by the cloud service provider. Thus, an increasing amount of cloud applications make use of Function-as-a-Service components, which are characteristic for the serverless cloud execution model. The possibilities for monitoring such cloud applications, are for the most part restricted to the cloud monitoring services provided by the cloud service providers, thus, the approaches to monitoring are limited. What is lacking is a flexible solution which allows its users an easy way to set up Service Level Objectives (SLOs) and which ensures that these are not violated unnoticed. This should be the case even if the monitored application is made up of different serverless components, such as Function-as-a-Service components, or even deployed across different environments. Although some monitoring tools are starting to support the monitoring of SLOs, these solutions are not freely available and not SLO-centered. The objective of this thesis is the extension of an existing software prototype to allow an SLO-centered monitoring of serverless cloud applications. In order for the resulting tool, which is called SoLOMON, to be as generally applicable and as useful as possible, it is implemented in such a way that it does not rely on the instrumentation of the monitored application and can easily be extended. To aid the selection of the most suited implementation approach for our goal, there is a prior investigation of what metrics apply meaningfully in serverless applications and which different possibilities exist to obtain them. Furthermore, a format for defining the SLOs which are to be modelled in SoLOMON is developed and presented. After implementing the new functionalities for SoLOMON, the tool is tested and evaluated in a real cloud application environment, namely the Connectivity Feature Service backend application developed by the company Vector Informatik GmbH. The evaluation shows that SoLOMON is successful in achieving its main goals and shows great potential to help its users manage the complexity of creating and monitoring SLOs for their serverless applications.

Kurzfassung

Das Serverless Paradigma im Bereich Cloud Computing hat weiterhin Zukunft, da es den Cloud-Nutzern die Möglichkeit gibt sich auf geschäftliche Aspekte zu fokussieren. Operationelle und infrastrukturelle Anliegen werden hier nämlich von dem Cloud-Serviceanbieter gehandhabt. Wegen dieser Vorteile verwenden immer mehr Cloud-Anwendungen sogenannte Function-as-a-Service Komponenten, welche charakteristisch für das Serverless Cloud-Modell sind. Die Auswahl der Monitoring-Lösungen für Serverless Cloud-Anwendungen ist größtenteils auf die Monitoring-Dienste der Cloud-Anbieter beschränkt und damit relativ limitiert. Was fehlt ist eine flexible Lösung die ihren Nutzern einen einfachen Weg zum Erstellen von Service Level Objectives (SLOs) anbietet und die sicherstellen kann, dass diese nicht unbemerkt verletzt werden. Dies soll sogar dann möglich sein, wenn die zu überwachende Anwendung aus verschiedenen Serverless-Komponenten, wie zum Beispiel den Function-as-a-Service-Komponenten, besteht, oder wenn die Anwendung über verschiedene Umgebungen hinweg verteilt läuft. Inzwischen unterstützen manche Monitoring-Tools das Überwachen von SLOs, jedoch sind diese Tools nicht kostenlos erhältlich und nicht SLO-zentriert. Das Ziel dieser Masterarbeit ist die Erweiterung eines existierenden Software-Prototypen, so dass dieser Serverless Cloud-Anwendungen in einer SLO-zentrierten Art überwachen kann. Damit die resultierende Anwendung, die SoLOMON heißt, so allgemein anwendbar und nützlich wie möglich ist, wird sie so entwickelt dass sie leicht erweiterbar ist und dass die zu überwachende Anwendung nicht instrumentiert werden muss. Für die Auswahl des Implementierungsansatzes der am besten für das Erreichen unserer Ziele geeignet ist, findet eine vorbereitende Untersuchung statt. In dieser geht es darum welche Metriken im Kontext von Serverless-Anwendungen sinnvoll sind und wie diese gesammelt werden können. Weiterhin entwickeln und präsentieren wir ein Format für SLOs, welches in SoLOMON verwendet werden soll. Nachdem die neue Funktionalität für SoLOMON entwickelt wurde, wird das Tool in einem realistischen Cloud-Umfeld getestet und evaluiert. Dieses Cloud-Umfeld ist durch die Connectivity Feature Service (CFS) Backend-Anwendung gegeben, die von der Firma Vector Informatik GmbH entwickelt wird. Die Evaluation demonstriert, dass SoLOMON seine Hauptziele erfüllt und großes Potenzial darin zeigt seinen Nutzern zu ermöglichen die Komplexität des Erstellens und des Überwachens von SLOs für Serverless-Anwendungen handzuhaben.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

API Application Programming Interface. 6, 7, 9, 13, 15, 18, 20, 21, 22, 23, 24, 30, 32, 35, 40, 41, 42, 47, 48, 49, 50, 51, 53, 63, 66, 70, 72, 73

APM Application Performance Management. 6, 13, 19

ARN Amazon Ressource Name. 37, 38, 45, 64

AWS Amazon Web Services. 3, 6, 12, 13, 15, 17, 18, 20, 21, 22, 23, 24, 25, 26, 28, 30, 31, 32, 34, 35, 37, 38, 41, 42, 43, 44, 45, 46, 48, 49, 50, 51, 52, 53, 55, 57, 58, 59, 61, 62, 63, 68, 69, 70, 71, 72, 73, 74

BaaS Backend-as-a-Service. 6

CA Certificate Authority. 46

CDK Cloud Development Kit. 43, 44, 52

CFS Connectivity Feature Service. iii, v, 3, 25, 38, 55, 57, 58, 59, 61, 62, 63, 64, 68, 70, 72, 73

CLI Command Line Interface. 52

CSLA Cloud Service Level Agreement. 11

DSL Domain Specific Language. 11

EC2 Elastic Compute Cloud. 35, 42, 45

ECS Elastic Container Service. 15, 35, 38, 43, 57, 62, 63, 64, 73

ECU Electronic Control Unit. 55

EKS Elastic Kubernetes Service. 61, 74

ELB Elastic Load Balancing. 35, 51, 62

FaaS Function-as-a-Service. iii, 1, 5, 6, 7, 11, 12, 15, 16, 17, 19, 67

FP Functional Programming. 51

FRP Functional Reactive Programming. 51

GUI Graphical User Interface. 18

IaaS Infrastructure-as-a-Service. 1, 7, 11, 19

IaC Infrastructure as code. 15, 43, 52, 53

ISTE Institute of Software Engineering. 9, 41

LCU Load Balancer Capacity Unit. 26

npm Node Package Manager. 36, 50, 52

OOP Object Oriented Programming. 51

PaaS Platform-as-a-service. 1, 7, 11, 19

PromQL Prometheus Query Language. 9

QoS Quality of Service. 1, 6, 7, 8, 9, 11, 71, 72

RDS Relational Database Service. 35, 38, 51, 57, 61, 64, 66, 73

SaaS Software-as-a-service. 1, 11

SDK Software Development Kit. 52, 62

SLA Service Level Agreement. 1, 8, 9, 11, 12, 13, 14, 36, 37

SLAaaS SLA-aware Service. 11

SLI Service Level Indicator. 8, 16, 17, 18, 25

SLO Service Level Objective. iii, v, 1, 2, 3, 8, 9, 10, 11, 12, 13, 15, 16, 21, 22, 23, 24, 25, 26, 28, 30, 31, 32, 35, 36, 37, 38, 40, 41, 42, 44, 45, 46, 47, 48, 49, 50, 52, 53, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75

SNS Simple Notification Service. 32, 35, 38, 42, 45, 46, 52, 58, 73

SOA Service Oriented Architecture. 5

SQS Simple Queue Service. 45, 51, 52

SSL Secure Sockets Layer. 46

URL Uniform Resource Locator. 41, 53, 74

VCS Version Control System. 46, 54

VPC Virtual private cloud. 35, 43, 45, 46

WSLA Web Service Level Agreement. 11

1 Introduction

Serverless is a relatively new paradigm in cloud computing which allows customers to focus even more on business logic as operational and infrastructural concerns are abstracted away from the user and handled by the cloud service provider. Especially for smaller applications that are built to serve many small individual requests, this can prove beneficial, as the scaling and pricing model is very fine-grained. On the other hand, the implementation of large cloud applications in serverless fashion, consisting only of Function-as-a-Service (FaaS) components and other managed services, can quickly lead to complex and complicated architectures. The large number of functions might even cause the total cost of ownership to be higher than a similar implementation using Platform-as-a-service (PaaS) components. This just shows that FaaS is not a replacement of PaaS and other deployment models. In reality one can often find bigger heterogeneous cloud applications consisting of different component types. They might use PaaS or even Infrastructure-as-a-Service (IaaS) components for some parts of the applications, and FaaS components or other managed services (e.g. storage, messaging, cache, etc.) as well.

1.1 Motivation and Problem Statement

Cloud providers usually guarantee a certain Quality of Service (QoS) for each service they provide to their users (see for example [Ama21d] and [Mic21]). These guarantees, also known as Service Level Agreements (SLAs), do, in the case of cloud service offerings, often not cover any QoS aspects beyond availability. This makes sense, as the providers are able to guarantee the availability of the infrastructure provided by them, but can make no promises about the performance, as the performance is largely dependent on the implementation of the application deployed on said infrastructure. Thus, in the context of cloud computing, SLAs, and as a consequence the Service Level Objectives (SLOs) they are made up of, are usually thought of as being located between the cloud provider and the cloud customer. In this thesis though, the perspective on SLAs and SLOs is different. Most of the big cloud providers have fixed SLAs for their cloud services and for most service consumers it is not feasible to even try to negotiate these. Although many cloud providers include price reductions or other perks in case of violation of their SLAs, our aim is not to monitor whether the cloud providers are actually keeping their promises. Rather, our aim is to ensure that a cloud application, consisting of different cloud components and cloud component types, is able to perform in the parameters of the SLOs defined by the application owner. What this means is that another viewpoint is taken: The cloud application owner may want to offer the functionality of his cloud application in a Software-as-a-service (SaaS) manner to his customers and might want to be able to ensure them a certain quality of service.

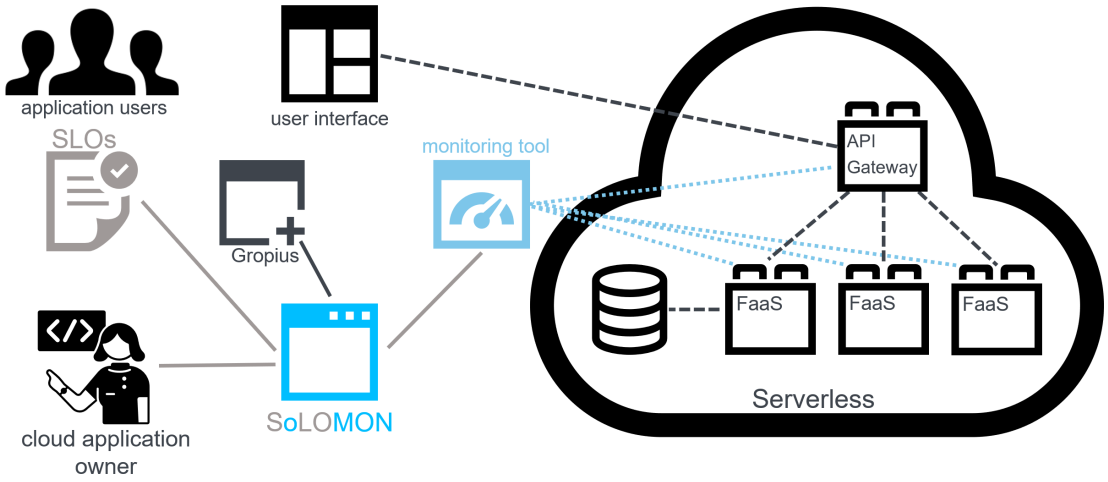

Figure 1.1: A basic overview visualizing the solution approach.

1.2 Solution Approach

In addressing the aforementioned challenge, the SoLOMON application[1] comes into play. Fig. 1.1 gives a very basic visual overview of our solution approach. The aim of SoLOMON is to increase the cloud application owner's confidence in the fact that his application (on the right in the serverless cloud) can meet the Service Level Objectives (SLOs) he defined for it with the help of SoLOMON, and to enable him to take action as soon as this is not the case anymore. The black-box view of the solution can be described very concisely: The user enters his SLOs into the SoLOMON application, gives it some information about the endpoints of the serverless cloud application that is to be monitored, and can then expect to be notified in case of violation of any of the SLOs. This means that an issue should be created in the issue management project associated to the application that is monitored when the application is not performing as expected. This is possible by SoLOMONs interface with the Gropius cross-component issue management system.

The contribution of this thesis thus includes multiple aspects:

1. An analysis of what kind of metrics are meaningful and can be observed in serverless cloud applications, and to what degree these metrics can be obtained without instrumentation and without using vendor-specific monitoring services.
2. A reasoned definition of an SLO format that is used to describe SLOs in SoLOMON.
3. A discussion of what we call the complexity problem of formulation SLOs and how the SoLOMON application could tackle the problem at different dimensions.
4. The actual implementation of the SoLOMON application which allows its user to create and monitor SLOs for serverless cloud applications.

[1] **https://github.com/ccims/solomon**.

The implementation will focus exemplary on the monitoring of serverless applications running in an Amazon Web Services (AWS) environment, but the general concept of the application is independent of AWS. SoLOMONs architecture is conceived in such a way that connectors for other cloud service providers can easily be added.

To test and evaluate the SoLOMON application, it is deployed and then used to monitor a real cloud application. This cloud application is the Connectivity Feature Service (CFS) backend application [Inf21b] developed and managed by the company Vector Informatik GmbH.

Thesis Structure

The remainder of this thesis is structured as follows:

Chapter 2 – Foundations and Related Work: Here, we provide an overview of some basic concepts from the domains of monitoring and cloud computing. Furthermore, we take a look at some related work and other approaches that exist to the problem presented in the thesis.

Chapter 3 – Approach: In this chapter, we present the goal of SoLOMON and a selection of different solution approaches that we considered. Furthermore, there is a preliminary investigation of what meaningful SLOs for a serverless application could look like.

Chapter 4 – Design: This chapter is dedicated to architectural descriptions of our solution and the requirements that we defined for it.

Chapter 5 – Implementation: Here, we describe some of the more complex challenges we faced during the implementation of SoLOMON and the different approaches we considered for solving them.

Chapter 6 – Evaluation: This chapter is all about the evaluation of SoLOMON as an experiment in a realistic cloud environment using the Goal Question Metric approach.

Chapter 7 – Conclusion We conclude our thesis by summarizing the results and, especially, the benefits of this work, while also looking at its limitations and possible future work in the domain of SLO-based monitoring.

2 Foundations and Related Work

In the first part of this chapter, we will defined some of the relevant concepts and terminology from the monitoring and cloud computing domain. The second part of the chapter takes a look at related work that inspired or was relevant to this work in some aspect.

2.1 Foundations

There are several foundational concepts that are the basis to an understanding the context of the thesis. They will be elaborated in this section.

2.1.1 Serverless and Function-as-a-Service (FaaS)

Baldini et al. call *serverless computing*, or simply *serverless*, a "new and compelling paradigm for the deployment of cloud applications" [BCC+17, p.2], in their book *Research Advances in Cloud Computing* from 2017. In 2021 it might not be that new anymore, but it seems to still be very compelling for many companies. According to Tunall, referencing the *2019 Forrester Global Business Technographics Developer Survey*, 49% of the surveyed companies were already using or planning to use serverless architecture in the next twelve months [Tun21].

In the following, we are going to look at some definitions of serverless. Some, such as Rehemägi, see **serverless architecture** as "an extension of the principles of the Service Oriented Architecture (SOA), where services (functions) communicate using messages (events)" [Reh21]. The definition proposed by Van Eyk et al. shares with the aforementioned one the element of events: "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" [VTT+18, p.9]. The last part of this definition thus means that the cloud provider is responsible for the app container, language runtime and all underlying layers of infrastructure, while the cloud user just has to load his code into the app container. This is also where the name *serverless* stems from. It obviously does not mean that no servers are needed for the application to be deployed, it just means that they are abstracted away from the users and, thus, they are thus not concerned with them. Additionally, the users can expect the application they deployed to be auto-scaling, and particularly to be scaled to zero when it is not used. In these time spans, the cost for the user is then also reduced to zero [BCC+17, p.4]. With this, we are already describing **Function-as-a-Service (FaaS)**, which is a concept closely related to serverless, and which is actually the main building block for any serverless architecture. A definition given by Sewak and Singh states: "Function as a Service (FaaS) is a small, discrete, reusable chunk of code which is stateless compute container modeled for an event-driven solution" [SS18, p.1].

It has become clear, that there is a huge overlap between the two concepts. In a webinar on the topic, Ribenzaft and Peretz put the two in relation to one another by claiming that serverless consists of FaaS components and managed services (APIs) and allows the focus of the cloud user to be on the business logic, as the infrastructure does not have to be managed by him or her [RP21, 2:24]. The managed services mentioned here, are of course nothing else but a specific type of backing services [Wig21], namely what is sometimes also called Backend-as-a-Service (BaaS), and which includes things like database, message queuing, cache and notification management services managed by the cloud provider. The last missing piece in the construction of serverless applications is what Microsoft calls *Durable Functions* and goes under *Step Functions* in the Amazon Web Services (AWS) ecosystem. These are function orchestrators that, as Fox et al. note, let the user "manage long running flows by combining multiple (small) FaaS invocations" [FIMS17, p.9]. Their advantage is that, unlike FaaS components, they are not stateless, and can thus maintain the application state. This could for example mean the conditional invocation of FaaS components, or that the output of one FaaS invocation can be used as input for a following one. Of course they can manage not only FaaS, but also BaaS invocations, and allow, by combining those, for the development of quite complex serverless architectures.

The limitations that come with building more complex serverless applications are pointed out for example by Sewak and Singh: "[T]he integration becomes more complicated and it's difficult to debug and troubleshoot. The higher level of granularity also affects tool and frameworks to be used with Serverless" [SS18, p.2]. Ribenzaft and Peretz note that it is harder to find bugs and get certain metrics precisely because the user has no access to the underlying infrastructure. The asynchronous, distributed and event-driven nature of serverless applications makes troubleshooting even more difficult [RP21, 6:23].

2.1.2 Monitoring

Application Performance Management (APM) is an important aspect of running business-critical applications. The general idea behind it, is gathering and then analyzing application performance data. This is useful for detecting potential performance problems, and guiding the administrators of the respective software system to their possible solutions. In traditional APM, there is a periodic sampling of monitoring data. A more modern approach is called **observability**. Here the sampling is not periodic, but constant and the monitoring data that is gathered is analyzed using machine-learning techniques [Edu21]. Although the observability approach is very well suited for cloud native technologies, microservices, Kubernetes and serverless functions [Edu21], our approach is based on the traditional APM, and more specifically on runtime monitoring. Cassar et al. give following definition of **runtime monitoring**: "Runtime Monitoring thus employs techniques for observing the internal operations of a software system and/or its interactions with other external entities with the aim of determining whether the system satisfies or violates a correctness specification." [CFAI17, p.15]. This means that by keeping a watch on important Quality of Service (QoS) metrics of the application, it tries to constantly answer the question: "Is my system running properly?" If the answer is no, business might be seriously affected by this, and its cause has to be identified as quickly as possible. But before even being able to talk about the identification of root causes, it is important that the aforementioned question is answered, and this is possible only through measurements. Thus, monitoring aims to continually measure certain defined QoS aspects, such as availability, performance and reliability, of the components of which the application is comprised. If anomalies

are detected in the measurements, an administrator can be alerted and the root cause for the anomaly can be looked for. An even more comprehensive understanding of monitoring would also include aspects such as trending analysis and performance prediction.

For the scope of this thesis, it will be of importance to distinguish between what we will call intrusive and non-intrusive monitoring. For **intrusive monitoring**, instrumentation is the way to go. The instrumentation of an application means that additional code, with the purpose of making measurements, is added to the application code. This could be code that records how long the execution of a function takes and then logs this measurement. The collection and storing of logs and measurements from different functions and components is called *telemetry* and is usually done by the monitoring system. Instrumentation has four disadvantages: It can impact the performance of the application, as additional code has to be executed in the production system. The second disadvantage is that the additional code snippets needed for instrumentation decrease the readability of the code and, thus, make maintenance more difficult. Thirdly, it can be quite tedious to manually instrument code and that, especially in larger systems, one can easily lose the overview over what components have been instrumented and how they were instrumented. The last aspect is that instrumenting code manually is error prone and errors in the instrumentation can cause either the creation of incorrect monitoring data, or even affect the production system that is to be monitored. That is why there are some monitoring solutions which promise automatic instrumentation. This offers the advantages of instrumentation, such as the generation of more detailed monitoring data, without the last two disadvantages just mentioned. In the related work part (see section 2.2) we will mention some of the existing monitoring tools that offer automatic instrumentation.

Non-intrusive monitoring, on the other hand, does not add additional code to the components that are to be monitored. It may use data provided through an Application Programming Interface (API) by the component (on which the code runs) itself. This presupposes that the component provider offers such an API and that the data offered through it can be used meaningfully for monitoring. A different way for non-intrusive monitoring is synthetic monitoring. In this approach availability, correctness of service and performance can be measured simply by calling the components that are to be monitored [Gra21]. The performance, for example, can be quantified by measuring the response time, the availability by checking whether there is a response, and the correctness of service by comparing the content of the response with the expected result. Although the monitoring data produced without instrumentation can usually not be as detailed as the one using instrumentation, non-intrusive monitoring offers other advantages such as a greater flexibility, no performance impact and no cluttering of the code.

2.1.3 Quality of Service and Metrics

Quality of Service (QoS) is generally understood to comprise non-functional aspects of a service such as performance, reliability and security [Se21]. For each of such a QoS aspect, multiple QoS metrics can be considered. **Metrics**, or measurements, are meant to quantify the service level with regards to the QoS aspects [SBK+13]. Not all standard QoS metrics can be applied meaningfully to FaaS components. It makes, for example, no sense to expect a FaaS component to be running constantly, as one might expect from a Platform-as-a-service (PaaS) or Infrastructure-as-a-Service (IaaS) component with a high availability. The very idea of FaaS components is that they are initialized on-demand. This means they do not have to be available, in the sense of running all the time. But they do have to be available in the sense of actually being callable all the time and

then also working correctly. **Aggregation** is something also often mentioned in the context of monitoring and metrics. Wagner notes that "aggregation is an important part of any measurement system and an reoccurring task in any measurement" [Wag13, p.47]. Without aggregation there might be way too many data points, resulting from monitoring measurements, for any given system. Thus, aggregation operations, such as grouping, rescaling, central tendency and dispersion, are necessary to make measured data intelligible and to allow for its visualization.

2.1.4 Service Level Agreements and Service Level Objectives

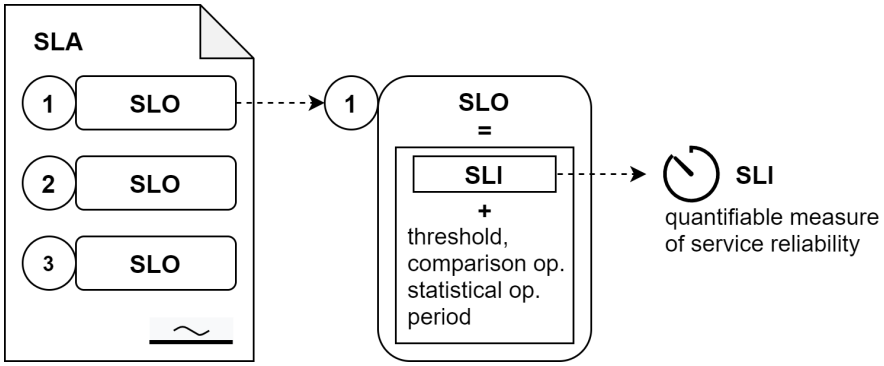

Figure 2.1: The relationships between SLA, SLO and SLI.

In the definition of runtime monitoring referenced above, there already appeared the term *correctness specification*. In a sense this is what a **Service Level Agreement (SLA)** is. It is an, often times legally binding, contract between two parties: the service provider and the service consumer. They usually consist of multiple **Service Level Objectives (SLOs)**. Each SLO is an aim for a service with regard to QoS aspects as measured through a defined metric. A **Service Level Indicator (SLI)** describes exactly such a defined metric: a quantifiable measure of service reliability. The SLO is thus nothing else but an SLI with a period of measurement, a statistical operator used to aggregate the measurements, a threshold and a comparison operator, describing at what point the objective is not fulfilled anymore. Fig. **??** provides a visual depiction of how SLA, SLO and SLI relate to each other. All these play an important role not only in cloud computing, but also in other web services, as Keller and Ludwig have shown already in 2003 [KL03]. Labidi et al. claim that "SLA is the principal means of control which includes Quality of Service (QoS) requirements and terms of engagement for the participating entities" [LMG+17, p.338]. Even though often there are also other SLOs, which are not QoS related (e.g. confidentiality SLO), the focus in our work is on QoS related SLOs, and more specifically on performance SLOs.

The SLAs that cloud providers offer for their services usually cover only the availability aspect, as they are responsible only for the cloud infrastructure used by the cloud consumers. Other QoS aspects such as reliability and performance are heavily dependent on the user specific component implementation, and the cloud providers can thus make no guarantees for them. One of the key problems regarding SLAs in the cloud context, as identified by Ludwig et al., is the **heterogeneity of interfaces**: "Services from different vendors have different instrumentation and use different service management systems making it difficult to collect and aggregate performance data for service level objective evaluation" [LSM+15, p.140]. This is a problem especially in cloud applications that make use of services from multiple different vendors: "The more external service vendors are involved

the less control an application owner has over the quality of the delivery of this service and must rely on the quality commitments of his or her vendors in the form of Service Level Agreements (SLAs)" [LSM+15, p.140]. But, as we have seen, the SLAs offered by most cloud providers are not sufficient to guarantee a certain QoS for a cloud application. This is why monitoring is needed.

2.2 Related Work

In the first part of this section, the Gropius application and the prototype for SoLOMON will be presented. While the second part is a quick look at different SLA languages that have been proposed to describe SLAs and SLOs, the third part will examine existing monitoring solutions.

2.2.1 Gropius and SoLOMON Prototype

The software we have called SoLOMON, is available as open source project on GitHub[1]. As the GitHub structure indicates, SoLOMON belongs to the Gropius project[2]. **Gropius** is a system to manage cross-component issues for component-based architectures. Cross-component issues are issues that can concern more than one component in a service-oriented or component-based architecture [SBB21, p.308]. In order to support these types of issues, Gropius "acts as a wrapper over traditional issue management systems and can consequently create and manage issues in the issue management systems of all affected components" [SBB20, p.83]. Its frontend allows to describe the static architecture of the system that is to be managed in a graphical building-block view, while the backend offers an API which allows the creation of new projects, components and their interfaces, as well as issues and their interlinking. SoLOMON, as well as Gropius are developed at the Institute of Software Engineering (ISTE) in Stuttgart. Fig. 2.2 shows, in a simplified way, how SoLOMON and Gropius are connected to allow the creation of issues in the traditional issue management system (e.g. GitHub) that Gropius is used as a wrapper around.

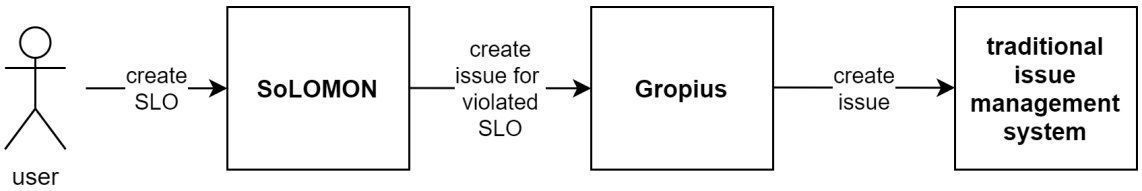

Figure 2.2: Issue creation based on SLO violations.

