



Project Thesis

# Service Chaining Path Calculation

Departement of Computer Science  
OST - University of Applied Sciences  
Campus Rapperswil-Jona

Autumn Term 2020

**Authors**            Julian Klaiber  
                              Severin Dellsperger

**Advisor**             Prof. Laurent Metzger  
**Co-Advisor**        Urs Baumann  
**Project Partner**    Cisco EMEA *represented by* Francois Clad

---

## Abstract

Today's networks often use traditional routing protocols and approaches, which have their fundamentals often before the millennium turn. In recent years, the network area has not seen such rapid changes as familiar as in other sectors of IT. With the growth of the digital world and the introduction of new fields and technologies, such as 5G and cloud computing, the amount of data that today is sent over networks is enormous and will increase in the future. Modern networks not only have to deal with the new incredible number of data transfers, but in addition to this, they have also to face new challenges to satisfy customer needs. One of these requirements is the concatenation of network services, such as firewalls, IDS systems, DDoS protection, load balancers, and more similar services. This mechanism is known as service chaining and can not be accomplished by conventional networks.

This thesis aims to find a solution, that can help to calculate and find the best service chain according to defined parameters. The most suitable service chain refers to the total path, which has the requested services included and has minimum costs. The whole solution is based on the Segment Routing protocol, which follows the source-based routing paradigm and introduces new approaches to evolving existing and new networks. The goal is to deliver a result, that informs which network path a packet, which has specific parameters, is steered through. Besides, the application should give information about which specific service instances process packets during its path traversal. The application has to get information about a present Segment Routing network to achieve this goal. This data is gathered and aggregated in an external system, provided by the industrial partner, and used for the calculation.

During this project thesis, a useful Service Chain Path Calculation software could be developed. The application consists of different microservices. The central part of the solution is a backend based on Python Django, which offers a standardized REST-API to perform calculations, handle the topology information, and more. One of the backend's crucial tasks is to maintain a graph representing the network with its nodes and edges. On this graph, the requested calculations are performed. This part of the application was optimized using a library that is written in C++ in its core. To ensure the correctness of the graph in the backend, a service was introduced to process the necessary network data and provide it to the backend. The polling service is also written in Python and uses a caching system to handle the volatile network items. In collaboration with the Institute for Networked Solutions, a modern and easy to use web frontend was developed, using the introduced API developed in this thesis. The user can view the topology, perform calculations, and view the results via the frontend.

---

# Management Summary

## Initial Situation

Many of today's networks were built in the past and were often only expanded to handle the additional load that has been added in recent years. Due to the constant expansion and the use of complicated technologies, the networks became more and more complex and thus less maintainable and expandable. Likewise, troubleshooting in such networks has also become very difficult. Customer requirements for networks have also changed with new technologies and the ever-increasing shift to the cloud. It has become apparent that modern customer requirements can no longer be met with traditional approaches.

Existing networks can be optimized and simplified with the help of Segment Routing. This technology also allows to satisfy the recent and latest customer needs. One of the requirements that networks have always been expected to meet is Service Chaining. Service Chaining refers to the targeted concatenation of network services. Nowadays, there are different services available: Firewall, Intrusion Detection System, or Anti Virus Systems, just to name a few.

Service chaining aims to allow the user to configure which network flows should be treated by which different services. The goal of this project thesis was to create a computation and find the most suitable service chain. The most suitable service chain can be different for various applications, networks, or other characteristics.

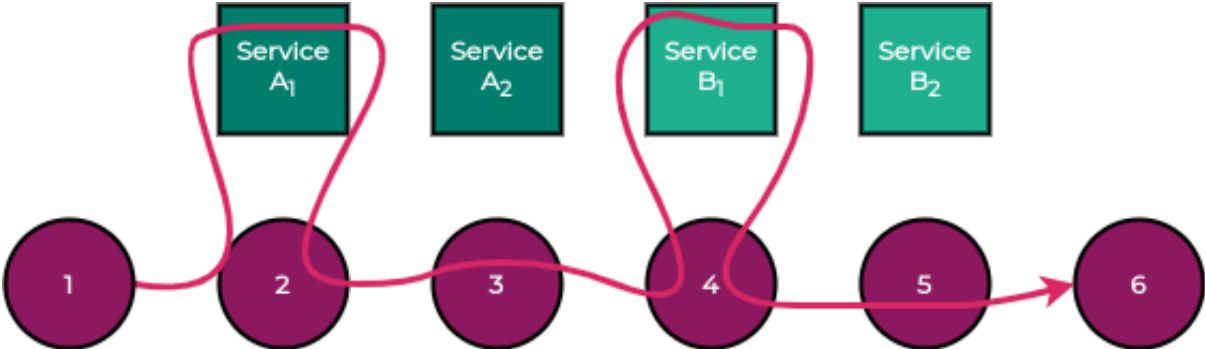


Figure 1: Service Chain Example

The introduction of this new technology and Service Chaining has not only a positive functional effect. It would also result in a lucrative benefit from the financial point of view. Many problems in the domain of network services can be optimized and solved in a fraction of time. It also allows using network department personnel in a more targeted and effective way, as the complete complexity is decreased, especially in traffic engineering.

## Procedure, Technology

The individual use cases were defined in a first phase together with the supervisor and the industry partner to achieve this goal. The definition of the use cases already gave an insight into the direction, in which this project should lead. In addition to the use cases, the various non-functional requirements were also defined, which have a major influence on the overall architecture and design decisions.

In the second phase, the architecture on which the application should be based was estab-

---

lished. It was decided that the development should be cloud native compliant and all components should be provided by Docker containers. In this phase, various performance tests were also carried out to elaborate the best possibilities to calculate the service chain.

The third phase was mainly concerned with the development of the application. In order to access the data of the Segment Routing network, which is supplied by the external software system of the industry partner, a separate polling service was written, which should always provide the backend with the latest data. The backend was developed with Python based on the Django framework. It thus provides the interface between the user and the data of the network as well as the calculation. The main responsibility of the backend is the processing of requests executed by the user and the calculation of the best service chain in the Segment Routing network.

It was decided that this project would be continued in the bachelor thesis after consultation with the supervisor. This decision had an influence on the further procedure in this thesis. Therefore, more emphasis was paid on stability and performance in order to lay a solid foundation for the continuing project.

## **Results**

In this project thesis, an Application Programming Interface was developed that allows service chain calculations to be performed in a Segment Routing network. Through the use of a standardized API, which has been developed along the lines of large software manufacturers such as Google or Facebook, various applications can access the data and functions of this application and perform visualizations and calculations using the API.

The API collects and uses all the data it receives from its integrated polling service. The data coming from a Segment Routing network is then processed and made available to the user. An external application, for example, a frontend, can then access the data via different endpoints.

A frontend, such as the one developed by the Institute for Networked Solutions, can display the complete topology and pass a user's parameters to the API, which then calculates the correct service chains. The results are then returned in a format that the frontend can process directly, for example, to display the path in the topology.

## **Outlook**

s The functionalities and the focus of this work has led to a stable foundation of an application which will be continued in the next semester in the form of a bachelor thesis. The focus will then be on improving the performance and stability as well as expanding the functionality. The goal is to bring the application to a point, where it can be used productively by a customer. Functionalities shall be developed, which transfer the application from Service Chaining to an evolved Service Programming software.

---

## Acknowledgments

We would like to thank several people for the support they have given us during this work:

**Laurent Metzger** who motivated us for this topic and reminded us throughout the work that this topic is the future in networking. Also his support with questions about segment routing have helped us a lot.

**Urs Baumann** who was the substitute for Laurent Metzger and always helped us with tips and advice.

**Francois Clad** who, as an industry partner and co-author of the Segment Routing books, has an enormous knowledge of this subject and has assisted us with it on many topics. With his knowledge he could always assess our progress and tell us what could be improved and how.

**Judith Manhart** who counter-read our work and checked our English. Since this was our first big work in English, we were very happy to have this help.

**Sharon Moll** who helped us with technical and architectural ideas and also counter-read our work.

---

# Contents

---

<b>Glossary and Abbreviations</b>	<b>ix</b>
<b>Bibliography</b>	<b>xiii</b>
<b>List of Figures</b>	<b>xiv</b>
<b>List of Tables</b>	<b>xvi</b>
<b>I Technical Report</b>	<b>1</b>
<b>1 Introduction</b>	<b>2</b>
1.0.1 Structure of this Work . . . . .	2
1.1 Traditional Routing and Networks . . . . .	3
1.1.1 Destination-based Routing . . . . .	3
1.1.2 Traditional Approaches . . . . .	4
1.1.3 Drawbacks . . . . .	5
1.2 Segment Routing . . . . .	6
1.2.1 Concept . . . . .	6
1.2.2 Example . . . . .	7
1.2.3 Terminology . . . . .	8
1.2.4 Advantages . . . . .	9
1.3 Aims and Objectives . . . . .	11
1.3.1 Problem . . . . .	11
1.3.2 Solution - Service Chaining . . . . .	12
<b>2 Results</b>	<b>15</b>
2.1 Distinction . . . . .	15
2.2 Achievements . . . . .	15
2.2.1 View Topology . . . . .	16
2.2.2 CRUD Service Chain . . . . .	16
2.2.3 Data Definition . . . . .	17
2.2.4 Service Discovery . . . . .	18
2.2.5 Path Calculation . . . . .	18
2.3 Implementation . . . . .	18
2.3.1 Architecture . . . . .	18
2.3.2 Polling . . . . .	19
2.3.3 Backend . . . . .	19
2.3.4 Calculation . . . . .	20

2.3.5	Frontend . . . . .	21
<b>3</b>	<b>Conclusion</b>	<b>22</b>
3.1	Retrospective . . . . .	22
3.1.1	Use Cases . . . . .	22
3.1.2	Discussion . . . . .	24
3.2	Outlook . . . . .	25
3.2.1	Improvements . . . . .	25
3.2.2	Innovations . . . . .	26
<b>II</b>	<b>Project Documentation</b>	<b>28</b>
<b>4</b>	<b>Requirement Specification</b>	<b>29</b>
4.1	Use Cases . . . . .	29
4.1.1	Actors . . . . .	29
4.1.2	Use-Case-Diagram . . . . .	30
4.1.3	Use-Case-Description . . . . .	30
4.2	Non-Functional Requirements . . . . .	37
4.2.1	Functionality . . . . .	37
4.2.2	Usability . . . . .	37
4.2.3	Reliability . . . . .	38
4.2.4	Performance . . . . .	38
4.2.5	Scalability . . . . .	38
4.2.6	Maintability . . . . .	38
4.2.7	Traceability . . . . .	38
<b>5</b>	<b>Domain Analysis</b>	<b>39</b>
5.1	Domain Model . . . . .	39
5.2	Administrative Concepts . . . . .	40
5.2.1	Node . . . . .	41
5.2.2	Network . . . . .	42
5.2.3	Segment . . . . .	42
5.2.4	Interface . . . . .	43
5.2.5	Link . . . . .	43
5.2.6	PhysicalLink . . . . .	43
5.2.7	LogicalLink . . . . .	44
5.2.8	ServiceInstance . . . . .	44
5.2.9	ServiceType . . . . .	45
5.2.10	ServiceChainCalculation . . . . .	45
5.2.11	ServiceChainResult . . . . .	46
<b>6</b>	<b>Architecture and Design Specifications</b>	<b>47</b>
6.1	General . . . . .	47
6.2	System Overview . . . . .	47
6.2.1	Jalapeño . . . . .	49
6.2.2	Frontend . . . . .	50
6.2.3	Backend . . . . .	50
6.2.4	Polling . . . . .	50
6.2.5	Database . . . . .	51
6.2.6	Redis . . . . .	51
6.3	12-Factor Methodology . . . . .	51

6.3.1	Codebase	51
6.3.2	Dependencies	52
6.3.3	Config	52
6.3.4	Backing services	52
6.3.5	Build, release, run	53
6.3.6	Processes	53
6.3.7	Port binding	53
6.3.8	Concurrency	54
6.3.9	Disposability	54
6.3.10	Dev/prod parity	54
6.3.11	Logs	55
6.3.12	Admin processes	55
6.4	Technology Decisions	55
6.4.1	Programming Language	56
6.4.2	Frameworks	56
6.4.3	Graph Library	59
6.4.4	Cache	61
6.4.5	Database	61
6.5	REST API	62
6.5.1	Documentation	62
6.5.2	Pagination	62
6.5.3	Nesting	63
6.5.4	Hyperlinking	65
6.5.5	Filtering	66
6.6	Backend Architecture	66
6.6.1	Django Daphne	67
6.6.2	serchio	67
6.6.3	updatetopology	68
6.6.4	graph	68
6.6.5	api	68
6.6.6	calculation	68
6.7	Sequence Diagrams	69
6.7.1	Polling Service	69
6.7.2	Get Data from API	69
6.7.3	Backend Data Update	70
6.7.4	Backend Calculation	71
6.8	Deployment	72
6.8.1	Current Deployment	72
6.8.2	Planned Deployment	73
6.9	Persistence	74
6.10	Packages and Classes	76
6.10.1	Backend	76
6.10.2	Polling	78
<b>7</b>	<b>Project Management</b>	<b>80</b>
7.1	Project Management	81
7.2	Scheduling	81
7.2.1	Iteration Planning	82
7.2.2	Estimates	83
7.2.3	Time Evaluation	83
7.3	Milestones	84
7.4	Meetings	85



---

7.5	Responsibilities . . . . .	85
7.6	Repositories . . . . .	85
7.7	Infrastructure . . . . .	86
7.8	Development Concept . . . . .	87
7.8.1	Definition of Done . . . . .	87
7.8.2	Code Style Guidelines . . . . .	87
7.8.3	Development Workflow . . . . .	87
7.9	Continuous Integration . . . . .	88
7.10	Code Metrics . . . . .	90
7.11	Risk Management . . . . .	91
7.11.1	Dealing with Risks . . . . .	93
7.11.2	Implications . . . . .	95
7.12	Quality Attributes . . . . .	95
7.13	Exception Handling . . . . .	95
7.14	Logging . . . . .	96
7.15	Testing . . . . .	96
<b>III</b>	<b>Appendix</b>	<b>98</b>
<b>A</b>	<b>Task Description</b>	<b>100</b>
<b>B</b>	<b>Class Diagrams</b>	<b>102</b>
<b>C</b>	<b>Test Protocols</b>	<b>103</b>
C.1	System Tests . . . . .	103
<b>D</b>	<b>Metrics</b>	<b>105</b>

---

# Glossary and Abbreviations

---

**5G** Fifth generation and current mobile communication standard. 5

**Affinity Attribute** Enables Coloring/Naming of links in Segment Routing Domains, which can be used for more enhanced Traffic Engineering policies.. 27

**API** Application Programming Interface. ix, *see* [Application Programming Interface](#)

**ArangoDB** Open-source graph database. 19

**ASGI** Asynchronous Server Gateway Interface. ix, 67, *see* [Asynchronous Server Gateway Interface](#)

**Asynchronous Server Gateway Interface** Providing a standard interface for enabling asynchronous communication between Python web servers, frameworks and applications.. 67

**Border Gateway Protocol** Exterior Gateway Protocol (EGP), which exchanges network information between different autonomous system (mostly on the internet). 11

**Cartesian Product** Set of all possible ordered pairs from two different sets. 20

**CD** Continuous Delivery. ix, 51, *see* [Continuous Delivery](#)

**CI** Continuous Integration. ix, 51, *see* [Continuous Integration](#)

**Cloud Computing** On-demand availability of computer system resources, e.g. storage, computing power, in a data center which is available over the internet. 5

**container** Processes which are running on the host system but in an encapsulated context. 12

**Continuous Delivery** Development practice in software engineering, where changes are automatically integrated into production. 51

**Continuous Integration** Development practice in software engineering, where the code is frequently integrated into the shared code base. 51

