

Institute of Software Engineering  
Software Quality and Architecture

University of Stuttgart  
Universitätsstraße 38  
D-70569 Stuttgart

Bachelor's thesis

# **Extending a Microservice DSL for Service Level Objectives**

Pascal Schur

**Course of Study:** Softwaretechnik

**Examiner:** Prof. Dr.-Ing. Steffen Becker

**Supervisor:** Sandro Speth, M.Sc.

**Commenced:** September 1, 2022

**Completed:** March 1, 2023



## Abstract

*Context.* Microservice architectures have been growing in popularity for years because they are an effective means to develop scalable and maintainable software. One challenge in developing a microservice architecture is to ensure that the architecture achieves the necessary quality characteristics such as scalability and reliability. Domain-specific languages can be used to model microservice architectures and configure different aspects of them.

*Problem.* Whereby the existing domain-specific languages lack support for quality attributes such as service level objectives.

*Objective.* In this thesis, we present a concept of how such a domain-specific language can be extended to support quality aspects like service level objectives.

*Method.* We developed a prototype using the domain-specific language MDSL, which allows the modelling of service-level objectives. In addition, we have extended the generator for OpenAPI specification, already contained in Microservice Domain-specific Language (MDSL), to support our extensions as well. We validated our approach with an experiment.

*Result.* The results of this experiment show that our prototype has been well accepted by the participants and is a proper tool for specifying quality attributes in microservice architectures. Our extension allows developers to better understand and control their microservice architectures, which improves their reliability and maintainability.

*Conclusion.* In summary, our work demonstrates how important it is to specify quality attributes during the development of microservice architectures and how domain-specific languages can support this process.



## Kurzfassung

*Kontext.* Microservice-Architekturen erfreuen sich seit Jahren zunehmender Beliebtheit, da sie ein effektives Mittel zur Entwicklung skalierbarer und wartbarer Software sind. Eine Herausforderung bei der Entwicklung einer Microservice-Architektur ist es, sicherzustellen, dass die Architektur die notwendigen Qualitätsmerkmale wie Skalierbarkeit und Zuverlässigkeit erreicht. Domänenspezifische Sprachen können verwendet werden, um Microservice-Architekturen zu modellieren und verschiedene Aspekte von ihnen zu konfigurieren.

*Problem.* Den vorhandenen domänenspezifischen Sprachen fehlt die Unterstützung für Qualitätsmerkmale wie Service Level Objectives.

*Ziel.* In dieser Arbeit wird ein Konzept vorgestellt, wie eine solche domänenspezifische Sprache erweitert werden kann, um Qualitätsaspekte wie Service Level Objectives zu unterstützen.

*Methode.* Unter Zuhilfenahme der domänenspezifischen Sprache MDSL wurde ein Prototyp entwickelt, der die Modellierung von Service-Level-Zielen ermöglicht. Darüber hinaus haben wir den Generator für die OpenAPI Spezifikation, die bereits in MDSL enthalten ist, angepasst um unsere Erweiterungen zu unterstützen. Wir haben unseren Ansatz mit einem Experiment validiert.

*Ergebnisse.* Die Ergebnisse dieses Experiments zeigen, dass unser Prototyp von den Teilnehmern gut angenommen wurde und ein nützliches und effektives Werkzeug für die Spezifikation von Qualitätsattributen in Microservice-Architekturen ist. Unsere Erweiterung ermöglicht es Entwicklern, ihre Microservice-Architekturen besser zu verstehen und zu kontrollieren, was deren Zuverlässigkeit und Wartbarkeit verbessert.

*Schlussfolgerung.* Zusammenfassend zeigt diese Abschlussarbeit, wie wichtig die Spezifikation von Qualitätsattributen während der Entwicklung von Microservice-Architekturen ist und wie domänenspezifische Sprachen diesen Prozess unterstützen können.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Foundations and Related Work</b>	<b>3</b>
2.1	Foundations . . . . .	3
2.2	Related work . . . . .	12
2.3	DSL Analysis & Selection . . . . .	15
<b>3</b>	<b>Concept</b>	<b>17</b>
3.1	Concept Overview . . . . .	17
3.2	SLO Integration into a DSL . . . . .	18
3.3	SLO Integration into Interface Description Language . . . . .	21
3.4	SLO Generation in OpenSLO . . . . .	22
3.5	Usage of OpenSLO . . . . .	22
<b>4</b>	<b>Architecture and Implementation</b>	<b>23</b>
4.1	Architecture . . . . .	23
4.2	Implementation of built-in SLOs . . . . .	23
4.3	Adding OpenSLO to MDSL . . . . .	29
4.4	Document Validation . . . . .	39
<b>5</b>	<b>Evaluation</b>	<b>41</b>
5.1	Experiment . . . . .	41
5.2	Results . . . . .	43
5.3	Discussion . . . . .	44
5.4	Threats to Validity . . . . .	45
<b>6</b>	<b>Conclusion</b>	<b>47</b>
6.1	Summary . . . . .	47
6.2	Limitations . . . . .	47
6.3	Lessons Learned . . . . .	48
6.4	Future Work . . . . .	48
	<b>Bibliography</b>	<b>49</b>
<b>A</b>	<b>Appendix</b>	<b>51</b>
A.1	OpenSLO file example . . . . .	51
A.2	Experiment . . . . .	53





## List of Figures

3.1	Thesis concept. . . . .	17
3.2	Service Level Objective model. . . . .	18
3.3	Service Level Indicator model. . . . .	19
3.4	Objective model. . . . .	20
3.5	Alerting model . . . . .	20
4.1	Architecture of our extension . . . . .	23
4.2	Example of an SLA in MDSL. . . . .	24
4.3	Service Level Agreement (SLA) which uses a simple measurement instead of a landing zone. . . . .	25
4.4	Endpoint which uses our SLA. . . . .	25
4.5	Provider which uses our SLA. . . . .	26
4.6	Service Level Objective model. . . . .	30
4.7	Example Service Level Objective (SLO) in MDSL . . . . .	30
4.8	Xtext grammar of the OpenSLO template. . . . .	31
4.9	Xtext grammar of the time window object. . . . .	31
4.10	Xtext grammar of the Duration object. . . . .	32
4.11	Example Service Level Objective with a rolling time window. . . . .	32
4.12	Xtext grammar of the Service object. . . . .	33
4.13	Service used in our example. . . . .	33
4.14	Xtext grammar of the objective object. . . . .	33
4.15	Service Level Indicator used in our example. . . . .	34
4.16	Xtext grammar of the Service Level Indicator. . . . .	34
4.17	Xtext grammar of the Ratio Metric. . . . .	35
4.18	Xtext grammar of the Metric Source . . . . .	35
4.19	Xtext grammar of the Data Source . . . . .	35
4.20	Data Source used in our example. . . . .	36
4.21	Xtext grammar of Threshold Metrics. . . . .	36
4.22	Xtext grammar of Alert Policies . . . . .	37
4.23	Alert Policy used in our example . . . . .	37
4.24	Xtext grammar of the Alert Condition . . . . .	38
4.25	Xtext grammar of the Alert Notification Target . . . . .	38
4.26	Alert Notification Target used in our example . . . . .	39
4.27	Error message display when entering an invalid URL . . . . .	40
5.1	Average rating of <b>Q1</b> . . . . .	43
5.2	Average rating of <b>Q2</b> . . . . .	43
5.3	Average rating of <b>Q3</b> . . . . .	43



## List of Listings

2.1	An Example of an interface description taken from MDSL main website [Zim]. . .	6
2.2	OpenSLO template of the SLO object showing tags and their values. Taken from the OpenSLO specification on their GitHub repository [BMb]. . . . .	8
2.3	OpenSLO template of the service object showing tags and their values. Taken from the OpenSLO specification on their GitHub repository [BMb]. . . . .	8
2.4	OpenSLO template of the objective object showing tags and their values. Taken from OpenSLO specification on their GitHub repository [BMb]. . . . .	9
2.5	OpenSLO template of the Service Level Indicator (SLI) object showing tags and their values. Taken from the OpenSLO specification on their GitHub repository [BMb]. . . . .	9
2.6	OpenSLO template of the threshold metric object showing tags and their values. Taken from the OpenSLO specification on their GitHub repository [BMb]. . . . .	10
2.7	OpenSLO template of the ratio metric object showing tags and their values. Taken from the OpenSLO specification on their GitHub repository [BMb]. . . . .	10
2.8	OpenSLO template of the alert policy object showing tags and their values. Taken from the OpenSLO specification on their GitHub repository [BMb]. . . . .	11
2.9	OpenSLO template of the alert condition object showing tags and their values. Taken from the OpenSLO specification on their GitHub repository [BMb]. . . . .	11
2.10	OpenSLO template of the alert notification target object showing tags and their values. Taken from OpenSLO specification on their GitHub repository [BMb]. . .	12
3.1	Example of multiple YAML objects in a single document. . . . .	22
4.1	Generated OpenAPI description from Figure 4.4. . . . .	27
4.2	Generated SLA description from Figure 4.2. . . . .	28
4.3	Inlined SLA in a OpenAPI specifcaiton. . . . .	29
4.4	Info object in the OpenAPI document. . . . .	39
A.1	Generated YAML document from our example part one. . . . .	51
A.2	Generated YAML document from our example part two. . . . .	52
A.3	MDSL file of the t2-inventory service provided to the study participants. . . . .	53
A.4	MDSL file of the t2-inventory service provided to the study participants. . . . .	54



# Acronyms

<b>API</b>	Application Programming Interface.	1
<b>CLI</b>	Command-line Interface.	6
<b>DSL</b>	Domain-specific Language.	1
<b>EMF</b>	Eclipse Modeling Framework.	3
<b>GQM</b>	Goal Question Metric.	42
<b>HTTP</b>	Hypertext Transfer Protocol.	3
<b>IDE</b>	Integrated Development Environment.	5
<b>IDL</b>	Interface Description Language.	21
<b>MDSL</b>	Microservice Domain-specific Language.	iii
<b>SLA</b>	Service Level Agreement.	ix
<b>SLI</b>	Service Level Indicator.	xi
<b>SLO</b>	Service Level Objective.	ix
<b>URL</b>	Uniform Resource Locator.	36
<b>WSDL</b>	Web Services Description Language.	6



# 1 Introduction

Microservice architectures have gained significant popularity in recent years as a solution for building large and complex software systems that can be deployed and maintained efficiently. Microservices are small, independent services that work together to form a larger system. One of the key advantages of microservice architectures is their ability to provide scalable and maintainable software systems. However, ensuring that microservice architectures meet the required quality characteristics, such as availability and performance, remains a challenge.

Domain-specific languages have been used to model and configure various aspects of microservices, including their Application Programming Interfaces (APIs), deployment. Domain-specific Languages (DSLs) provide a powerful way of describing the architecture of a system, as well as automating certain tasks associated with the development, deployment, and maintenance of microservices.

However, most existing DSLs for microservices do only provide support for modelling the functional contract of the architecture but provide no support for specifying quality characteristics like service level objectives. To address this gap, in this thesis, we present an extension to the MDSL domain-specific language that enables the specification of quality characteristics, like service level objectives, for microservice architectures. MDSL is an open-source domain-specific language that has been widely used for modelling microservices. Our extension enables developers to specify SLOs for their microservices, as well as integrate them into the overall architecture. We also extend the built-in OpenAPI generator to support these additional constructs.

To validate the effectiveness of our approach, we created a scenario using the extended version of MDSL and conducted an experiment with participants from the software engineering community. The results of our experiment show that the extended DSL is well-received by participants and provides a valuable tool for building better microservice architectures.

In summary, the main contribution of this thesis is the extension of MDSL to include support service level objectives. Our approach enables developers to better understand and control the behaviour of their microservices, improving the reliability and maintainability of the system. Our work highlights the importance of considering quality characteristics in the design and implementation of microservices, and the value of DSLs in supporting this process.

## Thesis Structure

Here, we provide an overview of the structure of this thesis.

**Chapter 2 – Foundations and Related Work:** Here, we provide an overview of existing techniques and technologies this thesis uses and builds upon. Also, related work is presented and discussed.

**Chapter 3 – Concept** Here, we provide an overview of the concept of this bachelor's thesis.

**Chapter 4 – Architecture and Implementation** In this chapter, we discuss the architecture, and implementation of the extension. Describe the choices made.

**Chapter 5 – Evaluation:** In the last part of the thesis the evaluation the planning and conducting of the experiment is described. Then the results of this evaluation are presented and described. Lastly, the threats to validity are discussed.

**Chapter 6 – Conclusion** In the last chapter we provide a summary of the thesis, reflect on lessons learned and provide an outlook to future work.



## 2 Foundations and Related Work

In this chapter, the foundations and related work are shown. For this purpose first, the necessary foundations are listed and discussed in Section 2.1. Followed by the literature and research methodology with which the related works were obtained. The related work is then listed and discussed in Section 2.2.

### 2.1 Foundations

In this section, the necessary foundations to understand this thesis are listed and explained. Starting with microservice architecture in Section 2.1.1 while model-driven software development is explained in Section 2.1.2. Following DSLs in general are explained in Section 2.1.2. As a framework to develop such DSLs, the Eclipse Modeling Framework (EMF) and Xtext are introduced and discussed. To end this subsection, the DSL MDSL will be introduced. Information about SLAs, SLOs and the OpenSLO specification can be found in Section 2.1.3.

#### 2.1.1 Microservice Architecture

Despite the microservice architecture style becoming more popular in recent years. The design philosophy behind the microservice architecture is heavily influenced by the UNIX philosophy. They can be summarized into three key aspects “Make each program do one thing well”, “Programs should be able to work together”, and “Use a universal interface” [MPT78, p. 1902] [Wol15, p. 2].

Microservice architecture is a style to develop software applications. The application consists of individual services each fulfilling one task. Each service communicates with the others using lightweight mechanisms like Hypertext Transfer Protocol (HTTP) resource APIs. These services are automatically deployable and scalable and built around business capabilities. There is little central management which can be written in different programming languages and use different data storage technologies. [FL14]. With this style, we can modularize our application and make it more loosely coupled [New21]. We also enable different teams to work independently of each other on their own service. The services are not technologically dependent on each other. This enables the different teams to use the programming languages and tools best suited to solve their respective problem without hindering the other teams. And when we begin to combine these services together they can fulfil a bigger task by working together. Imagine three services, one service can manage an inventory, another can process orders and one can offer baking services. With all three of them working together, we can build a small web shop application.