At the beginning of this thesis **SoLOMON** was in a prototypical state[3]. SoLOMON, at that point, allowed the creation of SLO objects through a user interface. An SLO created this way could then be applied to services in a Kubernetes cluster, which meant that a Prometheus operator was checking that they are not violated. Internally, the frontend component of the tool creates JSON objects for the SLO from the user input and sent them to the backend component of the tool. There, these object were used to create matching Prometheus Query Language (PromQL) expressions. PromQL

[1] **https://github.com/ccims/solomon**.
[2] **https://github.com/ccims**.
[3] **https://github.com/ccims/solomon/tree/e06df3e2d3445251ff8f29566f289cfae8d35f01**.

expressions allow users to select and aggregate time series data in real time [Pro21b]. Furthermore, the objects were used to create corresponding `PrometheusRule` objects. The Prometheus operator was then able to monitor these rules. If a rule is broken, an alert was sent to SoLOMON by an alert manager component which is part of Prometheus. Upon receiving an alert, SoLOMON checked to which SLO object it belonged. Because the user defines beforehand that the SLO belongs to a certain Gropius project, an issue referencing the SLO that was broken could be created automatically for the respective project. In Fig. 2.3 the initial SoLOMON prototype and its interaction with Gropius, Prometheus and the Prometheus Operator inside the Kubernetes cluster can be seen.

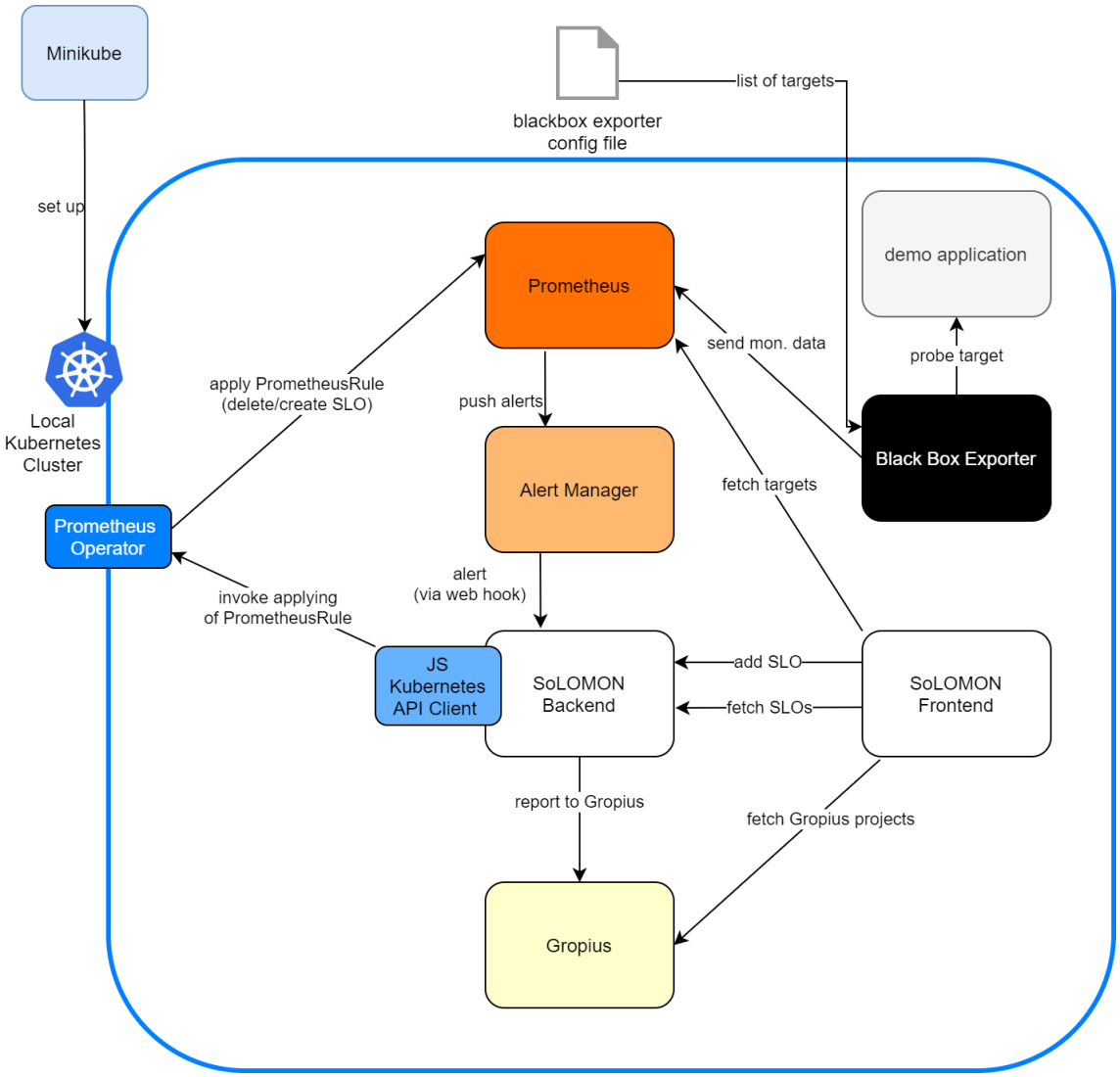

Figure 2.3: Overview of the initial SoLOMON prototype and its interactions with other systems.

One major limitation of this prototype was that it only supported monitoring services in Kubernetes clusters. Even within these confines, it allowed the monitoring of only two different metrics, namely availability and response time. Still, the basic idea of the prototype had potential and through a restructuring of the backend and the addition of new functionalities its applicability could greatly be improved.

2.2.2 Existing SLA languages

There have been multiple attempts at defining formal, machine-interpretable languages for the description of SLAs. Initially we thought about using an existing SLA language in SoLOMON, but we then realized that it would not make sense to base our implementation on such a language. Because of the approach that we selected for implementing SoLOMON (see section 3.4), our descriptions of SLOs would have to be based primarily on the expected description format for alarms in the cloud provider monitoring services. Looking at other SLA language descriptions and the frameworks that make use of them was still helpful in inspiring our own approach.

The **Web Service Level Agreement (WSLA) framework**, proposed by Keller and Ludwig in their famous 2003 paper [KL03], is a very influential approach for the definition of SLAs for web services. It was the first step toward automation of the SLA definition, negotiation, deployment and monitoring, by creating a flexible formal language to describe SLOs [KL03, p.58]. The framework also includes a set of services which are able to interpret the WSLA language, which is quite similar to something that we will try to achieve in the context of this thesis. The paper also presents a monitoring framework, although monitoring seems to be understood by them more as a function to examine whether the service provider keeps his part of the agreement. The same thing can be said about more recent approaches, like the one by Labidi et al., who want to "assist cloud consumers to automatically detect service violations" [LMG+17, p.338]. As already clarified in the introduction (Chapter 1), this is not the aim of this project. The idea of SoLOMON is not to detect SLA violations the cloud service providers might commit, but rather creating confidence in the QoS of the components implemented or configured by the users, and running on the cloud services of the cloud provider. The WSLA language is quite dated now and has among other things the drawback that it includes "no prescribed way to address the heterogeneous instrumentation found in today's Cloud scenarios." [LSM+15, p.142].

Serrano et al. have presented an approach called SLA-aware Service (SLAaaS) [SBK+13] which allows a systematic and transparent integration of SLAs into the cloud and is orthogonal to different cloud deployment models. The ones mentioned in the paper are IaaS, PaaS and SaaS. Whether the model might be applicable to applications using FaaS and, thus, become interesting for this project, remains uncertain without an in-depth study, as the papers related to this approach make no mention of the FaaS paradigm. Integrated into the SLAaaS model is the use of the **Cloud Service Level Agreement (CSLA)** language, which was proposed by Kouki and Ledoux [KL+12] in 2012. Its authors note that it is influenced by the already mentioned WSLA language, and by the SLA@SOI project [KL+12, p.587]. The CSLA language "allows to describe an SLA between a cloud service provider and its customer by defining QoS guarantees in the form of SLO clauses" [SBK+13, p.52]. Its additional benefit, compared for example to the WSLA language, is its readiness to be used in cloud contexts. What is special about it is that it integrates features that deal with QoS uncertainty and fluctuations typical to cloud environments (e.g. cloud elasticity) [KL+12, p.586]. Concretely, they call these features fuzziness, confidence and penalty.

Similar to CSLA, a team of IBM researchers under the lead of Ludwig who developed the **rSLA** approach, wanted their SLA language to be suitable for heterogeneous and dynamic cloud environments [LSM+15, p.142]. Additionally, they also developed a framework which allows flexible and scalable SLA monitoring and which we will describe in the next subsection. The rSLA language itself was designed as a Domain Specific Language (DSL) on the basis of the Ruby programming language. It has multiple elements: *Base metrics* describe how and how often the monitoring system

should obtain metrics. Thus it contains a *measurement directive*, pointing to a special plugin of the monitoring system which gets the metrics from the proprietary cloud interface. The base metric also contains a *schedule* describing the frequency at which measurements are taken. *Composite metrics* aggregate values from base metrics and other composite metrics and, thus, allow the building of more complex metrics that take into account multiple measurements. Another element of the language are *Service Level Objectives* (SLOs). In this case they refer to the commitments of the service provider to the customer and not the objectives an SLA author might want to describe for his application using the rSLA language as a whole. The last element of the language are *notifications*. As one might expect they define how external services are to be informed about the metrics and about violations of the SLOs. For a detailed specification of the language Ludwig et al. reference to a technical report from the IBM research lab in Almaden that is not publicly available. The fact that the language specification is not accessible make its adoption for this project impossible, even though from the description the language sounds suited for our use case.

2.2.3 Existing Monitoring Solutions

There exist a few products on the market that promise to solve the monitoring serverless problem. One example for this is **Epsagon**, which promises an easy way to monitor cloud services, container orchestrators and serverless functions [Eps21, "What is Epsagon"]. According to the documentation, it is able to integrate with AWS accounts and Kubernetes clusters. Through *tracing libraries* that exist for Node.js, Python and Java, applications that are deployed on containers of other cloud providers could technically also be monitored. This works through an automated instrumentation approach in which the user just has to load the library in his code and set some environment variables. This means the instrumentation effort is quite reduced [Eps21, "Monitoring Applications"]. For the integration of Amazon Web Services (AWS) services such as AWS Lambda, Epsagon uses the monitoring data generated by CloudWatch, which is a monitoring service offered by AWS [Ama21a]. This means that for such components, no code changes are required at all.

Dynatrace is another product with a similar approach and which also seems to support multi-cloud environments. Unlike Epsagon it allows easy integration not only for AWS and Kubernetes, but also for Azure, Google Cloud Platform, Cloud Foundry and OpenShift [Dyn21]. It also uses instrumentation, although the approach seems to be a bit different: Using what they call *OneAgent deployment*, they allow the users to generate a template for AWS Lambda functions after they selected some monitoring configuration parameters. After adding the actual business logic code into the template, the function can be deployed and monitored [Dyn21, "Deploy OneAgent as Lambda extension"]. For Azure Functions on the other hand, monitoring is as simple as installing a so called *Azure-Site-Extension* [Dyn21, "Integrate OneAgent on Azure Functions"].

As we have seen, both monitoring solutions allow the monitoring of FaaS components, although Dynatrace seems to have a better integration for a larger number of cloud providers. The disadvantage is that both either require instrumentation (even if it is not completely manual), or that they rely on the metric data provided by vendor specific monitoring services such as AWS CloudWatch, in order to monitor FaaS components. In contrast, the monitoring solution aimed for in this thesis should not be intrusive, meaning no components or code should have to be altered for the monitoring to work. In a preliminary investigation we wanted to find out if there are monitoring tools that

allow the monitoring of serverless components running on AWS and other cloud providers without instrumentation. As we found out, current APM solutions allow this only when they instead rely on the monitoring services of the cloud providers.

Initially we also aimed to not rely on the cloud monitoring services of the cloud providers, but the next chapter (Chapter 3) will explain why we dropped this self-imposed limitation.

A different approach that requires no instrumentation is *synthetic monitoring*. One software that pursues the synthetic monitoring approach for serverless architectures is **2 Steps**. Here is how the 2 Steps team describes synthetic monitoring: "Synthetic monitoring allows you to emulate the way an actual user would perform an action. This gives you a complete view of the user experience from start to finish regardless of what underlying technologies your software is built on" [Tea21]. Instead of tracking monitoring data from real users of the application, it simulates users and measures important metrics meanwhile. This also allows for very detailed error reports and a reduction of confounding variables in the production of a failure state in the application. While this approach is certainly interesting in that it is non-intrusive and vendor-agnostic, it is quite limited. It allows monitoring only on the application level and cannot offer any insights at the component level. A more in-depth discussion of synthetic monitoring follows in the next chapter (see section 3.3).

All of the monitoring solutions presented so far also have the disadvantage that they do not allow a direct integration of SLA or SLO management. This is a problem a team of researchers at the IBM lab in Almaden tried to tackle with the **rSLA framework**. Aside from the rSLA language we already presented in short, it consists of a set of lightweight adapters, called *Xlets*, that are responsible for the reading of metrics from different proprietary cloud monitoring interfaces, as well as a modular SLA evaluation service. The evaluation service interprets SLA documents written in the rSLA language and analyzes the monitoring data received from the Xlets in order to evaluate the SLA compliance [LSM+15, p.143]. This approach still relies on the retrieval of the relevant monitoring data through an API offered by the cloud service provider, although it is abstracted away through the Xlets. As the next chapter shows, it has become obvious that there is no good way to get relevant monitoring data except through such APIs or through instrumentation of the application code. This is why such a plugin-based approach as used by the rSLA framework seems to be useful and also inspired the solution implemented in SoLOMON. The effectiveness of the rSLA approach has been shown by the IBM research team in at least two case studies [LSM+15] [MAS+16]. One major drawback of rSLA is that neither the language specification nor the framework we just described are publicly accessible. Furthermore, it seems that the development of the framework was discontinued, as there are no related papers released after 2017 [IBM21]. In contrast to the rSLA framework, the SoLOMON application and its SLO specification format developed as part of this project will be published as open source software.

2.2.4 Research Methodology

In this section we want to shortly outline the methodology that was used for the research part of this work. The starting point for research of relevant literature was the *Google Scholar* search engine[4]. Here we used several keywords, such as the following, and combinations of them to discover the literature of interest: *SLA*, *SLO*, *SLI*, *language*, *service level*, *agreement*, *objective*,

[4] **https://scholar.google.com/**.

indicator, *monitoring*, *metrics*, *cloud*, *serverless*, *FaaS*. Some of the scientific articles we found in this way referenced to other insightful articles. In order to be as efficient as possible, we first read only the abstract of seemingly relevant papers. If the abstract indicated the relevance of the paper, we skimmed other sections of it, such as the end of the introduction, the results chapter and so on. The results from these searches were helpful to build the foundational knowledge needed for this work (see section 2.1), as well as the knowledge about related work, and more specifically about existing SLA languages (see subsection 2.2.2).

For the research of existing monitoring solutions, we used the *Google* search engine[5]. Here we used a combination of different keywords such as *monitoring*, *cloud* and the name of specific cloud providers (e.g. *AWS*, *Microsoft Azure*, *Google Cloud*). What these searches lead to were, among other things, online articles comparing different monitoring solutions, and the websites of those monitoring solutions. Often, it was necessary to skim through the documentation of the monitoring tools, in order to be able to get information about the support of monitoring serverless components.

[5] **https://www.google.com/**.

3 Approach

The general approach for this project was to first consider how a user of the SoLOMON application might generally use it and what goals he or she would want to achieve by using it. Ascertaining in the first step that the goal would be the creation and measurement of SLOs, next we would do some research regarding the question which SLOs can be applied meaningfully to serverless applications. Thus, this chapter will illuminate some of the findings of the research regarding the general approach to the topic, as well as the background behind some of the basic decisions made with regards to the project.

3.1 Usage and Goals of SoLOMON

The general idea of usage behind SoLOMON was guided by the existing prototype, which was already described a bit in section 2.2. The prototype was a proof of concept limited to monitoring applications running in a Kubernetes cluster. In contrast, our extension of SoLOMON has its main focus on monitoring serverless applications, which usually consist of multiple fully managed cloud services (such as FaaS, API Gateways and database services). Other than that, the main goal remains the same. It can be formulated as following: "The goal of SoLOMON is to allow the user to set up SLOs for serverless applications, and to make sure that eventual SLO violations do not go unnoticed."

The usage of SoLOMON is thus envisioned like this: The users have a serverless application which is deployed in a cloud environment. As already explained more in-depth in subsection 2.1.1, this means that different fully managed services, offered by the cloud providers, are combined and connected in order to create the serverless application. In practice this usually involves FaaS components, API Gateway services, database services and others. What all the major cloud providers also offer are monitoring services which have access to the services which make up the serverless applications. If it was not activated by default, the users would have to activate this monitoring service. In the case of AWS CloudWatch, monitoring of the used cloud services is activated by default. Then the users have to start SoLOMON. It can either also be deployed on AWS using for example the Elastic Container Service (ECS), or it can be run from a local machine. When running from a local machine, the users have to make sure they have the correct AWS security credentials on it, in order to be able to connect to AWS. It has to be mentioned here that the local deployment is not advised, as it leads to a problem described in more detail in section 5.2. When deploying on AWS, there are some important aspects that have to be taken care of, such as security groups, roles, and others. More details about this can be found in section 5.3, and also in the Infrastructure as code (IaC) part of the SoLOMON repository. When SoLOMON is correctly deployed, the users can use its web interface to create new SLOs for the different components of their serverless application. In order to do this, they select a component from a list of components of the serverless application that SoLOMON is aware of. Then they set the parameters for the SLO: name, description, metric

type, statistic, comparison operator, period and threshold. Additionally, they can link the SLO to a component from a Gropius project which models the serverless application. In this way a mapping between the actually deployed component and its modelled abstraction in Gropius is created. What this allows, is the creation of Gropius issues linked to a specific modelled component of a Gropius project in case the SLO is violated. Gropius also ensures that the created issues are reflected in the issue management systems that Gropius acts as a wrapper over [SBB20]. That is actually all the users should have to do. The rest is taken care of by the SoLOMON.

A more detailed and technical description of some use cases can be found in the next chapter, in section 4.2.

3.2 Meaningful SLOs

One of the first questions we had to ask, before even considering the approach for the implementation of the solution, is the following: What indicators can be used meaningfully to derive whether the serverless application is performing as desired? Of course what we are talking about here are Service Level Indicators (SLIs) and Service Level Objectives (SLOs). As part of our evaluation, we created a questionnaire to find out what meaningful SLOs are for the people working in the industry context in which the evaluation took place (see subsection 6.3.1). But even before this, we wanted to conduct a preliminary study, based on literature and other sources, that would give some general answers which could help us approach the topic.

One general approach for selecting and formulating SLOs, proposed for example by Quach, is to focus on scenarios in which customers are likely to experience significant pain [Qua21]. This means, as Sharma points out, that the SLIs should be directly connected to something the user experiences and not some internal metric, like for example CPU utilization [Sha21]. The SRE workbook, for example, recommends treating the SLI as a ratio between the number of successful events and the number of total events [TFHB18].

After taking a closer look at the components that typically make up a serverless application and also taking into account the information regarding SLI formulation, we came up with the following list of potentially relevant SLIs:

- **Availability**: The proportion of requests that resulted in a successful response
- **Throttles**: The proportion of invocation requests that are throttled (Request is rejected because all function instances are processing requests and no concurrency is available to scale up)
- **Latency**: The proportion of requests that were faster than some threshold.
- **Execution Duration**: The proportion of code executions that were faster than some threshold (For synchronous executions of FaaS this SLI coincides with the latency)
- **Freshness**: The proportion of the data that was updated more recently than some time threshold
- **Correctness**: The proportion of records (or inputs) injected into the database (or component) by a correctness prober that result in the correct data being read from the database (or returned from the component)

- **Custom Error Rate**: The proportion of requests that lead to a specific and defined type of error (custom error logging must be implemented in the code of the function)

With these seven SLIs we cover three of the ISO 25010 Quality Characteristics [Se21]:

- **Reliability**: Availability, Throttles
- **Performance Efficiency**: Latency, Execution Duration
- **Functional Suitability**: Freshness, Correctness, Custom Error Rate

As functional suitability falls more into the domain of software testing rather than monitoring in the usual sense of the term, the focus was on the first four SLIs. We also acknowledge that the abovementioned SLIs are very general, as they do not sufficiently take into account the fact that different component or service types allow for different metric types. Thus this selection of SLIs has to be understood as a first approach to serverless metrics and not as a definitive list of metrics that has to be supported. For the evaluation of SoLOMON we created a questionnaire which we used for an elicitation of metrics that ought to be supported in SoLOMON (see subsection 6.2.2).

Before taking a look at the possible sources of the SLI measurements, we want to shortly mention other possible SLIs and why we did not include them in the list above. Durability is an SLI used in relation to databases and refers to the securing of the integrity of the data. Monitoring this SLI is rather irrelevant, as the cloud service providers usually guarantee a very high durability of their DB-services (e.g. AWS S3 with 99.999999999% durability [Ama21f]). The number of invocations of FaaS components is also an information all cloud monitoring services provide and while this can be interesting to the person monitoring the application or its architect, this value is not something directly affecting the user experience (unlike for example throttles). Another SLI available when monitoring FaaS components, but also other managed services, are the number of errors. We did not list this SLI in the list above, as this metric is already covered by the availability SLI for standard errors, and by the custom error rate for custom errors. Cold starts are also often mentioned when talking about FaaS components. What the term refers to is the bigger amount of time the execution of a function call takes for such a component if the components was not called for some time and was thus in an inactive state. When calling the function when it is cold, which is another term for saying it is inactive, it takes additional time until the cloud service provider provisions the container your function is running in. In our opinion cold starts are also already covered in the latency SLI listed above.

3.3 Different Approaches to Monitoring

We want to distinguish between three different general approaches to monitoring: Black-box, grey-box and white-box monitoring. They will be sketched out in the following.

3.3.1 Black-box Monitoring

In the black-box monitoring approach the entire application that should be monitored is seen as a black-box. This means we have no information about the internals of the application, or more specifically, the components it is made out of and their connections. All that is known are the entry

points into the application, so basically what the end user also knows. This usually includes the user interface and maybe the API endpoints if the API of the application is public. We can than use what is called **probing** to generate some monitoring data. What this means is that we make requests to the application, either through the API or through the user interface, and then await the response. Based on if we get a response at all, we can thus determine the availability of the application, and based on the time elapsed until we receive it, we can determine the latency.

When taking this approach one step further, we arrive at what is often called **synthetic monitoring**. The idea behind synthetic monitoring is to model and then simulate the behavior of real users, especially their interaction with the user interface. The results the application returns in response to the actions simulated in the user interface can then be compared to the expected results. By modelling different use cases and creating the user interface workflows for them, we are able to also measure some aspects of the functional suitability of the application with black-box monitoring. It has to be noted here though that this approach is much closer to testing, and especially to automated GUI testing, than to actual monitoring.

3.3.2 Grey-box Monitoring

In grey-box monitoring the application that is to be monitored as a whole is not treated as a black-box anymore. It presupposes that we have an idea about the components out of which the application is made, and maybe even how they are connected. The components themselves on the other hand are treated as black-boxes now. Thus grey-box monitoring requires no changes to or knowledge about the internals of any given component. The only thing that has to be known is where the component can be accessed (usually using an identifier or an address). Similar to black-box monitoring, the monitoring data can then be generated for each component by calling it and awaiting the response, which again is called **probing**. In this way metrics such as availability and latency can be measured on the component-level. When looking at database components there is also the possibility to generate monitoring data using database queries. The idea behind this is that certain queries can be made on databases in order to evaluate SLIs like freshness and correctness. For determining the freshness of data within a database, one could make a query on the date of latest modification for all entries. To evaluate the correctness, a correctness prober could be implemented. This prober would inject synthetic data into the database and then, after some time, try to retrieve the data again and compare it to the expected result.

In order to get more detailed monitoring data we have to go beyond the probing approach. One possibility to do this and which still does not require knowledge about the internals of the components is to use the **standard metrics offered by the cloud providers**. What we are talking about here are services such as AWS CloudWatch, Azure Monitor or Google Cloud Monitoring. The cloud providers, being the ones who provision the different components and the cloud environment, are able to gather all kind of metrics and logs from them without having to add additional monitoring code into them. To access these metrics usually the cloud provider monitoring services have to be used.

3.3.3 White-box Monitoring

In what could be called white-box monitoring, we have full visibility into the application and into its components. This means the code inside the components is also accessible and can even be altered in order to implement monitoring. This procedure of altering a component in order to allow it to be monitored is often called **instrumentation**. We can further differentiate between two approaches, with different levels of instrumentation effort.

The one with the lower instrumentation effort is what we call **custom logs**. Beside the logs that a component, for example, a FaaS component, generates automatically when being called and when finishing execution, we may want to log additional information. To do that, we just have to add a single line of code per additional log output we want to have. This can be used to log specific error types which might be related to business logic or even just information logs which can help to create a better understanding of the probabilities of different control flows in the function. On a higher level, log analysis can be used to aggregate the different logs from different components and to analyze them. This way certain dependencies and influences of components and their runtime behavior on others can be deduced. Here we are already talking about what is often referred to as *observability*, which allows the identification of bottlenecks and can thus give insight into how to improve the application architecture.

The second white-box approach, which has a slightly higher instrumentation effort, is the implementation of **custom metrics**. This is similar to custom logs, in that code has to be added manually to the components. With custom metrics the monitoring data generated can be more detailed and comprehensive as a single data point can take complex forms and there is even the possibility of time series data. This increased complexity is also the cause for the increase in instrumentation effort.

3.4 Selecting a Monitoring Approach

After having laid out three major approaches to monitoring and some more specific ways how they could be applied to serverless applications, we want to present the rationale behind the decision for the actual approach used by SoLOMON. The decision tree that can be seen in Fig. 3.1 provides an overview.

The first question asked was whether the application we want to monitor uses virtualization. As this is the case for the type of cloud applications that we are interested in, we follow the left path. For other types of applications there are plenty existing Application Performance Management (APM) solutions [Nov21]. The next decision is based on the deployment type used for the application. “Classic” cloud applications, meaning for example applications build using the PaaS or IaaS deployment models, were not the focus of this work. The prototypical SoLOMON implementation that existed before was specialized for use with applications that are deployed using Kubernetes. Kubernetes is a system for the deployment and the management of containerized applications. The applications targeted with SoLOMON, are those using the serverless deployment model. Following the tree down this path, we now arrive at the point from which on we can arrive at five different major approaches to monitoring serverless applications. The distinction here is made using the view on the system, differentiating between black-box, grey-box and white-box monitoring. These general views one can take on a system and their differences were already elaborated in section 3.3.