**DDoS** Distributed Denial-of-Service. ix, 11, *see* [Distributed Denial-of-Service](#)

**Dijkstra** Mathematical algorithm which solves the problem of the shortest path for a given start-point. 20, 60

**Distributed Denial-of-Service** Malicious attempt to disrupt the normal traffic to e.g. a web-server. 11

- Django REST Framework** Django extension, which allows the development of web APIs. [57](#)
- Docker** Company which offers a set of platform as a service products. [18](#)
- DRF** Django REST Framework. [x](#), [57](#), *see* [Django REST Framework](#)
- ECMP** Equal-Cost Multi-Path Routing. [x](#), [13](#), [33](#), *see* [Equal-Cost Multi-Path Routing](#)
- Elasticsearch, Logstash and Kibana** Stack out of different softwares, that enable log management and monitoring. [55](#)
- ELK** Elasticsearch, Logstash and Kibana. [x](#), [55](#), *see* [Elasticsearch, Logstash and Kibana](#)
- Equal-Cost Multi-Path Routing** Strategy which enables forwarding packets on several best paths with equal routing costs.. [13](#)
- Firewall** Security system designed to prevent unauthorized access to network resources. [8](#)
- FW** Firewall. [x](#), [8](#), [38](#), [45](#), *see* [Firewall](#)
- HTTP** Hypertext Transfer Protocol. [x](#), [24](#), *see* [Hypertext Transfer Protocol](#)
- Hypertext Transfer Protocol** De facto standard protocol which enables communication between browser and web server and is used for exchanging hypermedia documents like HTML or JavaScript files. [24](#)
- IDS** Intrusion Detection System. [x](#), [8](#), [11](#), [38](#), [45](#), *see* [Intrusion Detection System](#)
- IGP** Interior Gateway Protocol. [x](#), [4](#), [8](#), *see* [Interior Gateway Protocol](#)
- INS** Institute for Networked Solutions. [x](#), [15](#), *see* [Institute for Networked Solutions](#)
- Institute for Networked Solutions** Computer science institute at the University of Applied Sciences of Eastern Switzerland. [15](#)
- Interior Gateway Protocol** A type of routing protocol used to exchange network information within an Autonomous System (AS). [4](#)
- Intermediate System to Intermediate System** Besides OSPF well-known routing protocol, which belongs to the Interior Gateway Protocol (IGP), and makes decisions along to link-state information.. [11](#)
- Internet Protocol Version 6** Most recent version of the Internet Protocol (IP) which introduces new techniques and capabilities. [10](#)
- Intrusion Detection System** Monitoring system, that detects suspicious activities and creates warning messages. [8](#)
- Jalapeño** Software system developed by Cisco, which aggregates and processes the data of a segment routing network. [24](#)
- JavaScript Object Notation** Standardized and easy to read format for exchanging data. [22](#)
- Jazzband** Community which is responsible for a lot of open source Python projects. [58](#)
- JSON** JavaScript Object Notation. [x](#), [22](#), *see* [JavaScript Object Notation](#)
- Kubernetes** Open-source container orchestration system, which allows easy scaling and management of the deployed applications. [18](#)

- metric** A variable that identifies the cost of a particular route on which routers make forwarding decisions. [3](#)
- Minimum Viable Product** Version of a product which includes the customer's minimum requirements so that it can be used in production. [22](#)
- Multi Protocol Label Switching** Routing protocol which routes packets based on path labels and not network addresses. [10](#)
- MVP** Minimum Viable Product. [xi, 22, 29](#), *see* [Minimum Viable Product](#)
- Object-relational mapping** Programming technique for converting data between objects in object-oriented programming languages and incompatible types. [61](#)
- Open Shortest Path First** Probably most famous fast and scalable link-state routing protocol, which belongs to the category of Interior Gateway Protocol (IGP). [11](#)
- ORM** Object-relational mapping. [xi, 61](#), *see* [Object-relational mapping](#)
- PE** Provider Edge. [xi, 33, 41, 42, 45](#), *see* [Provider Edge](#)
- PEP8** Document providing guidelines and best practices for the Python programming language. [87](#)
- PmQm** Project and Quality Management. [xi, 30](#), *see* [Project and Quality Management](#)
- Project and Quality Management** Module about project and quality management, taught at the University of Applied Sciences of Eastern Switzerland. [30](#)
- Provider Edge** Interface between end customer and service provider. [33](#)
- Redis** In-memory store, which allows fast data access. [19](#)
- ReDoc** Open source API reference documentation tool. [59, 62](#)
- SDN** Software Defined Networking. [xi, 11](#), *see* [Software Defined Networking](#)
- Segment Routing** Source-based routing protocol that enables new approaches and optimizes traffic engineering, network protection and more. [2](#)
- SerChio** Service Chaining Path Calculation. [xi, 15, 24, 47, 57, 59, 62, 63](#), *see* [Service Chaining Path Calculation](#)
- Service Chain** Concatenation of different services. [2, 11](#)
- Service Chaining Path Calculation** The name of this thesis and developed application. [15](#)
- Software Defined Networking** Technology that enable dynamic and programmable network configuration. [11](#)
- SQL** Structured Query Language. [xi, 51](#), *see* [Structured Query Language](#)
- SR** Segment Routing. [xi, 2](#), *see* [Segment Routing](#)
- Structured Query Language** Language which is used to communicate with databases. [51](#)
- Swagger** Well known documentation tool for web APIs. [59, 62](#)
- TE** Traffic Engineering. [xi, 8, 10, 33](#), *see* [Traffic Engineering](#)

- TI-LFA** Topology Independent Loop-Free Alternate. [xii, 10](#), *see* [Topology Independent Loop-Free Alternate](#)
- Topology Independent Loop-Free Alternate** Protection for the network that reaches sub-50ms rerouting of prefixes. [10](#)
- Traffic Engineering** Technique used to control and steer traffic to optimize the network utilization and performance. [8](#)
- Virtual Machine** Emulated computer system, that is running on a so-called hypervisor. [12](#)
- Virtual Network Function** Virtualized Network Service which is running in a virtual machine or container.. [12](#)
- Virtual Routing and Forwarding** Virtual router which lives on a physical. Enables the coexistence of e.g. different customers on the same router. [16](#)
- VM** Virtual Machine. [xii, 12](#), *see* [Virtual Machine](#)
- VNF** Virtual Network Function. [xii, 12](#), *see* [Virtual Network Function](#)
- VRF** Virtual Routing and Forwarding. [xii, 16, 64](#), *see* [Virtual Routing and Forwarding](#)
- WebSocket** Communication protocol which uses the HTTP protocol to provide full-duplex channels for communication. [19, 24](#)

---

## Bibliography

---

- [1] Google Cloud apigee. *Web API Design: The Missing Link. Best Practices for Crafting Interfaces that Developers Love*. apigee, Google Cloud, 2016.
- [2] Clarence Filsfils, Kris Michielsen, and Ketan Talaulikar. *Segment Routing. Part 1*. Independently published, 2016.
- [3] Clarence Filsfils et al. *Segment Routing. Part 2*. Independently published, 2019.
- [4] Wiggins Adam. *The Twelve-Factor App*. URL: <https://12factor.net/> (visited on 10/06/2020).
- [5] *Applying Evolutionary Requirements*. URL: [https://www.craiglarman.com/wiki/downloads/applying\\_uml/larman-ch5-applying-evolutionary-requirements.pdf](https://www.craiglarman.com/wiki/downloads/applying_uml/larman-ch5-applying-evolutionary-requirements.pdf) (visited on 09/23/2020).
- [6] Simon Brown. *C4 Model*. URL: <https://c4model.com/> (visited on 11/03/2020).
- [7] Clarence Filsfils et al. *Segment Routing Architecture*. URL: <https://tools.ietf.org/html/rfc8402> (visited on 12/07/2020).
- [8] Cloud Computing Foundation. *Cloud Native*. URL: <https://cncf.io/> (visited on 10/11/2020).
- [9] Craig Larman. *Applying Evolutionary Use Cases*. URL: [https://www.craiglarman.com/wiki/downloads/applying\\_uml/larman-ch6-applying-evolutionary-use-cases.pdf](https://www.craiglarman.com/wiki/downloads/applying_uml/larman-ch6-applying-evolutionary-use-cases.pdf) (visited on 09/23/2020).

---

## List of Figures

---

1	Service Chain Example . . . . .	ii
1.1	Example destination-based Routing . . . . .	4
1.2	Example Routing with Labels . . . . .	5
1.3	Example Convergence Problems . . . . .	6
1.4	Concept of Segment Routing . . . . .	7
1.5	Convergence in Segment Routing . . . . .	10
1.6	Service Consumptions in Traditional Networks . . . . .	12
1.7	Service Chaining in Segment Routing Networks . . . . .	14
2.1	Frontend Network Topology . . . . .	16
2.2	Frontend Calculation . . . . .	17
2.3	Backend API Root . . . . .	17
2.4	Backend Nested Endpoints Example . . . . .	18
2.5	Calculation Workflow . . . . .	21
4.1	Use-Case-Diagram . . . . .	30
5.1	Domain Model . . . . .	40
6.1	C4 System Landscape . . . . .	48
6.2	C4 Container Diagram . . . . .	49
6.3	DRF built-in Web API . . . . .	57
6.4	Example Endpoint with Nesting . . . . .	57
6.5	Heroku Overview Django Channels . . . . .	59
6.6	Calculation Performance Diagram . . . . .	61
6.7	API Redoc . . . . .	62
6.8	API Swagger . . . . .	62
6.9	Backend Architecture Overview . . . . .	67
6.10	Polling Service Data Update . . . . .	69
6.11	GET topology from API . . . . .	70
6.12	Backend Data Update . . . . .	71
6.13	Backend Calculation Sequence . . . . .	72
6.14	Docker Single Host Deployment Diagram . . . . .	73
6.15	Kubernetes Deployment Diagram . . . . .	74
6.16	Persistence Diagram . . . . .	75
6.17	Package Diagram Backend . . . . .	77
6.18	Class Diagram api . . . . .	77
6.19	Class Diagram updatetopology . . . . .	78
6.20	Class Diagram graph . . . . .	78

---

6.21	Class Diagram calculation	78
6.22	Class Diagram polling	79
7.1	Project Timeline	82
7.2	YouTrack Estimates	83
7.3	Milestone Overview	84
7.4	Development Workflow	88
7.5	Pipeline Workflow	89
7.6	Executed Pipeline Example	90
7.7	Sonarqube Dashboard	91
7.8	Risk Overview	92
A.1	Task Description	100
B.1	Class Diagram Backend api	102
C.1	System Tests Postman	104
D.1	Sonar Lines of Code	105
D.2	Sonar Lines of Code Timeline	105
D.3	Sonar Metrics	106



---

## List of Tables

---

4.1	Use Cases Color Description . . . . .	29
4.2	Actor Description . . . . .	29
4.3	UC01: View Topology - Fully Dressed Description . . . . .	31
4.4	UC02: CRUD Service Chain - Fully Dressed Description . . . . .	35
4.5	UC03: Data Definition - Casual Description . . . . .	35
4.6	UC06: Path Approval - Fully Dressed Description . . . . .	36
6.1	Dev/prod parity comparison . . . . .	55
6.2	Technologies . . . . .	56
6.3	Calculation Performance Comparison . . . . .	61
7.1	Version History Project Mangement . . . . .	80
7.2	Iteration Planning . . . . .	83
7.3	Milestone Description . . . . .	84
7.4	Project Structure . . . . .	86
7.5	Continuous Integration Stages . . . . .	89
7.6	Risk List . . . . .	93
7.7	Dealing with Risks . . . . .	94
7.8	Quality Attributes . . . . .	95

**Part I**

**Technical Report**

## Chapter 1

---

# Introduction

---

This chapter contains an overview of what the thesis is about. Besides, it contains an introduction of the topics that are treated in this project thesis. These chapters are inspired by the books *Segment Routing Part 1* and *Segment Routing Part 2*. [2, 3]

In order to understand the content of this thesis, some general knowledge about the existing network world and the traditional routing approach has to be understood and therefore is a part of this introduction. Furthermore, it is noteworthy to introduce future technologies and approaches. Especially the **Segment Routing (SR)** technology, which is deeply connected to this thesis, introduced and explained to some extent. A small insight into this technology is given to understand the advantages and why this technology will replace existing traditional approaches. This topic will also explain the term **Service Chain** and reveal what this concept can be used for.

The above mentioned paragraphs are to show how this work is structured and how the parts are organized in the content.

### 1.0.1 Structure of this Work

This work is divided into two main parts. The topics included in the appropriate parts can be found below.

#### Part 1 - Technical Report

The first part of this project thesis is the **Technical Report**. It is divided into three chapters and directed to engineers in the computer science field. The first chapter is the introduction, which includes the entrance to the project thesis. It provides an overview of the traditional networks and their routing approaches. Followed by an overview of Segment Routing which is present to understand what essential techniques this project is based on. Last, the aims and objectives of this work are described. The second chapter includes the results, which are reached during the project time. It is divided into a distinction that determines the project's scope, overlooking the achievements, and information about the implementation. The third part describes the conclusion. In this chapter, a self-reflecting retrospective is made, inspiring further discussions and changes in the future. As a last part, it includes an outlook, how the project will be continued.

#### Part 2 - Project Documentation

The **Project Documentation** part includes project-specific documentation. It is divided into several chapters. The first chapter contains the requirement specification split to use cases

(functional requirements) and non-functional requirements. The next chapter contains the whole analysis of the domain, where essential parts are mentioned. Chapter six - [Architecture and Design Specifications](#) - describes the most important aspects of the application's architecture and its system design. The last chapter contains information about how the project is managed to reach its goal.

## 1.1 Traditional Routing and Networks

Most of today's networks, especially of service providers or enterprise companies, were built 20 or more years ago. The applied network protocols and techniques were at least as old if not older. Many communication fundamentals, which are used in such networks, are even more aged. Moreover, it is a fact that the basics of these approaches are still the same in their core. The network world has not been improving in the last years as the rest of the fast-moving digital world. It can be compared to the banking sector applications that are still using programming languages invented in the 60s. This chapter contains information about the traditional routing approaches, techniques used, and some facts and imperfections and disadvantages. Because these techniques are not the main topic of this thesis, it is only covered as profound as necessary to understand the main topics of this project thesis.

### 1.1.1 Destination-based Routing

Most networks and network protocols follow the destination-based routing approach. Simplified, this technique works as follows: As soon as data should be transmitted, the data will be split into smaller sized packets. Each packet not only contains a part of the data but rather contains additional information like where the packet should be routed to - the destination field. On the way to its destination, a packet is sent over the network and different routers. The destination field is inspected at each router, and the packet is forwarded according to the router's known best path. The best path is calculated with a network protocol, and hence each router knows which is the most suitable path for each destination.

#### Example

Refer to the visualization in figure 1.1:

Packets from the source R1 should be transmitted to Network A. Therefore, R1 checks out its shortest path and forwards the packet in the direction of R4. R4 forwards the traffic according to its lowest [metric](#) path to R3. This process is repeated at each router until the packet is delivered to its destination - Network A.

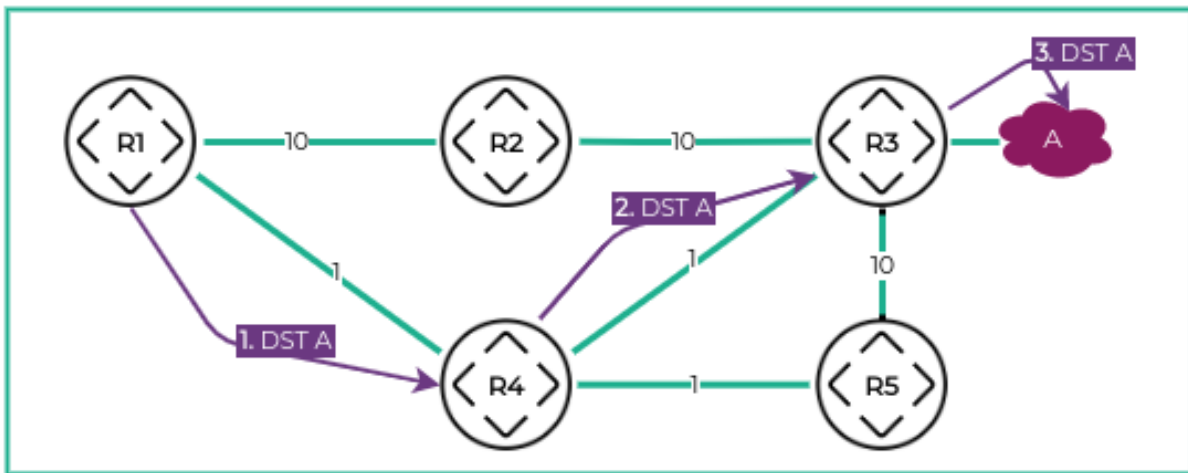


Figure 1.1: Example destination-based Routing

### 1.1.2 Traditional Approaches

In many networks, especially large ones, this destination-based routing approach is not enough to satisfy customer needs. In the past, on top of the destination-based routing approach, there were added some other concepts.