Microservice architecture also enables the scaling of agile development processes without generating a lot of communication and coordination effort. Microservices should be ideally small. Then they are clear and can be quickly developed further. In case of demand, they can be easily replaced

by a re-implementation. In large systems, you have often the problem that over time unwanted dependencies creep in. Dependencies between microservices must be introduced via the API. This can be complex and does not happen accidentally. Microservices can be scaled independently from each other. Each service can be secured against the failure of other services so that the whole system is robust. Another advantage is, if key services are identified, in case of overload non-critical services can be scaled down or terminated, to free resources for critical services.

But this new style also brings its own set of challenges. Dependencies between microservices are not obvious in a microservice architecture. Often it is not clear which service needs another service and which version this other service should have. The distributed nature of the microservice architecture generates additional complexity predominately in network latency, load balancing and fault tolerance. Because there are more systems than in monolithic services, the probability is higher that one component fails. The variety of services makes development and testing more complex.

To tackle the last challenge, especially the development one. We can use the model-driven development approach and domain-specific languages to model our architecture and generate all necessary artefacts. This ensures consistency among all artefacts, is less prone to human errors and needs less time and saves therefore money, compared to creating each artefact by hand and is beneficial for the developer and the company.

### 2.1.2 Model-Driven Software Development

Model-driven software development is a software development approach that uses models as specifications of the software. It focuses more on modelling and model analysis and less on programming, code generation and other development steps. These steps are automated [WB23, p. 21]. These models are then used to generate different artefacts of the software e.g. source code or documentation. This follows the DRY- principle. It states "don't repeat yourself" or "Once and Once only" [ES19, p. 2]. In software development, this means we should avoid redundancy and code duplicates because maintaining them is time-consuming. The models are generally more abstract than the source code for the system but this does not mean they are less precise but more compact than the system implementation. In model-driven software development, this principle is followed by only having one model of the architecture which contains all information. With this model and some generators, all other necessary artefacts can be generated. This has the advantage that all generated artefacts are consistent with each other. It saves time in the development process because if one thing in the software changes, you only have to make the change in the model and regenerate all artefacts. To achieve this domain-specific languages can be used.

#### Domain-specific Languages

Domain-specific languages are programming languages which are highly specialized to one particular domain [Fow10]. Additional characteristics are that domain-specific languages should be designed to be used by humans and should therefore support the user to make the task as simple as possible. Secondly, the language should have a sense of fluency where expressiveness comes not only from individual expressions but also from the way they can be composed. This makes the language easy to read and supports therefore the understandability. Thirdly domain-specific languages have

limited capabilities in comparison to general-purpose programming languages. This means their focus is highly specialized tasks from a particular domain. These tasks are carried out by experts with a lot of domain knowledge.

This makes them perfect for our use case to support software architects in maintaining software architectures and the corresponding artefacts. Because with domain-specific languages we can realize a programming language tailored to the needs of software architects and their respective domain knowledge. Additionally, we can support them in specialized tasks which are difficult to automate using general-purpose programming languages.

### **Xtext**

Xtext is a framework to develop domain-specific languages developed by the eclipse foundation [ES]. It comes with its own grammar language which can be used to define the syntax and semantics of your domain-specific language.

From this grammar, you can generate the full infrastructure to use your language including a lexer parser, abstract syntax tree and an editor for the domain-specific language fully integrated into the eclipse Integrated Development Environment (IDE). From this grammar, Xtext generates an Ecore model, from which a code generator generates fully functional java classes. Each of these sub-products can be used independently from Eclipse. Also, Xtext features full integration in the Eclipse IDE and supports syntax colouring, code completion, and static analysis. All the features are highly customizable and can be tailored to the needs of the language. The generated Java classes make it easy to build any feature on top of it e.g. generators for different technology specify descriptions.

Lastly, from this whole infrastructure, you can generate a plugin for the eclipse IDE to ship your language and tools to your target audience. In this thesis, it is used because the language on top which we build our SLO extension is based on this framework.

### **Eclipse Modeling Framework**

The EMF is a modelling framework and code generation facility for building tools and other applications based on a structured data model [Gro]. Xtext uses EMF to generate its Ecore model out of the Xtext grammar from the domain-specific language and the abstract syntax tree. It also supports the Language-server protocol which can be used to generate a Visual Studio Code plugin.

### **Microservice Domain-specific Language**

The key foundation of this thesis is the Microservice Domain-specific Language. The Microservice Domain-specific Language is developed by Zimmerman et al. at the Eastern Switzerland University of Applied Sciences [Zim].

Microservice Domain-specific Language is an interface description language. It distinguishes itself from other such languages by abstracting from technology-specific descriptions such as OpenAPI<sup>1</sup> formerly Swagger, WSDL<sup>2</sup> or Protocol Buffers<sup>3</sup>. It offers a language to describe the interfaces in a technology-agnostic way and generate technology-specific descriptions with the help of built-in generators.

Part of MDSL is an eclipse plugin which facilitates the popular eclipse editor to offer a graphical user interface, project support, and validation of the interface descriptions and generators for the different technology-specific formats. Some of the current, supported interface descriptions are OpenAPI, GraphQL, and Jolie Lang Specification.

It also includes a standalone Command-line Interface (CLI) which features validation, and generation out of the MDSL interface descriptions.

In Listing 2.1 an example description of an interface is given, with which we will give a little introduction into MDSL.

---

**Listing 2.1** An Example of an interface description taken from MDSL main website [Zim].

---

```
API description HelloWorldWorld

data type SampleDTO {ID, D<string>}

endpoint type HelloWorldEndpoint
exposes
  operation sayHello
    expecting payload "in": D<int>;
    delivering payload SampleDTO

API provider HelloWorldAPIProvider
offers HelloWorldEndpoint
at endpoint location "http://localhost:8000"
via protocol HTTP
  binding resource HomeResource at "/"
  operation sayHello to POST

API client HelloWorldAPIClient
consumes HelloWorldEndpoint
from HelloWorldAPIProvider
via protocol HTTP
```

---

In this example, we can see how to describe a simple API. It exposes one single endpoint with a sayHello operation. This operation takes an integer value as input and returns a sampleDTO Object. Note SampleDTO is specified as a data pair of ID and D. ID is an identifier and D is not specified and describes some data. Both parameters have no names. In difference to the payload specification of the sayHello operation here, we have a fully specified parameter with name “in”

---

<sup>1</sup><https://www.openapis.org/>

<sup>2</sup><https://www.w3.org/TR/wsdl20/>

<sup>3</sup><https://protobuf.dev/>

and data specified as int. Additionally to the HelloWorldEndpoint, we define an API provider and an API client. Both work with the specified contract and are both bound to a single home resource over HTTP.

### 2.1.3 Service Level Objectives

Service level objectives are a contract between a service provider and its client. SLOs define the objective that a service has to fulfil. This can be the availability or latency of the service. These objectives are further described with service level indicators which specify how to measure and evaluate them. Both service level objectives and service level indicators are part of a service level agreement which often contains compensations in case the objective is not met by the provider. They are often attached to a business contract. [Atl]. There are many languages to describe SLOs. In this thesis, we focus on OpenSLO [BMa].

#### OpenSLO

OpenSLO is an open-source specification for service-level objectives. This specification is developed by Bartholomew et al. [BMa]. We use it here because there is currently no standard model to describe service-level objectives. OpenSLO tries to establish itself as a standard model. It is not technology dependent like WS-Policy<sup>4</sup>, and uses the comprehensible description language YAML.

The specification will now be explained with examples taken from OpenSLO's GitHub repository [BMb].

We start with a Service Level Objective:

The specification of SLOs begins with an API version followed by the kind of document. Each element has a metadata object which contains the name of the element also used as a reference and an optional display name. Then follows the specification of the SLO. Starting with an optional description string. Then we can reference a service object. The service object is a fairly simple one it contains a name an optional display name and a description. It is used to group different SLOs together by referencing the same service object in each SLO. The template of a service can be seen in Listing 2.3.

Then we can reference a defined SLI by using the `indicatorRef` key or inline an SLI with the `indicator` keyword. Then we have to specify the time window in which the SLO will be budgeted. Here we can choose a rolling time window. Then we use the `isRolling` keyword and set it to true and give a duration shorthand. A shorthand is an integer number followed by a letter indicating the time unit. Currently, the following short hands for time units are allowed: m for minutes, h for hours, d for days, w for weeks, M for months, Q for quarters and Y for years. Or we can choose a fixed time window by using the `calendar` keyword and give a duration shorthand, a start time in the 24h format and a time zone. In this case, we can omit the `isRolling` keyword. Then we have to choose the budgeting method. The budgeting method specifies how our error budget is calculated. Here we can choose from Occurrences, Timeslices and Ratiotimeslices.

---

<sup>4</sup><https://www.w3.org/Submission/WS-Policy/>

**Listing 2.2** OpenSLO template of the SLO object showing tags and their values. Taken from the OpenSLO specification on their GitHub repository [BMB].

---

```
apiVersion: openslo/v1
kind: SLO
metadata:
  name: string
  displayName: string # optional
spec:
  description: string # optional up to 1050 characters
  service: string # name of the service to associate this SLO with
  indicator: # see SLI below for details
  indicatorRef: string # name of the SLI. Required if indicator is not given.
  timeWindow:
    # exactly one item; one of possible: rolling or calendar-aligned time window
    ## rolling time window
    - duration: duration-shorthand # duration of the window eg 1d, 4w
      isRolling: true
    # or
    ## calendar-aligned time window
    - duration: duration-shorthand # duration of the window eg 1M, 1Q, 1Y
      calendar:
        startTime: 2020-01-21 12:30:00 # date with time in 24h format, format without time zone
        timeZone: America/New_York # name as in IANA Time Zone Database
        isRolling: false # if omitted assumed `false` if `calendar:` is present
  budgetingMethod: Occurrences | Timeslices | RatioTimeslices
  objectives: # see objectives below for details
  alertPolicies: # see alert policies below for details
```

---

**Listing 2.3** OpenSLO template of the service object showing tags and their values. Taken from the OpenSLO specification on their GitHub repository [BMB].

---

```
apiVersion: openslo/v1
kind: Service
metadata:
  name: string
  displayName: string # optional
spec:
  description: string # optional up to 1050 characters
```

---

- Occurrences mean the error budget is calculated by counting good attempts e.g. successful requests against all requests and this ratio is then compared against the target defined in the SLO.
- Timeslices mean the error budget is calculated by measuring the good minutes of a system in the time slice. Good minutes mean here minutes where the system operates within the defined boundaries. Each timeslice has an additional allowance. An allowance is used to evaluate each timeslice. It can be considered a micro-objective. And when a timeslice met the micro objective it is considered good. And the ratio of good timeslices against all timeslices is then compared to the SLO.
- Ratiotimeslices are similar to timeslices but here an average of all timeslices' success ratios is used to do the budgeting.

Then we have to specify at least one objective or more. The template of an objective is shown in Listing 2.4 this has to be inlined and is here shown for simplicity. The objective has an optional

---

**Listing 2.4** OpenSLO template of the objective object showing tags and their values. Taken from OpenSLO specification on their GitHub repository [BMB].

---

```
objectives:
  - displayName: string # optional
    op: lte | gte | lt | gt
    value: numeric
    target: numeric [0.0, 1.0]
    targetPercent: numeric [0.0, 100]
    timeSliceTarget: numeric (0.0, 1.0]
    timeSliceWindow: number | duration-shorthand
```

---

display name. Then the op key is only needed if we use a threshold metric in the referenced SLI. The same applies to the operator key here lte stands for less-than equal, gte for greater-than equal, lt stands for less-than, gt for greater-than. Then we have to specify the target of the SLO. Here we can choose between target and targetPercent. The difference is with target we have to give a decimal representation of the percentage. And with targetPercent we can the percentage directly. TimeSliceTarget is only needed if we selected timeslices as a budgeting method here we have to give a percentage. This percentage is then used to evaluate if the timeslice was good. TimeSliceWindow needs a number or a duration-shorthand and sets the time window in which the slice is evaluated. If you provide only a number the unit minutes is assumed. And to complete the SLO shown in Listing 2.2 we have to inline an Alert Policy object or provide a reference to a defined Alert Policy. In the latter case, we have to use the alertPolicyRef keyword followed by a reference.

The second big part of the specification is the service level indicator shown in Listing 2.5

---

**Listing 2.5** OpenSLO template of the SLI object showing tags and their values. Taken from the OpenSLO specification on their GitHub repository [BMB].

---

```
apiVersion: openslo/v1
kind: SLI
metadata:
  name: string
  displayName: string
spec:
  description: string
  thresholdMetric: [...]

  ratioMetric: [...]
```

---

An SLI starts similar to the SLO. We have the API version followed by the kind and then the metadata object. Then we have an optional description of the SLI. Then we have to choose between a ratio metric or a threshold-based metric. They are here and shown separately for the sake of understandability. We start with the threshold metric shown in Listing 2.6.

A threshold metric compares raw values against the objective. The threshold metric consists of a single query to obtain data. First, we can reference a predefined data source object using the metricSourceRef keyword. A data source contains information on how to access a system to obtain

**Listing 2.6** OpenSLO template of the threshold metric object showing tags and their values. Taken from the OpenSLO specification on their GitHub repository [BMb].

---

```
thresholdMetric:
  metricSource:
    metricSourceRef: string
    type: string
    spec:
```

---

data. We then provide a type of data source e.g. Prometheus Datalog etc. and in the spec, we can insert any valid YAML to obtain the data this can be queries or an accesskey. MetricSourceRef and type are optional. Then everything needed to obtain the data must be specified under the spec keyword.

More complex is the ratio metric shown in Listing 2.7. The metric source objects are identical to

**Listing 2.7** OpenSLO template of the ratio metric object showing tags and their values. Taken from the OpenSLO specification on their GitHub repository [BMb].

---

```
ratioMetric:
  counter: true | false
  good:
    metricSource:
      metricSourceRef: string # optional
      type: string # optional
      spec:
  bad:
    metricSource:
      metricSourceRef: string
      type: string
      spec:
  total:
    metricSource:
      metricSourceRef: string # optional
      type: string # optional
      spec:
  rawType: success | failure
  raw:
    metricSource:
      metricSourceRef: string
      type: string
      spec:
```

---

them in the threshold metric. First, we have to indicate if our metric is monotonically increasing. This can be done with the counter keyword. If we choose to use raw the counter keyword can be omitted. Then the following combinations of good, bad, total and raw keywords are allowed.

- If you provide a good and a total metric object the ratio will be calculated in the following way  $ratio = \frac{good}{total}$ .
- If you provide bad a total metric object the ratio will be calculated in the following way  $ratio = \frac{bad}{total}$ .