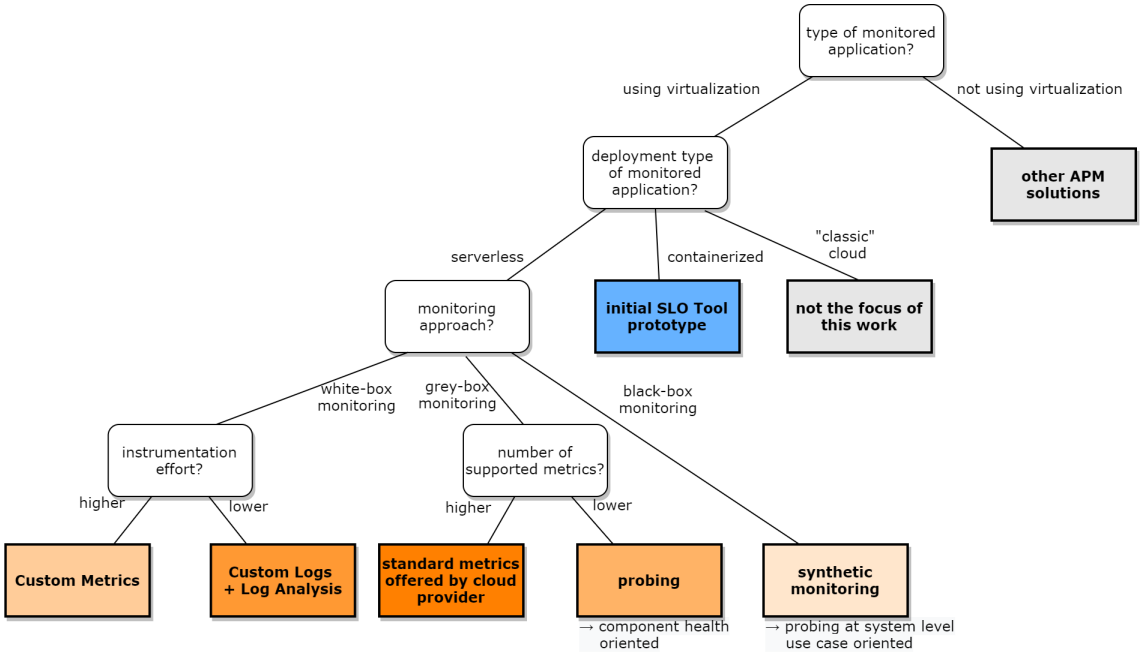

Figure 3.1: Decision tree for monitoring approach.

Black-box monitoring, where the entire application that is to be monitored is seen as a black-box, leads to the approach of probing at the application level, which is quite limited in its basic form. The more elaborate version of this is sometimes called synthetic monitoring. As already mentioned, we considered this approach to be rather an edge case of monitoring and being situated more in the domain of automated testing, which was not the topic of this thesis. White-box monitoring, on the other hand, is interesting, as it allows for a high degree of customization, at the cost of having to modify the code at least to add custom log statements or other code to send custom metrics. As the goal at the beginning of the thesis was stated to avoid a solution which depends on instrumentation, meaning the modification of the application which should be monitored. Thus, what remains is the grey-box monitoring path on which we made one further distinction, namely whether the number of supported metrics should be higher or lower. The probing approach on component level, is what was used in the initial SoLOMON prototype with the so called *blackbox exporter*. This is a tool which allows the probing of Kubernetes containers without having to know about their internals. One severe limitation is that through this approach only the availability and the latency of the response of a component can be measured. For a higher level of detail and a higher number of supported metrics, probing is not sufficient. This is where the final approach comes in: The standard metrics offered by the monitoring tools of the cloud providers. AWS CloudWatch, Azure Monitor and Google Cloud Monitoring are the three monitoring tools offered by the three biggest cloud computing providers for their cloud environments. The big advantage of these tools is that they already have insight into many of the metrics of the different cloud components and managed services, and can offer them without having to modify them in any way or without having to install additional monitoring agents. Specifically for serverless cloud applications that means that we get an insight into the application at an infrastructural level, even though this level is abstracted away from the user in the serverless paradigm. Additionally, these tools offer APIs that allow the retrieval of monitoring data

and logs, and even the setting of alarms. The fact that the logic for log analysis and alarm setting exists already means that we do not have to reinvent the wheel and implement it ourselves; we just have to make use of it and correctly connect it SoLOMON.

3.5 Consequent Architecture Considerations

As already argued in the section above, using cloud provider monitoring is the most promising approach for the implementation of SoLOMON. It offers the best insight into the cloud environments, even at the infrastructural level that is abstracted away from the user in the serverless paradigm. Using this approach, we are able to retrieve metrics as well as log data. Additionally, we can also use the functionality for setting up alarms on metric data, which is offered by the cloud monitoring providers. This knowledge opens up two possible architectural alternatives that are going to be discussed here.

3.5.1 Using a Metric Collector

The first alternative includes what we will call a metric collector. This is a component that has the task to retrieve the monitoring data collected by the cloud provider monitoring system. What this means concretely is the following: the cloud provider monitoring system (e.g. AWS CloudWatch) collects the monitoring data from all the components and services running in the cloud environment. The metric collector would then have the task to retrieve the metrics from CloudWatch using its API. After having extracted the monitoring data from the cloud environment, it can be analyzed and visualized. By applying certain queues on the data, the compliance to the SLOs we set up in SoLOMON can be checked. Fig. 3.2 depicts a simplified view of such an architecture.

The advantages of such an approach are that while the cloud provider monitoring system might delete the monitoring data after a certain period of time, after having extracted it from the cloud environment we would be able to store the data as long as we want and analyze it in any possible way. This might be useful for analyzing monitoring data over longer spans of time, but for the main focus of SoLOMON, it might not be that important. Another advantage of this architectural alternative is that it allows for a good extensibility. For connecting other applications, even from different cloud or non-cloud environments, you would just have to find a way to get the monitoring data into the metric collector. Once available in the metric collector, all the data can be analyzed and visualized there and the compliance to the SLO can be checked. The best candidate for what we have been calling metric collector so far would be Prometheus. Prometheus is an open source monitoring system and time series database. It is the monitoring system used in the SoLOMON prototype for Kubernetes applications. The biggest advantage Prometheus offers, and what makes it suited as metric collector in this architecture, is the fact that there are a multitude of so called *exporters* that can be connected to it. Exporters are libraries and servers which can be used to export existing metrics from other third-party systems, without having to instrument them directly [Pro21a]. The Blackbox exporter, which is used in the SoLOMON prototype for Kubernetes and which was already mentioned a few times, is one example of an exporter. There are also exporters for getting data from AWS CloudWatch, Azure Monitor, Google Stackdriver, Alibaba Cloudmonitor, but also

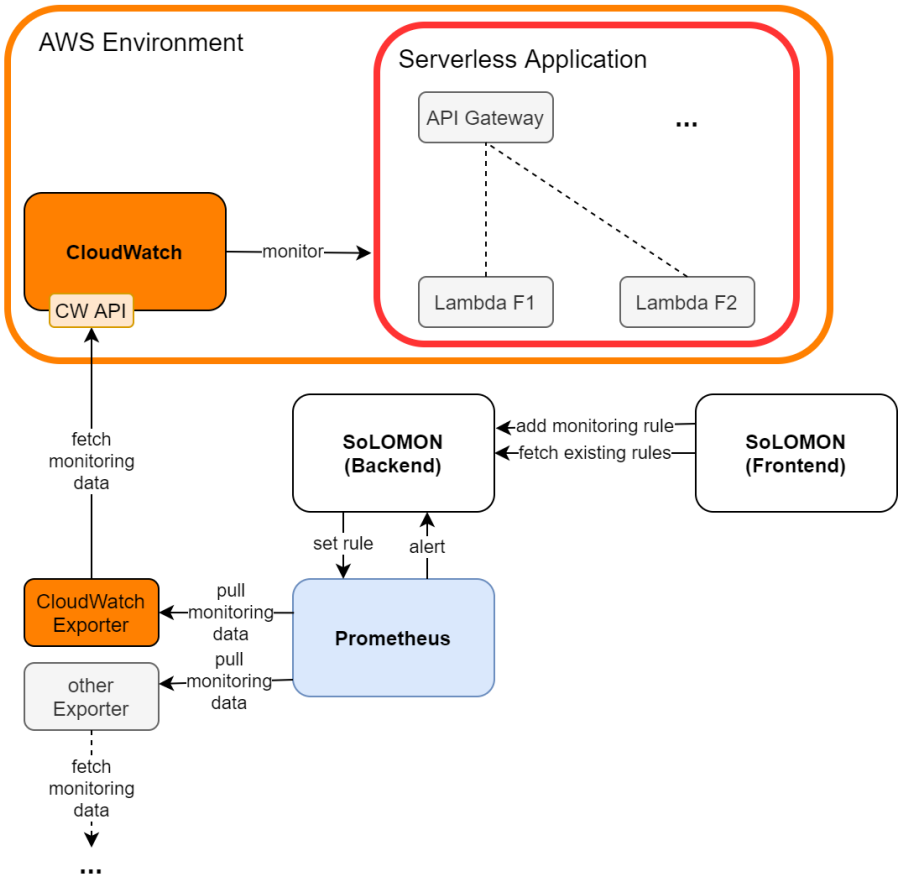

Figure 3.2: Architectural alternative using a metric collector.

from all kinds of databases, messaging systems and other APIs. These exporters have the advantage that they all export the monitoring data from the different systems in the Prometheus specific format, which means it can directly be used without further translation in Prometheus.

Now, to the disadvantages of this architectural alternative. First of all the setup and configuration of this architecture is not trivial. A constantly running instance of Prometheus needs to exist, as well as instances of the exporters. Prometheus has to be configured to receive the data from the exporters, and the exporters themselves have to be configured to be able to access the data from the cloud provider monitoring system. All in all, the general architecture of our entire system is more cluttered in this approach: There is one exporter for CloudWatch or any other cloud provider monitoring system, then there is Prometheus, which is itself also a monitoring system, and then there is SoLOMON. Another important drawback is the fact that fetching data using the APIs of the cloud monitoring providers is, at least in the case of AWS CloudWatch not free for any amount of monitoring data. This is closely connected to another problem: the timing. To receive notifications about the violation of SLOs timely, the monitoring data has to be polled as frequent as possible. But more frequent calls to the API also lead to higher costs for the API usage.

3.5.2 Connecting Directly to the Cloud Provider Monitoring Systems

It becomes clear that the approach using a metric collector, described in the previous subsection, is limited in a few important aspects. This raises the question why the monitoring data would have to be exported from the cloud provider monitoring system at all. Especially given the fact that these monitoring systems already support the creation of alarms and alerting in case of the alarms enter a critical state. In this *direct connection approach* we have less complexity and an easier setup, as we do not need an additional monitoring system which retrieves the data from the cloud provider monitoring system. SoLOMON connects directly to the APIs offered by the cloud provider monitoring systems and uses it only to set alarms for the SLOs, instead of exporting the monitoring data. This means we have much lower, if any, costs for using the cloud provider monitoring system API. A simplified view of this architecture can be seen in Fig. 3.3.

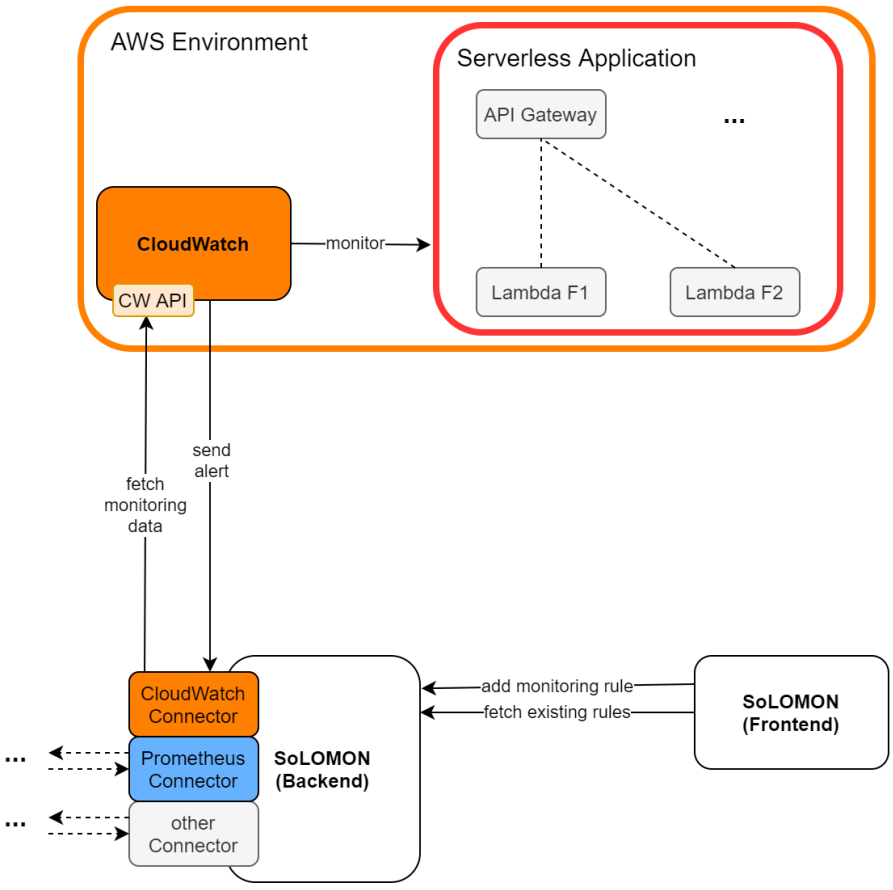

Figure 3.3: Architectural alternative with direct connection.

The fact that the monitoring data is not exported can be seen as an advantage: it means we do not have the monitoring data duplicated and thus in two places, for one of which we would have to provide the storage. On the other hand, there might be some cases where one would want to retain the monitoring data for longer periods than the cloud provider monitoring system retains them normally. Depending on the resolution of the metrics, in AWS CloudWatch for example, the monitoring data is stored between 3 hours and 15 months. The 3 hours apply for data points with a period of less than 60 seconds, and the 15 months retention applies for data points with a period of

1 hour. There are also two other categories between these two which have retention periods of 15 and 63 days [Ama21b]. This is sensible and should suffice for the majority of use cases. If, for any reason, the monitoring data for certain metrics should be retained longer, it can still be exported from CloudWatch, but we consider that our solution should not rely on the constant exporting of monitoring data for the reasons mentioned before.

There is one other aspect that we consider to be a slight drawback in this architectural alternative compared to the one using a metric collector. This aspect is the extensibility of the solution. The extensibility refers in this case to the question of how easy it is to connect SoLOMON to new cloud provider monitoring systems. In the scope of this thesis the connection to AWS CloudWatch is implemented, and the prototype existing before allowed the connection to Prometheus monitoring a Kubernetes cluster. In the approach described in the previous section, using a metric collector, and more exactly Prometheus as that metric collector, connecting a new cloud provider monitoring system, such as Azure Monitor or Google Stackdriver, is as easy as setting up the specific Prometheus exporters for these systems and configuring them. In the approach described in this section, which removes the metric collector between SoLOMON and cloud provider monitoring system, the connection between the two, using the API of the latter, has to be implemented in SoLOMON.

In spite of this disadvantage, which we are able to attenuate through abstraction and a modular design of the connectors in SoLOMON, we consider that the advantages of this architecture prevail. Moreover, we consider that this approach is closer to the spirit of serverless computing. Although it was not a requirement for SoLOMON, it is still very likely that its users can benefit from the fact that they do not have to set up and manage as many components and services as would have been the case in the approach with the metric collector and the exporters. Additionally, we can use the existing functionality for alarms on monitoring data provided by the cloud provider monitoring systems, and do not have to reimplement that on our own. This means we can simply map our SLOs to alarms in the respective monitoring systems. The implementation of this mapping and the calling of the specific API of different cloud provider monitoring systems is what we abstract away into the so-called connectors in SoLOMON. More details about this and the general architecture of SoLOMON can be found in section 4.3.

4 Design

The goal of this chapter is to present the requirements for SoLOMON and how they were elicited. Furthermore, the architecture of SoLOMON is specified using Kruchten's 4 + 1 View Model [Kru95]. We then present the interface we designed to be used in SoLOMON to describe SLOs and the reasoning behind it. The chapter is then ended by showing what design and architectural patterns were used in the design of the application.

4.1 Requirements

The number of strict requirements given for this project was rather limited. It can thus be understood more as an exploratory concept project without very strict boundaries. There were some guidelines and ideas mentioned by the supervisors, but even most of those were not defined as necessary requirements. Thus, there was quite a lot of freedom given to us regarding the design and the implementation of the system.

In spite of this, we created the following lists of requirements to guide the development. Some of them were implicit, for example through the functionality of the prototype on which SoLOMON is based on, others were given by the supervisors of the thesis or the domain experts contacted for the evaluation, and some were proposed by us.

4.1.1 Requirements Elicitation

The list of the basic functionality requirements is the result of the discussions with the supervisors with some of our own ideas added. The same is true for the non-functional requirements.

To evaluate SoLOMON in a realistic environment, we still needed to formulate a few realistic SLOs for the serverless application running the Vector AWS environment. As should be clear by now, metrics, which are nothing else but SLIs, are the basic component of SLOs. This is why we created a questionnaire in which we asked the people at Vector Informatik who have been working on the CFS serverless application what kind of metrics they would like to monitor for which kind of serverless components. As a result of the questionnaire, we thus added the functional requirements related to the metrics that should be supported when monitoring AWS serverless applications with SoLOMON. The questionnaire responses from the product owner of the CFS backend application were understood as being authoritative and of higher importance than those of the other two participants. This is why the requirements in this regard were formulated according to his results of the questionnaire. More details about the questionnaire and a more in depth discussion of its results can be found in Chapter 6.

4.1.2 Functional Requirements

For the formulations of the requirements that make up this section we oriented ourselves to Chris Rupp's FunctionalMASTER template [Rup16, p. 230].

Basic Functionality

R-F01 SoLOMON shall provide the user with the ability to create SLOs for the components of serverless applications running on AWS.

R-F02 SoLOMON shall provide the user with the ability to edit created SLOs for the components of serverless applications running on AWS.

R-F03 SoLOMON shall provide the user with the ability to delete SLOs for the components of serverless applications running on AWS.

R-F04 SoLOMON shall provide the user with the ability to view all created SLOs for the components of serverless applications running on AWS.

R-F05 SoLOMON shall notify the user in case of violation of existing SLOs by creating an issue in the Gropius issue-management system.

Supported Metrics

R-F06 SoLOMON shall support the invocations metric for AWS Lambda.

R-F07 SoLOMON shall support the errors metric for AWS Lambda.

R-F08 SoLOMON shall support the throttles metric for AWS Lambda.

R-F09 SoLOMON shall support the duration metric for AWS Lambda.

R-F10 SoLOMON shall support the concurrent executions metric for AWS Lambda.

R-F11 SoLOMON shall support the 4XX error metric for AWS API Gateway.

R-F12 SoLOMON shall support the 5XX error metric for AWS API Gateway.

R-F13 SoLOMON shall support the count metric for AWS API Gateway.

R-F14 SoLOMON shall support the active flow count metric for AWS Network Load Balancer.

R-F15 SoLOMON shall support the consumed LCUs metric for AWS Network Load Balancer.

R-F16 SoLOMON shall support the healthy host count metric for AWS Network Load Balancer.

R-F17 SoLOMON shall support the unhealthy host count metric for AWS Network Load Balancer.

R-F18 SoLOMON shall support the CPU utilization metric for AWS Relational Database Service.

R-F19 SoLOMON shall support the database connections metric for AWS Relational Database Service.

R-F20 SoLOMON shall support the free storage space metric for AWS Relational Database Service.

R-F21 SoLOMON shall support the CPU utilization metric for AWS ECS.

R-F22 SoLOMON shall support the memory utilization metric for AWS ECS.

4.1.3 Nonfunctional Requirements

Usability

R-NF01 SoLOMON shall be able to provide this basic user functionality to the user without the user having to modify the components of the serverless application.

Extensibility

R-NF02 The SoLOMON architecture shall provide future developers with the ability to easily add connectors for other cloud service providers.

R-NF03 The SoLOMON documentation should guide future developers with regards to the development of new connectors for other cloud service providers.

Maintainability

R-NF04 The SoLOMON code shall be documented well.

R-NF05 The SoLOMON code shall be developed after software engineering best practices.

Portability

R-NF06 SoLOMON should be deployable in different environments by offering the possibility to run in a Docker container.

4.2 Use Cases

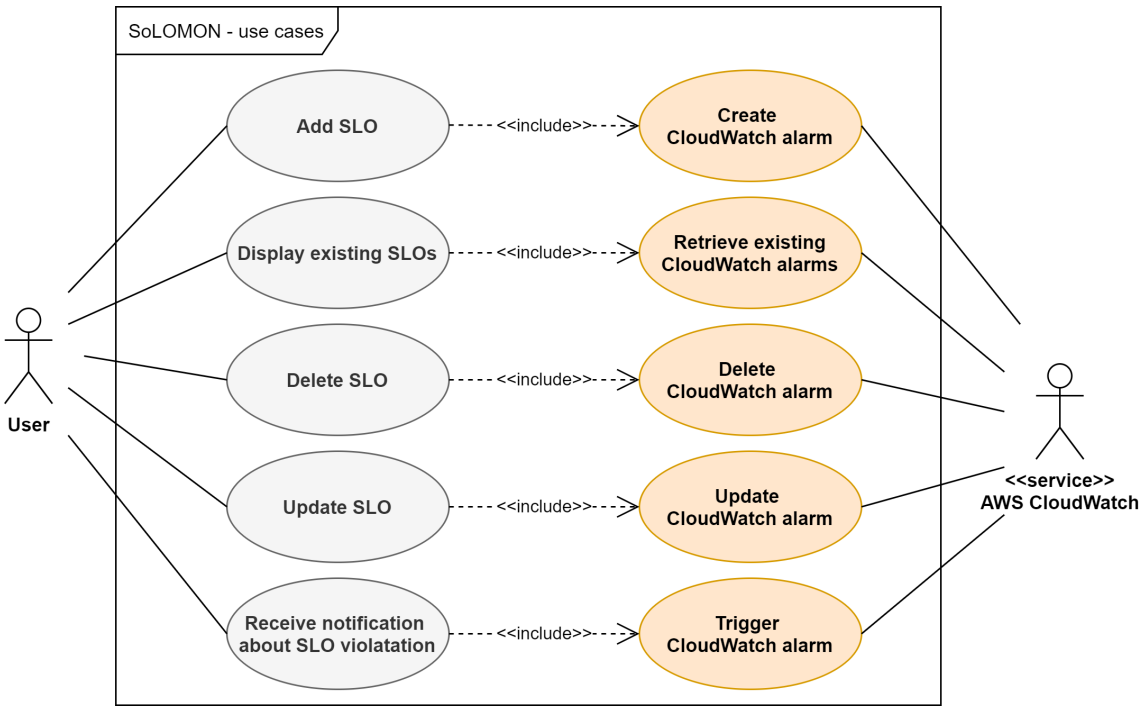

Figure 4.1: Use Case Diagram for SoLOMON.

A very general and textual description of the user of SoLOMON was already presented in section 3.1. An overview of the specific use cases for SoLOMON can be seen in section 4.1. The use cases connected to the user actor correspond to the basic user functionality as described in section 4.1. Each of these use cases, in turn, has an include relationship to another use case that is connected to the second modelled actor, namely the AWS CloudWatch service. For example, in order for the user to be able to add a new SLO and for it to be actively monitored, a new CloudWatch alarm needs to be created.

For the detailed description of the use cases we used the *Basic Use Case Template* proposed by Cockburn in 1996 [Coc98]. The two most important use cases are shown in detail in Table 4.1, and in Table 4.2.

Table 4.1: Use Case 1: Add SLO.

Goal in Context	The user wants to create a new SLO for his application.
Scope & Level	SoLOMON, primary task
Preconditions	The application is running. The SoLOMON is running and connected to the application.
Success End Conditions	A new SLO was created and a new Gropius issue gets created as soon as the SLO is broken by the application.
Failed End Conditions	The new SLO was not created, but the SLOs that existed before are not affected by this.
Actor(s)	user, AWS CloudWatch
Trigger	- button click in the UI "add SLO" - POST call of `/slos` API endpoint with an SLO object in the message body
Description	1. user enters `name, description, deploymentEnvironment, alarmAction, gropiusProject, gropiusComponent, target, metricOption, statistic, comparisonOperator, threshold, period` in the UI 2. user clicks on "save" button 3. an SLO object is created internally 4. an alarm gets created in CloudWatch using a configuration created out of the SLO entered by the user 5. when the threshold for the alarm is crossed, it is activated 6. CloudWatch notifies SoLOMON about the triggered alarm 7. SoLOMON triggers the creation of an issue related to the SLO in Gropius
Extensions	-
Sub-variations	1. user creates an SLO JSON object that contains all necessary attributes and sends it directly to the `/slos` API endpoint via POST request 4. a query on the monitoring data collected by Prometheus gets created using a configuration created out of the rule entered by the user 6. the Prometheus alarm manager notifies SoLOMON about the triggered alarm
Open Issues	-

Table 4.2: Use Case 2: Display SLOs.

Goal in Context	The user wants to see all the SLOs that he created for his application.
Scope & Level	SoLOMON, primary task
Preconditions	The application is running. SoLOMON is running and connected to the application.
Success End Conditions	The user is shown all the SLOs that were created.
Failed End Conditions	The SLOs cannot be displayed, but they still exist and are not affected by the failure of displaying them.
Actor(s)	user, AWS CloudWatch
Trigger	- loading the main page of the UI - GET call of `/slos` API endpoint
Description	1. user loads main page of the UI 2. the backend fetches the existing alarms from CloudWatch and maps them to corresponding SLOs 3. the SLOs are listed in the UI
Extensions	-
Sub-variations	1. user sends GET request directly to the `/slos` API endpoint 2. the backend fetches the existing alarms from Prometheus and maps them to corresponding SLOs 3. the API returns an array containing all the SLO objects
Open Issues	How are SLOs from different deployment environments (AWS, Kubernetes) displayed in the UI? Are they displayed in separate tabs?

4.3 System Architecture

This section contains an architectural description of SoLOMON. The presented views are based on Kruchten's *4 + 1 View Model* from 1995 [Kru95]. This model proposes describing the software architecture of a software using five concurrent views, each of which "addresses a specific set of concerns of interest to different stakeholders in the system" [Kru95, p. 42]. Fig. 4.2 gives an overview to this approach.

The *Scenarios* view, depicted in the middle of the graphic usually covers the use cases of the software. An overview of the use cases and a selection of use cases with a detailed description were part of section 4.2. In the following, the remaining three views of the system will be covered.

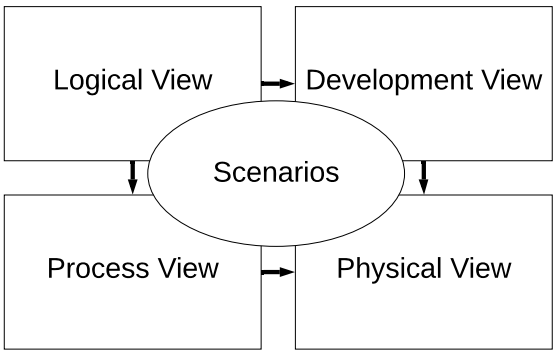

Figure 4.2: Simplified version of the 4 + 1 View Model by Kruchten [Kru95, p. 43].

4.3.1 Logical View

Fig. 4.3, shows a simplified logical view of the SoLOMON backend. The boxes in this class diagram represent the different classes of the application and the dotted lines their connection. The domain model classes, which contain mostly interfaces, were left out. As will be elaborated a bit more in section 5.7.2, in NestJS[1] there are two special component types: controllers and providers.

Controllers are classes that describe APIs of the application. In the case of SoLOMON these are the two topmost classes in the diagram: the *SoLOMON API* and the *Alert API*. The **SoLOMON API** is the main API of the backend, the one that the frontend of the application communicates with and which offers the client all the functionality of the application. The **Alert API** is defined in a separate class, as it is not called from the clients of SoLOMON, but instead it is called by the monitoring tools connected to it. In the example of AWS, CloudWatch initiates the sending of an HTTP/S request with the alert to a specific endpoint in the *Alert API* of SoLOMON.

Providers, or often called services, can be injected in other classes and usually provide some specific functionality. All other classes in the diagram, except the three Mappers, are providers. To underline the dependency injection relationship we used the *«use»* decorator for those dotted lines. The **Gropius Manager**, for example, is a provider that is injected in the SoLOMON API and in the Alert Handler. In the former it provides the functionality to get a list of currently existing Gropius projects and components, in the latter it provides the functionality to create a new Gropius issue based on an alert. The SoLOMON API also has the **Forwarder** injected which is responsible for handling most of the API requests. What it does is, it forwards the requests, for example to create a new SLO, to either the **Kubernetes Connector** or to the **CloudWatch Connector**, depending on a specific URL parameter. By using this forwarder we can keep the SoLOMON API controller clean of the forwarding logic. What was left out in this diagram is another provider that is injected in the CloudWatch Connector, the Kubernetes Connector, and the Gropius Manager, namely the **Config Service**. The Config Service allows to load environment variables which can then be used in the other classes. It was left out of the diagram in order to improve its readability and because it is not that important to the architecture.