These concepts include introducing different labels. Each router has to maintain which meaning a unique label has within a certain approach. In concrete, this means that every single router has to take care of where to forward traffic for each specific label.

If data in such networks have to be delivered, the packets were assembled with one unique label. The label, in this case, represents the whole path. After the label was attached, the packet is sent from router to router through the network. Because each router maintains information on where to forward the packet according to its unique label, each packet is treated independently.

This approach helps to introduce several paths for a specific destination according to specific requirements. For example, one label can stand for the shortest path in concern of a specific metric. Another label can stand for the same destination network but another treatment. Maybe the label stands for the total path, which has the fewest delay. This approach introduced many advantages but also has several disadvantages which are listed in chapter 1.1.3.

#### Example

Refer to figure 1.2:

In this topology, two packets are sent over the network. Each network packet has a unique label assigned, which stands for its own path. In a concrete network, the label IV could, for example, stand for the path with the shortest delay. On the opposite, label XI could, for instance, stand for the lowest [Interior Gateway Protocol \(IGP\)](#) path.

The first path, to router R4, is identical for both packets. On the node R4, both packets are sent over to different next-hop routers. This behavior is related to the special treatment for each unique label: Router R4 has maintained information per path for each label and treats the packets different.

The total result can be observed - the same destination network's packets take other paths to their destination.

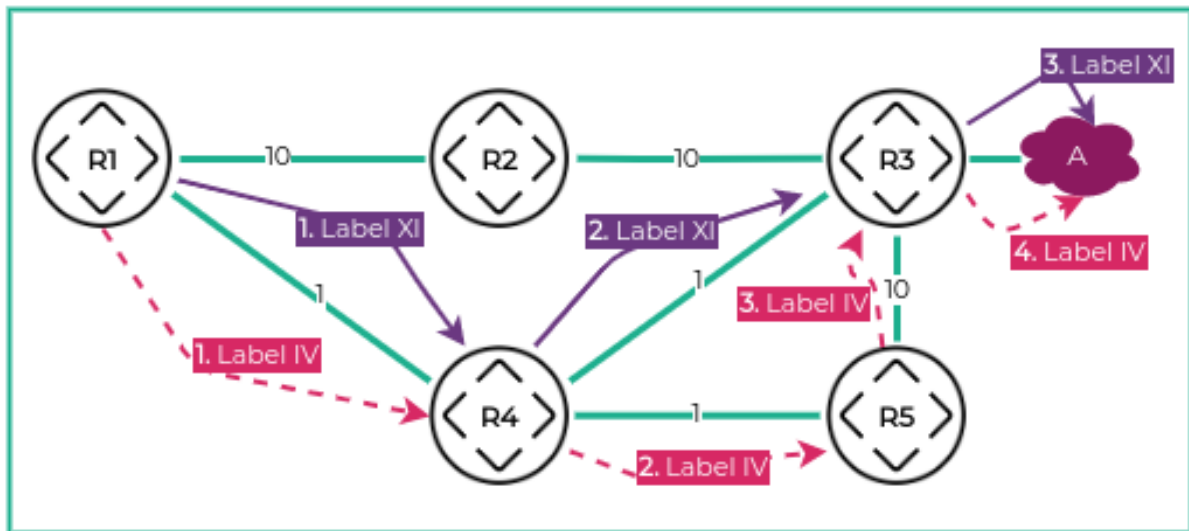


Figure 1.2: Example Routing with Labels

### 1.1.3 Drawbacks

The above-described techniques have improved the legacy networks and showed new features that were an excellent match for networks ten years ago. However, with the digital world's raising and introducing new technologies like **5G** and **Cloud Computing**, entirely new challenges overcome modern networks. The traditional approaches do not meet the requirements of modern networking anymore and have some drawbacks.

#### State in the Network

As mentioned above, in traditional networks, each router needs to maintain information about where to forward a packet to. This decision is made mostly according to unique characteristics like a label. This fact leads to one main drawback: The state relies on the network and has to be exchanged between each router. Consequently, there is heavily distributed intelligence that requires protocols to enable the understanding of every single router. Simplified, it can be said that many languages had to be used, so that the routers could talk to each other. This concept often leads to too complex architectures, in which the operating and also troubleshooting was debilitating.

Of course, there have also been approaches that brought up central coordinate instances. These concepts often had a lack of scalability and are not be discussed further in this thesis.

#### Convergence Problems

Another disadvantage the traditional approaches bring along is convergences.

As soon as the topology changes, for example a link disconnects, the network state must be recalculated. This process is a critical task; packets may not be transmitted successfully during the execution of the recalculation, and hence this task has to be executed as fast as possible. In the network jargon, this process is called network convergence.

As described, traditional networks follow the principle that every single router has to maintain the network state. This concept has two main disadvantages concerning network convergences:

First, each router had to be informed about changes in the network, which then triggers the recalculations. These exchanged messages can trigger a heavy load of the network and each

router's resources and may result in congestions.

Second, each router has to execute recalculations after a network topology change has occurred. As a result, network packets may not be transferred or get lost.

The convergence time could be improved in the past with proper network design, capacity planning and improvements in the protocol section. However, it was not always possible to ensure fast rerouting in each network. As a result, entirely new approaches had to be introduced.

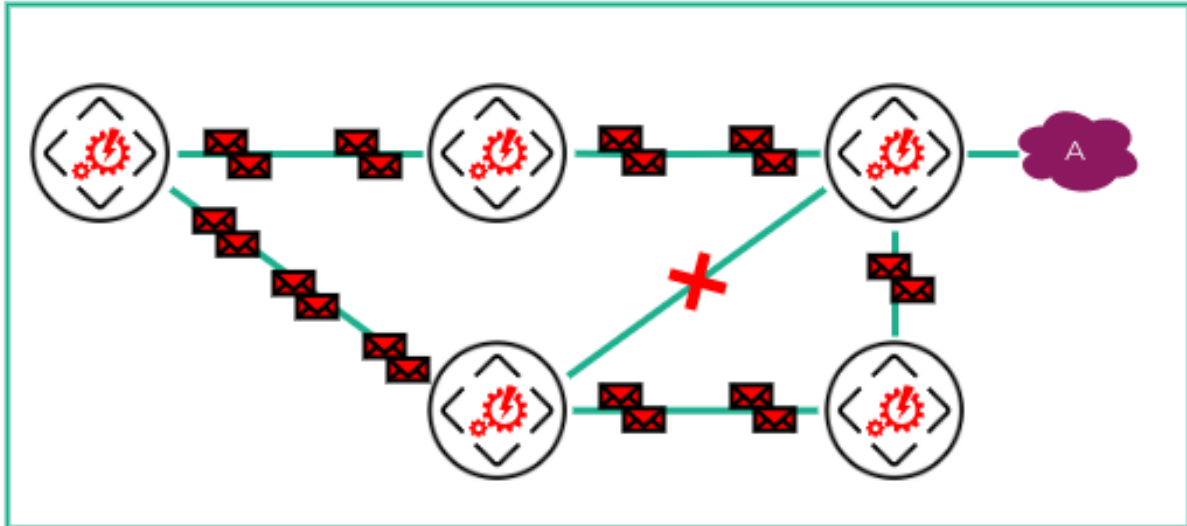


Figure 1.3: Example Convergence Problems

## 1.2 Segment Routing

Segment Routing is this new approach, which was praised in the earlier chapters. The following sections explain the necessary information about Segment Routing, which should be understood according to the project thesis's context. It also contains a chapter about the basic functionalities and concepts of Segment Routing on which this project is based on. Additionally, it implicates several terminologies used in Segment Routing, which are further used in this project thesis. Besides, it shows several advantages over common traditional routing concepts and lists why this approach will be heavily used in the future.

### 1.2.1 Concept

Segment Routing is a new routing approach that follows the source routing paradigm. The first router decides which path the packet should be steered through. This decision can be made upon several characteristics or rules, which can be configured.

After the path is determined, the path is expressed by adding instructions, so-called segments, to the packet header. A segment can represent many instructions; more information about a segment is given in chapter [Terminology](#). This list of instructions (segment list) in the network packet header ensures that the packet flows correctly through the path the source node had planned.

After the packet is assembled with its segment list, the packet can be sent through the network. Each node inspects the outermost segment of the packet and follows its instruction. The concept works because each intermediate node knows exactly which instruction it has to follow if a packet with a specific segment is received. After an instruction is finished, the

outermost segment is removed, and the next segment is examined and processed. This machinery is repeated for each segment in the segment list.

The correct maintained segment list ensures that the packet has reached its correct destination after all segments of the segment list are removed. The empty segment list at the end expressed that all planned instructions have been processed and executed.

### 1.2.2 Example

The following example, illustrated in figure 1.4, will bring clarity: Two packets, purple and pink, from different source networks and types have to be forwarded to network A behind router R3.

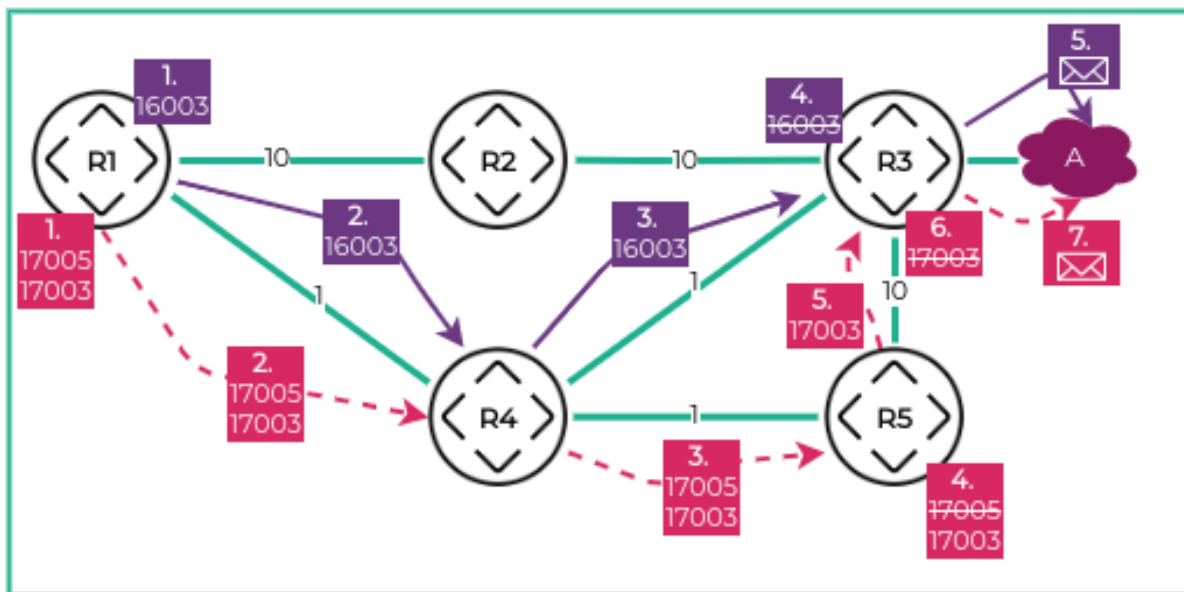


Figure 1.4: Concept of Segment Routing

The purple packet represents substantial business traffic, which has to be delivered on the shortest path to the destination network.

Therefore the source node, R1, decides to add the segment 16003 which tells all intermediate router to forward the packet on the shortest path to router R3. After the packet has assigned its segment list, which is composed of one instruction in this particular case, the packet will send over the shortest path to R3.

The first hop is the router R4. This router inspects the segment, recognizes its instruction, and therefore forwards it to router R3.

The destination router, R3, inspects the outermost segment and understands that the packet is destined to itself. Further, it can remove the outermost packet and send it to the end-destination because no more segments are assigned to the packet. Like mentioned before, the empty segment list ensures that the packet had been steered correctly.

The pink packet represents voice traffic, which has to be steered along the most sensitive delay traffic path. In this particular case, this path differs from the shortest path.

To express this path, the router R1 adds two segments, 17005 and 17003, to the packet. The segment list now includes two instructions. The first can be understood as "forward the packet on the path with the shortest delay path to R5". The second instruction refers to "forward the packet on shortest delay path to R3". Thus, the whole path, from source router to destination router, can be expressed with these two segments.



After the segment list is added to the packet header, the packet can be forwarded on the planned path. The first intermediate node is again R4. R4 inspects the segment 17005 and recognizes that the packet must be delivered to the R5.

After R5 has received the packet, it pops the outermost segment because the first segment of the path is achieved. Afterwards, the packet is redirected over the path with the shortest delay to the destination router R3.

R3 inspects the segment at the top, understands its commands, and therefore, it strips down the last segment. The next packet can be sent to the destination device. Hence, the packet was sent from source to destination over the designed path with the shortest delay.

### 1.2.3 Terminology

The terminology is based on the corresponding RFC standard: *RFC8402 - Segment Routing Architecture* [7] and is summarized to the minimum which is used in this thesis.

#### Node

A Node represents a device, which understands the Segment Routing protocol and participates in its functions. Most of the cases, this may be a router, but this can also be other devices, which can steer a packet, based on segments, through the network.

In figure 1.4 all routers, R1-R5, are Nodes.

#### Segment

A Segment expresses an instruction. Such an instruction can have a topological or service-based context. Basically, a Segment can have a local or a global meaning. For example, a Segment within a local topological context could refer to: *"forward the packet on a specific outgoing interface"*. A Segment in a global topological context could, for instance, be understood as *"forward the packet on a specific path through the network"*. This specific path could then follow specific metrics — for example the delay, IGP or **Traffic Engineering (TE)** metric.

A Segment could also be associated with a service-based instruction, which means something like *"the packet should be processed by a specific service or application"*. Such services could, for instance, be a **Firewall (FW)** or an **Intrusion Detection System (IDS)** application.

In figure 1.4 16003, 17005, 17003 are segments.

#### Segment Identifier

A Segment Identifier, often also Segment ID or SID, is an integer which is like a unique ID for a Segment. In this project thesis, Segment Identifier is used analog of Segment and does mean the same.

#### Segment List

The Segment List is the stack of different Segments, which is used to express the set of instructions that has to be followed by the traversal through the Segment Routing Domain. It is composed of one or several segments.

In figure 1.4 the Segment List can be found in the pink as well as the purple box and consists of the list of segments.

### Prefix Segment

A Prefix Segment, also often IGP-Prefix Segment, is a segment that refers to the instruction to forward the packet along a computed path to the node, which has the prefix with the interior gateway protocol advertised. The computed path can follow different specified algorithms, which use different metrics/characteristics to calculate the path.

In figure 1.4, for instance, 16003 or 17003 could refer to the prefix of Network A, if this prefix was advertised from Router 3 with IGP and different algorithms. In this particular example, the first algorithm was used to calculate the shortest IGP total weight path. The second algorithm, connected to the segment 17003, was used to calculate the total path with the shortest delay.

### Segment Routing Domain

The Segment Routing Domain refers to the set of devices which actively participate in the Segment Routing protocol.