- If the ratio is stored directly in the system we use raw then we have to specify how the stored ratios are calculated. We do this with the `rawType` success means  $ratio = \frac{good}{total}$  and failure means  $ratio = \frac{bad}{total}$ .
- Any other combinations are not permitted.

Now we can specify which objective we want to measure but we also want to specify what happens when an objective is not met or when the system begins to breach one. For this, we have the Alert Policy its template is shown in Listing 2.8. An Alert Policy has the standard header information with

---

**Listing 2.8** OpenSLO template of the alert policy object showing tags and their values. Taken from the OpenSLO specification on their GitHub repository [BMb].

---

```
apiVersion: openslo/v1
kind: AlertPolicy
metadata:
  name: string
  displayName: string
spec:
  description: string
  alertWhenNoData: boolean
  alertWhenResolved: boolean
  alertWhenBreaching: boolean
  conditions:
    - conditionRef:
  notificationTargets:
    - targetRef:
```

---

name and display name and an optional description string. Then we have to specify when the alert will trigger. Here we can them trigger when no data arrives, when the breaching ends and when the system begins to breach the SLO. To specify when we consider an SLO is breaching we use the alert condition object with the `conditionRef` keyword or by simply inlining the condition. The same goes for the Alert Notification Target. An alert condition specifies when the corresponding alert policy comes into effect. The model of an alert condition is shown in Listing 2.9 The alert condition

---

**Listing 2.9** OpenSLO template of the alert condition object showing tags and their values. Taken from the OpenSLO specification on their GitHub repository [BMb].

---

```
apiVersion: openslo/v1
kind: AlertCondition
metadata:
  name: string
  displayName: string
spec:
  description: string
  severity: string
  condition:
    kind: string
    op: enum
    threshold: number
    lookbackWindow: duration-shorthand
    alertAfter: duration-shorthand
```

---

has the same header information as the alert policy and an optional description field. Then we

have to specify the severity of this condition and the condition. Currently, only burnrate conditions are supported by the OpenSLO specification. A burn rate describes how quickly we consume our error budget. The operator specifies how to compare our consumption against the set threshold. The operator enum is the same as used in the objective object shown in Listing 2.4. A threshold is described by an integer number. Here one means we consume our error budget according to plan. Two means we consume our error budget twice as fast as the plan suggests it. Three means we consume our error budget three times as fast as the plan suggests it. This pattern continues for higher numbers. We have to specify the look-back window which determine how the threshold is calculated. The alert after window specifies how long the condition needs to be valid before we start alerting. The Last Object we have to specify to complete our OpenSLO specification is the alert notification target. It describes how the alert policy can reach a certain person in case it starts alerting. We can see the template of an Alert Notification Target in Listing 2.10. The alert

---

**Listing 2.10** OpenSLO template of the alert notification target object showing tags and their values. Taken from OpenSLO specification on their GitHub repository [BMb].

---

```
apiVersion: openslo/v1
kind: AlertNotificationTarget
metadata:
  name: string
  displayName: string
spec:
  target: string
  description: string
```

---

notification target starts with a target string. It specifies how we reach the person. Here we can specify an email or push messages or anything else. And the description is optional. This concludes the explanation about the OpenSLO specification.

## 2.2 Related work

In this section, the related work is discussed. Starting with the literature research methodology in Section 2.2.1. Following with the related work in Section 2.2, which mostly consists of other domain-specific languages.

### 2.2.1 Literature Research Methodology

To find related work for this thesis the search engine Google Scholar was used.<sup>5</sup> First keywords relevant to this thesis were collected and then entered into the search engine individually and in suitable combinations. For all results, the first ten entries were considered. The individual entries were checked for their relevance to this thesis in the following way. For the entries I first looked at the title, some entries could already be excluded. From the remaining entries, the abstract was read, and after that, some entries could be excluded again. After that, the resulting entries were analyzed

---

<sup>5</sup><https://scholar.google.de/>

in detail. Also, the technique of snowballing was used to analyze the referenced sources according to the same principle. For the results from Google, we were mostly searching for projects so the goals of the project were considered and evaluated.

When collecting relevant keywords they were divided into different categories and in these categories promising combinations were entered into the search engines. The first category was about domain-specific languages in regard to microservice architecture.

- domain-specific language
- microservice
- microservice modelling dsl
- language ecosystem modelling microservice

Because the search for domain-specific languages yielded not many results. The search was extended to the search engine Google.<sup>6</sup> As with Google Scholar the first ten entries from Google were considered when searching for the keywords. The keywords mentioned above were used plus these additional keywords.

The second category was then about service level objectives and metamodels for them.

- Service Level Objective
- Service Level Objective Specification
- Service Level Objective Model
- Service Level Agreement
- Service Level Indicator

This search only yielded the OpenSLO model mentioned in Section 2.1.3.

### 2.2.2 Silvera

Silvera is a DSL developed by Suljkanović et al. [SMID22]. It allows the modelling of distributed systems based on microservices. From this model, it is possible to generate code to support technology stacks. Currently, they only support code generation to Java. Silvera offers extensibility through registering custom code generators written in Python. They also generate documentation for the architecture and also allow the evaluation of that architecture. There is currently no IDE support for Silvera. They planning on implementing one and extending Silvera to support security concepts. Silvera focuses solely on the generation of code and documentation for each service. As our approach focuses more on the generation of technology-specific descriptions of microservice architecture like interface description languages e.g. OpenAPI or ProtocollBuffers which then can be taken by additional generators to generate code.

---

<sup>6</sup><https://www.google.com/>

### 2.2.3 MicroART

MicroART is a DSL aimed to support software architects in recovering software architectures. In the paper by Granchelli et al. [GCD+17]. They propose a whole approach to recover and refine microservice-based architectures and for this approach, they created the MicroArt DSL to support this approach. MicroArt is based on EMF and the Java Spring framework and offers support for Eclipse IDE. The DSL allows modelling software architecture and mapping properties of the architecture to specific concepts implemented in MicroArt. The model can then be displayed to make architectural refinements to it. MicroArt focuses more on recovering and reconstructing already designed software architectures and supporting the evolution of software architectures. They present an approach to how this could be done using the MicroART DSL. Our work focuses more on supporting the initial design process of microservice architectures by allowing the modelling of not only the functional contract but also modelling quality characteristics like service level objectives and supporting the development by enabling the generation of technology-specific descriptions to be used by code generation to increase the implementation of such architectures.

### 2.2.4 Olive

Flaconi et al. developed the model-centred microservice Olive [FW21]. Olive is integrated into the ADOxx graphical modelling environment. Its goal is to create model-aware microservices. Its aim is to configure and manage configuration for model-dependent functionalities, and data sources for processing mechanisms. They want to enable these mechanisms to have access to model data. They do this through their concept of model-aware microservices they call connectors. They achieve that by attaching a connector to each microservice and exposing the connector through a REST interface. The connector is developed from an OSGi<sup>7</sup> plugin model. With this, they can control the whole lifecycle of the microservice and the microservice gets access to model information provided in ADOxx [FW21]. Olive focuses more on the relationship of microservices with the ADOxx platform as we want to improve the development process of microservice architectures by allowing the specification of quality characteristics of such architectures using domain-specific languages.

### 2.2.5 A language ecosystem for modelling microservice architectures

LEMMA is an acronym for the Language Ecosystem for modelling microservice architecture. The whole framework is based on the Eclipse modelling framework EMF mentioned in Section 2.1.2 and Xtext mentioned in Section 2.1.2. It is developed by Rademacher et al [Rad22] in his PhD thesis. Like the name said it is an ecosystem to design, develop and deploy microservice architectures using model-driven engineering. To accomplish this LEMMA focuses on three key aspects according to their website [Rad],

- modelling languages to define microservices and their deployment
- model transformation the code for the microservice and the necessary configuration to deploy the service

---

<sup>7</sup><https://camel.apache.org/>

- model analysis to detect smells and faults in the software architecture before the development

For this thesis, the modelling languages are most interesting. Lemma defines each viewpoint on architecture as a domain-specific language.

- domain data modelling language to create a domain model of the architecture
- operation modelling language to model the capabilities of different vendors on which the microservices can operate on
- service modelling language to create a service model of the architecture
- technology modelling language to model technology information

Each of these modelling languages is written in Xtext and has therefore full editor support from Eclipse.

## 2.3 DSL Analysis & Selection

In this section, we want to discuss how we selected the DSL. We will first reason why we did not choose to extend different languages and then provide an explanation of why we choose MDSL in the end. We start with Silvera mentioned in Chapter 2. The first and biggest problem for Silvera is that there is currently no editor support which is essential to allow a good user experience and makes it possible to test our extension with users mostly not familiar with Silvera and DSLs in general. The second reason is that Silvera focuses only on the generation of application code currently Java and not interface descriptions. This makes it hard to integrate service-level objectives and monitoring capabilities into them. Because monitoring should be done from other servers and applications and not by the main application itself to prevent the monitoring service from going down with the main service, which contradicts its main purpose. The last reason is more of an organizational reason Silvera was made public in July of 2022 and is the newest of these languages so the documentation is not that extensive what makes the extension really time-consuming which was a problem regarding the time scope of this thesis.

The next language investigated was Olive. Here was the main problem the different goals of their and our approach. Olive focuses on integrating microservice architectures in the ADDOxx metamodeling framework. Also, it follows a low code approach. This does not fit because software architects are domain experts who understand how to read and write code.

The third language investigated was MicroArt. Also here the focus of the project does not align well with our concept. MicroArt wants to make microservices more recoverable by modelling them and then making precise analyses of them. Also, MicroArt generates no interface descriptions or other artefacts of the architecture like technology-specific descriptions.

The last investigated language was LEMMA. The reason we choose not to extend LEMMA was primarily that it is currently in development and not sufficient docs are available and the generators for technology-specific descriptions are not built.

To close the section we will explain why we then choose to extend MDSL. First reason MDSL is out there for more than three years. Meaning there are extensive documentation, tools and editors to build our extension, upon. Additionally, MDSL offers many interface descriptions for the

microservices including OpenAPI and gRPC from which we can choose the most suitable to extend this description for SLOs. It also is fully integrated into the eclipse editor which makes it easier to become familiar with it. This is particularly beneficial because to validate our extension we want to conduct an experiment with participants mostly not familiar with DSLs and then, the editor should make the introduction into MDSL and our extension easier for them. This also allows us to conduct feedback only on our extension and prevents the results to be overshadowed by the limitations of the base language.

## 3 Concept

In this section, the concept of this bachelor’s thesis will be discussed. At the start, a brief overview of the concept is given in Section 3.1. Followed by the integration of the OpenSLO model into MDSL via the Xtext grammar in Section 3.2. We also will explain how we modelled the SLO and why we have done this. Section 3.3 deals with the integration of SLOs into the OpenAPI specification. Here we will explain which mechanism we used to accomplish this integration. In a similar fashion, we will explain in Section 3.4 how we integrate the SLO into the OpenSLO specification and also talk about the limitation of the OpenSLO specification. And lastly, in Section 3.5 of the SLO we will explain how the extension can be used.

### 3.1 Concept Overview

Today most tools for designing microservice architectures only allow designing the functional properties of software architectures like interface descriptions and not quality characteristics like SLOs. The concept of this thesis deals with supporting software architects to develop higher-quality software architectures and take the above-mentioned shortcoming. The approach is depicted in Figure 3.1. The concept of this thesis is to enable the designing of microservice architectures not only on the functional level but also to design quality attributes of the software. We use therefore domain-specific languages. Domain-specific languages are aimed at the small domain with a very specific use case. So they can be tailored to the user’s needs and our use case. Additionally, they offer features like easy integration for generators to create specific artefacts for the microservice architecture. So we extend a domain-specific language for microservice architectures to support the modelling of quality characteristics. Additionally, we want to integrate the model of the SLO into the model of the architecture in the DSL. Then we want also to extend the generators of this language to also support the generation of SLOs specification and integrate them into the artefacts of the architecture. To allow additional generators which use the artefacts created by our DSL to use our extension. Like implementations for server stubs and monitoring systems for them.

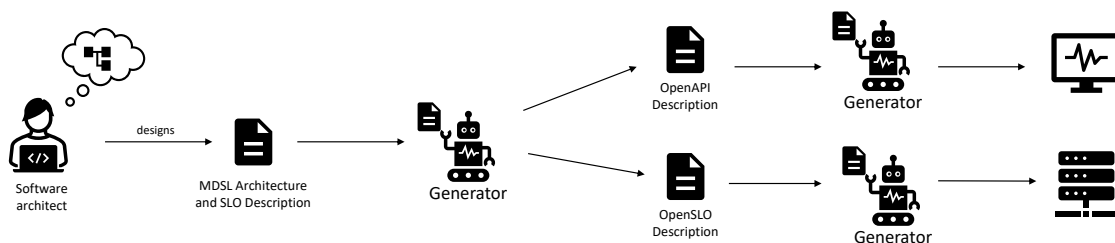


Figure 3.1: Thesis concept.

### 3.2 SLO Integration into a DSL

In this section, we will explain how such a DSL could be extended with the help of the UML model of Service Level Objectives depicted in Figure 4.6. In general, the model of the DSL should be extended with a model of the SLO. We will explain now what such a model can look like and which parts it should include and why. Also is explained what the intentions behind the different parts of the SLO model are.

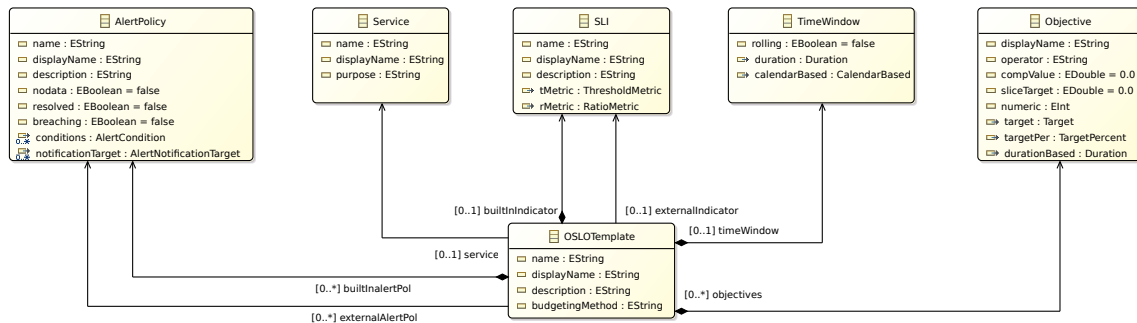


Figure 3.2: Service Level Objective model.