[1] **https://nestjs.com/**.

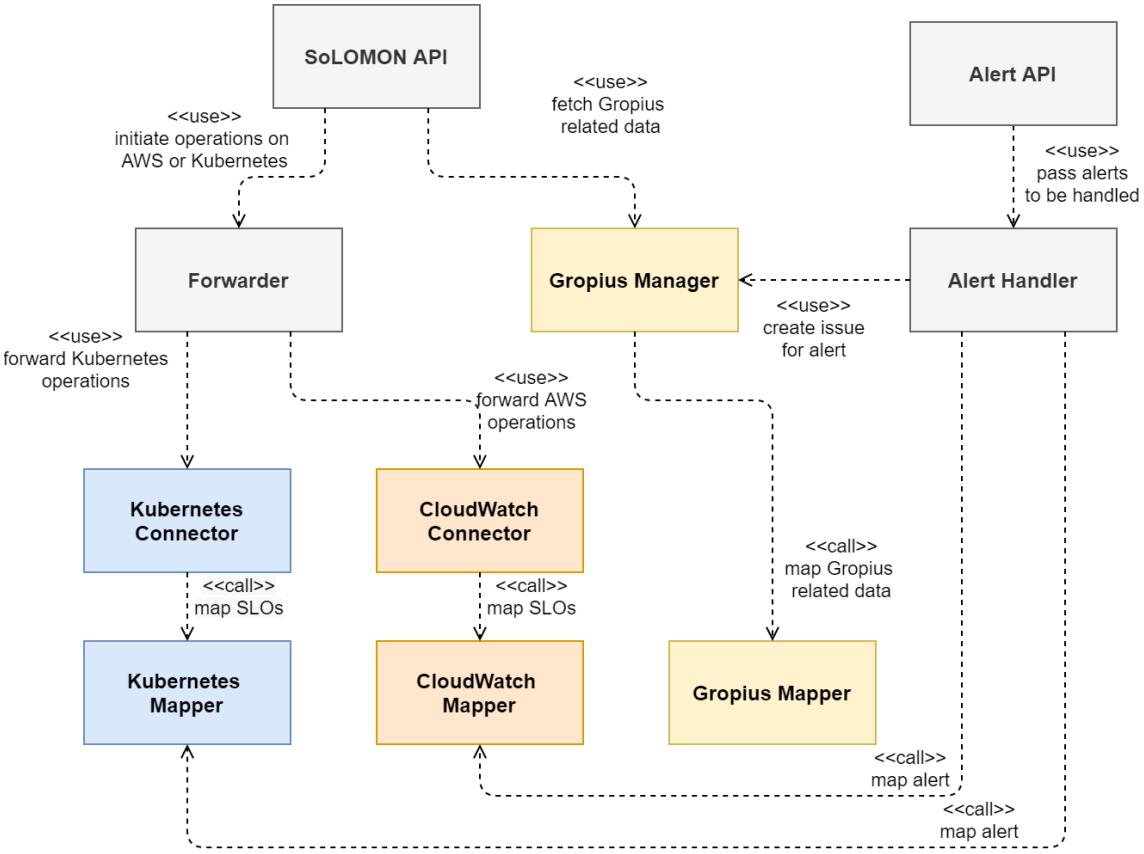

Figure 4.3: Simplified logical view of SoLOMON.

The remaining three classes are neither controllers, nor providers, but plain old classes with only static methods. They are three **Mappers**, one for Kubernetes, one for CloudWatch, and one for Gropius. The fact that they offer only static methods means that these methods do not have to be called on an instance of these classes, but can be called directly on the class. All three classes offer methods to map concepts from the respective environment (AWS or Kubernetes and Gropius) to SoLOMON concepts and back. The most important example is the mapping from SoLOMON SLO to CloudWatch alarm (AWS) or Prometheus monitoring rule (Kubernetes). The dotted lines leading to these classes have a different syntax (*«call»*) than those where providers are injected into the other classes, in order to be able to distinguish the types of relations.

4.3.2 Process View

The aim of the process view is to depict dynamic aspects of the system. This includes the communication between the user and the system, and between the different components of the system. The sequence diagram shown in Fig. 4.4, illustrates the messages exchanged by the components in their logical sequence for the use case of creating an SLO for a serverless application running in an AWS environment.

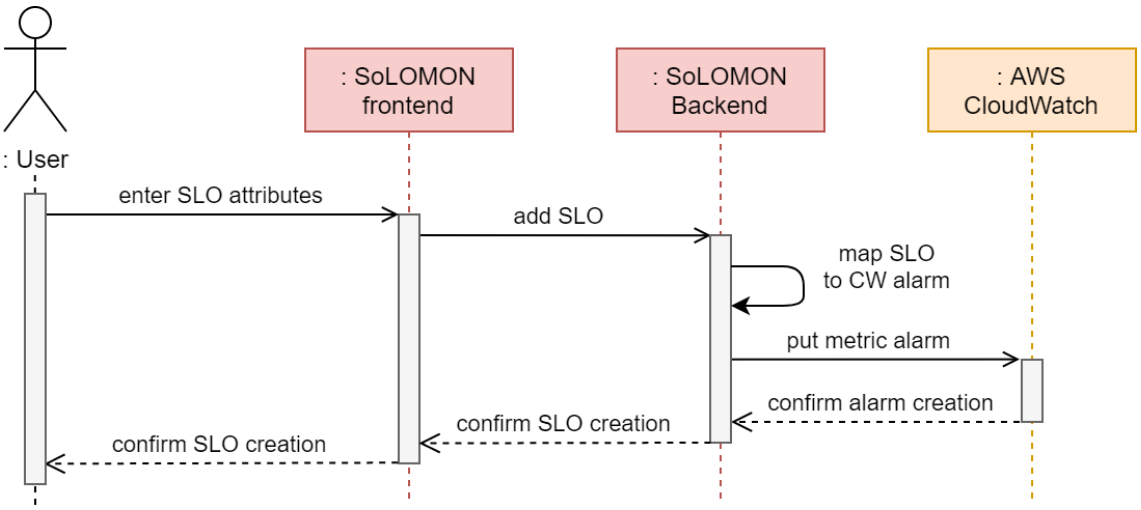

Figure 4.4: Simplified process view of SoLOMON for the creation of SLOs for AWS.

The SoLOMON frontend is what the user normally interacts with in order to use SoLOMON. To create a new SLO, there are a few attributes that the user has to enter or select in the user interface. The endpoint in the SoLOMON backend for adding a new SLO is called with a JSON object, which makes up the SLO, as payload. In the backend the SLO is mapped to the format which is used to describe alarms in CloudWatch. The CloudWatch alarm description is then sent as payload with the *put metric alarm* call to the CloudWatch API. What follows is a cascading confirmation of the successful creation of the alarm, respectively the SLO, which eventually reaches the user.

Fig. 4.5 helps to better understand the data flow in this process and shows what happens inside of the backend.

The SLO, as created by the user with the help of the frontend, arrives at the SoLOMON API on one of currently two possible endpoints for creating new SLOs. The endpoint for creating new SLOs for an AWS environment is `/slos/aws`, while the one for Kubernetes environments is `/slos/kubernetes`. The environment specific part of the URL is passed on to the Forwarder together with the SLO object itself. Based on the passed environment parameter, the Forwarder then either calls the *add SLO* function of the CloudWatch Connector, or the one of the Kubernetes Connector. In both cases the SLO object is again passed on, and the two Connectors each use their own Mapper which maps the SoLOMON-specific SLO description to either a CloudWatch alarm configuration or a Prometheus query configuration. Finally the API of the monitoring service in the respective environment is called, either the CloudWatch API for AWS, or the Prometheus API for Kubernetes and the metric alarm or the monitoring query is created.

Concerning the violation of created SLOs, Fig. 4.6 shows a sequence diagram with the process.

Multiple components are involved here: First of all there is AWS CloudWatch which triggers an alert when the threshold for an existing alarm is crossed. This alert is published to a specified topic in the AWS service called SNS. The Notifier Lambda function is a subscriber to this topic and gets triggered with the alert as a payload when a new alert is published to the topic. When the Lambda function is activated it sends a HTTPS request with the alert in the body to the specific alerting endpoint of the SoLOMON backend. In the SoLOMON backend, the CloudWatch-specific alert

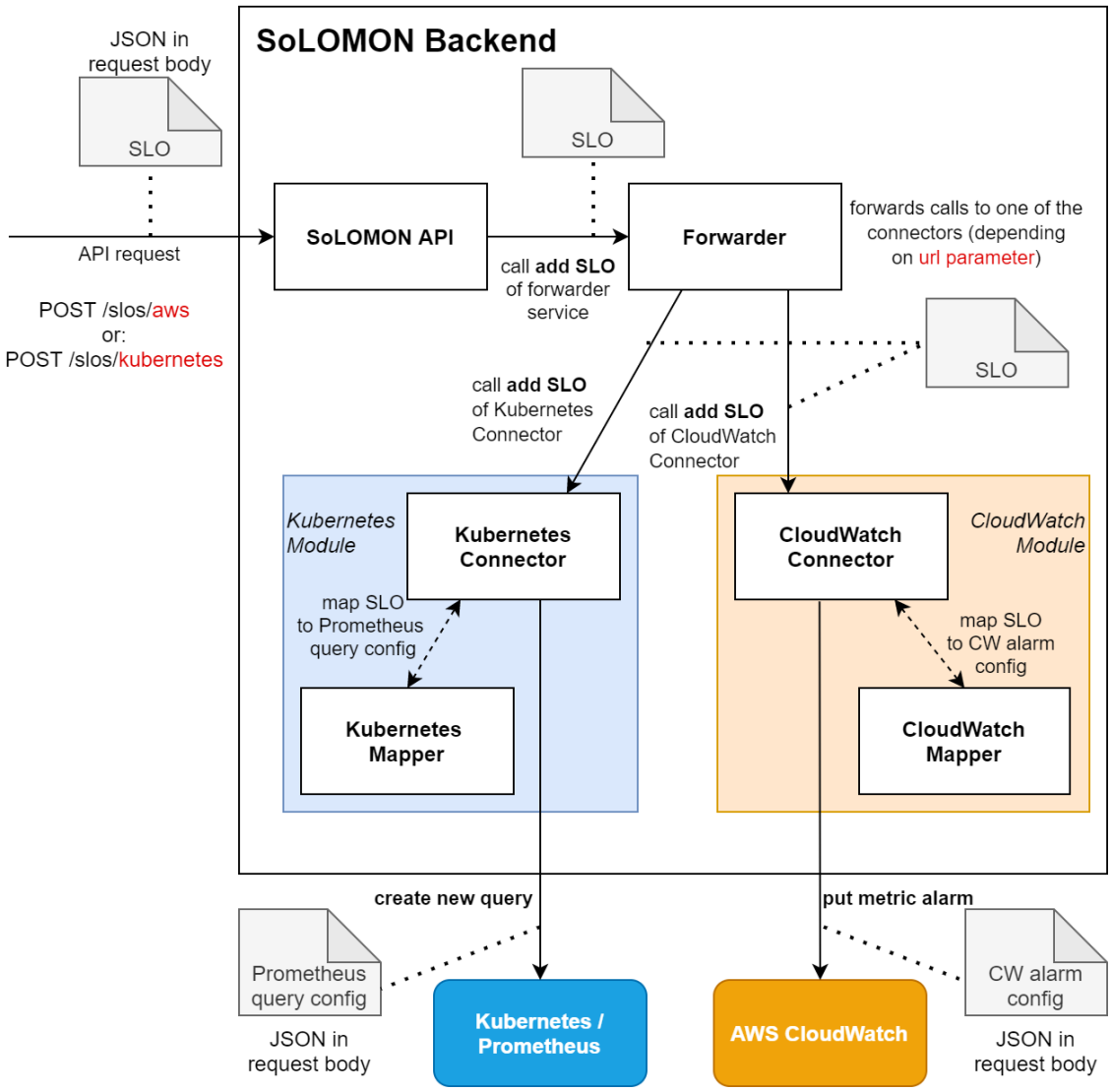

Figure 4.5: Data flow inside the SoLOMON backend for the creation of SLOs.

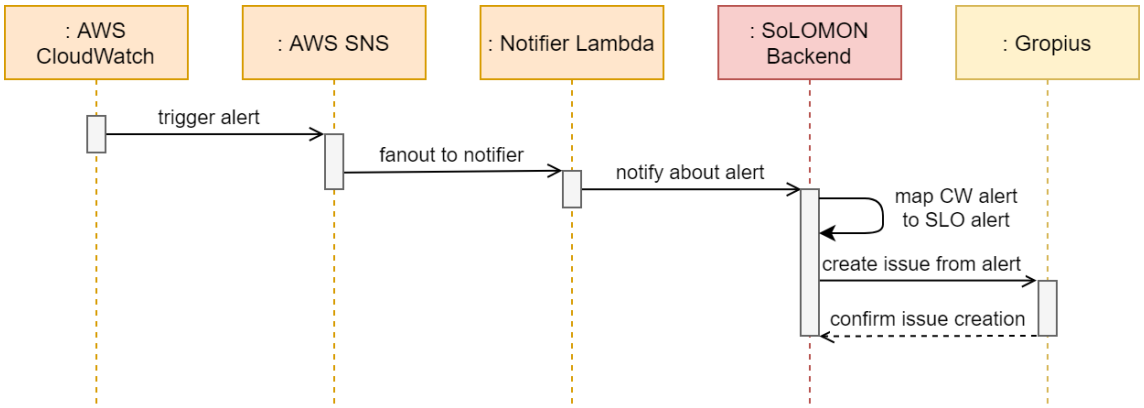

Figure 4.6: Simplified sequence diagram for the alerting process.

gets then mapped to a more generic format called *SLO alert*. Based on this SLO alert, an issue containing relevant information about the alert can be defined and then send to Gropius where the respective issue gets created. If the issue could be created successfully, Gropius confirms the issue creation to the SoLOMON backend. More details about the implementation of the alerting process and its specifics can be found in section 5.4.

4.3.3 Development View

What can be seen in Fig. 4.7 is a quite complex component diagram of the SoLOMON backend. Yet, this diagram is still a simplified one, which leaves out a few aspects for the sake of better readability. For example, it is concerned only with the components and the interfaces needed for operation of SoLOMON for an AWS environment. This is why no interfaces to Kubernetes or Prometheus are shown and the Kubernetes Connector is also left out. The Forwarder component, which would normally be located between the SoLOMON API and the CloudWatch Connector was also not modelled here.

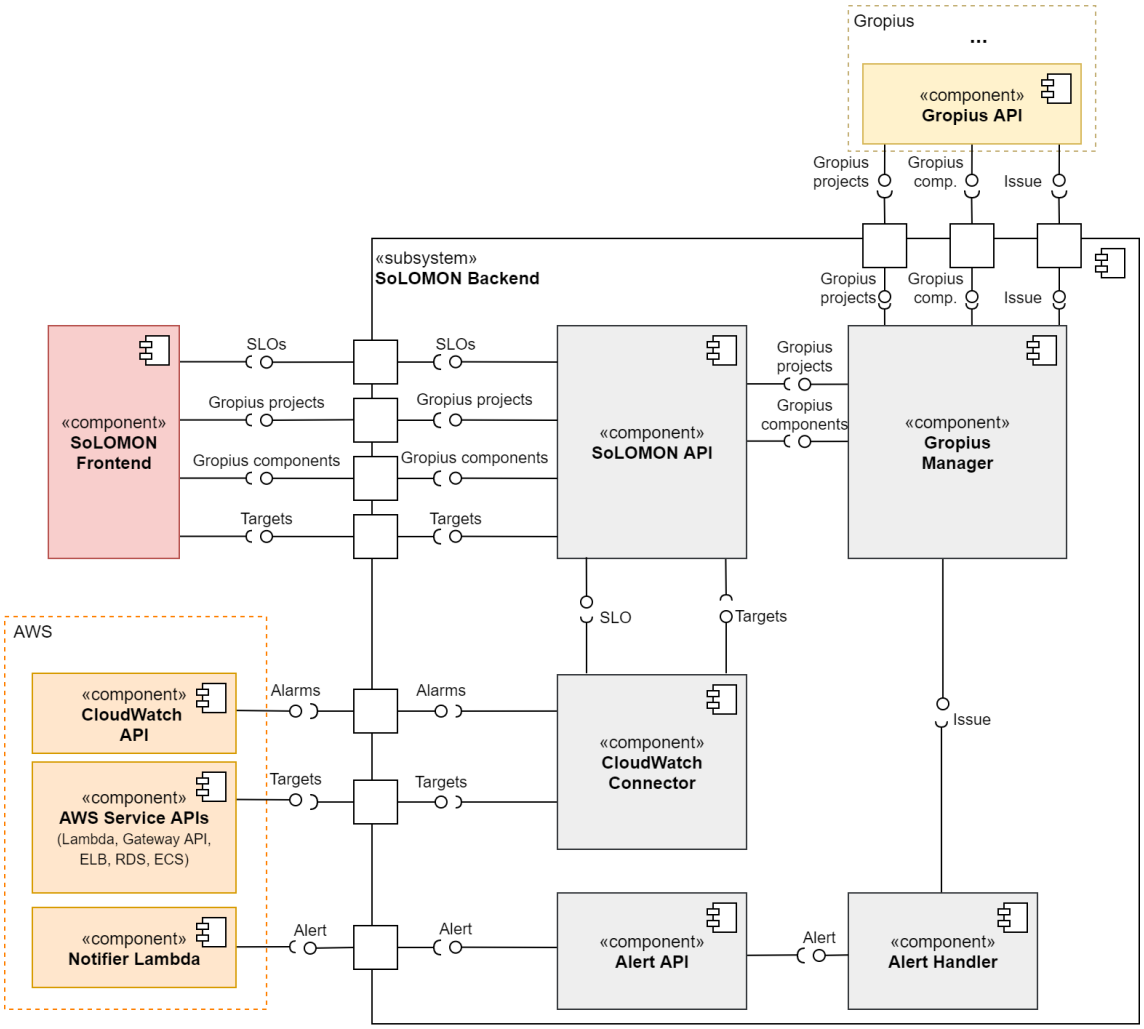

Figure 4.7: Simplified component diagram of the SoLOMON backend.

As can be seen, the two API components (SoLOMON API and Alert API) are those which offer a provided interface to the outside of the backend at the ports. The SoLOMON API exposes the basic functionality of the application to clients such as the SoLOMON frontend. This basic functionality consists of SLO-related operations such as adding, updating or deleting an SLO, as well as listing all existing SLOs. Additionally, this API also provides interfaces which allow to list all Gropius projects and the Gropius components which belong to a project. In order to be able to provide this information the SoLOMON API requires the interfaces from the Gropius Manager component which fetches it from the API provided by Gropius. The CloudWatch Connector component requires the interfaces provided by different AWS service APIs. The most important one is the CloudWatch API. It is used to fetch existing alarms and to add new alarms. The alarms correspond to the SLOs whose creation the SoLOMON API allows and which it can list. To fetch the lists of possible targets for the different target types such as Lambda, API Gateway, ELB, RDS and ECS, the APIs of these services have to be called. After having fetched this information, the SoLOMON API can provide this information to the frontend where it can be used in the creation of new SLOs. The Alert API provides an endpoint where the Notifier Lambda function can send an alert to SoLOMON. When an alert arrives it is passed on to the Alert Handler. This component creates an issue which relates to the alert and passes it to the Gropius Manager, which is responsible for invoking the external Gropius API. More specifically, it accesses the provided interface of the Gropius API which allows the creation of new issues.

4.3.4 Physical View

This final view has as its aim to visualize the topology of the components on the physical layer, as well as their connections. Fig. 4.8 shows the deployment diagram created for this view.

The depicted deployment is the one at the point in time when the evaluation was conducted, and is not seen as an optimal deployment yet. One of its biggest drawbacks is that the SoLOMON frontend is running on a local development server on the user client. Ideally the frontend would be deployed in the same ECS cluster as the backend, but as a different task. This would allow each of them to be accessible on its own IP address that it gets provided by the ECS service. Because of different limitations, this improvement in the deployment is estimated to require some effort, and as the focus of this work is mostly on the backend of the SoLOMON application, it was not implemented yet. The frontend accesses the backend via HTTPS. All the resources running in the AWS environment are part of the same Virtual private cloud (VPC). This allows an easy communication between them, as they can be addressed via private IPs. This is also an advantage regarding security, as the components do not need public IPs, which could expose them to malicious actors. As already mentioned, the SoLOMON backend container runs as a task in a specially defined ECS cluster. The cluster gets set up using an AWS service called CloudFormation. This process is described in more detail in section 5.3. The Gropius instance which was set up for testing and evaluation purposes is running on an EC2 instance. Docker Compose is installed on that instance and is responsible for running the Gropius backend container and the Gropius DB container. The SoLOMON backend accesses the Gropius API via HTTP. HTTP is also used by the SoLOMON backend to access the CloudWatch API and use it to create new alerts. When an alarm is triggered CloudWatch informs the SNS topic specified in the alarm. This communication is AWS internal. The same is the case for the Notifier Lambda function which is deployed using AWS Lambda. This Lambda function is a subscriber to the SLO Alerting topic. The Notifier Lambda then communicates the alert to the

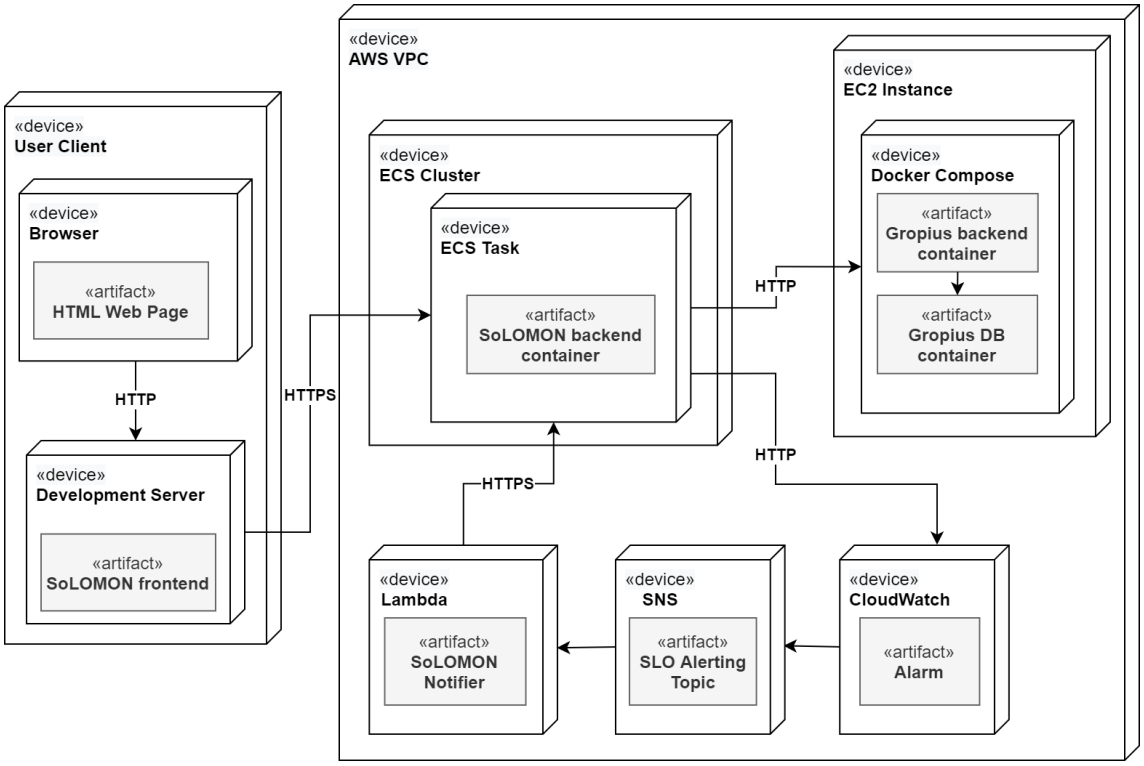

Figure 4.8: Simplified deployment diagram of SoLOMON.

SoLOMON backend via HTTPS. To be able to correctly do this, the Lambda function has to be correctly configured. Concretely, this means that it needs to get the IP address where the SoLOMON backend container is running passed as an environment variable.

4.4 SLO Interface

The design of the SLO interface was an important aspect of this work. It describes the structure of an SLO, as the SoLOMON backend expects it and as the SoLOMON frontend is able to create it. As this interface is shared between the backend and the frontend, and might in the future also be used by other clients, we decided to put the shared interfaces, such as this one for the SLO, into a separate repository[2]. Package managers such as Node Package Manager (npm) allow to install modules from GitHub repositories just as modules from package repositories.

Although we studied some of the SLA languages proposed in literature (see subsection 2.2.2), we could in the end not adapt any of it directly. The reason for this is that although there are some attributes that appear in one form or another in most of these languages, there are a few additional attributes needed specifically for the SoLOMON context which do not, as they are to specific. The core SLO attributes which appear in most SLA languages are for example the metric type, the comparison operator, the statistic (or aggregator), the period and finally the threshold. The problem

[2] **https://github.com/ccims/solomon-models**.

is that in addition to these attributes we also need additional ones for identifying the Gropius project and the modelled Gropius component with which the SLO should be associated when an issue gets created. Another attribute to distinguish which monitoring tool needs to be used to monitor the component is needed, as there are at the moment two, namely CloudWatch and Prometheus. What this means is that even if we would have used an existing SLA language to describe the SLOs in SoLOMON, we would have to wrap each SLO written in the specific language format in a special wrapper which would allow us to add the additional attributes. As this is unnecessarily complicated and brings no real benefit, we did not pursue such an approach, but decided to define our own SLO format. This allows us to define the SLO interface in such a way that all the attributes needed to define CloudWatch alarms and Prometheus query configurations are contained. This was important, as the main idea behind SoLOMON is the mapping from SLO to the alarm or query format of the specific monitoring tools. Being able to have this mapping work in both directions is what allowed us to implement the backend of SoLOMON as a stateless component. How this looks exactly is described in more detail in section 5.5.

Listing 4.1 SLO interface as used in SoLOMON.

```
export default interface Slo {
    id?: string;
    name: string;
    description?: string;

    deploymentEnvironment: DeploymentEnvironment;
    targetId: string;
    targetType?: TargetType;

    gropiusProjectId?: string;
    gropiusComponentId?: string;

    preset?: PresetOption;
    metricOption?: MetricOption;
    comparisonOperator?: ComparisonOperator;
    statistic?: StatisticsOption;

    period: number;
    threshold: number;

    alertTopicArn?: string;
}
```

Now to the actual interface we arrived at in the end and which can be seen in Listing 4.1. Its first attribute is the **id**. This attribute is optional because, when creating an SLO for the first time, the object that is sent to the SoLOMON backend has no ID yet. The ID gets generated in the case of AWS only when creating the corresponding alarm in CloudWatch, and for Prometheus it is generated in the Kubernetes Connector. For SLOs that target components in an AWS environment the SLO ID equals the Amazon Ressource Name (ARN) of the corresponding CloudWatch alert. The **name** and the **description** of the SLO are just user given strings which help to identify and understand it better, and which are shown in the SoLOMON frontend.