In figure 1.4 the Segment Routing Domain is the set involving the nodes R1-R5

### Ingress

The Ingress, often also Ingress Router, is the node closest to the source. This specific device is the interface between external (customer) networks and Segment Routing Domain. This router receives a specific packet, does the path decision, and puts the segment list into the packet header. The policies, which affect the packet's steering through the network, have to be written to the Ingress. Generally, routers which undertake the Ingress role are Provider Edges. Therefore several routers in the network could be Ingress nodes.

In figure 1.4 the router R1 is the Ingress Router for both packet flows, the purple and the pink one.

### Egress

The Egress, often also Egress Router, is the node closest to the destination. It is the specific router where the last segment terminates and hence the last node in the Segment Routing Domain. Therefore, this router is - related to a specific packet flow - the router has to strip down the last segment and forward the packet to the destination endpoint.

In figure 1.4 the router R3 is the Egress Router for both packet flows, the purple and the pink one.

## 1.2.4 Advantages

### State in the Packet

Segment Routing introduces some new approaches, which changed some general and essential concepts: In the traditional methods, the state had to be maintained by each router. These mechanisms led to different protocols that ensured that the per-path information has to be exchanged between the nodes. Rapidly, this could result in a too complex network and in a network overhead.

New in Segment Routing is, the whole path can be expressed within the packet header in the form of a segment list. The packet now contains all instructions in the header. Hence, the network nodes have to simply follow and execute these instructions. Following the nodes don't have to maintain state information anymore.

This core change leads to many advantages. First, this inherits simplicity because the protocol

for exchanging per-flow information can be omitted. Also, the network implementation gets easier to understand, and hence, the troubleshooting and changing of components improves. Second, the whole path decision happens at the Ingress. This concept allows introducing entirely new opportunities to steer the traffic through the network - easy and scalable.

### Traffic Engineering

Traffic Engineering, the specific steering of packets, is improved with Segment Routing. Because the Ingress router has full control over which path should be used, TE is simplified and gets more powerful than ever. The steering of packets also gets more enhanced, with the innovations made in the new approach. It is now possible to not only do topological decisions but also include services into the packet flow. Besides, it's possible to program guidelines, such as avoiding a specific path or a specific service. Following the new approaches introduces new levels of scalability and functionality. In addition to this, the networks get more intelligent and can interact closer with applications. It is a perfect fit for the future and customer's needs.

### Convergence

As seen, the convergence and re-routing was a big problem for traditional approaches and much computational power. Hence, this problem was faced and improved with Segment Routing. While all routers were included in calculating new paths and maintaining per-flow information in the classical approach, this has changed in Segment Routing.

Referring to figure 1.5: If a path breaks, the avoided router must detect it and send a notification to the Ingress. The Ingress then can adjust quickly and reroute the packet on the backup path to the destination. This technique is called **Topology Independent Loop-Free Alternate (TI-LFA)** and is not part of this project thesis but has to be seen as an eminent advantage of Segment Routing.

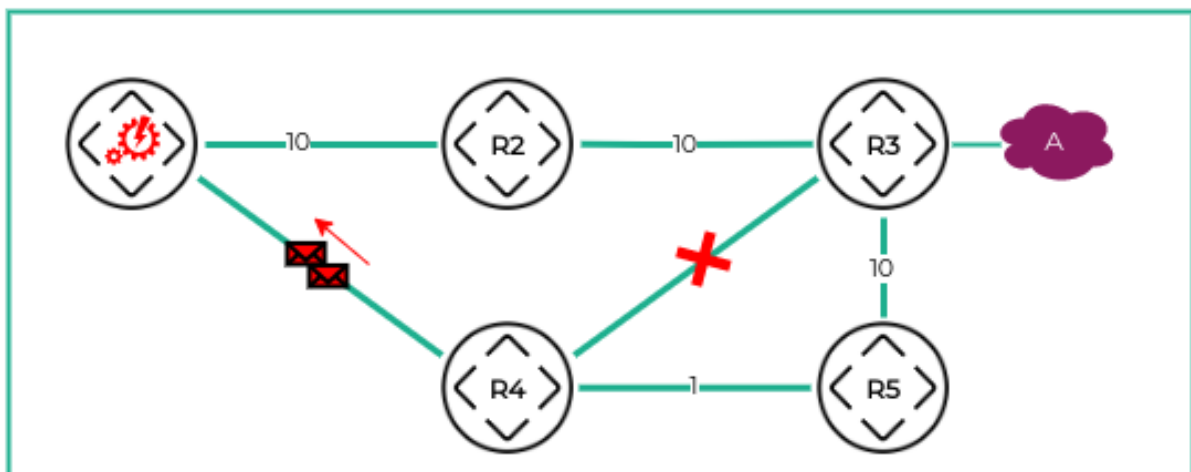


Figure 1.5: Convergence in Segment Routing

### Further Benefits

Segment Routing also brings up other benefits, which have to be mentioned:

- Segment Routing can seamlessly be introduced in existing infrastructures. It can be set up on existing dataplane protocols like **Multi Protocol Label Switching (SR-MPLS)** or **Internet Protocol Version 6 (SRv6)**. Further, it can implement well-know control plane

protocols like [Open Shortest Path First](#), [Intermediate System to Intermediate System](#), [Border Gateway Protocol](#)

- Segment Routing is standardized in RFC 8402 and is therefore vendor independent.
- Segment Routing architecture is made for [Software Defined Networking \(SDN\)](#).
- Many others which go over the scope of this project thesis.

## 1.3 Aims and Objectives

This chapter contains the aims and objectives of this thesis. It consists of two sub-parts. The first part shows the problem, which is present in today's network world. The second part mentions how this project contributes to solving the problem and introduces the reader to the term [Service Chain](#).

### 1.3.1 Problem

In traditional networks, service instances are often localized at the network's edges, centralized or not existent at all. The inclusion of services in networks has been a problematic topic since the early years. Customers wish that traffic is treated from services mainly to make the communication more secure, faster, or improve the service's quality.

The probably most present service in modern networks is firewalling. In emerging networks, this often does not come up to modern standards anymore and has to be extended with services like [IDS](#), [Distributed Denial-of-Service \(DDoS\)](#) protection, Anti Virus Scanner, and more.

A problem that can be faced in today's network is that there is no appropriate solution to differ traffic flows and consume services based on the traffic's requirement.

It is indeed possible to differ packet flows according to source networks, destination network protocols, or applications. For example, it is possible to differ voice traffic from mail traffic, and it is also possible to give them special treatment in the network.

What is missing in traditional networks is a closer relationship between the network with its services and applications. There is no scalable approach to consume a service according to the application's need. As a result, a non-optimized routing delivers the packets from source to destination on maybe not an optimal path.

Solutions, which exist at the moment, have a static manner. They always route the traffic through a service instance, but they can not witness a better service instance for a specific packet flow. What a customer wants from a network is more flexibility. It should be possible to create different traffic and network policies, which go hand-in-hand with user applications and dynamically adapt if better service instances are available. For example, it should be possible to let voice traffic follow the path with the shortest delay and be scanned from the most suitable IDS system. Another example would be, mail traffic that is sent through a firewall instance and a IDS systems by following best total path with the least utilization.

In figure 1.6 a potential problem of today's networks can be faced. It can be assumed that voice traffic is sent from Network A to Network B over the firewall and IDS systems according to a total smallest cost path. This path is shown in purple and is not optimal. A solution, which also would fit the customer's need, is shown in the pink dashed path. A customer wishes to send this voice traffic over the path with the shortest delay and just let it inspect by an IDS system. The problem, which is present, is simple: there is no appropriate solution for the existing customer need.

Imagine a packet flow, which should use the IDS first and then consume the firewall service. In such a traditional network architecture shown, this is simply a hard to solve problem but possibly a need.

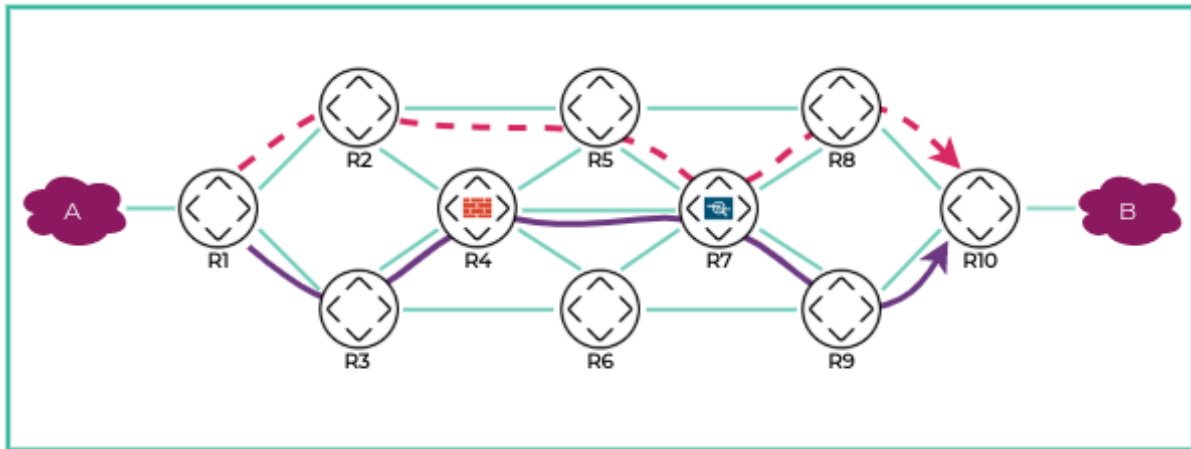


Figure 1.6: Service Consumptions in Traditional Networks

### 1.3.2 Solution - Service Chaining

The solution to the mentioned problem is called Service Chaining. In summary, Service Chaining refers to concatenating of different services and configuring the traffic to flow through these services.

A non-optimal, non-scalable, and in the end inadequate composition of services was also possible in the traditional approach. It was maybe possible to let traffic flow through several services. In the end, this was only static manner, and it was not a solution that fits the customer's needs of today. Because, it often ended toward suboptimal paths. Further, it had not the necessary scalability and flexibility to fit into the modern digital world.

New, with the intelligent implementation of Segment Routing and the source-based routing mechanism, Service Chaining can be made right. Finding an optimum service chain for each communication flow is possible, with the powerful Segment Routing traffic engineering machinery. This approach offers the necessary solution, which is scalable, robust, and flexible.

This project addresses the problem and delivers a way to calculate the most suitable service chain for requested properties.

#### Services

A service refers to any function or concept which can be offered by a network. Such a service can have different forms. It could either be a **Virtual Network Function (VNF)** that runs in a **Virtual Machine (VM)** or a **container** or refers to a physical bound function. A virtual network function could be, for instance, encryption or virus-scanning, whereas the physical-based function, for example, could be an IDS system or firewall.

It is important to differentiate between service instances and service types for the calculation. A service instance refers to a specific function instance of a specific service type. For example, FW-1 could be a service instance of the service type firewall.

## Operating Principle

In chapter [Segment](#) it was explained that a segment could be from a different context - topological or service-based. This mechanism can be used to calculate an optimal service chain for different purposes.

A service chaining calculation has to calculate the most suitable path to be used for a given source and destination network, a set of service instances considered in the calculation, and a metric type.

The result is a segment list that includes the instructions which steer the packets through the most optimal path and the most suitable service instances given in the calculation request. The most suitable path is the path that has the services included and the smallest total sum of the requested metric type.

It is possible to calculate and configure each optimal service chain path with this method. The result is a modern, flexible approach to answer the customer's needs of today.

## Example

The example in figure [1.7](#) shows three different packet flows that consume different services and follow different paths with different characteristics. This sample shows what is possible with Segment Routing and Service Chaining. It illustrates three fictitious examples which could be met within a real-world network.

The purple packet illustrates a packet sent from Network A to Network B. On its way, the packet has to be processed from the most suitable firewall instance and also from the most suitable IDS system. After the Service Chaining Path Calculation has calculated the best path and this rule was configured on the corresponding Ingress, the packet can be sent as planned: In the first step, the Ingress creates the segment list. It adds the label 27003, which in this example, stands for the treatment by the firewall instance behind R3. Besides, it adds the segment 28007, which stands for the IDS instance behind the node R7. The last segment has the ID 16001, which includes the instruction to forward the packet on the shortest IGP path to the destination.

With this instruction, the packet can be steered through the network. In the second step, the packet is processed by the firewall instance and the appropriate segment label removed. The same is done in step 3 from the IDS system. It is interesting to see how the packet from R3 gets to the IDS behind R7. There are two equivalent paths. Since Segment Routing is [Equal-Cost Multi-Path Routing \(ECMP\)](#) aware, both paths are used and load balancing takes place automatically. After the second service instance, the packet can be sent on the shortest path to R10, where the last segment can be removed.

The pink packet flow represents a flow, such as voice traffic, from Network A to Network B, which has only been processed by an IDS system. Therefore, the Service Chaining Path Calculation can calculate the best path for its need. The result is a Segment List that steers the packet through the IDS system allocated at router R3. After the service processing, the packet can send through the smallest delay path to the destination router, which then strips down the last segment and delivers it to the destination endpoint.

The green traffic represents an example, which has to be flooded first through an IDS system and then to a firewall. This is merely possible after the best path has been calculated and configured with this new approach. The segment list includes the instruction let the packet scan first by the IDS behind R2, after than handle by the firewall behind R4 and than send it over the delay most sensitive path to the Egress.

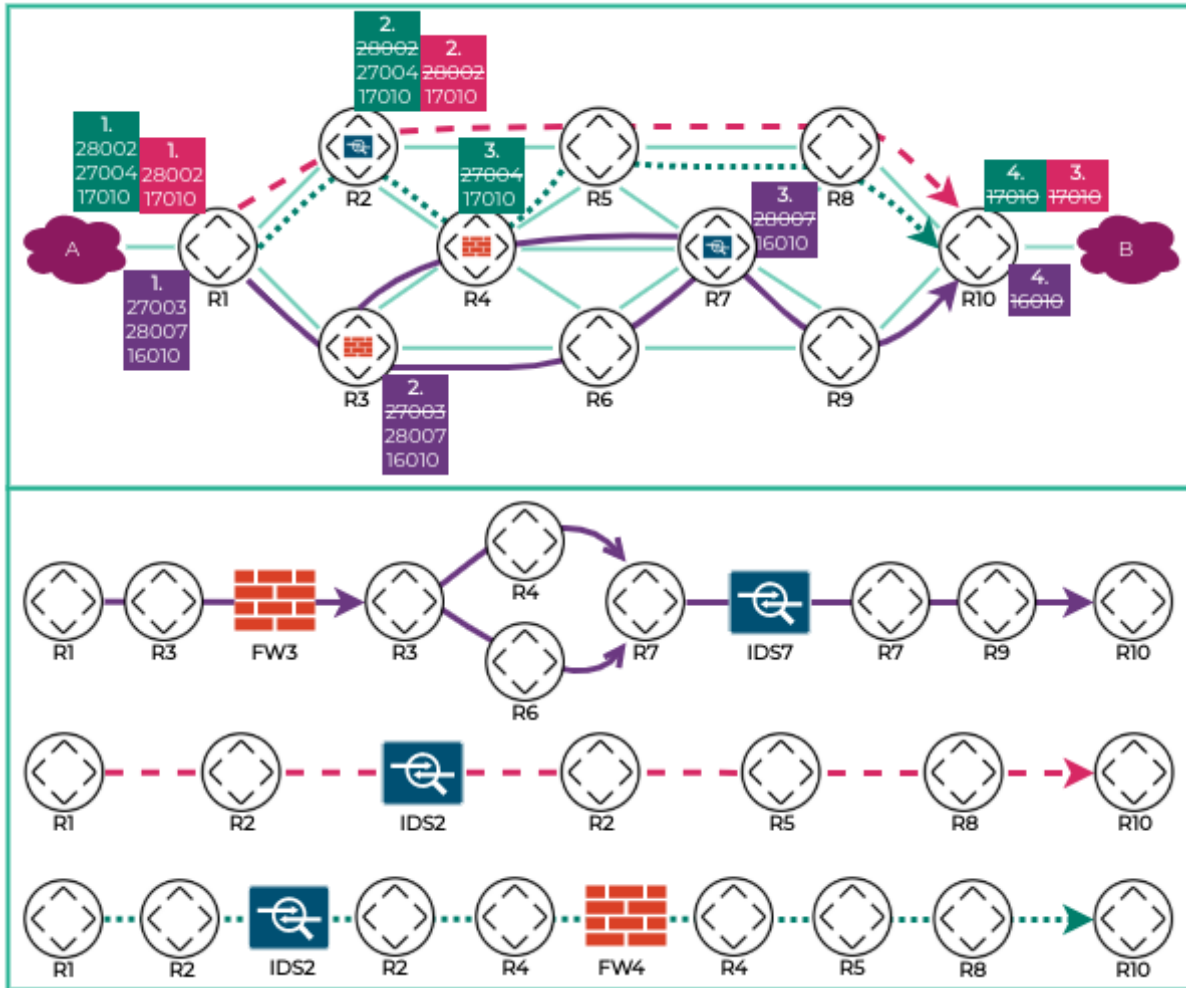


Figure 1.7: Service Chaining in Segment Routing Networks

### Conclusion

Service Chaining is a crucial feature which many customers worldwide would like to use. Up until now, the Service Chaining was not as practicable as it should be. The Service Chaining Path Calculation is a central point to calculate which Service Chain best meets a specific packet flow's characteristics. Together with Segment Routing, it helps to make the network world more intelligent and provides an important step to improve traffic engineering. Further, it helps to bring the network with its services and the applications closer together. All in all, it introduced a solution to give customers a scalable, simple, and stable approach to fit their customer's needs.