Here you see an extract from the model. There are the following entities. The SLO template itself serves as the container for all parts of the SLO. It should allow the software architect to make the SLO identifiable and distinguishable from other SLO objects. The SLO should have a name to make it referenceable and a display name to distinguish the reference from the actual displayed name. The SLO should also have a description to communicate its purpose. The budgeting method should be used to indicate how the described SLO is budgeted. This allows the software architect to make the SLO more adaptable to change and less strict in regard to its condition.

The next entity is the service entity which serves as a grouping mechanism. It consists only of its name and a display name and a description of its purpose. It should allow the software architect to group different SLOs together if they are applying for the same service.

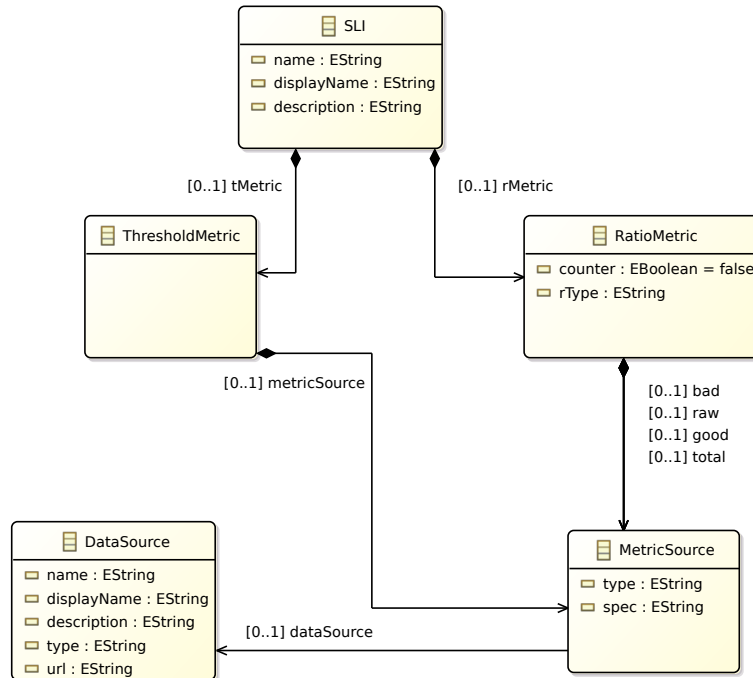
The time Window entity should be used to specify the period of time or interval over which the SLO is evaluated. This is to enable the architect to formulate precisely the timeframe over which this SLO is evaluated.

They specify the SLO and operational the condition the SLI it used. Is depicted in more detail in Figure 3.3.

The service level indicator is there to specify how we measure and calculate the metric to determine the state of the microservice and compare them against our objective. The SLI has like the SLO a name and display name for the same reasons. Then the SLI should contain metrics to measure the state of the microservice. Ideally, there should be the possibility to model different metrics to cover a wider variety of scenarios and give software architects the ability to describe the SLI more to the system need. In some cases more strict in others more volatile. Each metric should then contain a specification on how they obtain their data to calculate the metric and through which source they do this. To address the sources a data source entity should be present to encapsulate all these details and make the SLI in this case more readable. The metric source should encapsulate how we obtain the data from a data source here regarding how we establish a connection and if present how we



authenticate ourselves to the system to obtain data. This should allow software architects to use different tools and technologies to collect data about the microservice architecture and measure them.



**Figure 3.3:** Service Level Indicator model.

In Figure 3.4 the part which addresses the objective of the SLO. The purpose of the objective is there to specify the goal which the microservice should meet. This goal is expressed with the description of the objective and a per cent value and an operator to specify how to compare the objective against the metric defined in the SLI. Then the objective should specify over which period the objective is measured. The objective also is closely related to the budgeting method and should also specify how the objective behaves regarding the selected budgeting method if necessary. This is done to enable software architects to model different types of SLOs.

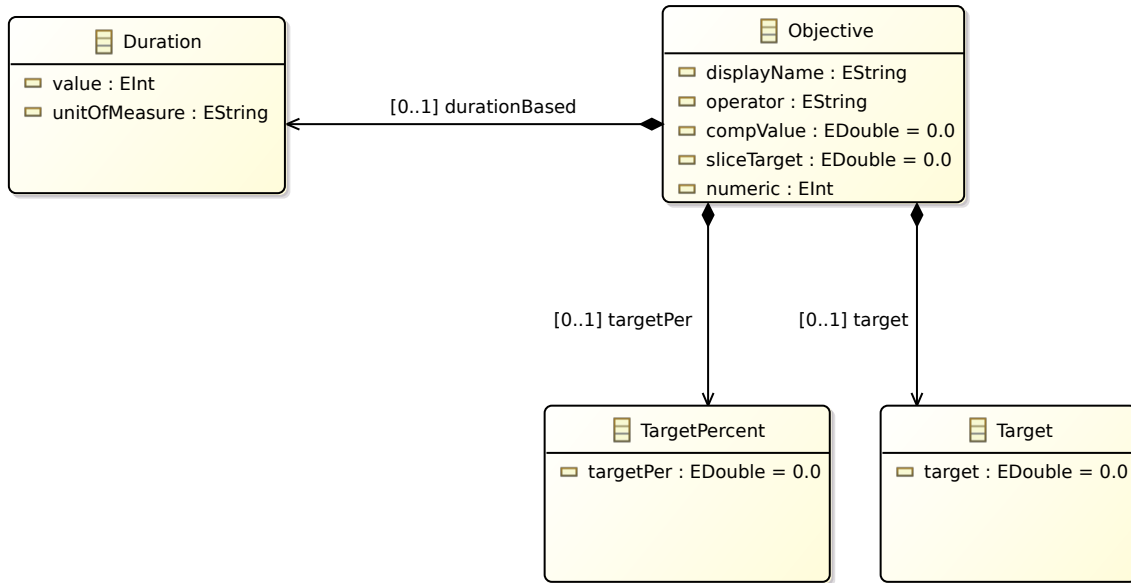


Figure 3.4: Objective model.

The last part of the model of the SLO is concerned with alerting.

Our SLO should also specify what happens when the defined SLO is breached or when we get no data indicating the state of our microservice. It should also specify whom to reach in that case and how we can do this. Concerned with this the alert policy entity in our model is depicted in Figure 3.5.

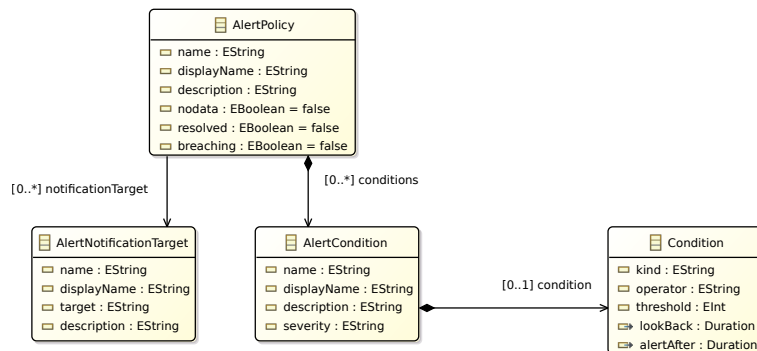


Figure 3.5: Alerting model

The alert policy entity is there to describe the different cases in which alerting should happen. An SLO can have more of such policies to cover different scenarios and enable the software architect to describe different behaviours regarding alerting. The alert policy should have a name display name. The name should be used to reference the corresponding alert policy in other objects. The display name is to be used to display the alert policy name to the user it uses. Then the description should be included to describe the purpose of this alert policy and give additional information about the alert policy. Then it should be indicated when the alerting happens. Three cases that should be covered are: when no data arrives to be compared against the SLO when the SLO begins to breach

this is further specified in the condition, and lastly when the breach is resolved to indicate no further action is necessary. They precisely specify the condition on which the alerting triggers the alert condition and the condition entities are introduced. The alert condition entity serves as a container for all the conditions regarding the specific alert policy. It has also a name to make it referenceable and a display name to separate the displayed name from the reference. Then the severity should be described to allow the software architect to indicate how high the impact breaching of this condition is. Then a description to describe the group of conditions under this alert condition. The condition entity describes the condition which is evaluated to determine if alerting should trigger. The condition should first describe which kind of condition it is. This could for example be a burn rate condition which describes how quickly we burn through our error budget. Then it should be specified how long we look back to determine if we breach and how long our defined condition should hold until we start alerting. This is to make the alerting more flexible and less strict and enable the software architect to specify precise alerting behaviour.

The last part of the alerting is the alert notification target. This entity is to separate the target description from the description of the policy and the condition. The alert notification target describes how who we reach in the specified cases in the alert policy. The alert notification target also has the name and display name scheme as the other entities mentioned above. The target field should be used to specify how we reach the target. Examples could be email-address or push messages. The description should be used to give additional information on the target. Maybe what his role is e.g. (Site Reliability Engineer).

The above-mentioned model and its entities should be displayed in a human-readable form. It should ideally be accessed through an editor to enable humans to describe them and get support. This is for the sole reason that these architectures are designed by people. The target group are software architects because they are concerned with the design and development of software architecture and its corresponding artefacts.

### 3.3 SLO Integration into Interface Description Language

This section describes how the SLO should be integrated into the Interface Description Language (IDL) which we generated with the extended DSL.

IDLs are used to enable programs to get information on how together data other programs by consuming this description.

IDLs achieve this by communicating the different interfaces of the software with each other.

These interface descriptions consist of a description of an API provider. Its purpose is to provide infrastructure to make the API accessible from outside the software system. The API has endpoints with which the user can interact to get data. Each of these endpoints offers different operations to get data, add additional data, edit existing data, or delete data.

Because the user mostly interacts with these endpoints we will integrate the SLO at the endpoint level. So the software architect can specify for each endpoint the SLO characteristics and is not bound to an SLO because another endpoint from the same provider uses it. We choose here to extend such a language for service level objectives because currently, they allow only the modelling of the functional contract of the architecture but the non-functional contract is missing.

## 3.4 SLO Generation in OpenSLO

In this section, we will explain how we can integrate a SLO into the OpenSLO spec.

OpenSLO offers here to possibilities. We can use each entity of the in Section 3.2 and generate a YAML document of it. This would lead to a lot of small documents. Additionally, because OpenSLO only requires objects to be referenced only by name there is no information on the location of the file in which this object might be defined. This would lead to a really extensive search if one wants to find a specific object referenced in another file. So we choose the second approach an example of it is depicted in Listing 3.1. We can use a single YAML file in which we

---

**Listing 3.1** Example of multiple YAML objects in a single document.

---

```
apiVersion: openslo/v1
kind: Service
metadata:
  name: MonitoringService
  displayName: Default Monitoring Service
spec:
  description: All SLOs associated with this service are for monitoring the t2-project
    application
---
apiVersion: openslo/v1
kind: DataSource
metadata:
  name: TeaShopPrometheusServer
  displayName: T2 Project Prometheus Main Server
spec:
  description: Tea Shop Prometheus Main Server
  type: Prometheus
  connectionDetails:
    url: http://www.t2-project.de/prometeus/api/v1
```

---

all write the different YAML objects and separate the different documents with a new line and three dashes. This mechanism is documented in the YAML spec section 2.2 [Net]. This allows grouping all necessary entities of an OpenSLO model into one file and excluding the entities which are not used by this SLO. Resulting in one clear document with all the necessary information easily accessible to a software architect or a developer.

## 3.5 Usage of OpenSLO

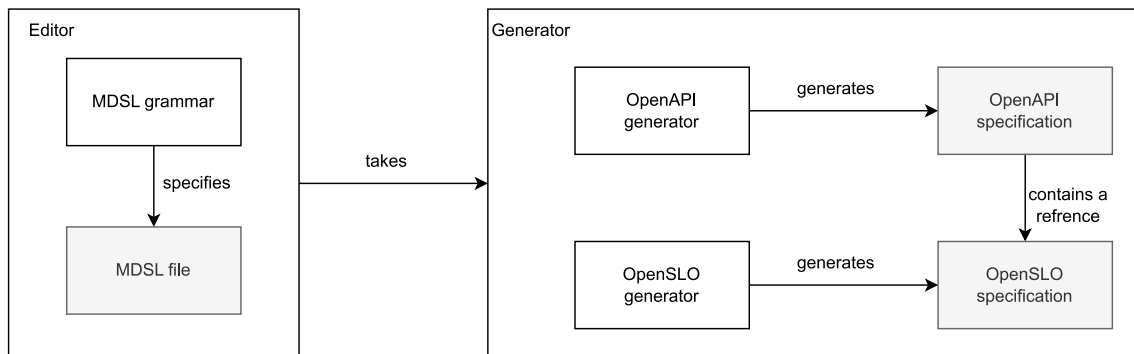
This section covers the last part of our concept and, therefore, closes this chapter. The generated extended interface description and the OpenSLO specification can now be used in additional generators to generate server implementations in different programming languages and libraries and configurations for different monitoring tools and systems. Lastly, the configurations can link these two and generate so a fully functional interface and the configured monitoring system which checks the defined SLO and alerts the responsible person in case the SLO is breached. Note that the parts described in this section are not covered by the following implementation chapter. The reason for this is the limited scope of this thesis.

## 4 Architecture and Implementation

In this chapter the implementation of the concept and how the extension of MDSL is described. We start with the architecture of the extension in Section 4.1. In section Section 4.2 we describe the technical implementaion of the SLA model already present in MDSL, how we extended it, and what the limitations of this implementation are. Following that, in Section 4.3 we describe how we extended the model of MDSL to accommodate the SLO model of the OpenSLO specification and discuss the implementational details. The extensions of the generators is presented in Section 4.3.2. Additional features we implemented are discussed and shown in Section 4.4.

### 4.1 Architecture

The architecture is depicted in Figure 4.1.



**Figure 4.1:** Architecture of our extension

### 4.2 Implementation of built-in SLOs

In this section, we describe the SLA model that MDSL already brought in Section 4.2.1 and how we extended the OpenAPI generator for this model in Section 4.2.2.