The **deploymentEnvironment** describes whether the component which is to be monitored is deployed in an AWS or a Kubernetes environment. Currently only these two environments are supported, but others could be added relatively easily. The **targetId** is what is used in the respective deployment environment to identify the specific component that is to be monitored. For AWS, this is the name of the component, e.g. of the Lambda function. Additionally, there is the **targetType** attribute which allows to specify the service type of the target. For AWS targets, currently the following types are supported: Lambda, API Gateway, Network Load Balancer, RDS and ECS.

The next two attributes, **gropiusProjectId** and **gropiusComponentId** help to map the target of the SLO to a modelled component in Gropius. The Gropius issue that will be created in case an SLO is violated will be linked to the Gropius project and component that are defined here.

Next, there is the **preset** attribute. This is an optional attribute, the use of which is not yet fully implemented in SoLOMON. The idea behind it is to be able to create presets of value selections for the next three attributes, namely *metricOperator*, *comparisonOperator* and *statistic*. Once this is fully implemented, it will ease the creation of SLOs as the user will not have to make selections for all three fields when creating similar SLOs for different targets. The attribute **metricOption** is the name of the actual metric that should be used to monitor the target. In AWS, there are different metrics that exist for different target types. For Lambda functions there is for example duration, the number of invocations, the number of errors and so on. For API Gateways there are 4XX errors, 5xx errors, the latency, and so on. The **comparisonOperator** is the mathematical operator that is applied to the threshold in order to check whether the target is in the accepted range of performance. It is to be noted here that the comparison operator of the SLO is the exact inverse of the comparison operator that will be used in the corresponding alarm in CloudWatch for example. This is because an SLO is positively formulated: "*The duration of the Lambda function should, on average in the period of 5 minutes, be lower than 3 seconds.*" The alarm formulation that this would correspond to would look something like this: "*Trigger an alert when the duration of the Lambda function is, on average in the period of 5 minutes, higher or equal to 3 seconds.*" Then there is the **statistic**. This attribute describes which statistical operator should be applied to the data points measured in the period of observation. The operations supported here are average, rate, sample count, sum, maximum and minimum.

The **period** attribute describes the evaluation period in seconds and the **threshold** the value against which the measurement should be done. Finally there is the **alertTopicArn** attribute. This attribute is needed specifically for AWS SLOs. It describes the ARN of the SNS topic to which alerts should be pushed. The user has to make sure that he has subscribed the SoLOMON Notifier Lambda function to this topic. Only this way can the SoLOMON backend be informed about alerts on the SLOs defined for AWS environments.

We want to finish of this section by presenting an exemplary SLO as it was created for the CFS backend application in the evaluation process. The SLO formulated in in natural language could look like this: *"The sum of 4XX errors of the OwnerRestApi API Gateway should be less than 1 in the period of 60 seconds"*. In the format as SoLOMON expects it, it can be seen in Listing 4.2.

Listing 4.2 Example of an SLO using the SoLOMON SLO interface.

```
{
    "id": "arn:aws:cloudwatch:eu-central-1:767491486699:alarm:APIGW4xx",
    "name": "APIGW4xx",
    "description": "4xx Errors for the API Gateway",
    "deploymentEnvironment": "aws",
    "targetId": "adev-frankfurt-OwnerRestApi",
    "targetType": "AWS-ApiGateway",
    "gropiusProjectId": "5eb221a67f566009",
    "gropiusComponentId": "5eb21f13d5d66005",
    "metricOption": "4XXError",
    "comparisonOperator": "LessThanThreshold",
    "statistic": "Sum",
    "period": 60,
    "threshold": 1,
    "alertTopicArn": "arn:aws:sns:eu-central-1:767491486699:SLO-Tool-Alerting"
}
```

4.5 Applied Patterns

In this section we want to shortly mention some of the design and architecture patterns used in the design and the development of the SoLOMON application. Two of the most important categories of non-functional requirements we set up for SoLOMON were extensibility and maintainability (R-NF02 - R-NF05). Both of these quality attributes are closely linked to the broader category of modifiability and, thus, to its inverse, cost of change. According to Northrop, “the cost of change [...] refers to the ease with which a software system can accommodate changes” [Nor04, p. 28]. By having a high modifiability, namely being able to accommodate changes easily, we can keep the cost of change low. As modifiability played such an important role in the design of SoLOMON, we were heavily inspired by some of the so called modifiability tactics as presented by Bass and Bachmann, and the patterns that implement these tactics. The two define modifiability tactics simply as “an architecture transformation that improves the modifiability of ... [the] architecture” [BBN07, p. 9]. On a very abstract level, a good approach to ensure a good modifiability is to localize expected modifications and encapsulate the parts of the system that are expected to change as well as to restrict visibility of responsibilities. The concrete patterns through which this was achieved will be presented in the following.

4.5.1 Microkernel Pattern

The microkernel pattern is influenced by two main forces: the need to cope with software and hardware evolution and, following from this, the aimed for portability, extensibility and adaptability of the application [BBN07, p. 45]. The main idea behind it is to separate a minimal core of the application from extended and specific functionality. These extension components are also called plug-in modules and provide specific additional rules or logic [Ric, p. 30]. In the case of SoLOMON, the plug-in modules are the modules for the connection to the specific environment and the monitoring service of the environment, namely the CloudWatch Connector and the Kubernetes

Connector. The extensibility we want to achieve for SoLOMON regards exactly such plug-ins for the connection to even more cloud environments such as the Google Cloud Platform and Microsoft Azure.

Some of the tactics used in the microkernel pattern, as described by Bass and Bachmann, are abstract common services, encapsulation, restricted communication paths and the use of intermediaries. For the Connector plug-ins for example there is a common abstract service from which both inherit. The abstract service describes methods for fetching, adding, editing and deleting SLOs, as well as a method for fetching the targets of the respective environment. The plug-ins each encapsulate all the logic for accessing the API of the respective environments and monitoring services. An intermediary called Forwarder service is also used in our architecture. This intermediary stands between the SoLOMON API and the specific connectors. The Forwarder also represents a restricted communication path from the SoLOMON API, which is part of the microkernel, to the plug-ins.

4.5.2 Adapter Pattern

The adapter pattern is also mentioned in Bass and Bachmann's description of the microkernel pattern [BBN07, p. 46], but to get a better understanding of it, a look at the 1993 classic *Design Patterns* [GHJV93] by Gamma et al. is useful. According to them, an adapter "makes one interface (the adaptee's) conform to another, thereby providing a uniform abstraction of different interfaces" [GHJV93, p. 411]. In the case of SoLOMON, the third party interfaces that need to be adapted are the CloudWatch API, and the Kubernetes API. The uniform abstraction of these two interfaces is the *ConnectorService* interface. The both adapters, the CloudWatch Connector and the Kubernetes Connector, each implement the ConnectorService interface. The adaptors are implemented in such a way, as to call the respective endpoints of the APIs of the environments and to process the results. The CloudWatch Connector, for example, calls the CloudWatch API and fetches all existing alarms that were created before. It then maps them onto their respective representations as SLOs and returns them. It works the same way for other methods, such as the one for adding or deleting SLOs. In this way, it makes the SoLOMON internal interface, which uses SLOs, conform to the interface of the CloudWatch API, which is unaware of SLOs and only knows about alarms.

5 Implementation

This chapter describes some of the most interesting aspects of the implementation of SoLOMON. Some of the challenges that occurred during the development and the deployment process are described, as well as different approaches to solve them and the reasoning behind them. The repository in which the implementation can be found is on GitHub[1]. The chapter ends with a presentation of the tools and technologies that supported the implementation process.

5.1 Existing Prototype

As already mentioned in section 2.2, SoLOMON builds upon an existing prototype which was developed at the Institute of Software Engineering (ISTE) in Stuttgart. One limitation of this prototype is that it only supports monitoring services in Kubernetes clusters, and that SoLOMON itself has to run in the same Kubernetes cluster as the services which are to be monitored. This has the obvious disadvantage that if there is a problem with the cluster itself or its availability the user will not be able to track this with SoLOMON. As a general rule, it makes much more sense to have the monitoring tool deployed separately from the application that is to be monitored.

As the prototype was developed more as a proof of concept for the context of Kubernetes Clusters, most of its code was logic concerned with Kubernetes- and Prometheus-specific operations. We extracted this code and put it into a separate module that we called the *Kubernetes Connector*. In the same way we created the *CloudWatch Connector* which contains the logic for communicating with AWS. In order to increase maintainability and extensibility, each of the services containing the logic for each of these connectors, inherits from an abstract service. Thus, we can ensure that new connectors support the basic operations of retrieving, adding, editing and deleting existing SLOs.

Other than that, the prototype contained a model for the SLO objects, which we updated so that it also contained certain attributes which are necessary for the definition of CloudWatch alarms. And finally there is the controller which defines the REST API of the backend. This controller was also modified so that the naming of the API endpoints corresponds to standard naming conventions. New endpoints were also added to the API so that it allows the fetching of a list of possible monitoring targets. Additionally, the endpoint were redesigned in such a way that a URL parameter indicated whether the information of interest (e.g. existing SLOs or targets) should be fetched from AWS or from Kubernetes.

[1] **https://github.com/ccims/solomon**.

5.2 Local Development and Its Limitations

In the initial phase of the project, SoLOMON was developed locally and tested by deploying an HTTP server which runs the backend on the local machine. For making different kinds of HTTP requests to the backend an API client called Postman was used. With this tool the REST API of the backend could be tested, as it allowed to make for example GET and POST requests with predefined payloads. Thus the HTTP requests that the backend would normally receive from the frontend could be mocked. This local deployment was useful in the first part of the project as it was fast, allowed quick testing without a complicated deployment process.

But eventually we reached a point at which the local deployment did not suffice anymore. Until this point we implemented the REST API of the backend, a mechanism to forward the requests to the specific connectors and a basic version of the CloudWatch connector, which allowed retrieving, adding, editing and deleting CloudWatch alarms based on the SLO passed to the backend. But now the backend should also be able to receive notifications when the alarms in CloudWatch would be triggered. One of the limitations of CloudWatch is, that the only way to handle the triggering of an alarm is using another AWS service called Simple Notification Service (SNS). SNS is based on the publish-subscribe pattern. We created a topic to which CloudWatch publishes the notifications about triggered alarms. The subscription to the alarms can be done through HTTP or HTTPS. In order to add a subscriber to the topic, an endpoint for which the subscription should be done needs to be specified. We intended this to be the SoLOMON backend. This is the point at which the local deployment of the backend is not sufficient anymore, as SNS cannot sent the HTTP/S requests to an instance of the backend which is deployed only on a local machine.

At this point in time we decided to deploy SoLOMON using AWS. This would allow assigning a public IP to the backend, and would make it possible for it to subscribe to the topic on which the notifications about triggered alarms are published. Such a deployment was also a logical step towards the evaluation of SoLOMON in a realistic environment.

5.3 Deployment on AWS

For the deployment of SoLOMON backend we chose the modern approach of deployment in the cloud. AWS was selected as cloud provider, as it is already used at Vector Informatik. For the deployment type of the application there were basically two alternatives: The application could be deployed either on a virtual machine, or it could be deployed using a container-based deployment. For virtual machines AWS offers the well known Elastic Compute Cloud (EC2). The disadvantage of deployment using EC2 are that the virtual machines have predefined sizes and compute power, and the cost for them is thus constant even if the compute power or the memory is not needed. AWS Fargate on the other hand, which is Amazon's serverless compute engine for containers, automatically allocates the right amount of compute capacity needed to run containers. This container-based approach also has the advantage that once SoLOMON is containerized, it also has a high portability: A Docker container, for example, can be deployed to a Kubernetes environments. A Kubernetes environment on the other hand can be provided either through the offerings of all major cloud providers (e.g. Amazon Elastic Kubernetes Service, Azure Kubernetes Service, Google Kubernetes Engine, etc.), or can be hosted on-premise. The next step was thus to write a Dockerfile that allows the creation of a Docker container image. A Dockerfile is a text-based script basically

containing the instructions that need to be called in order to build and run the application. But before building the application, a base image needs to be selected. For Node.js applications, a base image which contains Node.js and its package manager (npm) is needed. This allows for the next step in which the dependencies of the application are installed. After installing the dependencies the application is ready to be build. The final step is the command to start serving the application. After having thus dockerized the application we can use Docker Desktop in order to test the creation of an container image and finally the local deployment of the Docker container. Having done this a big step towards high portability of the application was taken: Using the Dockerfile that is committed with the rest of the application in the public repository, anybody can easily create a containerized version of the application with a single command, and deploy it all the same in all kind of different environments.

As already mentioned in the beginning, the public deployment of the SoLOMON backend was needed already during the development process. Naturally, it would be important to have a good deployment pipeline which allows quick debugging and rapid feedback for changes in the code of the application. In order to construct a pipeline which allows this, a framework, also provided by AWS, has proven useful: The AWS Cloud Development Kit (CDK). CDK is a software framework that is used to define the infrastructure of cloud applications. After having described an application using the CDK another AWS service called CloudFormation can be used for the deployment and the provisioning of the application. What we are thus talking about here is simply Infrastructure as code (IaC). By describing the deployment of the application in the form of code we make it explicit and much easier to understand and to maintain. Especially in cloud computing the temptation is big to add ever more cloud services and components to cloud applications in order to allow for new features or a better performance. But this can quickly lead to disadvantages. This is how Morris puts it in his book on IaC: “Adopting cloud and automation tools immediately lowers barriers for making changes to infrastructure. But managing changes in a way that improves consistency and reliability doesn’t (sic) come out of the box with the software” [Mor16, p.3]. Besides improving consistency and reliability, the IaC approach also offers other benefits: the biggest of those are the possibility to use logic and object-oriented techniques when defining the infrastructure, as well as the ability it gives users to define high level abstractions which can be shared and reused.

The main building block of the IaC description in CDK is a stack. A stack describes a unit of deployment and it contains the definitions of all resources that are to be provisioned for this specific deployment unit. More complex systems can of course consist of multiple stacks, but for SoLOMON we considered that one stack would be enough. This way our entire deployment is captured in a single file.

CDK provides a command line tool which allows to initialize new stack descriptions and to deploy stacks. We initialized a stack description in the repository of SoLOMON. As we will see later, this allows the referencing of the application code in the deployment description. The first thing that is defined in the stack is the Virtual private cloud that the resources that are to be deployed should belong to. Instead of only creating new resources CDK also allows the referencing of existing resources. Using its ID, we thus referenced to the already existing VPC that is used at Vector Informatik in our department. In the next step, we create a new ECS cluster to which the container we will create will belong. Similar to Kubernetes, a cluster in AWS is a logical grouping of services. As already explained above, the main goal of this whole process is to deploy an AWS Fargate service which can run the container containing SoLOMON. One important attribute of the Fargate service we need to define is the security group it should use. The security group defines the access to the

container hosted with Fargate. In the first phase, when still testing the deployment, we used this to allow access only from the IP address of my local machine. This was achieved by adding so called inbound rules to the security group. These rules can restrict from what kind of sources traffic is allowed on specific ports. In the beginning when the backend server was implemented using HTTP, we thus added a rule which allows inbound HTTP traffic on port 80, but only from the IP address of the local developer machine. This allowed for testing and debugging the deployed application without allowing access from other sources. In this way we can prevent security risks such as from bots that automatically scan open ports on the internet.

Another aspect of the stack definition is the task role. Without entering to much into the details, the task role is important, as it decides for example to which other AWS services the created resources should have access to. In the case of SoLOMON, this is a crucial aspect, as we explicitly need it to communicate with CloudWatch in order to be able to set and retrieve the existing metric alarms. To enable the creation of new alarms, we to use a policy which allows write access to CloudWatch. We also need read access to the Lambda service in order to be able to show the user of SoLOMON for which Lambda functions he or she could create SLOs. In order to ease troubleshooting and debugging we also need to add a policy with write access to CloudWatch Logs. Having done this we can easily check the logs emitted from the SoLOMON container in the CloudWatch web console.

Maybe the most important part of the stack definition is the one where we create the container image. As can be seen in the code excerpt (listing 5.1), the location of the directory of the source code needs to be passed to the *fromAsset()* method of the *ContainerImage* class. Using the created container image called here *sloToolBackendImage*, we can then create a container definition called *containerDef*, which we add to a task definition that we created before. Having also added the task role with the different access permissions to other AWS services that we talked about before to the task definition, as well as some other parameters regarding the compute power that is to be used with the container, we are ready to instantiate the Fargate service. We pass it the cluster, the task definition, the security group that we defined and set a flag to assign a public IP address to the deployed application. These were the most important parts of the stack definition.

Listing 5.1 Excerpt from the CDK Stack definition concerning the creation of the container image.

```
const path = require('path');
const sloToolBackendImage = ecs.ContainerImage.fromAsset(path.normalize(path.join(__dirname, '
../../solomon-backend/')));

const containerDef = taskDefinition.addContainer('slo-tool-backend-container', {
  image: sloToolBackendImage,
  containerName: 'slo-tool-backend',
  logging: ecs.LogDrivers.awsLogs({streamPrefix: 'SloTool'})
})
```

There is one other important file in the CDK part of the project. In it we basically create an instance of the stack according to the stack definition we just explained. In order to be able to do this we need to specify the ID of the AWS account on which we want to deploy our infrastructure. Initially this ID was hard-coded, but in order to be able to also publish this deployment-related part of the

project to the public Git repository, this should be prevented. Although publishing only the account ID in itself is not necessarily a security risk, we considered that it still would be better to load it from the environment variables of the machine triggering the deployment.

For the system test which was part of the evaluation, we also needed a running Gropius instance. As explained in section 2.2.1, Gropius is the issue management system in which the issues should be created in case there is an SLO violation. In order for SoLOMON to be able to communicate with Gropius, it needed to be deployed first. For the deployment of Gropius we created an EC2 instance on which we run Docker Compose. Docker Compose allows to run multi container applications, and Gropius was developed as a multi container application. The two containers which we needed to run with Docker Compose are one for the Gropius backend and one for the Gropius database, which holds all the information about created Gropius projects, components and issues. In order to ease the communication between the SoLOMON instance and the Gropius instance, we deployed them in the same subnet of the same VPC. Subsection 4.3.4 from the previous chapter elaborates more on the deployment and contains Fig. 4.8, which gives a graphical overview.

5.4 Receiving Alarm Notifications

Receiving the alarm notifications that CloudWatch sends when an alarm is triggered turned out to be one of the more difficult things in the conception of our solution. As already mentioned in section 5.2, in order to receive an alarm notification from CloudWatch, the Simple Notification Service (SNS) has to be used. We thus created a topic in SNS with the purpose to receive all the notification alerts form the triggered CloudWatch alarms. In order for this to work, we need to pass the identifier (Amazon Ressource Name (ARN)) of the SNS topic to the CloudWatch alarms when we create them. This is why we needed to add an attribute to the SLO interface describing the ARN of the SNS topic. We also added an additional endpoint to the SoLOMON backend which fetches all existing SNS topics for an AWS account.

SNS topics allows multiple types of subscribers. These are for example Email or SMS, other AWS services such as Lambda or SQS and HTTP or HTTPS endpoints.

The most straightforward approach would have been to use HTTP, as the SoLOMON backend was running in the local development environment using the HTTP protocol. Even in the beginning when deploying it on AWS we used still used HTTP in combination with a inbound-rule which allowed HTTP traffic only from explicitly listed IP addresses. Choosing this approach, it was necessary to somehow add an inbound-rule that whitelists SNS and allows it to send HTTP requests to SoLOMON. This turned out to be more difficult than initially thought: Some research showed that the only way to find out the IP address ranges of AWS services is to download a JSON file which contains thousands of objects and currently has a length of 33302 lines [Ama21c]. Besides the IP prefix, the objects have attributes that define the region and the service. But although there are categories for services such as EC2, API Gateway and others, there is currently no separate category for SNS. Instead there is a general category called *AMAZON*. Even when filtering by this general category and the region where the SNS is used, we were still left with 233 IP prefixes which would have to be whitelisted. Additionally, a security group is only allowed to have 60 inbound rules, which would mean that we would have to create multiple security groups to cover all of the IP prefixes, or requests a charged quota increase. This did not seem to be right.

The second approach was to implement the SoLOMON backend in such a way that it uses the HTTPS protocol and Basic Authentication. This would allow us to deploy it with a public IP address without having to allow traffic only from certain IP addresses and without risking unwanted access to it. In order to be able to do this we generated a self-signed Secure Sockets Layer (SSL) certificate for the server and defined a Basic Authentication user which should have access to the server. When the SoLOMON backend gets bootstrapped we pass the SSL certificate and key as HTTPS options, enable Basic Authentication with the user we are loading from a file and let the server listen on port 443. For security reasons the SSL certificate and the file defining the Basic Authentication user should not be checked into the Version Control System (VCS). Later we added the possibility to decide whether the backend server should run using HTTP or HTTPS via an environment variable called `HTTPS_ENABLED`. We expected everything to work now, as we could instruct SNS to send a topic subscription request via HTTPS to the public IP where we deployed SoLOMON. But the requests did not arrive. The reason was, as we found out after some time, was that SNS refuses to send HTTPS requests to endpoints that do not have certificates from a trusted Certificate Authority (CA).

Because a trusted certificate was not easily available to be used for SoLOMON, and it was also not feasible to expect other developers who in the future might continue the work on it to have one at hand, we decided for a third approach. This third approach integrated the changes made for the second approach which we just described and is based on the idea of using an AWS Lambda function as an intermediary. As already mentioned, SNS allows Lambda functions to subscribe to topics. We then implemented the Lambda function which does nothing more but forwarding the content of the notifications from the SNS topics to the endpoint of the SoLOMON backend. To allow this, the IP address where SoLOMON is deployed and the Basic Authentication user are passed as environment variables to the Lambda function. Another advantage of this third approach is that, unlike SNS, a Lambda function is able to send HTTP/S requests to SoLOMON even if it is deployed only with a private IP address. In order to allow this we just had to ensure that the Lambda function is also assigned to the same private subnet of the same VPC as our SoLOMON instance. Not having to deploy SoLOMON with a public IP address meant less security concerns.

Although this approach works, it is not seen as optimal as it increases the deployment overhead by the additional Lambda function, which is needed to use SoLOMON for a serverless application running in an AWS environment. Once the notification about a triggered alarm reaches the alert endpoint in SoLOMON, the content of the notification is mapped to a generic SLO alert format. The SLO alert is then used to generate a new Gropius issue with the relevant information.

5.5 Stateless Backend

One question that arose during the development of SoLOMON is whether the SLO objects should be stored, either in-memory in the backend or in a separate database that is connected to it, or whether SoLOMON should be stateless. At first sight it seemed more intuitive to store the SLO objects in the backend, as they contain information that is needed later. One example for this is the attribute called *gropiusTarget*, which is needed to be able to create an issue for the correct modelled component of a Gropius project in case an SLO is violated. This attribute initially got lost in the mapping from SLO object to CloudWatch Alarm definition and, thus, we thought it would be a good idea to persist it in the backend of our application. But we soon realized that this would bring

other problems. The main problem with this approach would be synchronization. As SoLOMON would not be the only client that could create, modify or delete CloudWatch alarms, this could definitely be a problem. Imagine for example following scenario: A user creates a new SLO using the frontend of SoLOMON. The SLO Tool backend persists this SLO and also initiates the creation of a corresponding CloudWatch alarm. The problem is that the user can also use the CloudWatch web console to view and edit existing CloudWatch alarms. If he now modifies the existing alarm and changes, for example, the threshold for the alarm, this change will not be reflected in the version of the SLO that is stored in the backend of SoLOMON.

When using the SoLOMON frontend to view the existing SLOs, the question arises where to get this information from? There seem to be two sources of truth: We could fetch the SLOs as they have been stored in the backend. But in this case the SLO would not reflect the actual current configuration of the CloudWatch alarm, and would thus be misleading. This is why it makes much more sense to use CloudWatch as the single source of truth, and to not store the state in the backend too. Back to the example: If the user wants to view all existing SLOs in the frontend, the backend just calls the CloudWatch API and fetches all existing alarms from there. These alarms can then be mapped back to SLOs and visualized in the frontend. This is an elegant solution, as this allows for a stateless backend. For example, if we shut down the backend server of SoLOMON, no information is lost: The existing SLOs can simply be derived from the existing alarms in CloudWatch. Of course this works just in the same way with the other connectors and their specific alarm interfaces.

One question that arises in this approach is how to store SLO attributes that have no direct counterpart in the CloudWatch alarm interface. Just not modelling these attributes in the CloudWatch alarm would lead to a loss of information when mapping the alarms back to SLOs. Two attributes for which this problem arises specifically are the `gropiusProjectId` and the `gropiusComponentId`. They describe to which Gropius project a target that is monitored belongs and what ID the modelled component in the Gropius project has. This information is important in order to be able to create Gropius issues for the correct Gropius project in case there is an SLO violation. Additionally, the created issue should also contain the information which modelled component a violation happened

The obvious solution to not lose this information and also keep the backend stateless is to somehow store it in the CloudWatch alarm and then to also restore it from there when mapping it back to an SLO object. The CloudWatch alarm interface does not foresee such attributes and there is also no possibility to just introduce custom attributes, unlike in Prometheus where alerting rules allow for custom labels and annotations. Still, by slightly misusing the description attribute of CloudWatch alarms and combining this with some logic applying regular expressions, we were able to build a satisfying solution. The basic idea was to just store the additional attributes in the description of the CloudWatch alarm using a certain syntax. This happens in the mapper class of the CloudWatch connector. When mapping an SLO to a CloudWatch alarm we just add the two additional attributes into the description field behind the actual description text. Additionally, we separate the description text from the two attributes with a visual demarcation. A alarm description generated in this way might then look like this:

```
Description of programmatically created alarm //// gropiusProjectId:grop-proj-234
gropiusComponentId:grop-comp-123
```

When mapping the other way round, from CloudWatch alarm to SLO, we use regular expressions to extract the two attributes. In listing 5.2, the function `getGropiusProjectId()` is shown, which extracts the Gropius project ID. A very similar method also exists for getting the Gropius component ID.

Listing 5.2 Method for extracting Gropius project ID from alarm description.

```
/**
  * extracts the Gropius project ID from the alarm description
  * @param alarmDescription Alarm description field of the CloudWatch alarm
  * @returns Gropius project ID of the SLO
  */
private static getGropiusProjectId(alarmDescription: string) {
    const matchRes = alarmDescription.match(/gropiusProjectId:([^\s])*/);
    if (matchRes == null) {
        return 'undefined'
    }
    return matchRes[0].replace('gropiusProjectId:','');
}
```

There is one more challenge we faced when implementing the stateless SLO Tool. This challenge was related to the fact that SoLOMON had to be aware of some configuration parameters. What this refers to at the minimum is the information about which connector should be used by the SoLOMON backend. When creating new SLOs, this is trivial. The user should simply make a selection in the frontend, whether the SLO is created for a component running in an AWS environment or a Kubernetes environment. This information is then an attribute of the SLO object. In the backend the forwarder service is able to forward the request for creating a new alarm to the corresponding connector by checking this attribute. This works perfectly for POST requests in which an SLO object is sent with the request, but it is not as straightforward for GET requests. Two examples of GET requests that the SoLOMON backend should be able to handle are those for retrieving the existing SLOs and those for retrieving the possible monitoring targets. For both of these actions the forwarder service needs to know which connector to forward the request to, or in other words, which deployment environment (AWS or Kubernetes) is to be queried for the information of interest.

The initial idea was to add an extra endpoint to the backend API which would allow setting the configuration of the backend, and more specifically, the deployment environment where the application that is to be monitored runs. An advantage of this approach would have been that the user does not need to specify for each action in the frontend for which deployment environment the action should be taken: After having once at the beginning defined the deployment environment and maybe even the Gropius project ID, these information are sent to the backend and stored in its configuration. The subsequent calls will not need to contain this information anymore, which would reduce the message size and complexity and would also declutter the user interface a bit. In the end we still decided against this approach, as it would break a desired property of the backend API, which could be called *atomic operations*. What this means is that ideally any operation on an API should be independent of previous operations. A concrete example: The operation of retrieving the existing SLOs should not be dependent on an operation for selecting the deployment environment before. In order to achieve this, we use different path variables in the design of REST

API. The GET request to retrieve all SLOs modelled for the AWS environment would thus go to the following endpoint: `/solomon/slos/aws`, while the one for the Kubernetes environment would go to `/solomon/slos/kubernetes`.