# Results

---

This chapter contains the results elaborated in this thesis. It should also show the delimitation of what has been developed in this work and what was not a part of this thesis (see [Distinction](#)). The chapter contains a detailed description of what exactly has been achieved (see [Achievements](#)) and how these results have been achieved (see [Implementation](#)). Furthermore, the solution aspects are analyzed and the procedure is shown.

## 2.1 Distinction

To obtain a clear distinction between the results worked out by the authors, the following describes what the authors have worked out themselves and which parts have been edited externally.

Due to the request of the Institute for Networked Solutions, which is represented by Professor Laurent Metzger, to develop the frontend as professionally as possible and to support further segment routing applications in the future, it was decided that the frontend does not fit into the scope of this project thesis.

In consultation with Professor Laurent Metzger, it was then decided that the frontend should be developed by a developer from the [Institute for Networked Solutions \(INS\)](#) in consultation with the authors of this thesis. This decision allowed the authors to fully concentrate on the development of the backend application. Weekly sessions with the frontend developer allowed both applications to be developed separately and to communicate with each other.

The frontend is mentioned again and again in this thesis because it helps to understand the complete application. However, it was not developed by the authors of this thesis. The complete communication between the frontend and backend application is done via an API which was part of this thesis.

## 2.2 Achievements

In the process of this thesis a backend application together with a polling service were developed, which with the frontend (which was not part of this thesis, see [Distinction](#)) builds the segment routing application [Service Chaining Path Calculation \(SerChio\)](#). In the course of this work, all use cases defined in the Minimum Viable Product could be achieved. These Use Cases should be described briefly in the following.



### 2.2.1 View Topology

A user can either view the network topology graphically on the frontend or access the topology data directly via API. The backend creates a response from the complete topology data, which enables the frontend to draw the network graphically for the end-user. The topology data is always up-to-date, because the current updates are always delivered by the polling service.

The topology as drawn in the frontend can be seen in figure 2.1. The frontend which was not part of this work (see [Distinction](#)) uses the API of the backend which can be seen in figure 2.3.

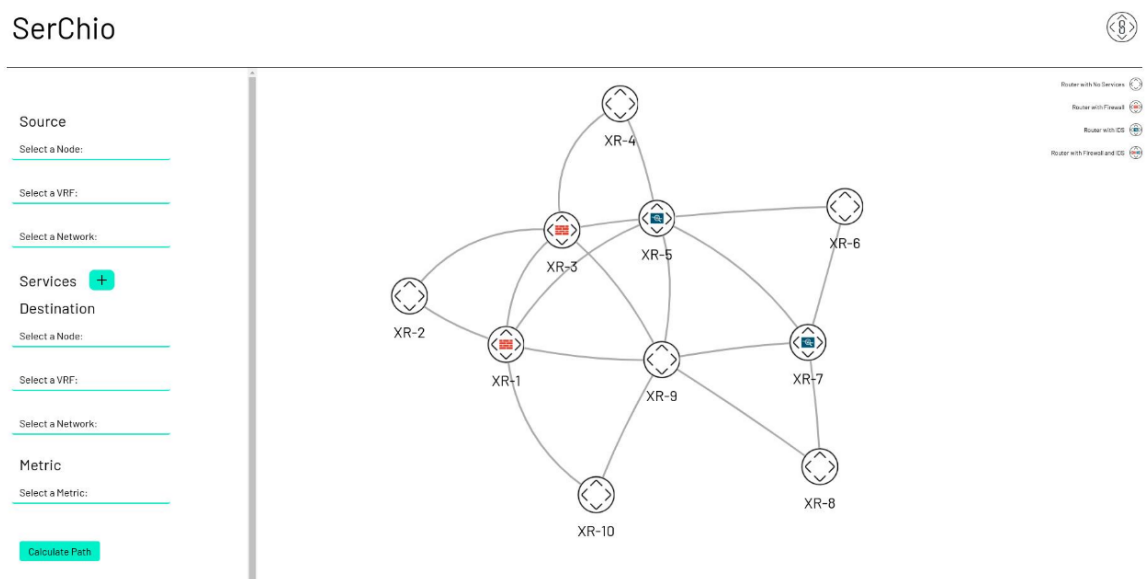


Figure 2.1: Frontend Network Topology

### 2.2.2 CRUD Service Chain

A user can select the source node, source [Virtual Routing and Forwarding \(VRF\)](#) and source network, destination node, destination [VRF](#) and destination network over the frontend. He can also define a service chain consisting of either service types, service instances or a mixture of both. In addition, the user has to select the metric, which has to be included in the calculation. The backend will then calculate the best path based on all selected services and return the result in a way that the frontend can display in the topology (see figure 2.2).

The backend will not only calculate the result for the visualization but also the segment list and provide it to the user.

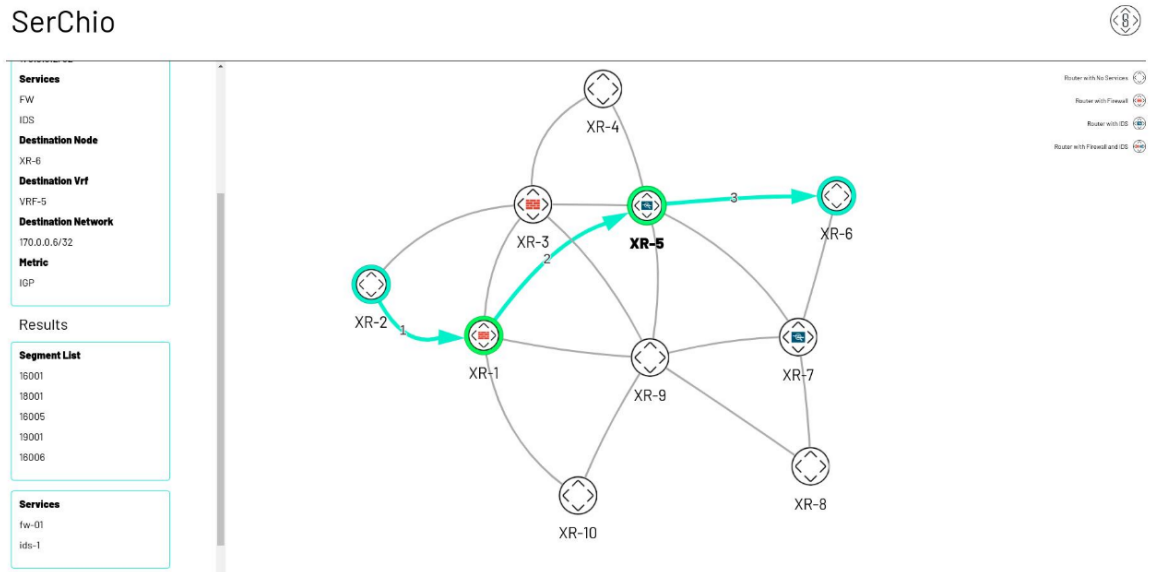


Figure 2.2: Frontend Calculation

### 2.2.3 Data Definition

A user can, via the frontend or directly via the API, view the topology and detailed information about the different nodes and links. The backend application is responsible for ensuring that all required topology or calculation information is available via a predefined format. A complete standardized API has been designed for this purpose. The user or the frontend can use this API to retrieve the defined data.

Figure 2.3 should show the different basic endpoints. The endpoints are nested, for better visualization only the API root is shown.

```

Django REST framework

Api Root

Api Root
The default basic root view for DefaultRouter

GET /api/

HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
  "services": "http://sr-000101.network.garden:8001/api/services/",
  "links": "http://sr-000101.network.garden:8001/api/links/",
  "topology": "http://sr-000101.network.garden:8001/api/topology/",
  "communities": "http://sr-000101.network.garden:8001/api/communities/",
  "vrfs": "http://sr-000101.network.garden:8001/api/vrfs/",
  "networks": "http://sr-000101.network.garden:8001/api/networks/",
  "calculations": "http://sr-000101.network.garden:8001/api/calculations/",
  "nodes": "http://sr-000101.network.garden:8001/api/nodes/",
  "endpoints": "http://sr-000101.network.garden:8001/api/endpoints/"
}

```

Figure 2.3: Backend API Root

An example of a nested endpoint can be seen in figure 2.4. The overview was created using

Swagger, which makes it easier to use the API.

GET	/endpoints/	endpoints_list	🔒
GET	/endpoints/{id}/	endpoints_read	🔒
GET	/endpoints/{parent_lookup_belongs_to_node}/vrfs/	endpoints_vrfs_list	🔒
GET	/endpoints/{parent_lookup_belongs_to_node}/vrfs/{id}/	endpoints_vrfs_read	🔒
GET	/endpoints/{parent_lookup_belongs_to_vrf_belongs_to_node}/vrfs/{parent_lookup_belongs_to_vrf}/networks/	endpoints_vrfs_networks_list	🔒
GET	/endpoints/{parent_lookup_belongs_to_vrf_belongs_to_node}/vrfs/{parent_lookup_belongs_to_vrf}/networks/{id}/	endpoints_vrfs_networks_read	🔒

Figure 2.4: Backend Nested Endpoints Example

## 2.2.4 Service Discovery

A user can view the various services available in the topology. In the topology it is clearly visible which service belongs to which node and to which service type the service instance belongs to.

Figure 2.1 shows that the frontend can classify the services in the topology through the API. For example, there is a firewall instance behind XR-1 and XR-3 and an IDS instance behind XR-5 and XR-7. The services are then also used in the calculation (see [Path Calculation](#)).

## 2.2.5 Path Calculation

The backend is responsible that the user gets the correct path for a calculation request. This use case is closely linked to the [CRUD Service Chain](#) use case.

The backend needs the information from the frontend which is provided by the [CRUD Service Chain](#) use case. The application can calculate the requested path together with the segment list and return it to the user. Therefore, it needs the source node, source VRF, source network, destination node, destination VRF and destination network and the current graph.

The result in the frontend can be seen in figure 2.2.

## 2.3 Implementation

In this section, we will proceed and discuss the solutions how the result was achieved. The implementation of the different components and the most important functions are explained in more detail.

### 2.3.1 Architecture

During the development of the application, the cloud native approach was used. Therefore, the complete architecture was planned and implemented accordingly.

All parts of the application were developed according to the Twelve-Factors[4] and run completely in [Docker](#) containers. This approach means that nothing stands in the way of a deployment on a [Kubernetes](#) cluster, for example. During the development of the containers, great importance was paid to their stability, security, simplicity and speed. All containers are built with a so called Multi Stage Build, which allows very small and fast productive containers.

The complete architecture is designed to be as stable and performant as possible. During performance tests, it was found that even large topologies with thousands of routers can be processed without any problems and calculations can be performed on them. Especially the architecture decisions to use caching systems like [Redis](#) contributed to this.

### 2.3.2 Polling

The polling service was developed to supply continuously the backend with the latest data. The data comes from an external software system that does not offer an API. To take the load from the backend and to generate an additional abstraction layer between the backend and the external software system, the polling service is designed to work independently from the backend.

The polling service was programmed entirely in the Python programming language and was designed to be as stable and robust as possible by means of special exception handling. Thus the service can still continue to run even if several external dependencies fail and resume its work immediately when the dependencies are working again.

The polling service fetches the data from the external software system and compares checksums from previous file versions, which it stores in a Redis Cache, with the new checksums. This way the service always has the certainty that it only transmits data, that is new, modified, or deleted. This takes an enormous load off the backend because it no longer has to decide what data is new, modified, or deleted, but only has to process it.

Through the consequent implementation of abstract classes and the use of dependency injection it was relatively easy to create various mocks which simplified the testing of the polling service. The many dependencies of the service on external containers such as Redis or the [ArangoDB](#) of the external software system could be minimized for testing.

### 2.3.3 Backend

The backend is the brain of the complete Service Chaining Path Calculation application. It processes all requests coming from the frontend and is responsible for data validation, data storage, and calculations of the different service chains.

The backend contains of five different packages, each of which has its purpose and is designed for a specific application.

The complete API is provided by the `api` package. The API is responsible for the entire request handling and is the entry point into the backend application. The API was implemented with the Django Rest Framework, which also emphasizes pagination, links, and nested routes as it is standard in large companies such as Google or Facebook. Through the API the frontend or a user can access, change, or delete all data. Besides, the API is also responsible for carrying out the calculations and returning the results to the user.

The `updatetopology` package was developed so that the API has access to data at all. The backend receives the data from the polling service (see [Polling](#)) via a [WebSocket](#). The `updatetopology` package is then responsible for keeping all data up to date in the backend.

A separate `calculation` package was implemented together with the `graph` package, to perform path calculations. All calculations, that are performed, are performed on a graph that is always kept up to date by the `updatetopology` package. By this procedure, the time needed

for calculations can be minimized, because it is not necessary to create a new graph for every calculation. How the calculation is done is described in section 2.3.4.

### 2.3.4 Calculation

The calculation of the service chain takes place in the backend in the so-called calculation package. A workflow was created which can be seen in figure 2.5, to display the whole calculation graphically.

The workflow gets the input parameters which the user has entered via the frontend. The first step is to create all service chain combinations. The different service chain combinations are created with the help of the **Cartesian Product**. Each service instance is combined with each other instance from the other service types. After the combination has been created, one of them is taken and the calculation is started.

For each service chain combination, the shortest path from the source to the first service instance is calculated using **Dijkstra**. Then it is checked whether the costs are already higher than the costs of the current best service chain combination, that has already been calculated. If this is the case, the calculation is aborted and it is checked, if there are other service chain combinations available. If there are, the calculation is restarted with this new combination, otherwise, the result is returned to the requester.

If the costs are not higher than in the current best calculation, then the calculation will check if there are more service instances available in the currently selected service chain combination. If this is not the case, you can jump directly to the calculation where the path from the last service instance to the destination is calculated. Otherwise, the shortest path from the service instance to the next service instance is calculated. Afterwards, it is checked again if the costs are higher than the costs of the current best service chain combination already calculated, if so, it can be checked for further combinations, if not, it is checked again, if there are more service instances, if so the whole process is repeated until there are no more instances and the costs are not higher than the costs of a combination already calculated.

The last calculation always takes place from the last service instance to the destination. Afterwards, it will be checked again if the costs are higher than for the current best combination already calculated. If this is not the case, the result can be saved temporarily and is in this case the shortest path for the time being. Afterwards, it has to be checked again, if there are still more service chain combinations outstanding for the calculation. If this is the case, the whole workflow starts again. If this is not the case, the previously saved path is the best and therefore the segment list can be extracted out of it. After extracting the segment list, the result can be returned to the requester.