#### 4.2.1 SLA model in MDSL

We started the implementation with an investigation of what MDSL already offers in terms of SLOs. We found that MDSL has already defined a model for SLAs. The model of SLA contains also SLOs. An example SLA model in MDSL is shown in Figure 4.2.

```
SLA template ExampleSLA
  type QUALITATIVE
  objective MaxUptime
  "Maximum uptime the service should have in one hour measured in minutes"
    minimal 55 minutes
    target 59 minutes
    optimal 60 minutes
  penalty "20% reduction on service fees"
  notification "Site reliability engineer"
  rate plan SUBSCRIPTION
  rate limit MAX_CALLS 6000
  within 2 hours
```

**Figure 4.2:** Example of an SLA in MDSL.

We begin an SLA with the two keywords SLA and template. Followed by the name of the template, in this case, ExampleSLA. This name is used when we want to reference this template elsewhere in the specification. This is shown later. Then comes the optional type of the SLA we can choose from the following options. NONE, IMPLICIT, INFORMAL, QUALITATIVE, QUANTITATIVE. Then we have to define the SLO. This is done via the objective keyword followed by the name of this objective. To specify the objective we must use a string which describes the goal of the objective. Followed by a measurement. The measurement provided is called landing zone. It consists of a minimum of two measurements minimal and target. And an optional third one called optimal. Each of these measurements consists of an integer number and a duration shorthand. The shorthand can be if we want to measure time hours minutes, or seconds. We define here as an example the following SLO to show the structure of our implementation. We want the service which uses this SLA to be available at least 55 minutes of each hour. The target is 59 minutes for each hour and optional describes the best case of 60 minutes per hour which is equivalent to availability of 100 per cent. We can also choose to only give a simple measurement instead of a landing Zone in this case we only provide a measurement after the quality sting. The example is depicted in Figure 4.3. Here the SLO requires an uptime of at least 59 minutes per hour and is there stricter than the landing zone example. We can then give a penalty if the SLO is not met. This is also done via one string which describes the penalty. In this case a 20 percent reduction on service fees. Followed by a String which describes whom to reach in case of breaching. We can then specify in the SLA the rate plan which specifies how to the customer pays for the service. Here we can select from three options. FREEMIUM, SUBSCRIPTION or USAGE\_BASED. In this example we choose SUBSCRIPTION. The last part of the SLA is a rate limit which defines how extensively we can use the service. It starts with the keyword rate limit then we have to select how we want to define the limit. First, we have to select the type of limit having two options MAX\_CALLS and DATA\_QUOTA. It is followed by an integer number to quantify the limit and an optional shorthand to describe the unit we want to measure. Lastly, we have to specify the time frame in which this limit applies. Here we use the within keyword followed by a number and unit. In the example, we choose a limit of six thousand calls every two hours.

Now that we have defined an SLA we can reference it in the MDSL description. For that, we have two points where we can define an SLA on the provider level which offers the service or per endpoint. An example of a provider which uses our SLA is seen in Figure 4.4 and an example of an endpoint which uses our SLA is shown in Figure 4.5.

```

SLA template ExampleSLA
  type QUALITATIVE
  objective MaxUptime
  "Maximum uptime the service should have in one hour measured in minutes"
  59 minutes
  penalty "20% reduction on service fees"
  notification "Site reliability engineer"
  rate plan SUBSCRIPTION
  rate limit MAX_CALLS 6000
  within 2 hours

```

**Figure 4.3:** SLA which uses a simple measurement instead of a landing zone.

```

API provider InventoryServiceProvider
  offers InventoryEndpoint
  at endpoint location "http://t2-project/api"
  via protocol HTTP
  binding resource ProductResource at "/inventory{id}"
  operation addProduct to POST
  operation addProduct to PUT
  operation deleteProduct to DELETE
  operation getProduct to GET

  offers RestockEndpoint
  at endpoint location "http://t2-project/api"
  via protocol HTTP
  binding resource RestockResource at "/restock"
  operation restocProducts to GET

  offers GenerationEndpoint
  at endpoint location "http://t2-project/api"
  via protocol HTTP
  binding resource GenerateResource at "/generate"
  operation generate to GET
  with endpoint SLA ExampleSLA

```

IPA

**Figure 4.4:** Endpoint which uses our SLA.

```
API provider InventoryServiceProvider
  offers InventoryEndpoint
  at endpoint location "http://t2-project/api"
  via protocol HTTP
  binding resource ProductResource at "/inventory{id}"
  operation addProduct to POST
  operation addProduct to PUT
  operation deleteProduct to DELETE
  operation getProduct to GET

  offers RestockEndpoint
  at endpoint location "http://t2-project/api"
  via protocol HTTP
  binding resource RestockResource at "/restock"
  operation restockProducts to GET

  offers GenerationEndpoint
  at endpoint location "http://t2-project/api"
  via protocol HTTP
  binding resource GenerateResource at "/generate"
  operation generate to GET
  with provider SLA ExampleSLA
IPA
```

**Figure 4.5:** Provider which uses our SLA.

In both figures, you see the same API provider. It models the Inventory Service of the T2-Project developed by Speth et al. [SSB22]. In the last line, you can see we once specified the SLA specifically for the generation endpoint and in the other case for the whole provider means all endpoints of this provider must meet this SLA.

After we familiarized ourselves with the model we checked if any of the implemented generators used this part of the model. We found out that none of the generators used this model so we choose to extend one generator to familiarize ourselves with the capabilities of Xtext and possible extension mechanisms.

#### 4.2.2 Extension of the Generator for preexisting SLAs

We choose to extend the OpenAPI generator because OpenAPI offers code generators for server stubs and other tools beneficial to developing APIs and microservice architectures.

It also offers benefits with its YAML format because:

- is Open Source and offers lots of preexisting works and tools



- is human readable making editing and checking the model for software developers and architects easy. This also makes it user-friendly to be used by version control systems like GitHub
- YAML offers integrations in the most popular languages making it feasible to be used by generators to get new artefacts.

It also offers an extension mechanism for SLA which we used. The mechanism is proposed by Frendandez et al. by the isa group of the University of Seville [Fre]. To extend the OpenAPI specification they extended the info object of the specification with a custom node called x-sla. Note the x- is needed to indicate that it is a custom node not covered by the specification jet. This allows tools, which use these specifications, to ignore these elements and function properly with the extension. Under this node, they use the \$ref key to reference the document in which the SLO specification is stored. So instead of one document, we get one with the OpenAPI specification and a reference to another document which contains the specification of the SLA.

We then extended the generator for OpenAPI and wrote a new one for the SLA model and integrated them into the tooling. When you now generate an OpenAPI specification with MDSL and a SLA is present the OpenAPI document with the reference and a second document will be generated in the same location which contains the SLA specification. As a language for the SLA specification, we also choose YAML for the benefits mentioned above and because the OpenAPI generator uses the YAML format for the OpenAPI specification so it seems unnecessarily complicated to use another format for the extension.

We take the examples in Figure 4.2 and Figure 4.4 and generate the OpenAPI is shown in Listing 4.1 and the referenced SLA file is shown Listing 4.2.

---

**Listing 4.1** Generated OpenAPI description from Figure 4.4.

---

```

openapi: 3.0.1
info:
  title: t2InventoryService
  version: "1.0"
  x-generated-on: 2023-02-15T17:38:42.0456589
  x-external-sla-file:
    $ref: ./t2-inventory-service-sla.yaml
servers:
- url: http://t2-project/api
- name: InventoryServiceProvider-ProductResource
externalDocs:
  description: InventoryEndpoint contract, Information Holder Resource role
  url: https://microservice-api-patterns.org/patterns/responsibility/endpointRoles/InformationHolderResource.html
  x-external-endpoint-slas:
  - ExampleSLA

```

---

Note here depicted is only a sniped for brifety.

You can see in Listing 4.1 that we have generated a node in the info object called x-external-sla-file. There you can see the reference to the SLA file in Listing 4.2, and for each endpoint, we have an additional node called x-external-endpoint-sla here we reference the name of the SLA used form the SLA file for this endpoint. Under this node, all external endpoint SLAss are listed.

---

**Listing 4.2** Generated SLA description from Figure 4.2.

---

```
sla-doc-version: "1.0"
sla-templates:
- name: ExampleSLA
  slas:
  - type: QUALITATIVE
    slos:
    - name: MaxUptime
      qualityGoal: Maximum uptime the service should have in one hour measured in
        minutes
      measurement:
        value: 59
        unit: minutes
      penalty: 20% reduction on service fees
      notification: Site reliability engineer
      ratePlan: SUBSCRIPTION
      rate limits:
      - rateLimit: MAX_CALLS
        measurement:
          value: 6000
        interval:
          value: 2
          unit: hours
```

---

If we define a provider-wide SLA each endpoint will get a reference to this endpoint under this node. We also allow the inlining of SLAs this will then lead to that under the specific endpoint which uses this SLA a node will be created named `x-internal-endpoint-sla` and under this node, the specification of the SLA will be shown. An example of this is shown in Listing 4.3. If we inline a provider SLA it will be inlined for each endpoint under their `x-internal-endpoint-slas` node.

We have seen what with this implementation is possible. we will talk now about the limitations of the model itself.

### 4.2.3 Limitaitons of the SLA model

Here want to address some of the limitations of the SLA model present in MDSL. The first limitation is that the model only allows things linge SLO and the notification to be modelled as strings. This makes it complicated to formulate precise SLOs. It puts all the responsibility on the architect and offers little support in designing them. Also, the current support measurements for the SLO are not sufficient measurements like ratios are not possible at the moment. Also, we have no possibility to specify from which source the SLO gets its to compare against a SLO. It also follows no official standard and is only used in MDSL.

All these limitations make it hard to build further generators upon them to for example generate configurations for monitoring systems. For this reason, we searched for a standardized model which addressed these limitations.

**Listing 4.3** Inlined SLA in a OpenAPI specifcaiton.

---

```

openapi: 3.0.1
info:
  title: t2InventoryService
  version: "1.0"
  x-generated-on: 2023-02-15T17:32:57.727828
servers:
- url: http://t2-project/api
tags:
- name: InventoryServiceProvider-ProductResource
externalDocs:
  description: InventoryEndpoint contract, Information Holder Resource role
  URL: [...]
x-internal-endpoint-slas:
- type: QUALITATIVE
  slos:
- name: MaxUptime
  qualityGoal: Maximum uptime the service should have in one hour measured
    in minutes
  measurement:
    value: 59
    unit: minutes
  penalty: 20% reduction on service fees
  notification: Site reliability engineer
  ratePlan: SUBSCRIPTION
  rate limits:
- rateLimit: MAX_CALLS
  measurement:
    value: 6000
  interval:
    value: 2
    unit: hours
- name: InventoryServiceProvider-RestockResource
- name: InventoryServiceProvider-GenerateResource
paths:
components:

```

---

## 4.3 Adding OpenSLO to MDSL

This section deals with the integration of the OpenSLO model into MDSL, this is covered in Section 4.3.1, and how the OpenAPI generator was extended is explained in Section 4.3.2.

### 4.3.1 Extending the MDSL model for Open Service Level Objectives

OpenSLO offers a more extensive model than the current present in MDSL. It offers the definition of the data source to specify where we take our data from to compare it to a SLO. It also includes metrics that can specify more detailed service-level objectives. It also includes extension mechanisms that future work can use to improve the model further or integrate new technologies or vendors.

OpenSLO

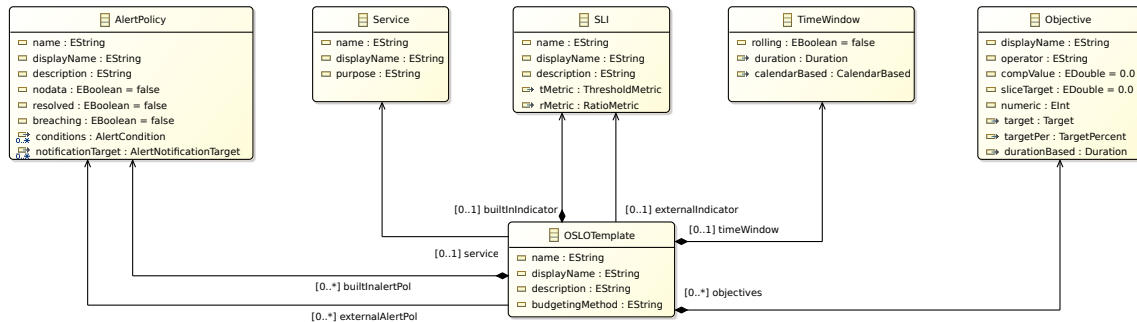


Figure 4.6: Service Level Objective model.

An overview of the OpenSLO model is depicted in Figure 4.6. An OpenSLO consists of the OSLO template. It houses all the different components of the model. The SLI defines the Service Level Indicator which describes how to measure the system’s state and how to calculate the metric which can then be compared against the objective. The objective describes the quality goal the SLO wants to achieve. The time window specifies in which time frame the SLO will be evaluated. The service is used to group different SLOs together. The Alert Policy describes what happens when the SLO breaches and whom to alert when this happens. We will now go through each of these components with an example to explain the model and show the excerpts from the Xtext grammar to visualise how its implemented.

Starting with the OpenSLO Template you see an example of this depicted in Figure 4.7. In Figure 4.8 you can see the corresponding Xtext grammar which defines this object.

```

OpenSLOTemplate BasicSLO displayed as "Basic Service Level Indicator"
  part of MonitoringService
  indicator SiteAvailabilityIndicator
  in time-window 30d
  starts at 2023-02-05 23:59:59
  in timeZone "Berlin/Europe"
  budgeted Timeslices

  with objectives
    displayed as "Site availability"
    with target percent 99
    with SliceTarget 90
    with SliceWindow 2h
    with Alert Policies BasicAlert
    
```

Figure 4.7: Examle SLO in MDSL .

```

OSLOTemplate:
  'OpenSLOTemplate'
  name=sName ('displayed' 'as' displayName=STRING)?
  ('description' description=STRING)?
  'part' 'of' service=[Service]
  'indicator' (builtInIndicator=SLI | externalIndicator=[SLI])
  'in' 'time-window' timeWindow=TimeWindow
  'budgeted' budgetingMethod=BudgetingMethod
  'with' 'objectives' objectives+=Objective
  'with' 'Alert' 'Policies' (builtInAlertPol+=AlertPolicy | externalAlertPol+=[AlertPolicy]);

```

**Figure 4.8:** Xtext grammar of the OpenSLO template.

This is a more refined version of the SLA shown earlier the goal of both is the same to have site reliability of 99 percent.