As a consequence, the frontend still needs to know for each call what the desired deployment environment is, in order to be able to call the correct endpoint in the backend API. This might be seen as a slight disadvantage, but it is one that can be mitigated quite well by caching the user's choices in the frontend. When creating a new SLO for example, the user might have to enter the Gropius project ID and select in which environment the application is deployed. Having then cached these choices, the frontend can pre-populate these fields for the next SLOs that the user might want to create.

Finally we want to point out one more problem that would have appeared, had we chosen the approach with the separate operation to set the deployment environment variable and the configuration state stored in the backend. This problem becomes obvious only when thinking about the possibility of having multiple users working with the same instance of the backend. One example where this would lead to much confusion is where one user would use SoLOMON to manage SLOs for an AWS deployment environment, and the user for a Kubernetes deployment environment. If one user sets the preferences of SoLOMON for his deployment environment and this configuration state is stored in the backend, this might lead to unexpected behavior for the other user. Of course the configuration state of SoLOMON could also be loaded into the frontend and shown there in order to at least inform the user that it has changed due to the actions of another user, but in the worst case the two users would end up getting in each others way in regards to the creation of SLOs for their deployment environment. Thus, having in perspective the extension of SoLOMON to being multi-user and maybe even multi-client, atomic operations on the REST API of its backend are almost an obligation.

5.6 Future Extension

There are multiple possibilities with regards to the future extension of SoLOMON. They are explained in more detail in section 7.5. This section has as its aim only to show how different kinds of extension are made easier through the current design and the implementation of SoLOMON.

First of all there is the **decoupling** of the frontend from the backend. Although this is very common in modern web applications, the aspect that we took special care of was the design of the SoLOMON API. The danger when implementing an API for a certain frontend is to design it to specific to that frontend. That is why we implemented the backend API as general as possible and with the idea of there possibly being other clients accessing the API in the future. One example of this is not relying on a configuration stored in the backend, but making each call to it atomic and containing all the configuration details necessary to properly process the call. This is explained in more detail in the previous section (see section 5.5).

The second important thing is that we wanted to make SoLOMON as easily expandable as possible. This is also reflected in requirement R-NF02. In order to this we designed the architecture of SoLOMON in a specific way. We decided for a **modular** approach of the backend in which the different parts of it are grouped in modules, based on functional relatedness of the classes. There is for example the CloudWatch module, containing the CloudWatch Connector service and the

CloudWatch mapper, as well as a file defining different CloudWatch related interfaces. In the same way there is the Kubernetes Module and the Gropius Module. Additional modules for adding support for the creation of SLOs for applications running in other cloud environments (e.g. Microsoft Azure, Google Cloud Platform) is possible and encouraged. To further aid this there is an **abstract class** for the connectors of these modules. The existing Connectors, the CloudWatch Connector and the Kubernetes Connector, both implement this abstract class. The abstract class serves as a guide to what functions have to be implemented in a connector in order for it to be usable in SoLOMON. Once the connector for Microsoft Azure, for example, is implemented, its use in SoLOMON is as easy as adding the case for this deployment environment to the Forwarder service which forwards the requests from the API to the different connectors. Of course it is also likely that a mapper for Azure would be needed, which would map the SoLOMON-specific objects to ones that Azure can understand.

As already mentioned in section 4.4, we put the SLO interface and for example the target interface into a separate GitHub repository. The interfaces from this shared repository are imported in the SoLOMON frontend and in the SoLOMON backend. This approach has the advantage that these interfaces can easily be imported into other projects as well. This makes the implementation of a new client for the SoLOMON backend straightforward. Additionally, we also provided some example SLOs in the repository, which might help developers approaching SoLOMON newly to quickly understand the structure and possible values for the SLO attributes.

5.7 Tools and Technologies

This section describes some of the underlying technologies of SoLOMON such as Node.js, NestJS and the different AWS services that are used. Additionally, it contains some information about tools such as Docker, Postman and Git which are useful regarding the development and the deployment of the application.

5.7.1 Node.js

Node.js, according to its developer documentation, is an “open-source and cross-platform JavaScript runtime environment” [Fou21b]. As it runs the V8 JavaScript engine outside of the browser and follows an asynchronous event-driven approach, it can have a very high performance and handle many connections concurrently without the developer having to deal with thread concurrency [Fou21b] [Fou21a].

Another advantage Node.js offers is that it allows the development of server-side code using the JavaScript language. This makes the development of the SoLOMON backend accessible to a wider variety of developers, and especially frontend developers, without them having to learn another programming language. The Node Package Manager (npm) is another factor that makes the Node.js ecosystem very attractive, as it offers developers the possibility to freely use the over 1 million open source packages it offers in their applications [Fou21b]. All these advantages mentioned make Node.js a good choice as the runtime environment for the SoLOMON backend application.

5.7.2 NestJS

NestJS is a framework that can be used to build scalable and efficient Node.js applications. By default it makes use of the Node.js HTTP Server framework *Express*, but provides a level of abstraction above it. The out-of-the-box application architecture that NestJS provides is heavily inspired by Angular. This architecture blueprint helps developers to build loosely coupled and easily maintainable applications. The code for a NestJS application can be written entirely in TypeScript and allows for elements from different programming paradigms such as Object Oriented Programming (OOP), Functional Programming (FP) and Functional Reactive Programming (FRP) [Nes21].

For SoLOMON, NestJS was chosen as the framework with which to implement the backend component. The application structure that NestJS provides is similar to that of Angular applications: There are **modules** which are used to organize components according to the relatedness of their capabilities. In SoLOMON for example, all the components related to the connection of the application to the CloudWatch API is grouped into the `ConnectorCloudWatchModule`. This is a way of encapsulating certain functionalities and, thus, helps to implement the principle of information hiding.

Other than this, the most important component types in NestJS are controllers and providers. **Controllers** handle incoming requests and define the responses that are returned to the client. This means that the API and in our case the REST API is defined in such a controller. NestJS allows for an easy definition of routing logic in the controller and also offers simple ways to handle the request object. Decorators can be used to define endpoints for all of the HTTP CRUD methods. Providers on the other hand are classes that can be used for dependency injection. Most of the logic of the backend happens in different service classes, which are a kind of **provider**. In the SoLOMON backend there is, for example, the so called `ForwarderService` which forwards certain operation prompts from the main controller of the backend to the different connector modules, depending on whether the application that is to be monitored runs in an AWS or a Kubernetes environment. The `ForwarderService` is injected into the main controller of the backend where we defined our API. Depending on the deployment environment, the Forwarder forwards the operation prompt (e.g. to list all existing monitoring targets) either to the connector service from the `ConnectorCloudWatchModule` or from the `ConnectorKubernetesModule`.

5.7.3 Amazon Web Services

The Amazon Web Services (AWS) are not only a very important part of this project, but also relevant on a global scale: AWS is the biggest player in the worldwide cloud infrastructure market, having a market share of around 32 percent [Ric21]. They offer a wide variety of over 200 fully featured web services and have a global network that is currently spanning 81 availability zones within 25 geographic regions [Ama21e].

Indirectly we are interested in the **services** needed to implement a serverless application on AWS. This is AWS Lambda, as well as other services such as Simple Queue Service (SQS), Elastic Load Balancing (ELB), Relational Database Service (RDS) and the API Gateway Service, for example. These services are mostly only of secondary interest, as we do not implement a serverless application ourselves in the scope of this thesis. The focus is much more on the one service that is able to monitor all the other services. This service is called CloudWatch.

CloudWatch monitors all of a user's AWS resources in real time, without the user having to configure or instrument anything beforehand. There are multiple ways to access CloudWatch: through the AWS CLI, the CloudWatch API, the web console or the AWS SDKs. The AWS **SDK for JavaScript** is the best way to connect SoLOMON to CloudWatch. It can easily be installed using the Node Package Manager (npm). In order to be able to actually connect to AWS using the SDK, the machine on which the code is running has to be authenticated. This can be done for example by having a **credentials** file, which contains the *AWS access key ID* and the *AWS secret access key*, in a specific location. When executing the application which uses the AWS SDK, it will check for the credentials file in the specific location and use the keys from it to authenticate the machine. In order to create these credentials a tool called *saml2aws* is used at Vector Informatik. This CLI tool allows its user to login to AWS and to retrieve temporary credentials using different identity providers. If valid credentials are present on the machine, using the AWS SDK works effortlessly. As the new major release of the AWS SDK for JavaScript, namely version 3, was released only in December 2020 [Kam21] and using it caused some problems in the beginning, we decided to use version 2, which was more stable and worked. Back to CloudWatch: Besides collecting and visualizing metrics of all used AWS services, it also offers a functionality which is crucial to our approach of implementing SoLOMON, namely alarms. **Alarms** can be set for any of the metrics that CloudWatch collects. They allow to take certain actions as soon as the threshold defined in the alarm for a specific metric is breached. The resemblance of the CloudWatch alarm definition with the structure of SLOs is the key here. This is what makes it possible to map SLOs unto CloudWatch alarms, and CloudWatch alarms back to SLOs. The AWS SDK for JavaScript in SoLOMON is thus mainly used to fetch, create, update and delete CloudWatch alarms. Furthermore, it is used for example to request the list of all existing Lambda functions on the AWS account. This allows the frontend of SoLOMON to offer the user a list of possible monitoring targets for which the SLO should be created.

Another crucial component of our solution is the AWS service called Simple Notification Service (SNS). **SNS** is a messaging service which allows both, application-to-person and application-to-application communication in a publish-subscribe manner. Especially the application-to-application communication aspect is important, as we need a way to let CloudWatch communicate the triggering of alerts to SoLOMON. In this regard SNS allows different options, for example fanout to HTTP or HTTPS endpoints, fanout to SQS and fanout to Lambda. The final solution for our project worked using the fanout to Lambda option. Section 5.4 contains a more detailed explanation of how SNS was used in the project.

Two other AWS services which are not part of our solution, but which helped with the deployment in the development and the evaluation process are Cloud Development Kit (CDK) and CloudFormation. **CDK** is a software framework that can be used to define the infrastructure of AWS cloud applications, meaning the different services, their connections and their configurations. After having created a description of the deployment in the manner of Infrastructure as code (IaC) using the CDK, the **CloudFormation** service can be used to deploy and provision the application. A more detailed description of the deployment and how the CDK was used for this project can be found in section 5.3.

5.7.4 Docker

Docker is a platform concerned with the development, shipping and running of applications and also follows the vision of describing the application infrastructure as code (IaC). The concept at the base of Docker are containers. **Containers** are lightweight and loosely isolated environments in which applications and all their dependencies are encapsulated [Doc21]. As a consequence, the application in the container is self-sufficient works just the same in any environment that can run Docker containers, regardless of operating systems, on a local machine or in the cloud [Doc21]. *Dockerizing* an application is thus a great step towards a good portability.

In order to dockerize SoLOMON, we needed to install Docker for Windows and to write a so called **Dockerfile**. The Dockerfile describes where the application files are located and contains the commands needed to install its dependencies and finally to build the application itself. Docker for Windows allows to locally build a Docker image from the Dockerfile. Once created, the Docker image can be shared and used on any platform running Docker, to instantiate a Docker container. Docker for Windows allows to do this directly, and can even run the Docker container on the local developer machine. The whole process of dockerizing SoLOMON is described in more detail in section 5.3.

5.7.5 Postman

Postman is an application that can act as an API client for REST, SOAP and GraphQL queries. Its intuitive user interface allows to quickly put together different queries, which can then be sent to a REST API for example. The result returned by the API to the client is also visualized in the application. The URL to which the request is sent can be written directly or can contain variables. The use of variables allows for example to quickly change the port or the host of the URLs of multiple requests using the variable. Similarly, variables can be used to define request parameters or the request body. Furthermore, Postman handles the use of authentication tokens for HTTPS requests.

Given all this, Postman was the perfect tool for testing the SoLOMON API during development. When modifying its REST API, we could quickly write a request testing the changes and checking whether the responses it returns are as expected. This meant that it was not necessary to also deploy the SoLOMON frontend during the development process, and that we could test the API endpoints specifically and with different payloads. This approach worked when running the SoLOMON backend on a HTTP server on the local developer machine, as well when having it deployed on AWS. In Fig. 5.1, the user interface of Postman can be seen for a GET request to the SoLOMON backend that fetches the existing SLOs.

We also used Postman to send GraphQL queries to the Gropius backend. This allowed us to easily test the GraphQL queries used in SoLOMON. Moreover, we could easily create new Gropius projects and components and we could check whether the creation of new issues by SoLOMON works correctly.

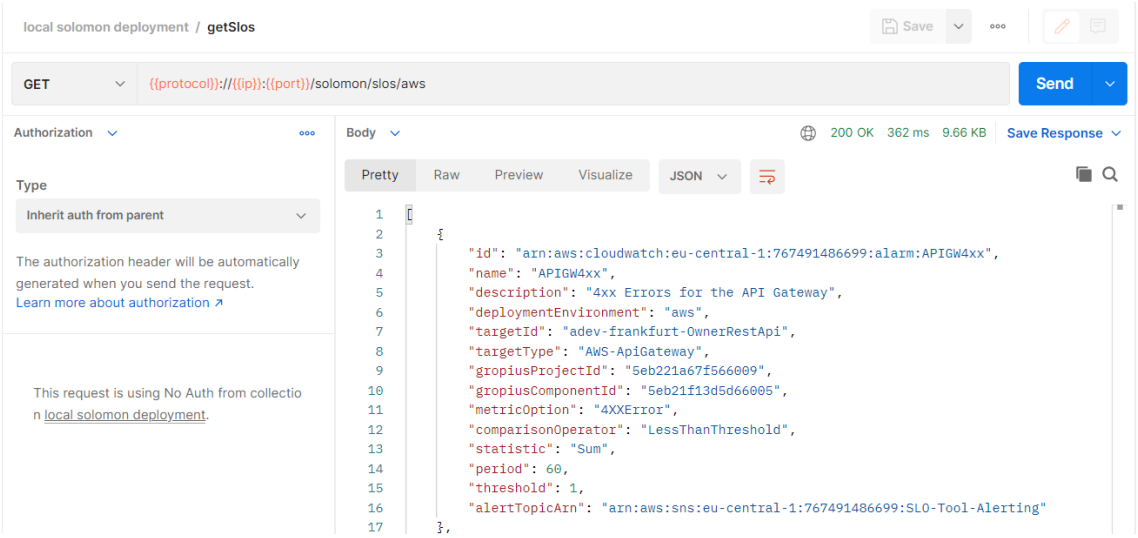

Figure 5.1: Request to SoLOMON backend for fetching existing SLOs in Postman.

5.7.6 Git

Git is a so called Version Control System (VCS). It is used mainly to store source code and documentation of software applications. Especially functionalities such as branching, and pull requests allow an efficient collaboration in the software development process.

As SoLOMON is an open source project, it was made openly accessible on the GitHub platform[2]. There, anyone can see its source code and even contribute to it by creating pull requests that add new features, expand the existing functionality or fix existing bugs. As the first prototype of SoLOMON was also already on GitHub, a new branch was created by us in which the total refactoring and restructuring of the backend of the application took place. Towards the end of the thesis project, the refactoring branch was merged back into the main branch. During the development process the collaboration aspect of GitHub also played a role: The developer of the initial SoLOMON prototype, Jonas W., was collaborating with us on the development of SoLOMON. More precisely, during the first half of the project we consulted with regards to different design and architecture choices, and during the second half he worked on the Kubernetes Module, in order for it to be compatible with the new backend design. Towards the end of the project, before the evaluation, Jonas W. also updated the frontend of SoLOMON in order for it to be able to use the new functionalities offered by the backend. GitHub allowed this collaboration process to work seamlessly.

[2] **https://github.com/ccims/solomon**.

6 Evaluation

In order to evaluate the approach of SoLOMON, and the merit of the idea behind it, we wanted to evaluate SoLOMON in a realistic industry environment. This is where Vector Informatik plays an important role.

6.1 Evaluation Context

Vector Informatik is a German company with more than 3000 employees that operates globally in the automotive industry. The motto of the company is "Simplifying the Development of Automotive Electronics" and hints at the fact that it supports suppliers and manufacturers who develop automotive electronics [Inf21a].

RDI is a department at Vector which aims to advance the development of dedicated projects. Its focus is on the conceptual and prototypical development of products and applications that may become important for Vector Informatik. One of the most successful products of Vector Informatik is called *CANoe*. This is a tool which can be used for the development, analysis and automated testing of electronic control units (ECUs) or entire networks of ECUs [Inf21c]. The Connectivity Feature Service (CFS) is a feature of *CANoe* which allows it to be connected to a cloud. This means that other participants, for example IoT applications, can be connected to *CANoe* via the internet [Inf21b]. To allow this feature to work, a cloud backend is needed, which is deployed in AWS.

In the period of this project, the team that was more or less working on the CFS backend counted six people. Most of them were not working full time on the CFS backend anymore, some of them were not working on it at all anymore by the end of the project. This all speaks of the fact that the CFS backend matured and was not in the prototype phase anymore. This means it was less and less in the domain of the department RDI, which also lead to the responsibility for it being slowly handed over to another department which is responsible for the product line development.

6.2 Goal Question Metric Approach

In this section we give a short conceptual overview of the Goal Question Metric approach in general and then take a look at it from a perspective specific to this thesis project. In the following, we describe the four metrics used in our evaluation.

6.2.1 Overview

The *Goal Question Metric* approach is a well known technique for evaluating the quality of specific processes and products that was published in 1994 by Basili, Caldiera and Rombach [CBR94]. The measurement model this technique proposes has three distinct levels:

1. **Conceptual level (Goal):** Goals refer to aims at the highest level of abstraction and can refer to different qualities of a product or a process from different points of view.
2. **Operational level (Question):** Questions are needed to assess the achievement of the goals or are raised by them.
3. **Quantitative level (Metric):** Metrics refer to ways of obtaining data that can answer the questions, ideally in a quantitative way.

An overview of all goals, questions and metrics for this project and how they are connected can be seen in 6.1.

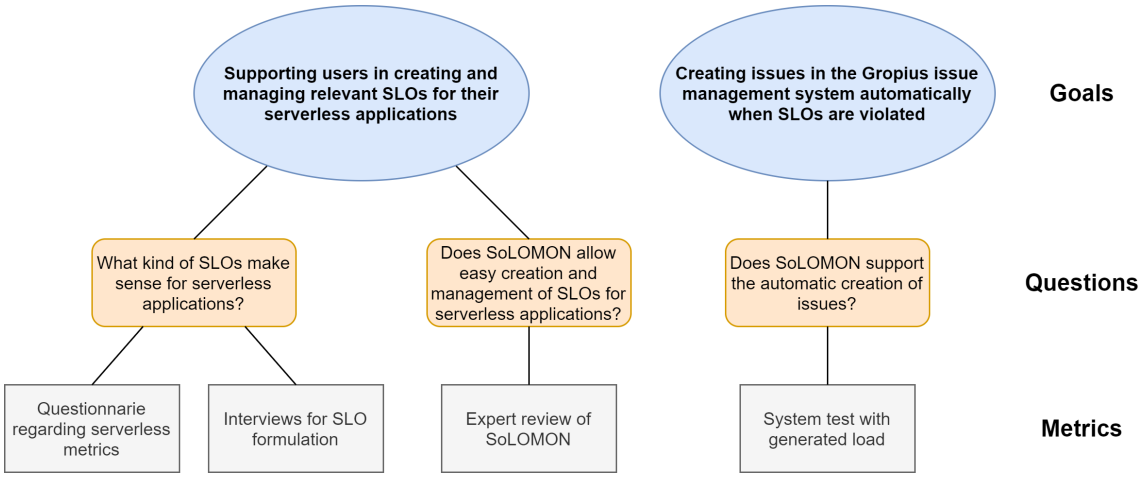

Figure 6.1: Goal Question Metric Plan.

Based on the general goal that we presented at the beginning of Chapter 3 (“The goal of SoLOMON is to allow the user to set up SLOs for serverless applications, and to make sure that eventual SLO violations do not go unnoticed.”), we specified two more explicitly formulated goals:

G1 Supporting users in creating and managing relevant SLOs for their serverless applications.

G2 Creating helpful issues in the Gropius issue management system automatically when SLOs are violated.

These goals raised the following three questions. The first two relate to G1 and the last one to G2:

Q1 What kind of SLOs make sense for serverless applications?

Q2 Does SoLOMON allow easy creation and management of SLOs for serverless applications?

Q3 Does SoLOMON support the automatic creation of issues?

In order to answer these questions we selected the following procedures and metrics. The first two are connected to Q1, the third to Q2 and the fourth to Q3:

M1 Questionnaire regarding serverless metrics

M2 Interviews for SLO formulation

M3 User test of SLO Tool and subsequent feedback

M4 System test

6.2.2 M1: Questionnaire regarding serverless metrics

The first metric regarding Q1 (What kind of SLOs make sense for serverless applications?), is a questionnaire regarding serverless metrics. The aim of this questionnaire was to limit the number of metrics available for the different serverless component types and, thus, to ease the process of formulating SLOs through a reduction of complexity. Of course the first and biggest limitation in this regard, is that we can choose only from the metrics that CloudWatch is offering to be monitored.

The questionnaire is made up of six major parts. The first one is general, regarding the types of serverless AWS components which are deemed relevant, in the context of monitoring, by the participant. Here the participants could select any amount of the five different components types (Lambda, API Gateway, Network Load Balancer, ECS, RDS) which we identified as the main building blocks of the CFS backend application. Additionally, the participants could add other component types which we might have omitted. Each of the following five major parts of the questionnaire are then concerned with one of the component types. In the first one, for example, the participant has to select from the 14 existing CloudWatch metrics for Lambda components, those which he deems most relevant. The 14 metrics for Lambda components are divided into three different sections (invocation metrics, performance metrics and concurrency metrics) and each of them also has a basic description of what the metric means. All of the metrics and their descriptions were taken from the official AWS documentation. At the end of each major part concerning one component type, we ask if there are any other metrics that would interest the participant with regards to the specific metric type. We end the questionnaire with a question were we want to know if the participant has anything else to say.

The sum of the metrics available for only these five different serverless component types is 67 (or even 79, if also counting some Network Load Balancer metrics in their more specific forms).

6.2.3 M2: Interviews for SLO formulation

The second metric connected to Q1 (What kind of SLOs make sense for serverless applications?) are interviews in which participants should formulate concrete SLOs. The first of these interviews was done with one of the members of the team responsible for the CFS backend application. This was a rather informal interview in a written form (through text chat). The second interview had the the product owner of the CFS backend application as participant. This second interview was prepared more thoroughly and the participant's input from the questionnaire M1 was used to guide it.

Similar to the questionnaire, the interview was structured around the different serverless component types. As the type of the outcome of the interview was predefined, namely SLOs, a structured interview made the most sense.

6.2.4 M3: Expert review

This metric was designed to answer the question about the usability and the functionality of SoLOMON (Q2). Although expert reviews are usually used primarily to evaluate usability and user experience [Har21], we wanted to focus more on functionality. The reason for this is that the user interface, which plays the biggest role regarding usability, was not developed by us as part of this project, but it was merely adapted slightly by the developer of the first SLO Tool prototype in order to support the functionality added by us. This means that we are acutely aware of the fact the user experience with SoLOMON is still lacking at the moment. An expert review is still purposeful, as it can reveal a lack of functionality or a need for change in the user interface, which might require other changes in the backend of the application. This can be useful for guiding future development of SoLOMON, on the backend part as well as on the frontend.

The basic idea of the expert review was to watch an expert use SoLOMON to create the SLOs from the list of SLOs we gathered as a result of Q1. This could already reveal problems in usability or a lack of functionality. Afterwards there would be an open discussion about the strengths and the weaknesses of the current implementation of SoLOMON.

6.2.5 M4: System test

After having answered the other questions, there remains one question regarding the functionality of automatic issue creation. Metric M4 is a system test in which load on the serverless application is simulated. This happens by using automated tests created for the CFS backend application. During the automated tests the metrics as provided by AWS for the different serverless components can be monitored using the CloudWatch web interface. A look at the metrics of the serverless application under load might also help with defining or refining the thresholds for the SLOs that are formulated for it.

This system test will thus test the interplay between all SoLOMON components (SoLOMON frontend, SoLOMON backend and SoLOMON notifier Lambda function) and their correct functionality. What will also be tested are the interfaces to external systems such as CloudWatch, SNS and Gropius. The SoLOMON frontend is used to create all the SLOs that were elicited by metric M2. Then the automated tests are used to put the CFS backend under load. The metrics for its components are observed, as well as the alarms. If an alarm gets triggered, a Gropius issue should be created. This Gropius issue should contain information about the SLO it is connected to and some more information about the component which caused it, and when the SLO violation took place. After the tests are run we check whether the expected issues were created. If no SLO is violated the tests or the components themselves can be modified temporarily in order to provoke errors or other performance problems.

6.3 Results

This section will present the results of the evaluation. Specifically this means that the answers to the questions Q1 - Q3 as obtained through metrics M1 - M4 will be presented.

6.3.1 Q1: Meaningful SLOs

What kind of SLOs make sense for serverless applications? This was the first question we asked regarding our goal (G1) of supporting users in creating and managing relevant SLOs for their serverless applications. A first approach to the question of what meaningful SLOs might be for a serverless application was already presented in section 3.2. There we tried to answer this question in a general way and based on literature and other examples such as the SRE workbook [BMR+18]. What become increasingly clear, especially after defining the interface for SLO objects in SoLOMON (see 4.4), is that defining SLOs is a task with a high complexity. This became obvious after the first interview which was part of M1. The participant struggled with defining any SLOs even though he had a relatively good understanding of the CFS backend application for which the SLOs should be defined. In the end the participant managed to formulate four concrete SLOs and indicated interest in other metrics too, but not being able to set a threshold for the metrics. These meager results for the first interview were in our opinion mostly caused by the lack of structure for the interview. Furthermore, this was the moment when we fully realized the complexity of formulating SLOs and the problems to which this leads. These are elaborated in more detail in the discussion in section 6.4.

In order to address the complexity of the CFS backend application for which the SLOs should be defined, we created an architectural diagram displaying an overview of its main components. In it all the different AWS component types or services are clearly distinguishable and the way the different instances of those are connected is also made obvious. In the interview we showed this diagram (Fig. 6.2) at the beginning of each of the sections concerning one specific component type.

Additionally, we reduced the number of metrics we are working with for each component. The reasoning behind this was to reduce the complexity at the level of metrics that exist for each different component types. In order to do this we used the questionnaire regarding serverless metrics (M1). From the total of six people in the team who are or were working on the CFS backend, and who were invited to participate in the questionnaire, a total of three persons participated. The first participant, who is also the product owner, selected 17 out of the 67 metrics available for the five different AWS services. The second participant selected 49 and the third a total of 36 metrics. If such as the latter two numbers of metrics would be supported, or we even would try to support the union of these three sets metrics, the task of defining new SLOs would remain highly complex and burdensome. The selection of metrics by the product owner is not only smaller, but also in general much closer to the metrics we ourselves also thought of as most relevant. This is why those 17 metrics selected by the product owner were also what we took as requirements for SoLOMON.