The whole calculation was built up in a way, that there are always abort criteria to the point, where you don't have to continue the calculation depending on the result. By these abort conditions, the performance and the calculation time can be minimized extremely. All calculations are done on a continuously updated and existing graph which is maintained by the graph package.

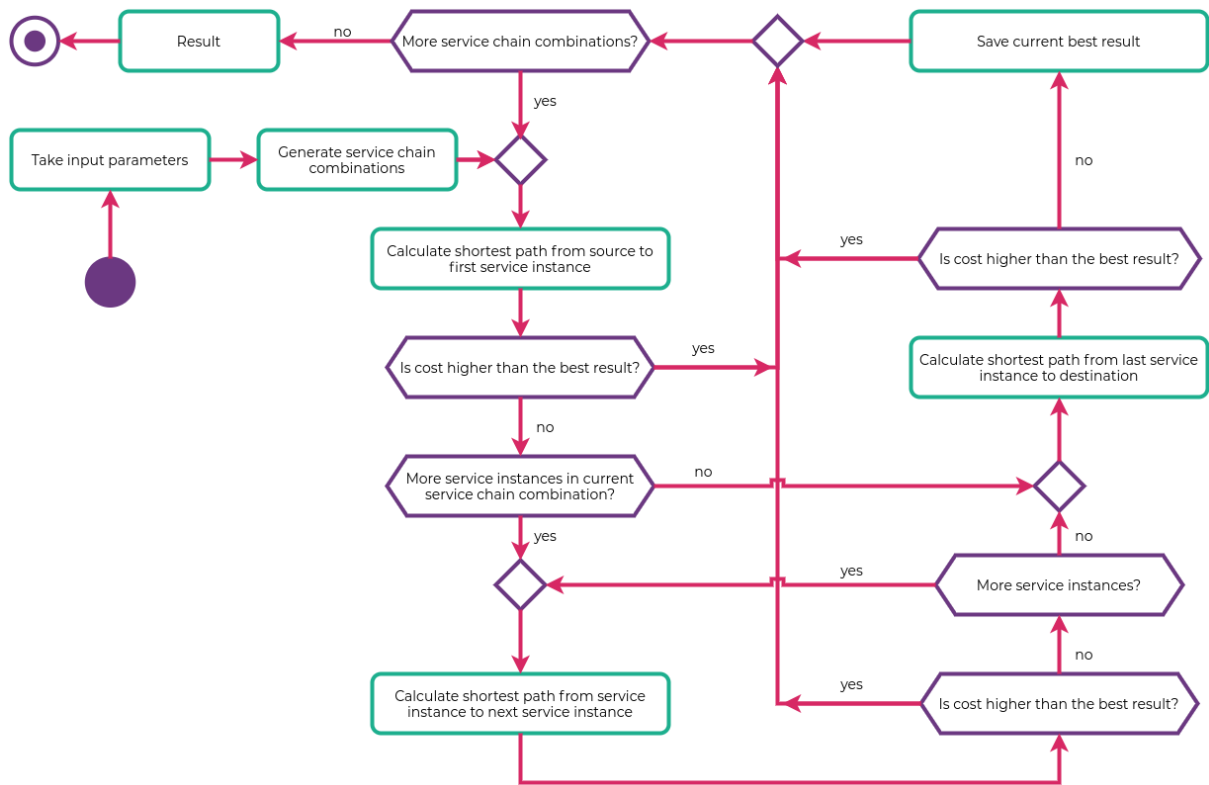


Figure 2.5: Calculation Workflow

### 2.3.5 Frontend

The frontend which was not part of the work (see [Distinction](#)) is the first thing, a user of the Service Chaining Path Calculation application sees. It allows the user to overview the topology and calculate the service chain.

The frontend uses the API provided by the backend application. The API is designed, so that the frontend does not need to contain any intelligence. Since the frontend was developed by a developer from the Institute for Networked Solutions, weekly sessions were held with the two authors to discuss the data definition and to provide feedback to the frontend developer on the usability of the frontend.

# Conclusion





---

The following chapter should take a detailed and critical look at this work. The individual goals should be broken down and examined critically. Besides, a discussion should be held about things that would be done differently in the aftermath. Finally, an outlook should be described how this project will be continued in the future.

### 3.1 Retrospective

The retrospective should provide an overview of the achieved and not achieved use cases. It should also take a critical look at the use cases that were not achieved and analyze, why they were not achieved within the context of this project. This is followed by a discussion section in which possible changes would have to be made in the next step of the project. This section should be inspiring and inform about possible changes and aspects which could be improved in a follow-up project.

#### 3.1.1 Use Cases

This sub-chapter contains information about each use case and if the use case could be covered (represented by ) within the application developed during this project thesis. If the use case could not be completed (represented by ) during the project, there is a description of why the use case could not be entirely completed. All the use cases which belong to the **Minimum Viable Product (MVP)** are marked with a **green** icon (for example ). The optional use cases are highlighted with a **purple** icon (for example )

Besides, there is a short overview about which things could or will be changed in the future.

#### **UC01: View Topology**

*"As User, I regard the network topology to get an overview of the network in place."*

The Service Chain Path Calculation application can fully cover this use case. The API, which was introduced, provides the frontend the topology information in a standardized **JavaScript Object Notation (JSON)** format. The frontend can fetch the topology data and visualize them with this implementation.

It is highly possible that the structure of the topology data may be changed, with changes, improvements, and feature additions of the existing frontend. These changes would happen

in consultation with the frontend developer and improve the fetch and visualization process. Because of a stable implementation, the change of the data representation can be adapted fast.

#### ✔ UC02: CRUD Service Chain

*"As User, I can select the source (ingress) node, source VRF and source network, destination (egress) node, destination VRF, destination network, and a metric, plus a chain of services through which the traffic will be steered."*

This use case could be completed during the project thesis. The application provides the necessary logic and data to create a service chain and process a calculation. Besides, it provides the result of which service chain is most suitable. This result can then be visualized in the frontend.

Because the frontend will change the result's representation in the future, it might be possible that the structure of the result response also has to be changed. These changes will be discussed with the front end team. It is possible to adjust the result structure without any significant backend modifications.

#### ✔ UC03: Data Definition

*"As User, I need an appropriate data definition, in order to use the application properly"*

The data definition use case was accomplished entirely in this project. The retrieval of information is simplified, during a present well documented API.

As described in the use cases above, changes in the frontend may also result in structural changes in the data definition. As mentioned before, it should not take much time to adapt to these changes.

#### ✔ UC04: Service Discovery

*"As User, I observe the different service types and nodes to get an overview of which various services are present and where they are located in the network."*

The created application can deliver the information, where services are located in the network and which service types are present. These data lead to the visualization of the services in the frontend. Hence, the use case is satisfied.

However, it has to be mentioned that the external system can not discover the services, and therefore, the backend can not receive relevant information through the polling service. Thus, it was decided to implement the feature to create the services manually in the backend application to implement the service chain calculation.

In the future, when the external system can deliver the necessary service information, the application will implement that. Because the services can already be managed, it is expected to adapt to new changes quickly.

#### ✔ UC05: Path Calculation

*"As User, I get the most appropriate path after selecting the service chain."*



This use case could be completely fulfilled. The computations all take place in the backend with a performance, that allows calculations on topologies with thousands of nodes.

In the future and as soon as further required data is available in the external software system [Jalapeño](#), the segment list can be further improved and optimized.

#### ✖ UC06: Path Approval

*"As User, I check the re-calculated path after a topology change to verify the calculation from the application."*

Despite the possibility to inform the backend about topology changes, it was decided together with the supervisor, that the authors will focus on performance and system stability especially for the computation. Because of this focus, this use case did not fit into the scope of this project.

However, the use case is not completely off the table but will be continued in further work. More about this can be read in [Outlook](#).

#### ✖ UC07: Login

*"As a non-authenticated user (Anonymous), I log in to the application to authenticate and use the application's additional functionalities."*

By deciding together with the supervisor, that the core functionalities of the [SerChio](#) application have priority over the additional use cases, the focus was completely on these functionalities and to implement them as clean, stable, and performant as possible. Therefore, this use case was no longer within the scope of this project.

#### ✖ UC08: CRUD Global Configuration

*"As Admin, I change the global configuration, to change roles or give access to different functionalities."*

Since this use case is strongly linked to the use case *Login* which did not fit into the scope of this project, it did not make sense to continue the use case *CRUD Global Configuration*. This use case would come back into focus, with an implementation of the use case login.

### 3.1.2 Discussion

The following chapter contains possible topics to improve or discuss. It contains a collection of critical aspects existing in the current solution. It does not contain explicit solution approaches. These will be mentioned in the [Outlook](#) section.

#### Microservice Communication

The communication between the microservices, specifically the backend and the polling application, is currently made via the [WebSocket](#) protocol. This protocol is standardized, runs over [Hypertext Transfer Protocol \(HTTP\)](#), and is easy to use. However, this protocol also has a disadvantage: if a microservice goes down, messages could be lost. The loss of messages has never occurred during the development and testing phase but could theoretically happen. It has to be further discussed, if another communication protocol or concept can improve communication.

### **Revision Persistence Mechanism**

Updates are received, processed, and information persisted into a relational database in the current software solution. The translation from Python objects into the database takes its time. Also, if the information is requested afterwards, the corresponding information must be fetched from the database. This process, indeed, is optimized but still takes time and can lead to a bottleneck. It has to be evaluated, if information can be moved to a cache system to improve overall performance.

### **Preprocessing**

The calculation will be triggered in the current solution after a user requests a calculation with specific input parameters. Afterwards, the calculation will be executed on the maintained graph, and the result will be returned. It must be examined if a preprocessing, the precalculation of all possible service chains, can improve the calculation process, especially according to updating and finding new, better suitable paths after a topology change.

### **Adoption Data Definition**

In the future, the requirements for the frontend could change noticeably, which would mean that the data definition is no longer sufficient to meet all needs. Therefore, it has to be thought about, if, and how the data definition in the backend can be changed. So that the logic can be move from the frontend to the backend, in such a way that the frontend can still display the data as easily as possible. The goal should be, that the frontend does not have to implement logic for the data visualization in the future.

### **Optimize Caching**

The polling service and the backend currently use a cache to have the data quickly accessible. This works without problems but can lead to inconsistencies if, for example, the cache of the polling service is no longer available. In such a case the polling service would send all data to the backend again, because it has no previous data to compare the checksums. The question is, if there is a way to have the advantages of cache speed but still keep some security of data in case of crashes, without losing the advantages of the in-memory cache.

## **3.2 Outlook**

A reliable and robust application was created within this project thesis to process network updates from an external system and offer a standardized interface to calculate the optimal service chains.

In particular, it should be mentioned that this project thesis will be continued within a follow-up bachelor thesis. Further work will be done with the consisting team, advisor, and project partner.

This chapter contains an outlook about which elements the authors like to improve in the futural work. Specific solutions were considered in advance, which may be applied in the bachelor thesis. The chapter also contains a description of possible innovations, which can be considered in the bachelor thesis.

### **3.2.1 Improvements**

In the future, various aspects that were implemented in this project should either be improved or exchanged with another technology. The planned improvements can be seen in the sections

below.

### **Communication**

In the communication between the microservices, the web sockets are to be replaced by queues. These queues provide more security against data loss and simplify communication between the microservices. This would especially affect the communication between the polling service and the backend, where there is currently complex error handling, which would be simplified by using queues.

### **Persistence**

It has to be verified, whether it is possible to outsource data into a cache system or not. The first clarification is to examine which specific data are possible to save in a cache system. The next step is to show, if the movement has a relevant positive influence on the system's overall performance.

### **Calculation**

It is possible to optimize the calculations and save time through the targeted precalculation of service chains. For this purpose, a proof of concept should first be prepared to see whether this procedure leads to the expected result. If so, this change would be taken into consideration for the implementation.

### **Data Provision**

The data definitions via the API should be updated in consultation with the frontend developer of the Institute for Networked Solutions. The definitions should be changed to allow the frontend developer to access the different aspects of, for example, the topology and the different topology information as easily and without a lot of logic. This should make features in the frontend available faster because no complicated implementations have to be made.

### **Caching**

A Redis cluster should be put into operation to increase the security against data loss in the cache system. It should replace the individual Redis containers which are currently in use and increase the availability and stability of the cache system with its cluster functions. This change would also lead to less error handling in the backend and the polling service.

## **3.2.2 Innovations**

In the following sections, the planned innovations for the Service Chaining Application should be presented. All these functionalities are currently not yet included in the application and should be added to the application alongside the [Improvements](#).

### **Service Management**

As soon as the external system can aggregate the network services, the information should be made available in the Service Chain Path Calculation application. For this purpose, it should be possible to retrieve and manage the data, as is already being done with current information. The service instance information should thereby automatically be added, updated, or removed in the application context if the appropriate change occurs in the related network.

### **Flexible Algorithm**

Flexible Algorithms, a powerful concept of Segment Routing, should help evolve the Service Chain application into a Service Programming application. This use of this new mechanism should add more flexibility, features, and intelligence to the application. For instance, it should be possible to optimize the segment list for non-IGP paths. Another example could be to tell the application which paths should be included or avoided. In this context, the topic of [Affinity Attribute](#) is indispensable.

### **Write Policies**

The functionality of the application should be extended to write rules on Ingress routers. It should be possible to check the service chain result visualized in the frontend and then release it to be written to the appropriate Ingress node. The policy should enforce, that the appropriate packets are steered through the correct path and are processed by the most suitable service instances, which are calculated by the application. Furthermore, it should also be possible to update existing policies or delete them completely.

### **Automated Recalculation**

As soon as the backend learns of changes in the topology by the polling service, it should recalculate all already calculated service chains. This is to ensure, that the user always receives an up-to-date service chain. The changes should be accepted automatically if the old service chain can no longer function due to the topology changes. A service chain, that is no longer optimal due to a change but still works, should be confirmed manually by the user. This should ensure, that a functioning service chain always exists in the system and that the user has control over his service chains.

### **Update Notification**

In the future, the frontend should be notified automatically when there have been changes in the topology. This would ensure that the user can always work with the most current topology, without having to update the topology manually.

### **Kubernetes Deployment**

The future productive deployment should take place on a Kubernetes cluster, this is not yet the case and should therefore be implemented in the follow-up project. Deployment on the Kubernetes Cluster would open new ways to scale and maintain the application. The service chain application and the external software system Jalapeño would also agree on a joint deployment and continue to follow the cloud-native approach.

**Part II**

**Project Documentation**

---

# Requirement Specification

---

This chapter contains the complete requirement specification which was elaborated for the project. The requirements were adapted to the wishes of the industry partner. Additionally non-functional requirements were compiled which can be found in the second part of the requirement specification.

## 4.1 Use Cases

Different colours were selected to distinguish whether a user story or a use case belongs to the MVP or not. The explanation of the colours can be found below.

Use Case - Minimum Viable Product
Use Case - Additional

Table 4.1: Use Cases Color Description

The Minimum Viable Product was defined together with the supervisor and the industrial partner. In the following section, the Minimum Viable Product only refers to the Main Scenario within a Use Case. The Extensions are always optional and will be implemented if enough time is available.

### 4.1.1 Actors

Actor	Description
Anonymous	Anonymous describes the role of an unauthenticated entity. Successful authentication is required to grant the necessary permissions so that additional use cases can be executed.
Admin	An admin is an authenticated entity that has additional rights on the application. The admin can use the various functionalities of the application and also configure and edit the different users and roles.
User	A user is an authenticated entity that can use the various functionalities of the application.

Table 4.2: Actor Description

### 4.1.2 Use-Case-Diagram

The use case diagram below is intended to provide a quick and easy overview of the application's functionalities.