We start the definition of an OpenSLO template with the keyword `OpenSLOTemplate` indicating that we want to reference it later in other objects like Providers or Endpoints. This mechanism is shown later. We have then given a name for the OpenSLO template in this example `BasicSLO`. We then can give an optional Name which is used to display it. Here we choose the fully written out name "Basic Service Level Indicator". The optionality is indicated in the grammar by enclosing the blue keywords `displayed` as `and` the `displayName` attribute in brackets and making this section optional by annotating it with a `?`. This can also be seen in the next attribute in the grammar we define an optional description string. In this example, we not have provided a description. After that, we see how we can realize references in Xtext we define an attribute in this case called `service` and give him the name of an object in Xtext. In this case, we want a service object. In the example, you can see this we provided after the part of keywords we only provide the name of a Service object, in this case, `MonitoringService`. The Service object will be discussed later. The service follows the service level indicator object. Here we have two choices we can provide a reference to an existing object. This is shown in our example where we reference the `SiteAvailabilityIndicator`. Or we can inline the object. So we can choose if we want the object only one time or else make it referenceable and use it more times. We only allow one SLI per SLO. Then follow the time window which describes in which time frame the SLO will be evaluated. The Xtext grammar of the time window is shown in Figure 4.9.

```

TimeWindow:
  duration=Duration
  (calendarBased=CalendarBased | 'is' 'rolling' rolling=Boolean);

CalendarBased:
  'starts' 'at' startDate=DATE
  'in' 'timeZone' timeZone=STRING;

```

**Figure 4.9:** Xtext grammar of the time window object.

First, we have to specify the duration of the time window. The duration can be seen in Figure 4.10.

Duration:

```
value=INT
unitOfMeasure=('m' | 'h' | 'd' | 'w' | 'M' | 'Q' | 'Y');
```

**Figure 4.10:** Xtext grammar of the Duration object.

It consists of a number which quantifies the time window followed by a duration-shorthand. Currently, we support the following shorthands: m for minutes, h for hours, d for days, w for weeks, M for months, Q for quarters and Y for years. In our example, we selected 30d meaning thirty days. We then have the choice between a rolling time window and a fixed start time. In the example we choose the latter: then we have to give a start date and time in the following format: YYYY-MM-DD hh:MM:ss. and a string indicating in which time zone we want to evaluate the timestamp. In our case, the time window starts on the 5th of February 2023 and last 30 days from then. If we want to evaluate the SLO every 30 days in the past we can choose a rolling window in this case we set is rolling to true. In Figure 5.3 you see an excerpt of the same SLO in Figure 4.7 with such a time window.

```
OpenSLOTemplate BasicSLO displayed as "Basic Service Level Indicator"
  part of MonitoringService
  indicator SiteAvailabilityIndicator
  in time-window 30d
  is rolling true
  budgeted Timeslices
```

**Figure 4.11:** Example Service Level Objective with a rolling time window.

The next part of the main model of the Open SLO is the budgeting method. This determines the method with which the error budget will be calculated. We have currently three possibilities to specify the budgeting method: occurrences, timeslices, and ratiotimeslices. These are to make the SLO a bit more fine-grained. In the example, you see we selected timeslices. The budgeting methods will be explained with the objectives object because in it we specify the values for the budgeting method. This in combination with the objective will be used to evaluate the SLO and calculate the error budget.

The penultimate attribute of the OpenSLO model is the objective it specifies the objective to which the data from the SLI will be measured against. The last attribute is the Alert Policy it can also be inlined or referenced. A SLO can have more than one Alert Policy.

This completes the OpenSLO model we will now go through each of the referenced objects and explain their purpose and implementation.

## Service

The first object we referenced in the Open SLO template was a service. He is used to grouping different SLOs together. Each SLO refreezes the service object to which it should belong. The excerpt from the Xtext grammar in Figure 4.12 shows the structure of the service object.

Service:

```
'Service'
name=sName ('displayed' 'as' displayName=STRING)?
('purpose' purpose=STRING)?;
```

**Figure 4.12:** Xtext grammar of the Service object.

It consists of a name and a display name. and a short description which defines the group. The service used in the example is shown in Figure 4.13.

```
Service MonitoringService displayed as "Standard Monitoring Service"
  purpose "All SLOs associated with this service are for monitoring the t2-project inventory API"
```

**Figure 4.13:** Service used in our example.

In our example went to group all SLOs which are monitoring the inventory API of our application. He should be displayed by his name John Miller.

## Objective

In Figure 4.7 you can see the objective object it starts with the keywords with objectives. The Xtext grammar for these objects can be seen in Figure 4.14.

```
Objective:
('displayed' 'as' displayName=STRING)?
('with' 'operator' operator=OPERATOR compValue=Double)?
('with' target=Target | 'with' targetPer=TargetPercent)
('with' 'SliceTarget' sliceTarget=Double)?
('with' 'SliceWindow' (numeric=INT | durationBased=Duration))?;
```

**Figure 4.14:** Xtext grammar of the objective object.

The objective also has an optional display name we named it here “Site Availability” to indicate that this is our goal with this objective. We have then to methods to specify how we want to compare the values from the SLI against our goal. We can do this by giving a percent value then we use the keywords target percent and give a number from 0.0 to 100 to indicate the percentage. If we want to give the percentage as a decimal we use the keyword target and provide a number between 0.0 and 1.0. For this to work properly, we have to use a ratio metric in the SLI object. If our SLI outputs only raw values we have to give an operator shorthand possible are: gt - for greater than, gte for greater than equal, lt - for less than, or lte - for less than equal, followed by a number to compare against instead of percentages In our example, we choose to measure the site reliability and set a goal of 99 percent. Now we can explain the budgeting method mentioned earlier. There are three possibilities occurrences timeslices ratio timeslices. The functionality of these different budgeting methods is described in Section 2.1.3 in Chapter 2. In our example, we divide the 30-day period

into slices of 2 hours. For each of these slices, we measure if our service has an availability of 90 percent because we specified the slice target to be 90. This can be seen as a micro service level objective defined for each timeslice. Then 99 percent of all timeslices in this 30 days must meet this objective to meet the whole SLO. The Ratio timeslices will build an average overall timeslice's success ratios and compare this against the SLO. timeslices and ratio timelines are more flexible and allow the SLO to define a threshold. This also leads to less frequent alerting.

### Service Level Indicator

The next big part of the OpenSLO model is the Service Level Indicator its purpose is to specify how the state of the service will be measured and calculated. In Figure 4.15 you see the Service Level indicator used in our example. The corresponding Xtext grammar for Service Level Indicators is depicted in Figure 4.16.

```
Service Level Indicator SiteAvailabilityIndicator displayed as "Availability"
  description "Indicator describes availability of the t2-project application"
  with Ratio Metric
    counter true
    total Metric Source
      with Data Source T2PrometheusServer
      type "Prometheus"
      with spec "http_requests_total"
    good Metric Source
      with Data Source T2PrometheusServer
      type "Prometheus"
      with spec "http_requests_total{satus_code=\"200\"}"
```

**Figure 4.15:** Service Level Indicator used in our example.

The service level indicator has at the start a similar design to the OpenSLO template. Starting with his name to make it referenceable. Note that this is the name we used in the OpenSLO template object depicted in Figure 4.6. Then we used the optional display name, in this case, “Availability”. When then also provided a little description to clarify what this SLI is about. The last part of the SLI is a metric object. Currently, we support two types of metrics a Threshold Metric and a Ratio Metric. This is due to the fact that OpenSLO currently supports only these two metrics. New metrics could extend the model and this implementation by defining new rules in the grammar for them and integrating them in this or- a condition seen in Figure 4.16.

SLI:

```
'Service' 'Level' 'Indicator'
name=sName ('displayed' 'as' displayName=STRING)?
('description' description=STRING)?
'with' (tMetric=ThresholdMetric | rMetric=RatioMetric);
```

**Figure 4.16:** Xtext grammar of the Service Level Indicator.

In our example, we used a ratio metric. The Xtext grammar of such a metric can be seen in Figure 4.17



```

RatioMetric:
  'Ratio' 'Metric'
  'counter' counter=Boolean
  ('total' total=MetricSource
  ('good' good=MetricSource | 'bad' bad=MetricSource)) | 'raw' raw=MetricSource
  ('with' 'raw' 'type' rType=('success' | 'failure'))?;

```

**Figure 4.17:** Xtext grammar of the Ratio Metric.

In the Xtext grammar, we see the first attribute of our ratio metric is a counter. It can be set to either true or false, indicating the metric is monotonically increasing. This is true for our case because we measure the total incoming requests and they can not decrease in a given time frame. Then we have to provide a combination of metric sources and objects which defines the ratio we want to measure. All allowed combinations and their meanings are also described in Section 2.1.3 in Chapter 2. In Figure 4.18 you see the Xtext grammar of the Metric source object. This object is used to specify how we get the numbers for our metric.

```

MetricSource:
  'Metric' 'Source'
  ('with' 'Data' 'Source' dataSource=[DataSource])?
  ('type' type=STRING)?
  'with' 'spec' spec=STRING
;

```

**Figure 4.18:** Xtext grammar of the Metric Source

This object only consists of an optional data source object which specifies how we connect to our resource. The type string is used to specify what kind of source it is. In our implementation, we currently only support Prometheus servers. In the spec string, we have to give a PromQL query to retrieve data from a Prometheus server. The server connection will be specified in the Data Source object which will be referenced in the Metric Source. The grammar of such a data source is depicted in Figure 4.19.

```

DataSource:
  'Data' 'Source' name=sName ('displayed' 'as' displayName=STRING)?
  ('description' description=STRING)?
  'type' type=STRING
  'connection' 'details' 'URL' url=STRING;

```

**Figure 4.19:** Xtext grammar of the Data Source

The data source has also a name to make it referenceable an optional display name, and a description. Like in the Metric Source, we have here also to specify the type source as a string. Under the connection details, like server address and keys for authentication. Also here we currently only

support Prometheus servers for that reason we have to provide Uniform Resource Locator (URL) to Prometheus HTTP API. In combination with the metric source, we can then get data from the Prometheus servers.

In Figure 4.20 you see the data source used in the SLI used in our example.

```
Data Source T2PrometheusServer displayed as "T2 Project Prometheus Main Server"  
description "T2 project Prometheus main Server"  
type "Prometheus"  
connection details URL "http://www.t2-project.de/prometheus/api/v1"
```

**Figure 4.20:** Data Source used in our example.

Here you see we defined as type Prometheus and provided a URL to a Prometheus server. This example is also inspired by the T2 Project developed by Speth et al. [SSB22]. In the SLI Figure 4.15 you see that we then provided in both metric sources the same server. The difference between the two metric sources is only the query in total we want to get the total HTTP requests to our service. In the good, we want only the amount of requests which we answered with a 200 HTTP response code. With this ratio, we calculate how available our service is for each time slice.

We also can use threshold metrics to define our SLI. The Xtext grammar of such a metric is shown in Figure 4.21.

**ThresholdMetric:**

```
'Threshold' 'Metric' 'from' metricSource=MetricSource  
;
```

**Figure 4.21:** Xtext grammar of Threshold Metrics.

In this case, we only have to provide one metric source which defines which value we want to compare against our threshold. This threshold is then defined in the objective object of our SLO where we specify the threshold the operator used for comparison seen in the grammar in Figure 4.14.

The percentage of the metric will then be calculated by taking the obtained data points from the data source compare them against our threshold. If we meet the condition the data point will be counted as good in the other case it will be counted bad. The ratio will then be calculated as good data points against all data points and this ratio must meet our target percentage specified in the SLO.

### **Alert Policy**

The last big part of our SLO model deals with alerting. What happens when we begin to breach an SLO, how do we define breaching, and whom to contact if that happens?

This is covered in the alert policy object. The grammar defining such objects are depicted in Figure 4.22.

```
AlertPolicy:
  'Alert' 'Policy'
  name=sName ('displayed' 'as' displayName=STRING)?
  ('description' description=STRING)?
  'alert' 'when' 'no' 'data' nodata=Boolean
  'alert' 'when' 'resolved' resolved=Boolean
  'alert' 'when' 'breaching' breaching=Boolean
  'conditions' conditions+=AlertCondition
  'notifies' notificationTarget+=[AlertNotificationTarget];
```

**Figure 4.22:** Xtext grammar of Alert Policies

The alert policy is also referenceable and has a display name and description like some of the other objects mentioned above. Then we have to give three boolean values the keywords indicate when we want to be alerted. We can choose to be alerted when no data arrives when the SLO breaches. Breaching will be defined in the inlined Alert Condition. and lastly when the breach is resolved.

In the example, in Figure 4.23 you see that after the keywords we simply provided boolean values. In this case, want to be alerted in all three cases. To further specify how we define breaching we have to give at least one alert condition this is indicated by the grammar by the += operator after the conditions attribute. After the condition comes at least one reference to an Alert Notification Target Object which specifies who we want to contact and how we reach him.

```
Alert Policy BasicAlert
  alert when no data true
  alert when resolved true
  alert when breaching true
  conditions Alert Condition SlightlyBelowTarget
  severity "page"
  condition
    kind "burnrate"
    operator gt
    with threshold 1
    look back 30d
    alert after 6h
  notifies sreEngineer
```

**Figure 4.23:** Alert Policy used in our example

Starting with the objectives object you can see the grammar of this object in Figure 4.24. We have first to provide a String which indicates what kind of condition we want to specify the alerting. OpenSLO currently only supports burnrates as alert conditions. Burnrates describe how quickly we burn through our error budget. An error budget is the number of errors allowed to make and still

meet our SLO. We have then to specify the operator for comparing the threshold. The threshold is an integer value which describes how quickly we are allowed to burn through our budget 1 means we burn through it according to plan. Two means we burn twice as fast through it as we should. And so on. We then have to give the window on which we want to look back to calculate our burnrate. Lastly, we have to provide a shorthand that indicates how long this condition must hold to start alerting. Our example here specifies a burnrate condition and when we burn faster through our error budget as expected for 6 hours we start alerting. This is evaluated in a 3-day time window. We must provide at least one. We can also provide more than one so we can notify whole groups of people.

### Condition:

```
'kind' kind = STRING
'operator' operator=OPERATOR
'with' 'threshold' threshold=INT
'look' 'back' lookBack=Duration
'alert' 'after' alertAfter=Duration;
```

**Figure 4.24:** Xtext grammar of the Alert Condition

The last part of our alert policy is the alert notification target. The structure of such a model is depicted in Figure 4.25.

### AlertNotificationTarget:

```
'Alert' 'Notification' 'Target'
name=sName ('displayed' 'as' displayName=STRING)?
'target' target=STRING
('description' description=STRING)?
;
```

**Figure 4.25:** Xtext grammar of the Alert Notification Target

An alert notification target starts with the three keywords alert notification target. Then we must specify a name, under this name we can reference the alert notification target in the alert policy. We can specify a display name. We use the text after the target keyword to describe how we can achieve the target. In our example, we have specified that we send a push message to the target. We can add an optional description to describe the object in more detail.

In our example in Figure 4.26 we specified we notify the responsible site reliability engineer. He will be displayed by his name John Miller. and in the description, we explain why we contact him.