Having the results of the questionnaire from a specific participant allowed us to prepare the interview for this participant much better. For example in the second interview, which was the one with the product owner of the CFS backend application, this made a significant difference: Instead of going through a total of 79 metrics, this allowed us to focus on the 17 metrics that the participant already selected as relevant in the questionnaire. Additionally, the participant stated that he was interested

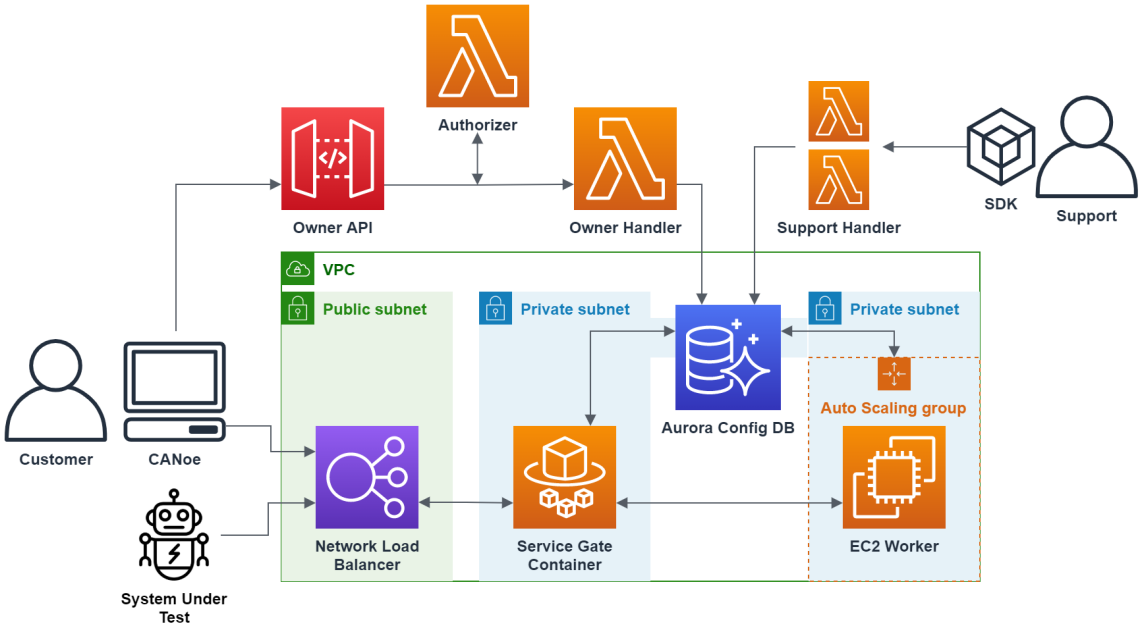

Figure 6.2: Architecture of the CFS using AWS services.

in setting SLOs per component type and not per instance type. This means for example that the SLO regarding the number of errors of a Lambda function should apply to all Lambda functions in the serverless application. A higher degree of granularity would be to formulate different thresholds for the SLOs on a per instance basis, but this would also once again lead to a higher complexity. In consequence the participant was able to formulate 17 potential SLOs. For a part of the 17 metrics the participant was able to formulate fully actionable SLOs containing all of the five metric related attributes: metric type, comparison operator, statistic, period and the threshold. For example the product owner said that the sum of errors on Lambda functions in the period of one minute should not exceed 1. This is an example of a fully actionable SLO that can be mapped to a CloudWatch alarm and is thus suited for testing SoLOMON. For some of the metrics the product owner showed interest purely in observing them but without defining a threshold up to which the service level is acceptable. He said for example that he would be interested in monitoring the sum of all invocations of lambda functions in the period of an hour. This information is helpful for defining monitoring views or dashboards in CloudWatch, but is not an actionable SLO description. For some other metrics the product owner said in the interview that he would have to know a baseline before being able to define a threshold for the SLO.

As a result we were able to capture a total of seven fully actionable SLOs that we could create in SoLOMON without any other information needed. Three of the SLOs, namely the two for Lambda Functions, and the second one for the API Gateway, are the same ones we extracted from the answers of the first interview. The table 6.1 gives an overview over all actionable SLOs captured from the two interviews.

Furthermore, we captured five SLOs in the second interview for which no threshold was defined yet. In order to define the threshold in the future, the application would have to be monitored over a period of time with normal activity, so that a baseline for these different metrics can be established. As for the SLOs that are not fully actionable, here there is also an overlap between the results from

Table 6.1: Fully actionable SLOs captured in the interviews.

Component Type	Metric Name	Stat.	Comp. Op.	Threshold	Period
Lambda Function	Errors	Sum	<	1 error	60s
Lambda Function	Throttles	Sum	<	1 error	3600s
API Gateway	4XX Errors	Sum	<	1 error	60s
API Gateway	5XX Errors	Sum	<	1 error	60s
Network Load Balancer	Healthy Host Count	Sum	>	0 hosts	60s
Network Load Balancer	Unhealthy Host Count	Sum	<	1 host	60s
Relational Database Service	Free Storage Space	Min.	>	20 percent	3600s

Table 6.2: SLOs captured in the interviews for which no threshold was defined yet.

Component Type	Metric Name	Stat.	Comp. Op.	Threshold	Period
Lambda Function	Duration	Max.	<	?	60s
Relational Database Service	CPU Utilization	Avg.	<	?	60s
Relational Database Service	DB Connections	Max.	<	?	60s
ECS Cluster	CPU Utilization	Avg.	<	?	60s
ECS Cluster	Memory Utilization	Avg.	<	?	60s

the two interviews: The participant of the first interview also showed interest in the potential SLO regarding the CPU utilization of the RDS, but did also not know what a sensible threshold would be. This one, and all other potential future SLOs which can be seen in table 6.2, were captured in the second interview with the product owner of the CFS backend application.

Other findings from the questionnaire are now going to be presented. To the question if there are any other AWS component types, beside the five already mentioned, that the participant would be interested in monitoring two participants responded. One mentioned the Elastic Kubernetes Service (EKS) and the other AWS ElastiCache. ElastiCache is already part of the CFS backend application, but was overlooked when creating the architectural overview of the serverless application and, thus, not included in the questionnaire. The product owner also mentioned two other AWS services which might be integrated into the CFS backend soon, and which might become relevant then. These services are called Cloud Map and Cognito. The first one is a cloud resource discovery service and the second one a user sign-up and authentication service. One of the participants of the survey wrote in the end that he or she would be interested in knowing how long the serverless application as a whole or just components of it have been unavailable to the user. Such a kind of downtime metric would be interesting, even though it can not be directly translated into an actionable SLO.

6.3.2 Q2: Easy creation and management of SLOs

Does SoLOMON allow the easy creation and management of SLOs for serverless applications? This was the second question we asked regarding our first goal (G1). In a way this question relates to many of the requirements we set up for this application, described in Chapter 4.

The setup for the expert review which we used in order to answer Q2 was the following: The participant was the product owner of the CFS backend application and can thus be qualified as a domain expert. He was given control over the screen where the SoLOMON frontend was opened, as well as the tables containing the results of M2 (see Tabular 6.1 and Tabular 6.2). He was then given the task to use the SoLOMON user interface to create some of the SLOs from the tables. In the meantime he was to comment on the things he did or things that were unclear. After having created some SLOs there was a final discussion with feedback.

The results of this expert review concern mostly the user interface design, but there also were some findings that have implications for the backend of the application. In order to offer some context to the feedback regarding the user interface of SoLOMON, we included a screenshot of its *Edit SLO page* (see Fig. 6.3). The screenshot depicts the user interface at the date of the 21.07.2021 when the expert review was conducted. It is possible that the user interface has already changed since then.

The following aspects were noted during the expert review with regards to the user interface design:

1. There should be additional information provided about what the attributes mean.
2. For all selectors, it should be possible to type in the beginning of a selection and it should filter the selection possibilities accordingly and on-the-go.
3. For the selectors, the selection possibilities should be sorted alphabetically.
4. The selection possibilities for the *Metric Option* should be filtered: only the metrics that are supported by the selected target type should be shown.
5. For the comparison operator, the signs (e.g. >) should be shown next to the name of the operators.
6. The unit of the threshold should be made clear.
7. For the period field, either the unit should be made clear, or even better, a parser should be integrated which parses entry with a unit (e.g. *m* for minute or *s* for seconds).
8. A cancel button should be added to cancel the editing activity.

Other than this, there were some other critical findings during the expert review. When the participant tried to load the targets for the target type *Network Load Balancer*, none could be loaded. This is an error we were already aware of before. At the moment it seems that this is caused by a limitation in the AWS JavaScript SDK for the ELB service, but it is to be further investigated. Another thing that the participant mentioned was related to the ECS target type. In the current implementation, SLOs can only be created at the ECS cluster level. The participant said that it would probably be more useful if SLOs could be created at the level of the different tasks running in a cluster. This would allow more precision in locating bottlenecks or faulty behavior.

On the whole, the participant was pleased with the current state of SoLOMON. He liked the response time of the interface and the fact that the existing target components of the CFS application were shown directly and could be selected in the user interface of SoLOMON. The result regarding goal G1, namely supporting users in creating and managing relevant SLOs for their serverless applications, is thus understood as mostly fulfilled. Merely the creation of the SLOs for the Network Load Balancer target type, was not possible.

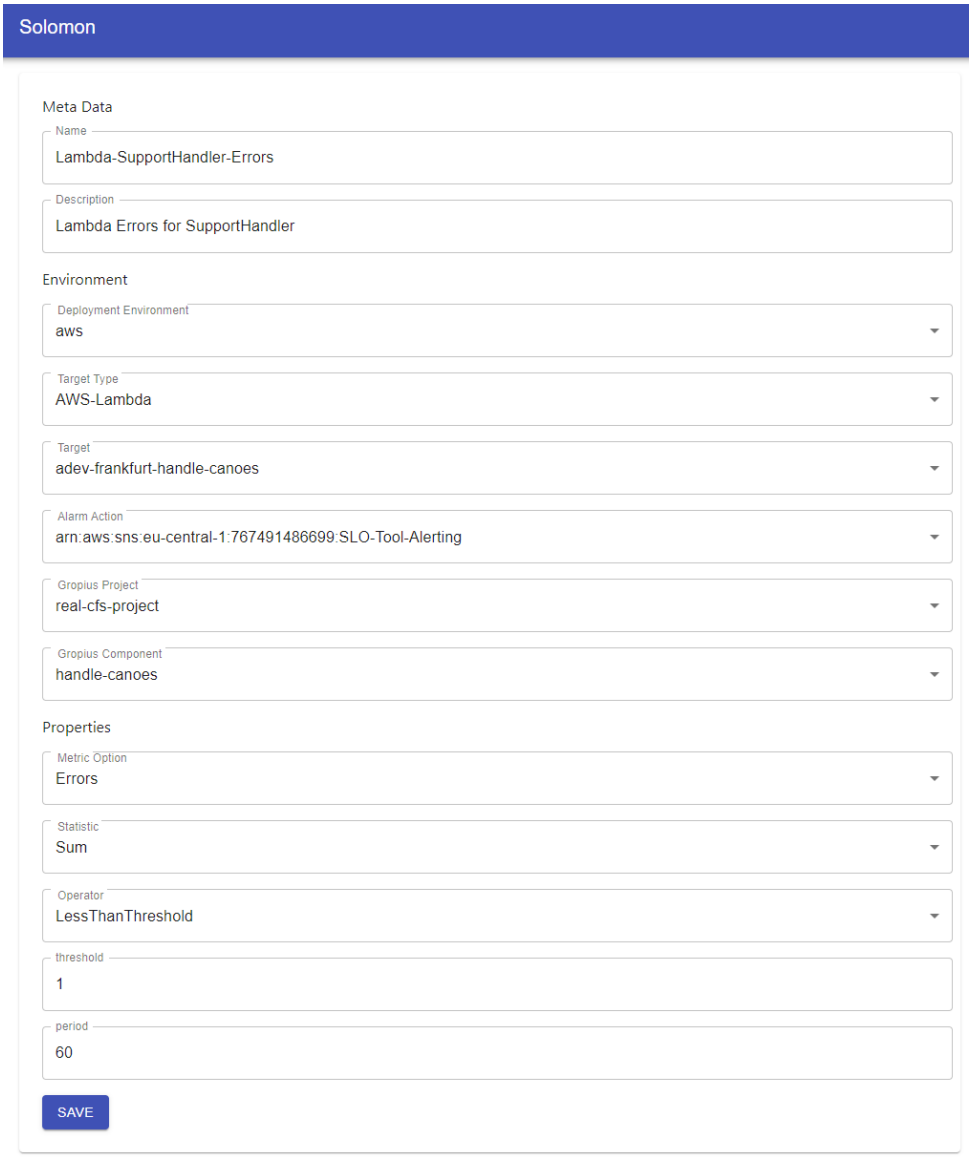

Figure 6.3: Screenshot of the prototypical user interface.

6.3.3 Q3: Automatic creation of issues

Does SoLOMON support the automatic creation of issues? This is the third and final question we asked for the *Goal Question Metric* approach. It is related to the goal G2, which is the creation of issues in the Gropius issue management system in case of SLO violation.

For this final test we created a total of 13 SLOs. We created all the SLOs from Table 6.1 except the ones for the Network Load Balancer, as it was not possible to fetch the Network Load Balancers using the API AWS provides for them. The Lambda-related SLOs were created for all four of the Lambda functions of the CFS backend application. Additionally, we also created two SLOs from Table 6.2, namely the ones for the ECS cluster. We set up a generic threshold of 85 % for both.

When we first triggered the automated tests for the CFS backend, we could observe that the different services were active by looking at the metrics in the CloudWatch web interface. Looking at the alarms in CloudWatch, we observed that none of them was triggered, and thus, we also could not expect new Gropius issues to be created. In a way this was not surprising, as we expected the system to perform within the threshold set by the SLOs. On the other hand the alarms all were in a state of insufficient data. This meant that there could also be a fault in the way the alarms were set up. Upon further inspection, and especially when comparing the alarms created directly in CloudWatch with the ones that get generated by SoLOMON from SLO descriptions, we noticed that there was a bug: SoLOMON passed the ARN of the components as the *targetId*, while the attribute in the CloudWatch alarm that the *targetId* gets converted to is normally just the name of the component. After fixing this bug, we started a second test round.

In this second test round, we could immediately see that the results from the first one were caused by the bug. Now instead of the CloudWatch alarms showing that there was insufficient metric data to evaluate for the Lambdas, the API Gateways and the RDS, they showed that they were in an *OK* state. We concluded from this that the we managed to fix the bug, and the CloudWatch alarms now get created correctly from the SLO. For ECS, CloudWatch still said that there was insufficient data which indicated that there is still some problem with these target types, which we also identified and located before, during the development process.

Name	State	Last state update	Conditions	Actions
RDSCPUUtil	OK	2021-07-23 12:16:15	CPUUtilization >= 85 for 1 datapoints within 1 minute	1 action(s) enabled
Lambda-SupportHandler2-Errors	OK	2021-07-23 12:09:53	Errors >= 1 for 1 datapoints within 1 minute	1 action(s) enabled
APIGW4xx	OK	2021-07-23 12:04:39	4XXError >= 1 for 1 datapoints within 1 minute	1 action(s) enabled
APIGW5xx	OK	2021-07-23 12:04:36	5XXError >= 1 for 1 datapoints within 1 minute	1 action(s) enabled
Lambda-Authorizer-Errors	OK	2021-07-23 12:03:42	Errors >= 1 for 1 datapoints within 1 minute	1 action(s) enabled
Lambda-SupportHandler-Throttles	OK	2021-07-23 12:03:24	Throttles >= 1 for 1 datapoints within 1 hour	1 action(s) enabled
Lambda-OwnerHandler-Throttles	OK	2021-07-23 12:03:21	Throttles >= 1 for 1 datapoints within 1 hour	1 action(s) enabled
Lambda-SupportHandler-Errors	OK	2021-07-23 12:03:13	Errors >= 1 for 1 datapoints within 1 minute	1 action(s) enabled
Lambda-OwnerHandler-Errors	OK	2021-07-23 12:03:12	Errors >= 1 for 1 datapoints within 1 minute	1 action(s) enabled
Lambda-Authorizer-Throttles	OK	2021-07-23 12:03:10	Throttles >= 1 for 1 datapoints within 1 hour	1 action(s) enabled
Lambda-SupportHandler2-Throttles	OK	2021-07-23 12:02:50	Throttles >= 1 for 1 datapoints within 1 hour	1 action(s) enabled
ECSMemUtil	Insufficient data	2021-07-22 11:24:55	MemoryUtilization >= 85 for 1 datapoints within 1 minute	1 action(s) enabled
ECSCPUUtil	Insufficient data	2021-07-21 14:47:02	CPUUtilization >= 85 for 1 datapoints within 1 minute	1 action(s) enabled

Figure 6.4: Screenshot of the CloudWatch dashboard showing the alarms which correspond to our SLOs.

Fig. 6.4 shows the alarms in the CloudWatch web interface. The 13 alarms are the ones which were created by the CloudWatch Connector in SoLOMON based on the SLOs we defined. As can be seen, the CFS backend application performs within the bounds defined by the SLOs that the product owner provided.

First we wanted to make sure the alerting functionality itself works. This is why we looked at another metric, namely the duration of the lambda functions. In the CloudWatch dashboard that we set up, we saw the measurements, which can also be seen in Fig. 6.5. The invocation widget shows

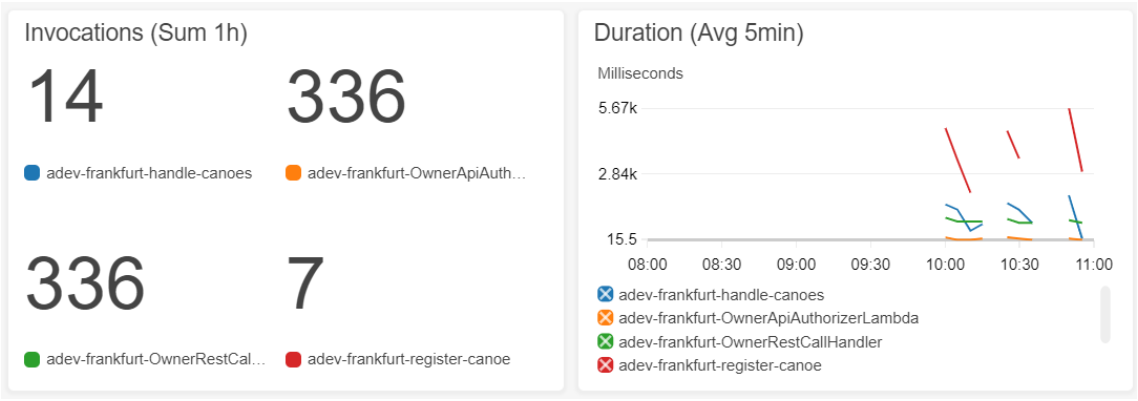

Figure 6.5: Screenshot of the CloudWatch dashboard showing some metrics for the Lambda functions of the CFS.

how often the Lambdas were invoked in the last hour. From this we can deduce that the automated tests, invoke the *OwnerApiAuthorizer* and the *OwnerRestCallHandler*, much more often than the other two Lambdas. Much more interesting though, is the duration widget on the right. Here we can see that the duration of the execution of the Lambda functions is on the average over five minutes between 5671 and 15 milliseconds. The *register-canoe* function is taking longer than the other ones. In order to test whether the alarms are triggered correctly, we decided to add four more SLOs which would target the duration of these Lambda functions. Setting the threshold for all four functions at 1000 milliseconds we would expect an alarm to be triggered only for the *register-canoe* and the *handle-canoes* Lambdas. After running the automated tests again, we could observe that the alarm corresponding to the SLO for the duration of the *register-canoe* Lambda was triggered, and now in the *In alarm* state. The *handle-canoes* Lambda remained in the *OK* state. As Fig. 6.6 shows, for the last execution of the tests before which we created the SLOs for the duration (at around 11:00), the duration of this lambda function actually remained under the defined threshold of 1000 milliseconds.

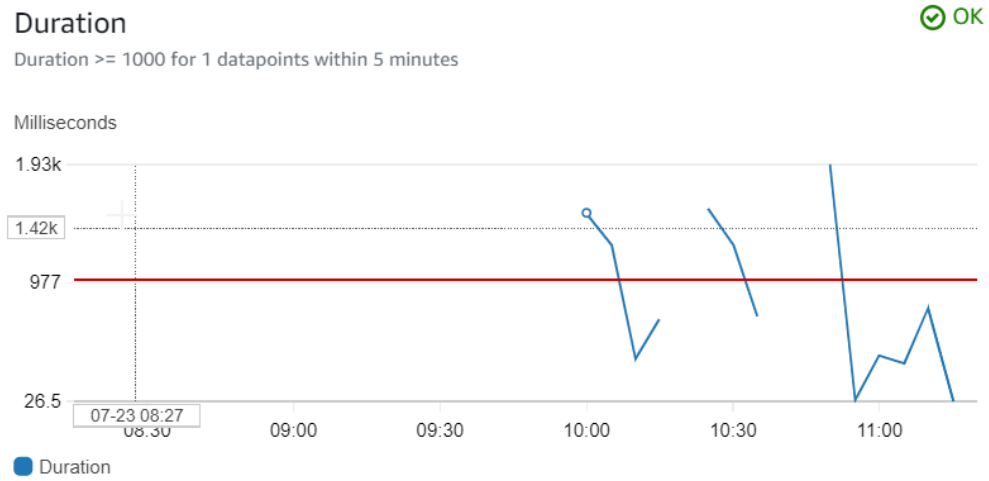

Figure 6.6: Graph for the duration metric of the handle-canoes Lambda function.

As we also wanted to test the functionality for other target types. This is why we lowered the threshold for the RDS SLO regarding the CPU utilization to 7.5%. The alarm got triggered again, which indicated that the alarms work for this component type too.

For the API Gateway it was a bit more difficult to trigger one of the created SLOs, as a 4XX or a 5XX error cannot easily be simulated, and is also not part of the automated tests. This is why, once again, we created a new SLO which was aimed at the *count* metric. This metric indicates the total number of API requests in a given period. Again, we chose a value which would be passed when running the automated tests, and again the alarm entered an alarm state soon after triggering the automated tests.

Having made sure that the alarms enter into an alarm state when their respective SLOs are violated, we wanted to check if the automatic issue creation worked. Because we did not install the Gropius frontend, we could not use a user interface to find that out. Instead we used Postman to send HTTP requests to the API provided by the Gropius backend. By sending the right query, we managed to receive all issues that were created in Gropius. As we did find the issues we expected to be created from the violations of our SLO, we concluded that the issues creation and the interplay between SoLOMON and Gropius worked as expected.

Listing 6.1 Issue created in Gropius for the SLO regarding the duration of the handle-canoes Lambda function.

```
{
    "title": "Lambda-SupportHandler-Duration-11:38:33-23/07/2021",
    "body": "Threshold Crossed: 1 out of the last 1 datapoints [1630.14 (23/07/21 11:33:00)]
            was greater than or equal to the threshold (1000.0)
            (minimum 1 datapoint for OK -> ALARM transition).",
    "components": {
        "nodes": [
            {
                "id": "5eb21edd18166003"
            }
        ]
    }
}
```

In Listing 6.1, one of the issue which was created for the SLO for the duration of the *handle-canoes* function can be seen. The title of the issue is the name of the SLO with the time when the issue was created added at the end. The body of the issue contains a more detailed description. Here we can see what the threshold and the comparison operator for the alarm was, at what point in time and with what value the threshold was crossed. In the components attribute, we can also see that the ID of the Gropius component to which the SLO was linked, is listed.

During this part of the evaluation a total of eight issues were created for Gropius. Here we also became aware of another potential issue: For one of the SLOs a total of three issues were created, all in the period of 41 minutes, and all related to the same component. This problem is sometimes called alert fatigue and will be discussed among other things in the next chapter.

Resuming this second part of the results, even though here too, there is still room for improvements with regards to the readability of issues and the mentioned alert fatigue, the goal G2 of creating issues in the Gropius issue management system automatically when SLOs are violated, was achieved fully.

6.4 Discussion

In this section we define the complexity problem and suggest possible solutions to approach it at its different dimensions. In the second part we discuss some problems related to the creation of issues and what is often called alert fatigue.

6.4.1 The Complexity Problem and Possible Solutions

In the first part of the previous section there was lot of talk about the complexity of different aspects which have to be considered when formulating SLOs. The combination of these aspects compound in such a way that in our opinion it can be called a real problem in this domain. We would state the complexity problem this way: "It is still very difficult for administrators to formulate SLOs, as this process involves a high degree of complexity with multiple dimensions." These are the dimensions of complexity:

1. **Complexity of application:** If the application is not of the smallest dimension, it can quickly become difficult to keep an overview of the serverless application. It is in the nature of serverless applications that they are more spread out and distributed than other applications.
2. **Complexity of component type metrics:** For each type of serverless component (e.g. FaaS, API Gateway, Load Balancer, etc.) different metrics are available. Although this makes sense, it adds to the complexity of the formulation of SLOs as these different types of metrics come with different units of measurement and different aggregation operators and comparison operators make sense for them. These have to be known and understood in order to be able to formulate meaningful SLOs for different component types.
3. **Complexity of threshold setting**: Setting a sensible threshold for a metric has also proven to be a challenge. Without knowing a baseline value, which can be approximated only after a period of observation.

Each of the three dimensions of complexity mentioned above can be tackled in a sensible manner only if the previous dimension has been taken care of sufficiently. In order get more thorough result with the second interview conducted as part of M2, we tried to tackle at least the first two dimensions of the complexity problem we just described. The previous section (see section 6.3) describes this in more detail.

The question how this problem can be tackled in general or maybe even in the context of SoLOMON remains. But there might be some things from the approach of the evaluation that can be integrated into SoLOMON. Regarding the first dimension of complexity concerned with the complexity of the application which is to be monitored, we came to the following conclusion: The results from the evaluation indicated that the architectural representation of the system aided the user who had to define the SLOs for the system, as it allowed a quick overview over all the components and their

connections. The idea that this lead us to, is that it would be ideal if the user could create the SLOs directly in a graphical user interface which consists of an architectural representation of the system that is to be monitored. This graphical user interface could show a diagram similar to the one created by us for the CFS (see Fig. 6.2). Additionally, it could offer two buttons on each of the components: One for listing the SLOs already modelled for this component in a new pop-up window, and one for adding a new SLO for the component. When adding a new component a pop-up window could open which could look similar to the page currently used in the SoLOMON frontend to add SLOs. But, an advantage could be that a lot of the attributes would not have to be filled out by the user, and could thus be removed from the view, and reducing the complexity of the process of SLO definition. The attributes that could be removed because they would be implicit when clicking on a certain component in the diagram to create the SLO are the deployment environment, the target type, the target ID, the Gropius project and the Gropius component ID. Additionally, the alarm action, which is needed only for AWS applications could also be preset in the project metadata. Six attributes less that the user would have to define would mean a massive improvement regarding the usability of SoLOMON and would greatly reduce the cognitive load on the user defining SLOs with it. In our opinion this would be a great first step in addressing the complexity problem surrounding SLO creation.

Addressing the second dimension of the complexity problem, namely the one regarding the component type metrics, might be a bit easier. The first step with this regard was found out through the feedback in the expert review M3 (related to Q2) described in subsection 6.3.2. One of the findings of this session was that the selection possibilities for the metric option attribute, which describes the type of metric, should be filtered according to what the component type of the target is. This means that when creating an SLO for a Lambda function, only the metrics that can meaningfully be applied to Lambda functions should be showed in the selector in the user interface. With this relatively small improvement in the implementation, once again we could reduce the complexity of the SLO creation.