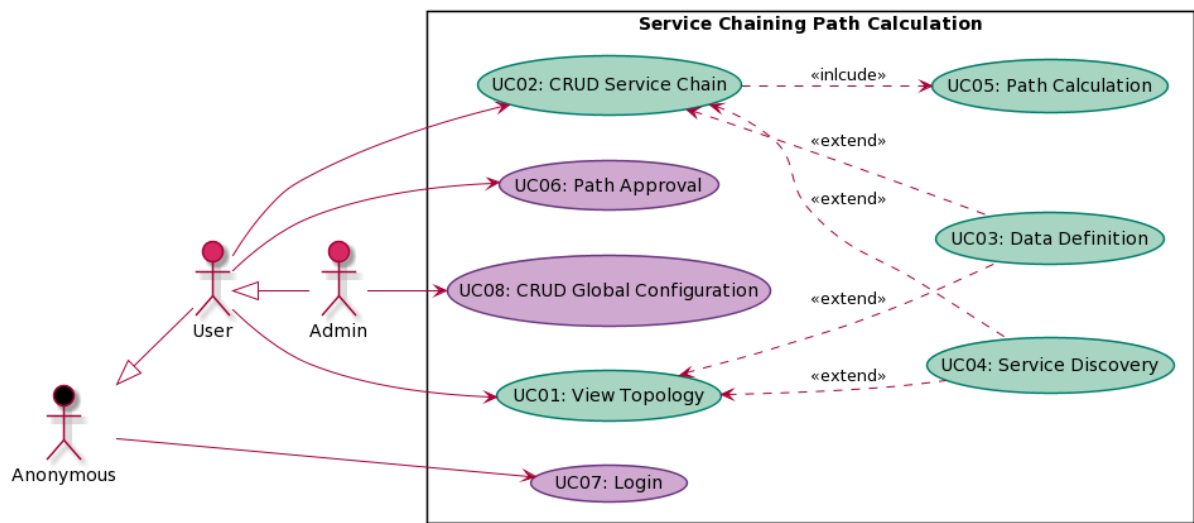


Figure 4.1: Use-Case-Diagram

### 4.1.3 Use-Case-Description

All use cases are described starting from a standardized user story template, which can be found below. This method was taught at the module [Project and Quality Management \(PmQm\)](#) at the University of Applied Science of Eastern Switzerland. Through the use of this method, all use-stories have the same format, which makes it easier to read and to understand. The last part of the user story has been kept optional. The motivation is only added if it gives the reader an additional value.

#### 👤 User-Story 00 - User Story Template

As »actor«, I »verb«»function«, »motivation«.

To describe the more complex use cases not only as user stories but also in a standardized and a lot more detailed form, work was done according to Larman [9]. This approach is also taught at the University of Applied Science of Eastern Switzerland and follows international standards.

Information that is not relevant to the reader is omitted. The following information belongs to a fully-dressed use case according to Larman, are not relevant for this thesis: Scope, Level, Extensions, Special Requirements and Technology and Data Variations List.

**UC01: View Topology****User-Story 01 - View Topology**

As User, I regard the network topology to get an overview of the network in place.

<b>Primary Actor</b>	User
<b>Overview</b>	A User should overview the whole present network topology with all services.
<b>Stakeholders and Interests</b>	User: Wants to get an overview of the existing network topology.
<b>Preconditions</b>	<ul style="list-style-type: none"> <li>• The User is authenticated within the application.</li> <li>• The User has the permission to overview the topology.</li> </ul>
<b>Postconditions</b>	<ul style="list-style-type: none"> <li>• The User can overview the topology in the form of graphs and vertices.</li> </ul>
<b>Main Success Scenario</b>	<ol style="list-style-type: none"> <li>1. The User wants to have an overview of the network topology.</li> <li>2. The User can regard the network topology.</li> </ol>
<b>Frequency of Occurrence</b>	As often as required

Table 4.3: UC01: View Topology - Fully Dressed Description



**UC02: CRUD Service Chain****User-Story 02 - CRUD Service Chain**

As User, I can select the source (ingress) node, source VRF and source network, destination (egress) node, destination VRF, destination network, and a metric, plus a chain of services through which the traffic will be steered.

<b>Primary Actor</b>	User
<b>Overview</b>	As User, I can select the source (ingress) node, source VRF and source network, destination (egress) node, destination VRF, destination network, and a metric, plus a chain of services through which the traffic will be flooded.
<b>Stakeholders and Interests</b>	User: Wants to get the most appropriate path between the source and destination node, which necessarily leads through the instances of the selected services in the service chain.
<b>Preconditions</b>	<ul style="list-style-type: none"> <li>• The User is authenticated within the application.</li> <li>• The User can regard the network topology.</li> <li>• The User belongs to a group with permission to do at least one of the CRUD operations.</li> <li>• All service instances are discovered and known to the application.</li> </ul>
<b>Postconditions</b>	<ul style="list-style-type: none"> <li>• The user gets the path with the smallest metrics of the selected metric type, which flows from source to destination and includes the instances of the selected services.</li> </ul>

**Main Success Scenario**

1. The user likes to calculate the best service chain path.
  2. A list of possible source nodes, which are available in the network, is shown. A node has to be a **Provider Edge (PE)** that it can be used as a source node in a calculation. The user selects the source node by choosing the router's hostname.
  3. A list of possible source VRF's, which are configured on the previously selected source node, is shown. The user selects the source VRF by choosing the name of a specific VRF.
  4. A list of possible source networks, which belongs to the previously selected source VRF, is shown. The user selects the source network by choosing a specific network. A network consists of an IP address and inherent subnet mask (e.g., 170.0.0.1/32)
  5. A list of service types and the configured service instances are shown. The user defines the service chain by selecting up to two different service types (FW or/and IDS) and the inherited service instances.
  6. A list of possible destination nodes, which are available in the network, is shown. A node has to be a **PE** to be a candidate for a destination node in a calculation. Besides, to be a valid destination node, the destination node has to possess at least one VRF where one export tag matches the selected source VRF's export tags. The user selects the destination node by choosing the router's hostname.
  7. A list of possible destination VRF's, which are configured on the previously selected destination node, is shown. Only the VRF's which has at least one export tag with the source VRF in common are valid and shown. The user selects the source VRF by choosing the name of a specific VRF. The name of the source and destination VRF has not to be the same.
  8. A list of possible destination networks, which belongs to the previously selected destination VRF, is shown. The user selects the destination network by choosing a specific network. A network consists of an IP address and inherent subnet mask (e.g., 170.0.0.2/32)
  9. The user selects the metric, which is used for the calculation of the most suitable path. Available metrics are **TE**, IGP, or Delay.
  10. After the calculation, the application returns the links included in the most suitable equal cost multi-paths from source to destination, which steers through the selected service instances. A segment list is also included in the response, which can be used to forward the traffic on the calculated **ECMP** paths through the network. The order of the service instances is essential and has to adhere. If there are two or more total paths with equivalent metric numbers, the first calculated complete path will be returned to the user.
  11. The user can inspect the links of the best path and the segment list from source to destination with the most appropriate service instances, according to the full path, in it.
-

---

**Extensions**

- 5a The user selects a combination of specific service instances and general services or only general services. (e.g., Firewall-instance-01 and IDS service)
1. A list of general service types and specific service instances available in the network is shown. The user selects for one service specific instances, which should be included in the calculation. The second input is a general service type, where no specific instance is selected. This choice implicitly triggers that all available instances of this service type should be considered in the calculation.
  2. - 5. Identical with step 6 to 9 in the Main Success Scenario.
  3. After the calculation, the application returns the links included in the most suitable equal cost multi-paths from source to destination, which steers through the most suitable service instances. The result contains information on which specific service instance of the selected general service type is used to end up as the best total path. Like in the main scenario, also, the segment list is transmitted. In this calculation, the order of the service instances is essential and has to adhere as well. If there are two or more total paths with equivalent metric numbers, the first calculated complete path will be returned to the user.
  4. Identical with step 11 in the Main Success Scenario.
- 5b The user selects one or more affinity constraint(s).
1. Additionally to 5a: The user selects the different affinity constraint(s), which should not be included in the best path.
  2. Identical with 5a.
  3. Additionally to 5a: The returned best path does not include affinity constraints.
  4. Identical with 5a.
- 10a There is more than one path with an equivalent metric.
1. The application has calculated several most suitable paths with equivalent metrics and return the paths to the user. Like in the above use cases, the path calculation depends on selecting the specific service instances and/or general service types and the order of the services is independent.
  2. The user can view the list of the most appropriate paths, which all have the same metric count.
  3. The user can select its personally desired path to configure it in the network from the list of best paths.
- 10b The diversity of the different service instances will influence the return of the most appropriate path.
1. The application calculates the most suitable path from source to destination, like in the other scenarios, returns it to the user. If there are two or more paths with equivalent metric numbers, the diversity of the various service instances will be considered. Diversity means that for each specific service instance, it is calculated how many best paths already pass through that particular service instance. Then for each potential best path, a total value from all service instances is calculated. The path that has the least total diversity value is returned.

---

<b>Frequency of Occurrence</b>	As often as required
--------------------------------	----------------------

---

Table 4.4: UC02: CRUD Service Chain - Fully Dressed Description

**UC03: Data Definition**
 **User-Story 03 - Data Definition**

As User, I need an appropriate data definition, in order to use the application properly.

---

<b>Primary Actor</b>	User
<b>Overview</b>	The User wants to check the topology visualization and CRUD the service chain. Therefore he needs data in the frontend application, to achieve this an appropriate data definition is required, in order to exchange data between frontend and backend.
<b>Stakeholders and Interests</b>	User: The User wants overview the topology and to CRUD the service chain.

---

Table 4.5: UC03: Data Definition - Casual Description


**UC04: Service Discovery**
 **User-Story 04 - Service Discovery**

As User, I observe the different service types and nodes to get an overview of which various services are present and where they are located in the network.

**UC05: Path Calculation**
 **User-Story 05 - Path Calculation**

As User, I get the most appropriate path after selecting the service chain.

## UC06: Path Approval

 User-Story 06 - Path Approval

As User, I check the re-calculated path after a topology change to verify the calculation from the application.

<b>Primary Actor</b>	User
<b>Overview</b>	The User can check the path, which was re-calculated after a topology change, and approve it.
<b>Stakeholders and Interests</b>	User: The User wants to have the opportunity to check the re-calculated path and to approve it manually.
<b>Preconditions</b>	<ul style="list-style-type: none"> <li>• The User is authenticated within the application.</li> <li>• The User has the permission to approve re-calculated paths.</li> <li>• A topology change, which leads to a re-calculation, shows that another path which leads through other service instances is better. Therefore a more appropriate path was found, however the path has to be approved, because other service instances are involved.</li> </ul>
<b>Postconditions</b>	<ul style="list-style-type: none"> <li>• The User has the possibility to approve the new path manually.</li> </ul>
<b>Main Success Scenario</b>	<ol style="list-style-type: none"> <li>1. The user likes to manually approve the re-calculated path.</li> <li>2. A message, with the information about a path change, is shown.</li> <li>3. The user can regard the new and old path in the topology.</li> <li>4. The user can manually approve the new path.</li> <li>5. The new path is shown in the topology.</li> </ol>
<b>Extensions</b>	<p>*a The user will cancel the approval.</p> <ol style="list-style-type: none"> <li>1. The application will not make any changes to the topology.</li> <li>2. The re-calculated paths are listed until they have been approved.</li> </ol> <p>2a The message is not shown because an approval attempt from the user has already been made.</p> <ol style="list-style-type: none"> <li>1. An icon is displayed which shows the current amount of unapproved paths.</li> <li>2. The re-calculated paths are listed in a separate list.</li> </ol>
<b>Frequency of Occurrence</b>	After each topology change, which leads to a path change.

Table 4.6: UC06: Path Approval - Fully Dressed Description

**UC07: Login****User-Story 07 - Login**

As a non-authenticated user (Anonymous), I log in to the application to authenticate and use the application's additional functionalities.

**UC08: CRUD Global Configuration****User-Story 08 - CRUD Global Configuration**

As Admin, I change the global configuration, to change roles or give access to different functionalities.

**4.2 Non-Functional Requirements**

The Non-functional Requirements were created according to the FURPS+ principle, which Larman[5] describes in his book and as it is also taught at the University of Applied Sciences of Eastern Switzerland.

**4.2.1 Functionality****Suitability**

Because the application is only accessible internally by specific networks and users, an extensive security implementation is not necessary for the time being.

**Accuracy**

At any time, the application should return at least one of the most appropriate paths between the source node and the destination node according to the selected service types and instances.

**Interoperability**

The application should interact with an Arango database that contains similarly structured data like the Cisco Jalapeño application version 1.0 is using. This non-functional requirement has the positive effect that the application can be tied up effortlessly with a mature and utterly operating Jalapeño application in the future.

**4.2.2 Usability****Understandability**

The various service instances should be visually emphasized to recognize where the network topology's different service instances are located.

**Operability**

A user should calculate the most appropriate service chain path from source to destination without needing an introduction to the application first.

### 4.2.3 Reliability

#### Availability

The application should be available for 99% of the time.

#### Recoverability

The software should be programmed according to the Cloud Native Standard[8], which includes a simple redeployment.

#### Fault Tolerance

Wrong values from the frontend should not influence the correct calculation of the path. If the wrong values arrive in the backend, an error message is returned.

### 4.2.4 Performance

#### Capacity

The application should be able to calculate paths in topologies with up to 1000 routers and up to two different services (FW and IDS) instances in it.

#### Time behavior

As soon as topology changes are saved in the Arango database, it should not take more than one minute to process these updates and make the updates available in the backend.

### 4.2.5 Scalability

The settings should be adjustable so that the backend can process updates before the next request is made to the Arango database.

The back-end counter should be scaled automatically, which means if there are many path calculations requests, an additional back-end pod should be started to share the load.

### 4.2.6 Maintability

#### Analysability

It should be possible to change the log level of the application. If the log level is set to debug mode, the application should write information about the current events to the standard output.

An all-time up to date log file should be written that includes information about the service path calculations and also occurred errors.

### 4.2.7 Traceability

All code changes should be traceable and commented on within the version control tool (Git-Lab).

## Domain Analysis

---

This chapter contains the analysis of the system under development. It shows the administrative concepts and their relations. It should help give an overview of which components are present in the domain and describe how each element interacts with each other logical block.

### 5.1 Domain Model

The following illustrations visualize the given domain. It shows the different models and how they are related to each other. A more detailed description of each element and the different Associations can be found under the section [5.2](#).