---

```
Alert Notification Target sreEngineer displayed as "John Miller"
  target "push message"
  description "John Miller is the site reliability engineer responsible for this service"
```

**Figure 4.26:** Alert Notification Target used in our example

### 4.3.2 Extending the OpenAPI Generator

We extend the generator with the mechanism explained in the concept chapter in Section 3.4. For each SLO we use in our MDSL document we will create a YAML file. In this file, all used elements by this SLO are stored as separate YAML documents separated by a new line and three dashes. We choose this approach because having only one central document for each SLO makes it easier to feed these specifications into other generators, compared to having more small documents. The generated document of our example can be seen in the Appendix A.2.2. We integrated this document into the OpenAPI document with a custom node we placed in the info object. The node is called `x-openslo` under it we store a list of all used OpenSLO files. The `x-` indicates that this is a custom node inserted into the specification. This is necessary to ensure that generators that work with standard OpenAPI specifications still function properly. The info object of the generated OpenAPI specification is seen in Listing 4.4.

---

**Listing 4.4** Info object in the OpenAPI document.

---

```
info:
  title: ExampleService
  version: "1.0"
  x-generated-on: 2023-02-17T20:04:25.4707732
  x-openslo:
    - example-service-openslo-BasicSLO.yaml
```

---

## 4.4 Document Validation

OpenSLO specifies for some attributes additional semantic constraints that we can't cover with the grammar alone. Xtext offers the possibility to define additional static validation rules for the different objects. These are supported by the Eclipse editor. If a document violates one of these rules, an error message is displayed and the part of the document that violates the rule is underlined in red and thus highlighted. This works the same way as static semantics checking in popular programming languages. We use this mechanism to implement these constraints

OpenSLO dictates that each name can be a maximum of 255 characters long and a description should have a maximum length of 1050 characters. We implemented this check for each object in the OpenSLO model. For the data source object, we are checking if the URL provided is a valid HTTP or HTTPS URL. For the service level object, we check if an operator and a threshold are provided if in the referenced service level indicator object a threshold metric is defined. This is necessary because ratio metrics do not work with thresholds and the defined SLO would not function properly. The last check we implemented regards the percentage values we give the SLO.

**connection details URL** ["www.example.com/prometheus/api/v1"](http://www.example.com/prometheus/api/v1)



**Figure 4.27:** Error message display when entering an invalid URL

We have the option to provide this as a percentage from 0 to 100 or as a double value from 0,0 to 1,0. Because Xtext does not allow the specification of range for attributes in the grammar we covered this in a validation check.

In Figure 4.27 you can see what such a validation looks like. Here we entered an invalid URL in the data source object.

## 5 Evaluation

This chapter describes the last step in this thesis the evaluation of the build extension. For the evaluation of the thesis, an experiment was carried out. We describe the setup of the experiment in Section 5.1. The results are presented in Section 5.2 and discussed in Section 5.3. The last part of this chapter focuses on the threats to validity Section 5.4.

### 5.1 Experiment

To evaluate the extension an experiment was conducted. We designed an experiment consisting of a scenario in which the participant had to model an SLO using our extension. We then designed and questionnaire to gather feedback regarding our extension. We describe in Section 5.1.1 the experimental setup and in Section 5.1.2 the design of the questionnaire.

#### 5.1.1 Experiment design

For our experiment, we modelled the InventoryAPI of the T2-Project by Speth et al. [SSB22] using our extended version of MDSL. The inventory service consists of an HTTP API which offers basic functions like restocking the inventory, reserving items, or getting single items. This is facilitated through different endpoints and HTTP verbs. The MDSL file provided can be seen in Appendix A.2.

We also modelled an SLO for the experiment to be integrated into the description of the inventory service. The description of the SLO can be seen Appendix A.2.2

The experiments were conducted online using Microsoft Teams. The participants were given a standalone Eclipse IDE. With a project containing the API description of the inventory service and a description of the SLO. We then introduced the participants to MDSL and our extension with an example and explained the different parts of the model and their purpose. We then explained the different parts of the provided SLO and let the participants model the SLO, and test the extension. During that, we were there to ask questions if the participant had any. After that, we gave them a questionnaire online via Google Forms<sup>1</sup> and let them fill it out. The questionnaire covered questions about the user experience with our extension and some to get some general feedback.

---

<sup>1</sup><https://www.google.de/intl/de/forms/about/>

### 5.1.2 Goal Question Metric approach

The questions were developed using the Goal Question Metric (GQM) approach by Basili et al. [BCR94]. The first step in the Goal Question metric approach is to define one or more goals which characterize what this thesis wants to achieve. The goal of this thesis is to allow software architects to model not only functional properties of microservice architecture like interface descriptions but also quality characteristics like service level objectives. This led to the one and only goal of this thesis.

**G1:** Improve the modelling capabilities of a DSL for microservices architectures to also support quality characteristics like service level objectives for software architects and developers.

The second step of the Goal Question metric approach is to define questions to measure if the goal was achieved and how well. A goal consists of three parts, the issue itself here is the limited capabilities of domain-specific languages modelling microservice architectures. Secondly the process we want to enable in this thesis, here the modelling of quality characteristics like service level objectives with such domain-specific languages. Lastly, the person's view from which this goal must be fulfilled. In our case software architects and software developers. All these things considered we came up with the following questions:

**Q1:** How user friendly would you rate the MDSL SLO extension? (1-5 1 Not user-friendly at all, 5 Very user-friendly)

**Q2:** How quickly were you able to model the given scenario? (1-5 1 Very slow, 5 Very fast)

**Q3:** How big would you estimate the time saving enabled by the MDSL, SLO extension? (1-5 1 Very low, 5 Very high)

**Q4:** What was particularly good using the MDSL SLO extension?

**Q5:** What was particularly bad using the MDSL SLO extension?

**Q6:** Where features missing you would expect?

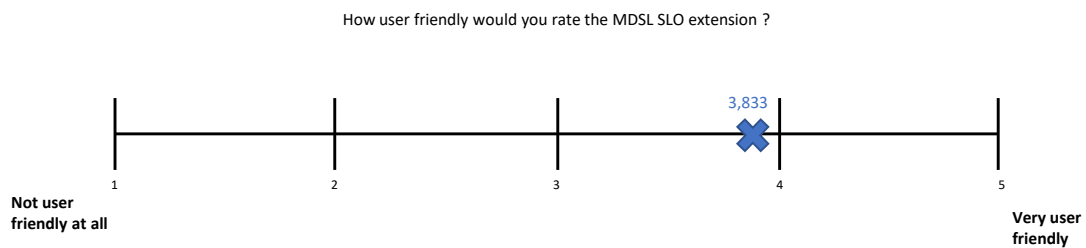
**Q7:** If you have to model an SLO and generate an OpenAPI specification out of it. Would you use MDSL instead of doing it with other tools?

The last part of the Goal Question metric approach is to identify appropriate metrics to measure each question. In this thesis, we plan to conduct an experiment with a prototype of the extension and let the participants provide feedback via a questionnaire. Now it needs to be analyzed if this is an appropriate metric for each question. For **Q1** and **Q2** one experiment is an appropriate metric because each of our participants has experience with developing software and understanding problems in doing this and using code editors like eclipse and can therefore determine if our extension is user-friendly. Not every participant has to be experienced with using domain-specific languages since we provide an introduction into MDSL and our extension. It is also beneficial if our participants have different experience levels using domain-specific languages and our extension because we expect that the problems vary with different experiences. For **Q3** our experiment is an appropriate metric because getting insight into our extension its capabilities and also its limitations. This allows them to evaluate if the prototype can be usable for them. For **Q3** to **Q6** the questionnaire is an appropriate metric because we provide free text questions which allow participants to write down their experiences. For all questions, our experiment is an appropriate metric we use this as our only metric.

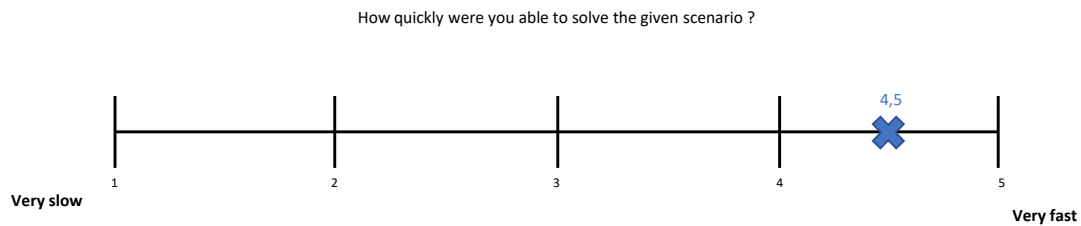


## 5.2 Results

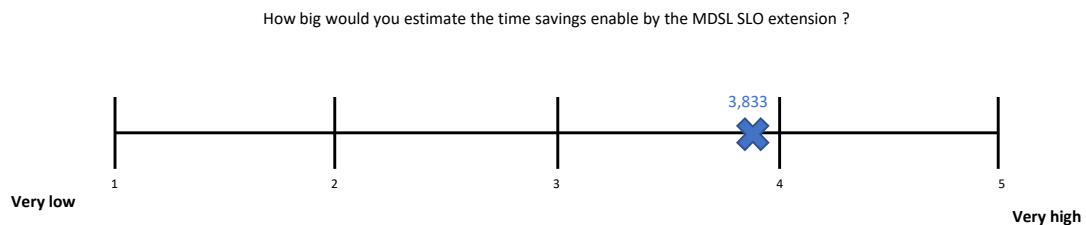
Of the six people who agreed to take part in our experiment, all were able to complete the experiment and fill out the questionnaire. Three of them are full-time employees from the industry with different levels of experience. The rest were from academia two with jobs as student assistants. The companies ranged from a well know German carmaker to an energy startup, and one IT consulting firm. All of our participants had more than three years of coding experience.



**Figure 5.1:** Average rating of Q1.



**Figure 5.2:** Average rating of Q2.



**Figure 5.3:** Average rating of Q3.

The general feedback regarding the extension was good. **Q1**, **Q2** and **Q3** were aimed to assess if our extension is usable. With an average of 4.0 between these three questions the extension is generally used but there is room for improvement. All participants were able to model the SLO into the scenario and use the generators to create artefacts.

The second part of the questionnaire is directed to get qualitative feedback on our extension. **Q4** covers all parts which the participants liked about the extension. The parts of the extension the participants not liked are covered by **Q5**. All features that are missed by the participants are covered in **Q6**. The questions also serve to determine where the extension could be improved in future work.

Question four yielded a lot of praise for the extension. The participants mentioned the excellent integration into the Eclipse IDE. Here the jump-to-definition feature and auto-complete feature were mentioned as great timesavers. Another well-received feature was the issue or error highlighting. The design of the syntax of the SLO extension to be readable was also mentioned as an advantage of the extension.

Question five deals with the things the participants did not like about the extension. Here one participant mentioned that the auto-complete behaviour can not be configured. Another also mentioned that the auto-complete feature is inconsistent and could not be applied everywhere. One participant criticized that several keywords have to be used to describe an attribute. He also mentioned that the terms and syntax have minor inconsistencies. One participant also mentioned that the editor has quirks that come from the used framework Xtext.

The last question dealt with the things participants thought were missing with our extension. Here some feature specific to the Eclipse IDE were mentioned. It was recommended that we have more syntax highlighting. Automatic indentation was mentioned. Lastly mentioned was that we should not focus on Eclipse with our extension and also support things like the language server protocol.

The last question asked was **Q7**. If they have to model a service-level objective and generate an OpenAPI document, would they use our extended version of MDSL? All participants agreed that they would use our prototype.

### 5.3 Discussion

In this section, hypotheses were created to determine if our goal **G1** was achieved. **G1** was achieved if every hypothesis here can be accepted.

The following hypotheses were created:

**H1:** The extension is beneficial in developing microservice architectures

**H2:** The industry could imagine using such a tool

For the validation of **H1** several validation questions were asked. **Q1** stated how user friendly the MDSL SLO extension is. With an average rating of 3.8 out of 5, we conclude the extension is in fact usable. **Q2** aims to assess if the participants were able to model the scenario. This was done to rule out that the given scenario is too complex and leads to bad usability. All participants were able to solve the task so this can be ruled out. **Q2** also was asked to find out how quickly the user

perceived the modelling of SLOs. Here **Q2** received an average rating of 4.5 out of 5. This means the participants were not only able to model the given scenario but also did this really fast. **Q3** was asked to validate if the extension is able to solve time in the development process. Since **Q3** received an average rating of 3.8 out of 5, we also conclude that our extension is able to solve time in the development process. In addition to that **Q4** also added evidence that our extension is able to save time features implemented by our extension were described as great time savers by our participants. One example mentioned was the auto-complete feature our extension provides for the Eclipse IDE. In summary, all participants indicated that the MDSL SLO extension is beneficial for developing microservice architectures and therefore **H1** is accepted.

**Q7** was asked to verify **H2**. Each participant could see themselves using our extension of MDSL to model and SLO and generate an OpenAPI specification out of it. For this reason, **H2** is accepted as well.

All our hypotheses were accepted. So it can be concluded that our defined target G1 has been reached.

## 5.4 Threats to Validity

In this section, the threats to validity are discussed according to Runeson and Höst [RH09]. Validity consists of construct validity, internal validity, external validity and reliability.

Starting with construct validity in this regard we have a threat.

It is possible that our participants interpreted the asked questions differently than we did. This is countered by the fact that the participants had the possibility to ask questions to clarify misunderstandings. For this reason, this threat is minimal.

The next aspect investigated is internal validity. We have a threat here because our extension was tested by participants and is therefore subjective. They may have personal opinions and preferences regarding the used techniques and tools which influence their feedback. Also, external factors not covered by this experiment could influence the participant's responses.

Another aspect to pay attention to is external validity. Here we have a threat because of our group size and background. Therefore some of the findings of this experiment may not be generalizable. The sample size for this experiment was small with only six participants. Also, we took people from industry and students from academia who all have or had an affiliation with the University of Stuttgart and our department. So this experiment should be seen as preliminary. All six of our participants mentioned similar things to fix in our extension. For this reason, it is likely that future experiments with other participants yield similar results. We would advise first addressing the issues mentioned by our participants and then repeating the experiment with a bigger sample size consisting of people with different backgrounds.

The last aspect investigated is reliability. It describes the extent to which the data and analysis are dependent on the specific researcher. This threat is small because we explained the setup of our experiment in Section 5.1.1 and the used questionnaire and documents can be found in the Appendix A.2. For the code, and plugin of our extension we still have this threat because is hosted online and maybe not be available forever.



## 6 Conclusion

With this chapter, we conclude this thesis. We give a summary of this thesis in Section 6.1. Followed by the limitations of this thesis in Section 6.2. We then go into what lessons we have learned while working on this thesis in Section 6.3. Finally, in Section 6.4 we give an outlook for future work.