Finally, there is the third dimension: the setting of meaningful thresholds for the SLOs. As it became most clear during the system test M4 (related to Q3), it certainly can be useful to look at the monitoring data before creating SLOs. As described in detail in section 6.3.3, we were able to define thresholds for metrics, for which we did not know what a baseline performance would be before. Creating dashboards that contain graphs and visualize the metric data is certainly a good step in the right direction. In the case of the evaluation, the baseline was defined using the load put on the system by automated tests. Of course it is more than questionable that this load corresponds to the load that the system might experience under production conditions. But one big advantage that SoLOMON offers is that SLOs can be edited with ease. This allows a simple and quick fine-tuning of thresholds. So in a way this last dimension of the complexity problem seems to be the least problematic one. A very advanced feature that SoLOMON might offer sometime in the future and which could help with the setting of thresholds might be the following: When creating or editing an SLO, after having selected the target ID and the metric option, SoLOMON could fetch some historical metric data from the past for this specific component and metric. This would help the user in establishing a baseline for the metric. The CloudWatch API would certainly allow for such a functionality, but the implementation on the SoLOMON side would probably require some effort.

6.4.2 Issues and Alert Fatigue

One problem that we became aware of during the last part of the evaluation process M4, is that there were multiple issues created for the same SLO during a relatively short amount of time (see subsection 6.3.3). Upon closer inspection of the metrics we realized that this is not directly related to what is called *alert fatigue*, even though we initially thought so. Alert fatigue, in the domain of incident management, is "when an overwhelming number of alerts desensitizes the people tasked with responding to them, leading to missed or ignored alerts or delayed responses." [Ltd21]. The way alerting is implemented in CloudWatch, this is already counteracted to a certain degree. For the CloudWatch alarms an alert gets triggered only for a state change. This means that it is not the case that alerts continue to get sent when the metrics for a component remain beyond the alarm threshold after crossing it. An alert gets sent only in the moment the alarm changes the state from *OK* to *In Alarm* (although other settings are possible too). This implementation has a positive impact on the behavior of SoLOMON too, because the SoLOMONs issue creation for applications running on AWS relies on the CloudWatch alerting system. The fact that multiple Gropius issues were created for the same SLO in a short time span can be explained by what is called *state flapping*. The Nagios Core documentation defines state flapping as what happens "when a service or host changes state too frequently, resulting in a storm of problem and recovery notifications."[LLC21]. This seems to have been what happened during our evaluation too: In the times between the activation of the automated tests, the alarms could enter an *OK* state, before being brought back into the alarm state again by the load created by the tests. A perfect visualization of this can be seen in Fig. 6.6, from the previous section. The duration of the Lambda can be seen to be at times above and at times below the 1000 millisecond threshold in a relatively short amount of time. Of course this was only possible because we set the threshold so low that it would be crossed with the load of the automated tests on the components. This is why flapping can also be an indicator of a threshold that is set too low (or too high, depending on the metric).

A more advanced technique to counteract flapping that is sometimes used is called *flapping detection*. This is a mechanism which allows to detect state flapping and which as a consequence avoids the alert fatigue to which flapping often leads. The implementation of such a mechanism in either SoLOMONs Alert Handler or in Gropius might be worth a consideration in the future. Otherwise, the problem of alert fatigue might be transposed to what we could call *issue fatigue*, where many issues are created for the same root cause, an the issues thus lose their meaning. One possible approach to handle this would be to update an SLO issue, if it already exists, instead of creating a new one for each time the threshold for the metric of a specific component is crossed.

6.5 Threats to Validity

In our evaluation approach, we are aware of some threats to **internal validity** with regards to goal G1 (Supporting users in creating and managing relevant SLOs for their serverless applications). The first has to do with the instrumentation. The interviews (M2) for the SLO formulation were conducted differently. While the first one was an informal interview conducted through text chat, the second one was more formal and conducted via video call. This change in instrumentation might have affected the results. We still think that the impact of this is low, as the results from the first interview overlap in great part with the results from the second interview.

Regarding instrumentation, we concede that in the process of eliciting the relevant SLOs through means of M1 and M2, we limited the options of the participants from the beginning in two regards. First of all, we selected the different target types that would be available to chose from in the questionnaire M1. Of course this selection was mostly on the types of components that make up the CFS backend application to our knowledge. Being aware of this limitation we imposed, we also put a question in the questionnaire where we asked the participants about other component types they would be interested in, in the context of monitoring serverless applications. As the responses to this question showed, there was at least one component type (AWS ElastiCache) that already was part of the CFS backend application and which was not considered by us neither in the evaluation nor in the implementation of SoLOMON. With regards to the different metrics available for each target type, we again let the participants of the questionnaire M1 select only from the metrics that CloudWatch offers for the respective component type. This was a limitation that we willingly accepted for practical reasons. We were aware of this limitation too, and thus, added questions asking about additional metrics for each component type to the questionnaire. There was only a small number of additional metrics that the participants proposed, but we could not implement them in SoLOMON because of the limitations of the CloudWatch API. This might be seen as a threat to the validity of our claimed achievement of goal G1.

An additional general threat to **external validity** is the size of the sample population for our questionnaire, the interviews and the expert review. We concede that the sample populations we had to work with are small. For the questionnaire (M1) we had three participants, for the interviews (M2) two, and the expert review (M3) was conducted with only one participant. On the other hand, the results of these metrics are meant to be generalized primarily for the evaluation context, namely for the team at Vector Informatik that works on the CFS backend application, which consists of six people.

7 Conclusion

In this final chapter we want to conclude the thesis by providing a short summary and pointing out the benefits, as well as the limitations of the solution implemented by us. Furthermore, some of the lessons learned during the project will be presented. The last section describes some possibilities for future work in the domain of thesis and with regards to SoLOMON specifically.

7.1 Summary

Monitoring serverless applications based on SLOs is a problem that existing monitoring solutions do not solve sufficiently. SoLOMON offers this functionality, at the current state, for serverless applications running on AWS or in a Kubernetes cluster. Additionally, it is embedded in the Gropius ecosystem, which allows the automatic issue creation in Gropius as soon as an SLO is violated. During the evaluation of SoLOMON, which took place in the context of the thesis, we noticed that users had certain difficulties related to the formulation of SLOs. We summarized them in what we called the *complexity problem*: "It is still very difficult for administrators to formulate SLOs, as this process involves a high degree of complexity with multiple dimensions." These dimensions are the complexity of the application, the complexity of component type metrics and the complexity of threshold setting. We discussed some ways in which the complexity at the different dimensions can be tackled, and how this knowledge can be taken into account in the future development of SoLOMON. During the evaluation we also became aware of another problem related to the creation of issues. More specifically, it has to do with the creation of multiple issues for the same SLO in a relatively small time span. This could be called *issue fatigue*. We discussed this and finally proposed a solution to counteract it.

7.2 Benefits

SoLOMON is a tool which can benefit administrators of serverless applications. Together with the insights provided in this thesis, it can help them to elicit SLOs and monitor them. What this enables, is the building of trust in the performance of the application. This confidence, in turn, allows the application owner to guarantee its users a certain Quality of Service (QoS). QoS guarantees can be very important businesswise, and especially with regards to the marketing of the application.

The alarm functionality that the cloud provider monitoring services already offer is in many regards similar to the SLO monitoring SoLOMON implements. The similarities are that both, SLOs and alarms, are related to specific metrics of specific components and that they are defined by a threshold. The difference is that SLOs are objectives, and thus, they describe the range of acceptable measurements, while alarms describe the range of inacceptable measurements. While the perspective

is different in these two, they can easily be derived from each other by inverting the comparison operator. This is also one of the reasons why the SLO monitoring of serverless components in SoLOMON builds upon the alarm functionality of AWS CloudWatch, for applications running in an AWS environment. Technically, there is not much of a difference between the concepts, but SLOs offer a different way to think about QoS. This is why some argue that SLOs are the key to making data-driven business decisions about reliability [TFHB18].

Another advantage of SoLOMON over the use of only the cloud provider monitoring services is that multiple (at the moment two) environments are supported. The way the architecture of SoLOMON was conceived and the way it was implemented allows for an easy extension of it to support even more environments. Some applications use services from different cloud providers or may even be deployed in a hybrid cloud fashion. In such cases, SoLOMON offers a single user interface in which all SLOs for all components of an application can be accessed, even if the components are deployed in different environments. It is even possible to monitor multiple different applications with a single SoLOMON instance. This helps application administrators to keep an overview over the SLOs even for complex applications.

The main contribution of our work is the development of an open source tool, namely SoLOMON, which allows to connect the useful concept of SLOs directly with the actual monitoring of serverless components. This is done by using the API of the cloud provider monitoring service of the environment, like CloudWatch for AWS, which means that no monitoring tool has to be configured in order to start the monitoring of the SLOs for the application components. While other paid monitoring tools are also starting to offer SLO tracking capabilities, SoLOMON is a free and open source solution which integrates with the Gropius issue management system and does not rely on third party monitoring tools.

The second main contribution of our work is the discussion of what we called the *complexity problem* with regards to the formulation of SLOs. This is something that we have not been able to find anywhere in the existing literature related to SLOs, even though it seems to us that it is important to be considered when working with SLOs.

7.3 Limitations

There are some limitations that we acknowledge in our work. First of all, SoLOMON currently supports only AWS as cloud environment for serverless applications. Kubernetes clusters are also supported to some extent, but this was not part of our work. We have a high degree of certainty that the design of SoLOMON allows for a relatively easy development of connectors for other cloud environments, such as Microsoft Azure and Google Cloud. We still have to admit that as long as these connectors are not implemented and tested, we cannot guarantee that it will actually work as envisioned by us.

As we decided for a non-intrusive monitoring approach, using the metrics that CloudWatch provides for components in the AWS environment, we had a strong bias towards metrics directly offered by CloudWatch. This means that there might be metrics that could possibly be important, maybe even in the context of the evaluation with the CFS backend, that were left out because of our focus on existing metrics. The questionnaire results from the evaluation showed that there was some interest in additional metrics among the participants. Even though CloudWatch allows so called *custom*

metrics, which we might have been able to configure in such a way as to allow the monitoring of the additional metrics asked for, we did not pursue this in the context of this work. We also did not have enough time to test SoLOMON with log-based metrics.

The number of component types that are supported for AWS currently is also limited. In this work we were focused on the component types that are most relevant in the CFS backend. This lead to the implementation of the CloudWatch connector in such a way that five different AWS component types should be supported. These are Lambda functions, API Gateways, Network Load Balancers, RDS components and ECS clusters. During the evaluation, we found out that the AWS API of the Network Load Balancer service does not work as expected, which leads to SoLOMON not being able to create SLOs for this component type. Another component type which is often used in serverless applications, but which was not used in the CFS backend, is the function orchestrator, or what is called *Step Function* for AWS and *Durable Function* for Microsoft Azure. This component type is not yet supported even though it might be important for many serverless applications.

7.4 Lessons Learned

There are things we learned at different levels during the development of SoLOMON and the work on this thesis.

Something we learned on a domain-specific level is that cloud environments often impose certain limitations, at times almost to the extent of what is called a *vendor lock-in*. One sort of limitation is, for example, that you have to use more of their services in order to use another service fully. In the context of AWS this could be seen most clearly in the alerting scenario for CloudWatch alarms. It is not possible to be notified directly by the CloudWatch service, but one has to use the Simple Notification Service (SNS) if one wants be informed about alerts. From SNS, theoretically, it is possible to have topic subscribers that get informed via HTTP/S, but in our case this did not work, and we had no other choice but to use another AWS service, namely a Lambda function. More generally, for monitoring serverless cloud components without having to instrument them, users have to use the monitoring tool offered by the cloud provider, in the case of AWS CloudWatch. These things show just how the cloud service providers are trying to make their users use more and more of their services and make it more difficult to use them in combination with external tools or services.

On a more general level, we found out that the collaboration in the development of software can sometimes be quite challenging. This seems to be the case especially when there exists no hierarchy between the involved parties. For SoLOMON, this became obvious especially in the first half of the development process when there were many important decisions to be made with regards to the architecture of SoLOMON or the design of the backend API. Most of the time, when there were differing opinions between the us and the other developer who developed the initial prototype of SoLOMON, they could be solved by discussing different arguments for and against the possible options. Sometimes, even after arguing, no conclusion could be reached and it was necessary to appeal to the supervisor and ask him about his opinion on the matter.

7.5 Future Work

We want to summarize our suggestions for future research and the future development of SoLOMON.

With regards to the *complexity problem*, there is still great potential for future research. It might be interesting to find out whether there are even more dimensions of complexity than the three presented by us. This could be found out by conducting more field studies in different industry contexts where SLOs are used. Furthermore, there is more research potential with regards to the solutions for the complexity problem. What we discussed in section 6.4 are some possible approaches to tackle the complexity at each of the dimensions. While they might make a good starting point for future research, they are on their own too specific for the context of the SoLOMON application. This is why further research is necessary regarding this topic.

With regards to the future development of SoLOMON, there are many suggestions we are able to make. First of all, more component types for serverless applications could be supported. We already mentioned the function orchestrator called *Step Function* in AWS. Other component types of interest that were mentioned by the participants of our questionnaire are the Elastic Kubernetes Service (EKS), ElastiCache, Cloud Map, and Cognito.

Thinking one step further, we would also want more different cloud environments to be supported. This would be a great step towards an SLO-based monitoring approach for heterogeneous cloud applications. In our opinion, the focus for new cloud environments that should be supported are the two other big cloud providers with the greatest market share, Microsoft Azure and the Google Cloud Platform.

Another thing that should be improved is the issue creation. As already mentioned, neither Gropius nor SoLOMON are currently able to handle what we have called *issue fatigue*. It would make sense to add the flapping detection functionality to either SoLOMONs Alert Handler or to Gropius, in order to prevent the creation of too many issues related to the same SLO in a short span of time. One approach, proposed by us, to handle state flapping, is to merely update an existing issue instead of creating new ones for each state change.

Another improvement of SoLOMON that was already requested, is that applications other than Gropius should also be able to be notified about SLO violations. Currently the issue creation in Gropius is hardcoded into SoLOMON. In the future it might be possible to change this in the user interface of SoLOMON and to allow the user to enter any URL to which the alert about a violated SLO should be sent to.

Finally, we want to talk about some future developments of the user interface of SoLOMON. In our opinion, this might be one of the most important areas in which SoLOMON must improve, in order to find a greater adoption and make it stand out event more from the pool of charged monitoring tools which offer SLO-based monitoring. One big advantage that SoLOMON could make use of is its connection to Gropius. Gropius in turn offers its users the possibility to model application architectures in a graphical notation [SBB20]. A graphical notation of the application that is to be monitored is an adequate approach to tackle the first dimension of the complexity problem, namely the complexity of the application. What we thus suggest, is that SoLOMON should be able not only to load the Gropius projects and components as it currently does, but to also load the graphical notation of the architecture of a project. This would allow the users of SoLOMON to get a

much better overview of the application for which SLOs are to be created. It could be implemented in such a way that the users can interact with the graphical representation of the application in order to create SLOs for the different components. We describe this possibility in more detail in subsection 6.4.1.

To address the complexity of threshold setting, the third dimension of the complexity problem, we suggest the following improvement of SoLOMON: When creating or editing SLOs, SoLOMON should be able to fetch historical monitoring data for the component and metric that were selected. The historical monitoring data could easily be fetched from the cloud provider monitoring service and would support the users greatly in defining meaningful thresholds for the SLOs, by providing them with a baseline performance of the components.

In conclusion we want to say that, in our opinion, further research on the complexity problem and then addressing it in the implementation of SoLOMON, is crucial for the future success of SoLOMON.

Bibliography

[Ama21a] Amazon. *Amazon CloudWatch*. 2021 (accessed 24.08.2021). URL: https://aws.amazon.com/cloudwatch/ (cit. on p. 12).

[Ama21b] Amazon. *Amazon CloudWatch FAQs*. 2021 (accessed 11.05.2021). URL: https://aws.amazon.com/cloudwatch/faqs/#:~:text=Extended%20retention%20of%20metrics%20was,are%20available%20for%203%20hours. (cit. on p. 24).

[Ama21c] Amazon. *AWS IP address ranges*. 2021 (accessed 30.06.2021). URL: https://docs.aws.amazon.com/general/latest/gr/aws-ip-ranges.html (cit. on p. 45).

[Ama21d] Amazon. *AWS Service Level Agreements (SLAs)*. 2021 (accessed 22.08.2021). URL: https://aws.amazon.com/legal/service-level-agreements/ (cit. on p. 1).

[Ama21e] Amazon. *Global Infrastructure*. 2021 (accessed 15.06.2021). URL: https://aws.amazon.com/about-aws/global-infrastructure/ (cit. on p. 51).

[Ama21f] Amazon. *Object Storage Classes - Amazon S3*. 2021 (accessed 20.04.2021). URL: https://aws.amazon.com/s3/storage-classes/ (cit. on p. 17).

[BBN07] F. Bachmann, L. Bass, R. Nord. *Modifiability tactics*. Tech. rep. CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2007 (cit. on pp. 39, 40).

[BCC+17] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, et al. "Serverless computing: Current trends and open problems". In: *Research Advances in Cloud Computing*. Springer, 2017, pp. 1–20 (cit. on p. 5).

[BMR+18] B. Beyer, N. R. Murphy, D. K. Rensin, K. Kawahara, S. Thorne. *The site reliability workbook: practical ways to implement SRE*. O'Reilly Media, Inc., 2018. URL: https://sre.google/workbook/table-of-contents/ (cit. on p. 59).

[CBR94] G. Caldiera, V. R. Basili, H. D. Rombach. "The goal question metric approach". In: *Encyclopedia of software engineering* (1994), pp. 528–532 (cit. on p. 56).

[CFAI17] I. Cassar, A. Francalanza, L. Aceto, A. Ingólfsdóttir. "A survey of runtime monitoring instrumentation techniques". In: *arXiv preprint arXiv:1708.07229* (2017) (cit. on p. 6).

[Coc98] A. Cockburn. "Basic use case template". In: *Humans and Technology, Technical Report* 96 (1998) (cit. on p. 28).

[Doc21] Docker. *Docker overview*. 2021 (accessed 16.06.2021). URL: https://docs.docker.com/get-started/overview/ (cit. on p. 53).

[Dyn21] Dynatrace. *Dynatrace Help*. 2021 (accessed 17.01.2021). URL: https://www.dynatrace.com/support/help/ (cit. on p. 12).

[Edu21] I. C. Education. *Application Performance Management (APM)*. 2019 (accessed 23.08.2021). URL: https://www.ibm.com/cloud/learn/application-performance-management (cit. on p. 6).

[Eps21] Epsagon. *Epsagon Documentation*. 2020 (accessed 17.01.2021). URL: https://docs.epsagon.com/ (cit. on p. 12).

[FIMS17] G. C. Fox, V. Ishakian, V. Muthusamy, A. Slominski. "Status of serverless computing and function-as-a-service (faas) in industry and research". In: *arXiv preprint arXiv:1708.08028* (2017) (cit. on p. 6).

[Fou21a] O. Foundation. *About Node.js*. 2021 (accessed 11.08.2021). URL: https://nodejs.org/en/about/ (cit. on p. 50).

[Fou21b] O. Foundation. *Introduction to Node.js*. 2021 (accessed 11.08.2021). URL: https://nodejs.dev/learn (cit. on p. 50).

[GHJV93] E. Gamma, R. Helm, R. Johnson, J. Vlissides. "Design patterns: Abstraction and reuse of object-oriented design". In: *European Conference on Object-Oriented Programming*. Springer. 1993, pp. 406–431 (cit. on p. 40).

[Gra21] Grafana. *Synthetic Monitoring*. 2021 (accessed 16.01.2021). URL: https://grafana.com/docs/grafana-cloud/synthetic-monitoring/ (cit. on p. 7).

[Har21] A. Harley. *UX Expert Reviews*. 2018 (accessed 19.07.2021). URL: https://www.nngroup.com/articles/ux-expert-reviews/ (cit. on p. 58).

[IBM21] IBM. *rSLA: Managing SLAs and Quality-of-Service in Cloud Environments*. 2017 (accessed 17.01.2021). URL: https://researcher.watson.ibm.com/researcher/view_group.php?id=9572 (cit. on p. 13).

[Inf21a] V. Informatik. *About Vector*. 2021 (accessed 01.07.2021). URL: https://www.vector.com/int/en/company/about-vector/ (cit. on p. 55).

[Inf21b] V. Informatik. *Connectivity Features Service for IoT*. 2021 (accessed 01.07.2021). URL: https://www.vector.com/int/en/products/products-a-z/software/canoe/for-iot-applications/ (cit. on pp. 3, 55).

[Inf21c] V. Informatik. *Testing ECUs and Networks with CANoe*. 2021 (accessed 17.01.2021). URL: https://www.vector.com/int/en/products/products-a-z/software/canoe/ (cit. on p. 55).

[Kam21] T. Kamat. *Modular AWS SDK for JavaScript is now generally available*. 2020 (accessed 16.06.2021). URL: https://aws.amazon.com/blogs/developer/modular-aws-sdk-for-javascript-is-now-generally-available/ (cit. on p. 52).

[KL+12] Y. Kouki, T. Ledoux, et al. "CSLA: A Language for Improving Cloud SLA Management." In: *CLOSER*. 2012, pp. 586–591 (cit. on p. 11).

[KL03] A. Keller, H. Ludwig. "The WSLA framework: Specifying and monitoring service level agreements for web services". In: *Journal of Network and Systems Management* 11.1 (2003), pp. 57–81 (cit. on pp. 8, 11).

[Kru95] P. B. Kruchten. "The 4 + 1 view model of architecture". In: *IEEE software* 12.6 (1995), pp. 42–50 (cit. on pp. 25, 29, 30).

[LLC21] N. E. LLC. *Detection and handling of state flapping*. 2021 (accessed 24.07.2021). URL: https://www.assets.nagios.com/downloads/nagioscore/docs/nagioscore/3/en/flapping.html (cit. on p. 69).

[LMG+17] T. Labidi, A. Mtibaa, W. Gaaloul, S. Tata, F. Gargouri. "Cloud SLA modeling and monitoring". In: *2017 IEEE International Conference on Services Computing (SCC)*. IEEE. 2017, pp. 338–345 (cit. on pp. 8, 11).

[LSM+15] H. Ludwig, K. Stamou, M. Mohamed, N. Mandagere, B. Langston, G. Alatorre, H. Nakamura, O. Anya, A. Keller. "rSLA: Monitoring SLAs in dynamic service environments". In: *International Conference on Service-Oriented Computing*. Springer. 2015, pp. 139–153 (cit. on pp. 8, 9, 11, 13).

[Ltd21] A. P. Ltd. *Understanding and fighting alert fatigue*. 2021 (accessed 24.07.2021). URL: https://www.atlassian.com/incident-management/on-call/alert-fatigue (cit. on p. 69).

[MAS+16] M. Mohamed, O. Anya, T. Sakairi, S. Tata, N. Mandagere, H. Ludwig. "The rSLA framework: Monitoring and enforcement of service level agreements for cloud services". In: *2016 IEEE International Conference on Services Computing (SCC)*. IEEE. 2016, pp. 625–632 (cit. on p. 13).

[Mic21] Microsoft. *SLA summary for Azure services*. 2021 (accessed 22.08.2021). URL: https://azure.microsoft.com/en-us/support/legal/sla/summary/ (cit. on p. 1).

[Mor16] K. Morris. *Infrastructure as code: managing servers in the cloud*. Ö'Reilly Media, Inc.", 2016 (cit. on p. 43).

[Nes21] NestJS. *Introduction*. 2021 (accessed 15.06.2021). URL: https://docs.nestjs.com/ (cit. on p. 51).

[Nor04] L. Northrop. *Achieving product qualities through software architecture practices*. Tech. rep. CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2004 (cit. on p. 39).

[Nov21] Novatec. *OpenAPM Landscape*. 2021 (accessed 24.08.2021. URL: https://openapm.io/landscape (cit. on p. 19).

[Pro21a] Prometheus. *Exporters and Integrations*. 2021 (accessed 07.05.2021). URL: https://prometheus.io/docs/instrumenting/exporters/ (cit. on p. 21).

[Pro21b] Prometheus. *Querying Prometheus*. 2021 (accessed 24.08.2021). URL: https://prometheus.io/docs/prometheus/latest/querying/basics/ (cit. on p. 10).

[Qua21] C. Quach. *Setting SLOs: a step-by-step guide*. 2020 (accessed 13.04.2021). URL: https://cloud.google.com/blog/products/management-tools/practical-guide-to-setting-slos (cit. on p. 16).

[Reh21] T. Rehemägi. *The Ultimate Guide to Monitoring Serverless Applications*. Sept. 2020 (accessed 15.01.2021). URL: https://dashbird.io/blog/ultimate-guite-monitoring-serverless-applications/ (cit. on p. 5).

[Ric] M. Richards. *Software architecture patterns*. Vol. 4 (cit. on p. 39).

[Ric21] F. Richter. *Amazon Leads $130-Billion Cloud Market*. 2021 (accessed 15.06.2021). URL: https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/ (cit. on p. 51).

[RP21] R. Ribenzaft, H. Peretz. *Best Practices to Monitor and Troubleshoot Serverless Applications*. 2019 (accessed 15.01.2021). URL: https://www.youtube.com/watch?v=vGY3yk64W_U (cit. on p. 6).

[Rup16] C. Rupp. *Requirements-Engineering und -Management aus der Praxis von klassisch bis agil*. Hanser, 2016. ISBN: 978-3-446-43893-4 (cit. on p. 26).

[SBB20] S. Speth, U. Breitenbücher, S. Becker. "Gropius—A Tool for Managing Cross-component Issues". In: *European Conference on Software Architecture*. Springer. 2020, pp. 82–94 (cit. on pp. 9, 16, 74).

[SBB21] S. Speth, S. Becker, U. Breitenbücher. "Cross-Component Issue Metamodel and Modelling Language." In: *CLOSER*. 2021, pp. 304–311 (cit. on p. 9).

[SBK+13] D. Serrano, S. Bouchenak, Y. Kouki, T. Ledoux, J. Lejeune, J. Sopena, L. Arantes, P. Sens. "Towards qos-oriented sla guarantees for online cloud services". In: *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. IEEE. 2013, pp. 50–57 (cit. on pp. 7, 11).

[Se21] I. J. 1. 7. Software, systems engineering. *ISO/IEC 25010:2011*. 2011 (accessed 20.04.2021). URL: https://www.iso.org/standard/35733.html (cit. on pp. 7, 17).

[Sha21] A. Sharma. *Choosing the Right Service Level Indicators*. 2020 (accessed 14.04.2021). URL: https://devopsinstitute.com/choosing-the-right-service-level-indicators/ (cit. on p. 16).

[SS18] M. Sewak, S. Singh. "Winning in the era of serverless computing and function as a service". In: *2018 3rd International Conference for Convergence in Technology (I2CT)*. IEEE. 2018, pp. 1–5 (cit. on pp. 5, 6).

[Tea21] 2. S. Team. *What You Need to Know About Synthetic Monitoring*. Aug. 2019 (accessed 17.01.2021). URL: https://blog.2steps.io/what-you-need-to-know-about-synthetic-monitoring/ (cit. on p. 13).

[TFHB18] S. Thurgood, D. Ferguson, A. Hidalgo, B. Beyer. *The site reliability workbook: Chapter 2 - Implementing SLOs*. O'Reilly Media, Inc., 2018. URL: https://sre.google/workbook/implementing-slos/ (cit. on pp. 16, 72).

[Tun21] A. Tunall. *5 Things to Know about Serverless in 2020*. June 2020 (accessed 15.01.2021). URL: https://thenewstack.io/5-things-to-know-about-serverless-in-2020/ (cit. on p. 5).

[VTT+18] E. Van Eyk, L. Toader, S. Talluri, L. Versluis, A. Uță, A. Iosup. "Serverless is more: From paas to present cloud computing". In: *IEEE Internet Computing* 22.5 (2018), pp. 8–17 (cit. on p. 5).

[Wag13] S. Wagner. *Software Product Quality Control*. Springer Berlin Heidelberg, 2013 (cit. on p. 8).

[Wig21] A. Wiggins. *Backing services*. 2017 (accessed 15.01.2021). URL: https://12factor.net/backing-services (cit. on p. 6).

Declaration

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

---

place, date, signature