### 6.1 Summary

This thesis provides a concept of how domain-specific languages can be extended to not only support the modelling of functional contracts but also model quality characteristics of microservice architectures like service level objectives.

This concept was implemented using the well-established domain-specific language MDSL and the service level objective model provided by OpenSLO. Also, the generator for OpenAPI documents of MDSL was extended to also feature service level objectives.

The built extension was then evaluated with an experiment. The extension was well received by the participants. Some improvement was suggested mainly regarding the integration of the extension into the used eclipse IDE.

### 6.2 Limitations

This thesis looked only at service-level objects. It is not clear if the concept can be used to extend domain-specific languages for other quality characteristics like scaling policies or service effect specifications.

Also regarding the implementation in MDSL we only extended the OpenAPI generator but MDSL also features generators for other interface description languages like GraphQL or Protocol Buffers. The implemented OpenSLO model currently also only features the use of Prometheus HTTP API as a data source for the service level indicators. Here also other monitoring service providers can be investigated and possibly integrated. Currently, also only some static checks were done on the model these could be extended to check the model more extensively.

In our evaluation, we only had a small group of participants. So the group may not be representative. Results therefore may not be generalizable. Also, the feedback from the first evaluation should be first implemented before repeating the experiment.

### 6.3 Lessons Learned

The first thing I learned during this thesis is to use model-driven software development and domain-specific languages. Also during the development of the extension for MDSL I learned a lot about developing domain-specific languages using the Xtext framework. Because Xtext is not that extensive documentation available, I gained a lot of knowledge about the internal of the Xtext framework by simply trying things out and looking into the source code. Lastly, I gained experience in planning and executing an evaluation using the goal question metric approach and discussing threats to validity.

### 6.4 Future Work

A first starting point where future work could build upon is to fully implement the concept and write a generator which takes the extended OpenAPI specification and OpenSLO document and generates a configuration for a Prometheus server to automatically monitor the generated server stub.

Like mentioned in the limitation section another starting point is to look into the other interface description languages MDSL offers and how these could be extended to also include service level objectives. Also, other future work could start and look for other quality characteristics, which could be integrated into MDSL.

It also could be investigated how and which other vendors could be integrated into the OpenSLO model to serve as data sources for our service level indicators.

A last point where future work can start is taking the feedback to the evaluation and implementing it to conduct a further experiment. This experiment could be conducted with a bigger group and people with different backgrounds and experience levels to validate if the suggested improvements are actually improving the prototype.

## Bibliography

- [Atl] Atlassian. *SLA vs. SLO vs. SLI - differences*. URL: <https://www.atlassian.com/incident-management/kpis/sla-vs-slo-vs-sli> (cit. on p. 7).
- [BCR94] V. R. Basili, G. Caldiera, H. D. Rombach. “The goal question metric approach”. In: *Encyclopedia of software engineering* (1994), pp. 528–532 (cit. on p. 42).
- [BMa] I. Bartholomew, N. Murphy. URL: <https://openslo.com/> (cit. on p. 7).
- [BMb] I. Bartholomew, N. Murphy. *OpenSLO/openslo: Open specification for defining and expressing service level objectives (SLO)*. URL: <https://github.com/OpenSLO/OpenSLO> (cit. on pp. 7–12).
- [ES] S. Efftinge, M. Spoenemann. *Xtext*. URL: <https://www.eclipse.org/Xtext/> (cit. on p. 5).
- [ES19] K. Eilebrecht, G. Starke. *Patterns kompakt*. Springer Berlin Heidelberg, 2019. DOI: [10.1007/978-3-662-57937-4](https://doi.org/10.1007/978-3-662-57937-4). URL: <https://doi.org/10.1007/978-3-662-57937-4> (cit. on p. 4).
- [FL14] M. Fowler, J. Lewis. *Microservices*. 2014. URL: <https://martinfowler.com/articles/microservices.html> (cit. on p. 3).
- [Fow10] M. Fowler. *Domain-specific languages*. Pearson Education, 2010, p. 27 (cit. on p. 4).
- [Fre] P. Frenandez. *Isa-Group/SLA4OAI-specification*. URL: <https://github.com/isa-group/SLA4OAI-Specification> (cit. on p. 27).
- [FW21] D. Falcioni, R. Woitsch. “OLIVE, a Model-Aware Microservice Framework”. In: *The Practice of Enterprise Modeling*. Ed. by E. Serral, J. Stirna, J. Ralyté, J. Grabis. Cham: Springer International Publishing, 2021, pp. 90–99. ISBN: 978-3-030-91279-6 (cit. on p. 14).
- [GCD+17] G. Granchelli, M. Cardarelli, P. Di Francesco, I. Malavolta, L. Iovino, A. Di Salle. “Towards Recovering the Software Architecture of Microservice-Based Systems”. In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. 2017, pp. 46–53. DOI: [10.1109/ICSAW.2017.48](https://doi.org/10.1109/ICSAW.2017.48) (cit. on p. 14).
- [Gro] R. Gronback. *Eclipse modeling project: The eclipse foundation*. URL: <https://www.eclipse.org/modeling/emf/> (cit. on p. 5).
- [MPT78] M. D. McIlroy, E. N. Pinson, B. A. Tague. “UNIX Time-Sharing System: Foreword”. In: *Bell System Technical Journal* 57.6 (July 1978), pp. 1899–1904. DOI: [10.1002/j.1538-7305.1978.tb02135.x](https://doi.org/10.1002/j.1538-7305.1978.tb02135.x). URL: <https://doi.org/10.1002/j.1538-7305.1978.tb02135.x> (cit. on p. 3).
- [Net] I. d. Net. *Yaml Ain’t markup language (YAML™) version 1.2*. URL: <https://yaml.org/spec/1.2.2/> (cit. on p. 22).

- [New21] S. Newman. *Building microservices*. 2nd ed. Sebastopol, CA: O'Reilly Media, Sept. 2021. ISBN: 978-1-492-03402-5 (cit. on p. 3).
- [Rad] F. Rademacher. *LEMMA Documentation*. URL: <https://seelabfhdo.github.io/lemma-docs/> (cit. on p. 14).
- [Rad22] F. Rademacher. “A Language Ecosystem for Modeling Microservice Architecture”. PhD thesis. Kassel, Universität Kassel, Fachbereich Elektrotechnik / Informatik, 2022 (cit. on p. 14).
- [RH09] P. Runeson, M. Höst. “Guidelines for conducting and reporting case study research in software engineering”. In: *Empir. Softw. Eng.* 14.2 (2009), pp. 131–164 (cit. on p. 45).
- [SMID22] A. Suljkanović, B. Milosavljević, V. Indić, I. Dejanović. “Developing Microservice-Based Applications Using the Silvera Domain-Specific Language”. In: *Applied Sciences* 12.13 (2022). ISSN: 2076-3417. DOI: 10.3390/app12136679. URL: <https://www.mdpi.com/2076-3417/12/13/6679> (cit. on p. 13).
- [SSB22] S. Speth, S. Stieß, S. Becker. “A Saga Pattern Microservice Reference Architecture for an Elastic SLO Violation Analysis”. In: *Companion Proceedings of 19<sup>th</sup> IEEE International Conference on Software Architecture (ICSA-C 2022)*. IEEE, Mar. 2022. DOI: 10.1109/ICSA-C54293.2022.00029 (cit. on pp. 26, 36, 41).
- [WB23] A. Wąsowski, T. Berger. *Domain-Specific Languages*. Springer International Publishing, 2023. DOI: 10.1007/978-3-031-23669-3. URL: <https://doi.org/10.1007/978-3-031-23669-3> (cit. on p. 4).
- [Wol15] E. Wolff. *Microservices*. de. 1st ed. Heidelberg, Germany: dpunkt, Nov. 2015. ISBN: 978-3-86490-313-7 (cit. on p. 3).
- [Zim] O. Zimmermann. *Microservice domain-specific language (MDSL) homepage*. URL: <https://microservice-api-patterns.github.io/MDSL-Specification/> (cit. on pp. 5, 6).

All links were last checked on March 1, 2023.



# A Appendix

## A.1 OpenSLO file example

---

**Listing A.1** Generated YAML document from our example part one.

---

```
---
apiVersion: openslo/v1
kind: SLO
metadata:
  name: BasicSLO
  displayName: Basic Service Level Indicator
spec:
  service: MonitoringService
  indicatorRef: SiteAvailabilityIndicator
  timeWindow:
    duration: 30d
    calendar:
      startTime: 2023-02-05 23:59:59
      timeZone: Berlin/Europe
    isRolling: "false"
  budgetingMethod: Timeslices
  objectives:
  - displayName: Site Availability
    targetPercent: 99.0
    timeSliceTarget: 90.0
    timeSliceWindow: 2h
  alertPolicies:
  - BasicAlert
---
apiVersion: openslo/v1
kind: SLI
metadata:
  name: SiteAvailabilityIndicator
spec:
  description: Indicator describes the applications availability
  ratioMetric:
    counter: true
    good:
      metricSource:
        metricSourceRef: ExamplePrometheusServer
        type: Prometheus
        spec: http_requests_total{status_code="200"}
  total:
    metricSource:
      metricSourceRef: ExamplePrometheusServer
      type: Prometheus
      spec: http_requests_total
```

---

### Listing A.2 Generated YAML document from our example part two.

---

```
---
apiVersion: openslo/v1
kind: AlertPolicy
metadata:
  name: BasicAlert
spec:
  alertWhenNoData: true
  alertWhenResolved: true
  alertWhenBreaching: true
  conditions:
  - kind: AlertCondition
    metadata:
      name: SlightlyBelowTarget
    spec:
      severity: page
      condition:
        kind: burnrate
        op: gt
        threshold: 1
        lookbackWindow: 30d
        alertAfter: 6h
      notificationTargets:
      - targetRef: sreEngineer
---
apiVersion: openslo/v1
kind: AlertNotificationTarget
metadata:
  name: sreEngineer
  displayName: John Miller
spec:
  target: push message
  description: John Miller is the site reliability engineer reposable for this service
---
apiVersion: openslo/v1
kind: Service
metadata:
  name: MonitoringService
  displayName: Standard monitoring service
spec:
  description: All SLOs associated with this service are for minotirng the application
---
apiVersion: openslo/v1
kind: DataSource
metadata:
  name: ExamplePrometheusServer
  displayName: Example prometheus main server
spec:
  description: Example prometheus server
  type: Prometheus
  connectionDetails:
    url: http://www.example.com/prometheus/api/v1
---
```

---

## A.2 Experiment

### A.2.1 MDSL model of the inventory service

---

**Listing A.3** MDSL file of the t2-inventory service provided to the study participants.

---

```

API description t2InventoryService

data type Identification D<string>
data type HTTPStatusCode {"successStatusCode":D<int>}
data type Product {"id": D<string>,
                  "name": D<string>,
                  "description": D<string>,
                  "units": D<int>,
                  "price": D<double>}

endpoint type InventoryEndpoint
  serves as INFORMATION HOLDER_RESOURCE
  exposes
    operation addProduct
      expecting payload Identification
      delivering payload HTTPStatusCode compensated by deleteProduct
    operation deleteProduct
      expecting payload Identification
      delivering payload HTTPStatusCode
    operation getProduct
      expecting payload Identification
      delivering payload Product

endpoint type RestockEndpoint
  serves as DATA_TRANSFER_RESOURCE
  exposes
    operation restocProducts
      delivering payload HTTPStatusCode

endpoint type GenerationEndpoint
  serves as DATA_TRANSFER_RESOURCE
  exposes
    operation generate
      delivering payload HTTPStatusCode

////////////////////////////////////Begin Open SLO //////////////////////////////////////

//////////////////////////////////// End Open SLO //////////////////////////////////////

```

---

---

**Listing A.4** MDSL file of the t2-inventory service provided to the study participants.

---

```
API provider InventoryServiceProvider
  offers InventoryEndpoint
  at endpoint location "http://t2-project/api"
  via protocol HTTP
  binding resource ProductResource at "/inventory{id}"
  operation addProduct to POST
  operation addProduct to PUT
  operation deleteProduct to DELETE
  operation getProduct to GET

  offers RestockEndpoint
  at endpoint location "http://t2-project/api"
  via protocol HTTP
  binding resource RestockResource at "/restock"
  operation restocProducts to GET

  offers GenerationEndpoint
  at endpoint location "http://t2-project/api"
  via protocol HTTP
  binding resource GenerateResource at "/generate"
  operation generate to GET
IPA
```

---

### A.2.2 Predefined Service Level Objective

The data source represented a Prometheus server it has the following attributes.

- Name: T2PrometheusServer
- Display name: "T2 Project Prometheus Main Server"
- Type: "Prometheus"
- URL: "http://www.t2-project.de/prometheus/api/v1"

The service had the following attributes.

- Name: MonitoringService
- Display name: "Default Monitoring Service"
- Description: "All SLOs associated with this service are for monitoring the T2-Project application"

The alert notification target had the following attributes

- Name: ShopDeveloper
- Display name: "John Doe"
- Target: "push message"

- Description “Person to contact when an SLO breaches”

The alert policy following attributes

- Name: BasicAlert
- AlertWhenNoData: true
- AlertWhenResolved: true
- AlertWhenBreaching: true
- Alert condition: SlightlyBelowTarget

The alert condition looks as follows.

- Name: SlightlyBelowTarget
- Severity: “page”
- Kind: “burnrate”
- Operator: gt
- Threshold: 1
- Look-back: 30d
- Alert after: 2d
- Alert notification target: ShopDeveloper

The service level indicator used has the following attributes

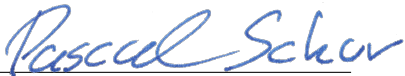
- Name: SiteAvailabilityIndicator
- Display name: “Availability”
- Description: “Indicator describes availability of the T2-Project application”
- Ratio metric
  - good:
    - \* Counter: false
    - \* Data Source: TeaShopPrometheusServer
    - \* Type: “Prometheus”
    - \* Spec: "http\_requests\_total"
  - total
    - \* Data source: TeaShopPrometheusServer
    - \* Type: “Prometheus”
    - \* Spec: "http\_requests\_total{satus\_code=“200”}"

The service level objective has the following attributes

- Name: BasicSLO
- Service: MonitoringService
- Service Level Indicator:
- Time-window: 30d
- Starting point: 2023-02-05 23:59:59
- Time-zone: "Europe/Berlin"
- Budgeting: Occurrences
- Objective:
  - "Site Availability"
  - Target percent: 99
  - Alert policy: BasicAlert

## **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.

Stuttgart, 01.03.2023 

place, date, signature