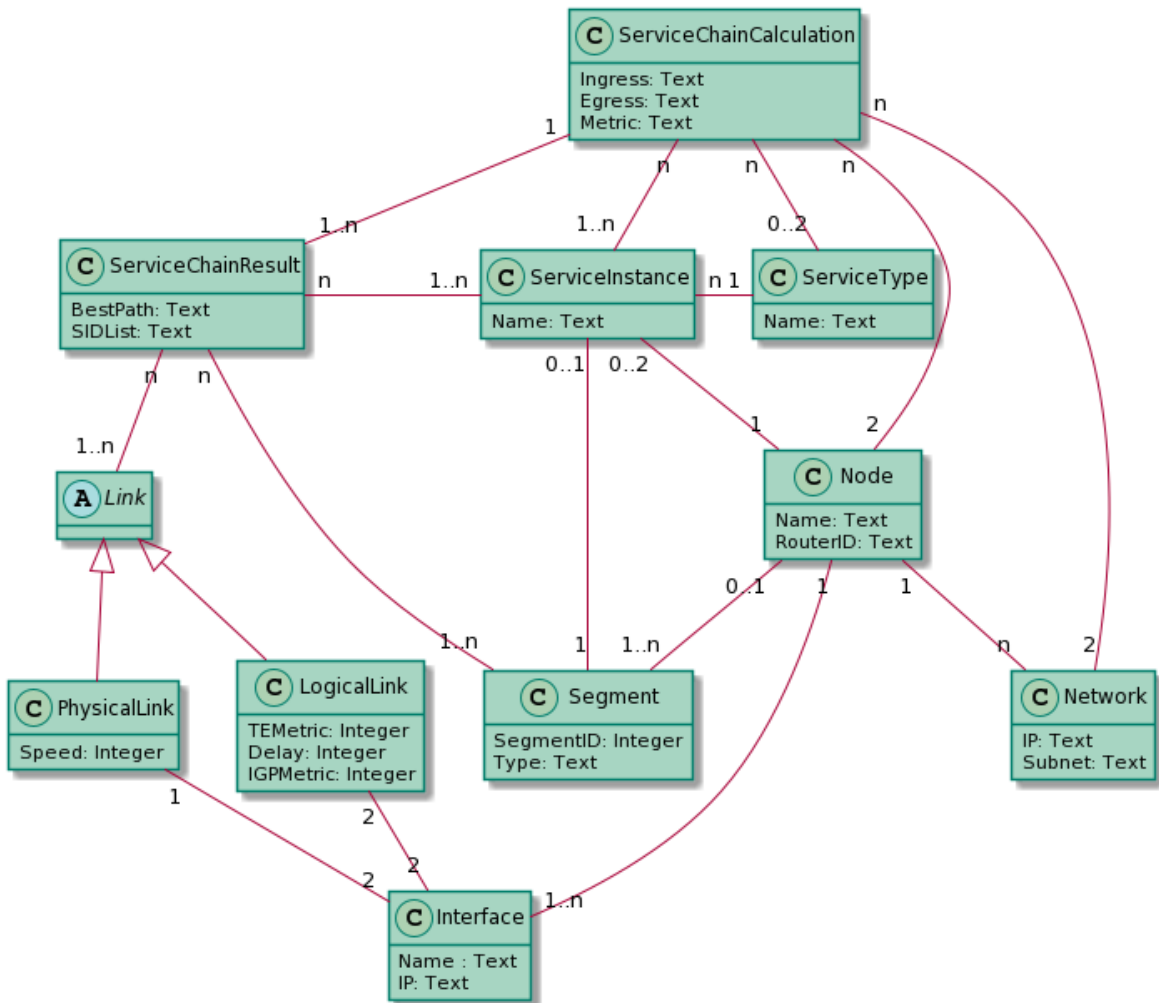


Figure 5.1: Domain Model

## 5.2 Administrative Concepts

To understand the individual administrative concepts, a basic knowledge of segment routing is required. An introduction to the topic Segment Routing can be found in the chapter [Segment Routing](#).

Because the thesis's concepts, especially Segment Routing, are not trivial, it was tried to attempt a fictive analogy from real life. This analogy is greatly simplified and hence does not always make 100% sense. Still, it should help understand the concepts from a high-level perspective and give non-tech people the necessary information to understand and discuss these topics. Analogy information is highlighted in boxes like the following:

### Example Analogy

Here comes the description of the example analogy.

This fictional example is about a bus transport company which offers a unique service. Instead of just bringing people from a city to another city, there is some stopover in some attractions.

The attractions can be, for example, an amusement park or an outstanding shopping mall or similar things.

The goal of the thesis and the application under development is to create a Service Chaining Path Calculation. More detailed, this means the most suitable path and segment list of one network of the source node (Ingress) to another network of the destination node (Egress) should be found. The packets should necessarily flow through the selected services on their way from source to destination. The best suitable path and hence the result depends on the calculation metric. If different metrics are used in a calculation, it's possible that also the result differs.

For the analogy, this means the transport business likes to find the most appropriate path from a city's District to another city's District with a given set of stops at attractions or other cities. The company also likes to introduce some different metrics/characteristics, which can influence the calculation. Besides the shortest path, it should also calculate the path, including the shortest delay or the route with the fewest possibilities to be involved in a traffic jam. According to the characteristics, the result of the journey can turn out differently.

More detailed information about this domain's administrative concepts and the analogies to each of these concepts can be found below. It should help to gain enough basic knowledge to understand the domain.

### 5.2.1 Node

A Node refers to any device in a specific Segment Routing Domain which understands and actively participates in Segment Routing. Typically, a Node is a router but could be any host or device in general. The main criterion is that the device can choose and steer a packet through the given topology by following the segments' instructions.

Usually, a Node has several network Interfaces connected. Some Nodes have Networks assigned, some not. Only the Nodes with Networks assigned (PE) can be used as source and destination Nodes for the ServiceChainCalculation. In Segment Routing, the source Node is also called Ingress because the packets from the source Network enter the Segment Router Domain at this Node. The same applies to the destination Node of a ServiceChainCalculation: This Node is called Egress because the packet leaves the Segment Routing Domain at this point.

Because of the above reasons, a Node can be included in aServiceChainCalculations but has not to. It can have configured one or more Segments, known by other routers in the Segment Routing Domain. Several Nodes have ServiceInstances connected. In particular, this stands for that a specific ServiceInstanc can be reached over this unique Node. It's only possible to have one ServiceInstance of each ServiceType connected to a particular Node.

#### Node Analogy: City

A Node refers to the City in which Bus Stops are possible. Bus Stops can be where people are getting on the bus or do leave the bus. The latter is possible on the destination City of the journey or at some stopover at an Attraction (ServiceInstance).

On a journey, there may not only the Cities be involved, which are the source or destination City, but also the Cities where the bus drives through are essential for the Journey Calculation (ServiceChainCalculation).

Source and destination Cities are only taken into account if at least one District (Network) is present. Accordingly, there are Cities, which not have a District and some they have.

A City can therefore be involved in a Journey Calculation or not. In a City, there can be an Attraction, but it does not have to. It's valuable to know that only one Attraction of a specific Attraction Type (ServiceType) can exist per City. Furthermore, a City always has at least one BusStop (Interface) attached because a City without a possibility to stop for the bus is not of interest to the company. A City always has at least one Address/Coordination (Segment) attached, which tells how to reach the City.

### 5.2.2 Network

In this domain, a Network is a specific computer network assigned to a PE and typically belongs to a specific customer. Networks are used as source and destination information in a ServiceChainCalculation. The ultimate goal of the application is to be able in the future to influence the communication between the Networks in such a way that the individual packets can be sent through different Service Instances.

A Network always belongs to exactly one Node, more precisely a PE. A specific Network can be involved in non or many ServiceChainCalculations.

#### Network Analogy: District

A Network refers to a City District, where the transport company picks or drops off people. In a Journey Calculation (ServiceChainCalculation), it's crucial to mention which is the source District and which is the destination District. Each District belongs to exactly one City (Node). Moreover, a District can be included in no or multiple Journey Calculations.

### 5.2.3 Segment

The Segment is an essential construct of Segment Routing. Each Segment describes a different operation, which action a Node has to execute if a packet with a specific Segment is received. These instructions can be versatile and have to be observed in a particular context.

A Segment in our domain has to belong either to a particular Node or to a specific Service Instance. As mentioned above, the Segment has other meaning if the Segment belongs to a Node or a ServiceInstance. A Segment may be available in zero or more ServiceChainResults.

#### Segment Analogy: Coordination/Address

A Segment refers to Coordination or Address, which shows the bus driver the interim destination, where the guest wants to go next. It can be imagined like a roadmap, that tells exactly where the bus driver has to drive through to reach the desired destination. After reaching a destination, the bus driver has to head to the next address. Therefore it types in the next address in the navigation device. This is repeated until the final destination is reached. The Coordination/Address belongs either to a City (Node) or an Attraction (ServiceInstance). Also, it belongs to zero or more best paths (ServiceChainResults).

### 5.2.4 Interface

An Interface in this domain is the cut between the involved Segment Routing Domain and a particular Node. Furthermore, the Interface can be seen as the point of intersection between the Node and the Links.

An Interface always has to belong to precisely one Node. Besides, an Interface belongs to precisely one PhysicalLink. Furthermore, an Interface is involved two times in a LogicalLink. These associations should be more apparent when taking a look at the chapters [LogicalLink](#) and [PhysicalLink](#).

#### Interface Analogy: Bus Stop

An Interface refers to a Bus Stop. A Bus Stop always belongs to precisely one City (Node). Moreover, a Bus Stop is assigned to just one Street (PhysicalLink). However, the Bus Stop occurs in exactly two Bus Connections (LogicalLinks). One connection in one direction and the other in the other direction.

### 5.2.5 Link

A Link is a connection between two Interfaces and, in the end, between two Nodes. At least one Link has to be present in a ServiceChainResult, but there can also be more.

A Link can be directed or undirected. This information mentions if the direction of the traffic is of interest or not. Because the direction declaration has an essential role for the ServiceChainCalculation and thus for the ServiceChainResult, it is essential to make this difference. In the end, a Link is a generalization of PhysicalLink and a LogicalLink, which means a Link is either logical or physical nature. More information about the similarities and differences can be found in the chapters [PhysicalLink](#) and [LogicalLink](#).

#### Link Analogy: Association

Because a Link is a placeholder for the more specific types PhysicalLink and LogicalLink it best refers to an Association between two Bus Stop (Interfaces) with not much meaning.

The meaning or the context is given by the specific implementation of a Street (PhysicalLink) or a Bus Connection (LogicalLink). For a better understanding, it is recommended to refer to chapter [PhysicalLink](#) and [LogicalLink](#)

### 5.2.6 PhysicalLink

The PhysicalLink can be seen as the physical connection between two Nodes. Simplified, this is the cable that is connected between two Interfaces of two different devices. It has physical characteristics of the connection assigned and does not know any direction.

The PhysicalLink has a massive influence on the visualization of the topology. It is mostly more clear to draw the physical links of a topology. The outline of each LogicalLink results shortly in an overwhelming unstructured topology overview.

Because this type of link is physical, it is compound by precisely two Interfaces, and as a consequence, only one specific Interface can be connected to a PhysicalLink.

### PhysicalLink Analogy: Street

A `PhysicalLink` refers to a `Street` that connects two `Bus Stops` (`Interfaces`). Simplified, we can assume that the `Street` enables traffic between the two `Bus Stops` but does not say anything about the traffic characteristics. That means it only says there is a road between two `Bus Stops`. But it does not describe important bus route characteristics like in which direction the traffic goes or how long a bus ride takes. This information is stored on the `Bus Connection` (`LogicalLink`).

A good thing to remember is that there is no `Bus Connection` (`LogicalLink`) if there is no `Street` (`PhysicalLink`).

## 5.2.7 LogicalLink

A `LogicalLink` is an abstract connection between two `Interfaces` and thus in conclusion between two `Nodes`. Such a type of `Link` has relevant information stored on it, like in which direction the traffic flows and which metrics exist on it. `LogicalLinks` is a crucial part of the `ServiceChainCalculation` and can therefore be found in the `ServiceChainResult`. Because the `LogicalLink` is directed, there are always be two pairs of `LogicalLinks` for one `Interface`: the first in one direction and the second in the opposite direction.

### LogicalLink Analogy: Bus Connection

A `LogicalLink` refers to a `Bus Connection` between two `Bus Stops` (`Interfaces`). It has characteristics assigned, like how long a bus has to drive from one to the other `Bus Stop`. Another example characteristic could be how long the current delay is to reach the other `Bus Stop`. Because these properties are directly related to the direction in which the traffic flows, it makes sense that a `Bus Connection` always has just two `Bus Stops` connected. It's important to mention that two `Bus Stops` are always in precisely two `Bus Connections`. In the first `Bus Connection`, one `Bus Stop` is the source, and the other is the destination `Bus Stop`. In the second one, these roles are changed: the source `Bus Stop` is now the destination `Bus Stop` and vice-versa.

## 5.2.8 ServiceInstance

A `ServiceInstance`, as the name already mentions, is a specific instance of a service. A `ServiceInstance` can be a specific firewall blade or IDS device through which packets should be steered.

Exactly one `Segment` belongs to a `ServiceInstance`. A `Segment` describes that packets with a specific `SegmentID` should be forwarded to an unique `ServiceInstance`. In our domain, a `ServiceInstance` is always attached to one specific `Node`. Furthermore, a `ServiceInstance` is exactly from one `ServiceType`. A `ServiceInstance` doesn't have to belong necessarily to a `ServiceChainCalculation`. The same applies to the `ServiceChainResult`.

### ServiceInstance Analogy: Attraction

A ServiceInstance refers to an Attraction which the guests like to stop by. An Attraction (ServiceInstance) can be of one specific Attraction Type (ServiceType), either it is an amusement park, or it is a shopping mall. It can not be both. Furthermore, an Attraction can be found in zero or more Journey Calculations (ServiceChainCalculations). The same applies to the Journey Calculation Result (ServiceChainResult). It's also essential that an Attraction has only one Coordination/Address (Segment). In addition, it follows that an Attraction belongs exactly one City (Node).

#### 5.2.9 ServiceType

Like the name already tells, a ServiceType gives specific information about which type a specific ServiceInstance is from. For now, there are only two ServiceTypes present in the domain: **FW** and **IDS**

In our domain, there are zero or more ServiceInstance of a specific ServiceType. The same applies to ServiceChainCalculations: A unique ServiceType can be included in one ServiceChainCalculation, but it has not to.

### ServiceInstance Analogy: Attraction Type

The ServiceType refers to an Attraction Type. It gives information about if an Attraction (ServiceInstance) is, for example, either from the type amusement park or of the type supermarket. In the domain, there do not have to be Attractions of a specific Attraction Type. The same is for a Journey Calculation (ServiceChainCalculation): There is no need for a journey type to be present in a Journey Calculation.

#### 5.2.10 ServiceChainCalculation

ServiceChainCalculation is the central concept of this domain. It consists of information to calculate the best path and also the segment list for a service chain.

In a ServiceChainCalculation, at least one ServiceInstance is involved. If a ServiceType is selected in a ServiceChainCalculation, all ServiceInstances of this ServiceType are considered for the ServiceChainCalculation implicitly. There is no need to select a ServiceType, but the current maximum of ServiceTypes available are two: **FW** and **IDS**.

Also, precisely two Nodes have to be selected for the ServiceChainCalculation. These Nodes have special terms concerning to Segment Routing: They are called Ingress and Egress. The Ingress is the Node in which the source Network is selected. The Egress the Node which the destination Network is selected. That a ServiceChainCalculation makes sense, both Nodes must be **PE** and have Networks assigned.

Like already mentioned also two Networks have to be selected. These Networks must be assigned to the Ingress and Egress.

A `ServiceChainCalculation` delivers at least one `ServiceChainResult` at each execution. Therefore there is at least one `ServiceChainResult` to a specific `ServiceChainCalculation` present.

#### **ServiceChainCalculation Analogy: Journey Calculation**

A `ServiceChainCalculation` refers to a Journey Calculation the transport business likes to perform. Because the Attractions (`ServiceInstances`) belong to the transport company's unique service, there has at least one Attraction involved in the Journey Calculation. Another possibility is to select only the Attraction Type (`ServiceType`). This selection will implicitly trigger that all Attractions of the Attraction Type are considered for the Journey Calculation. A Journey Calculation always belongs to two Cities (`Nodes`) and two Districts (`Networks`) located in the selected Cities. Because a Journey Calculation can be executed several times, it can be connected to one or more possible Journey Calculation Results (`ServiceChainResults`).

### 5.2.11 ServiceChainResult

A `ServiceChainResult` is the result of a finished `ServiceChainCalculation`. It includes all the information about the most appropriate `BestPath` and the Segment List or short `SIDList`.

The best path consists of a list of links which clearly defines where the traffic has to steer through, to achieve what was selected in the `ServiceChainCalculation`. There has to be at least one `Link` to be included in the `ServiceChainResult`.

Furthermore, the `ServiceChainResult` contains the information about which `ServiceInstance` has to be followed for the best path.

To manage a correct list of Segments, there has to be included at least one Segment in the `ServiceChainResult`. Mostly there would be more than just one Segment present in the `SIDList`. Following, a `ServiceChainResult` belongs to exactly one `ServiceChainCalculation`.

#### **ServiceChainResult Analogy: Journey Calculation Result**

A `ServiceChainResult` refers to a Journey Calculation Result. The result contains the information on which route the bus should take to have the most pleasant journey and satisfy the customers.

A Journey Calculation Result consists of the best path belonging to one or several Bus Connections (`LogicalLinks`).

Additionally, the Journey Calculation Result consists of different Attractions (`ServiceInstances`), where the bus has to stop. There has to be at least one Attraction involved.

So that the bus driver knows where to drive next, the journey result also includes a list of Coordinations/Addresses (`Segments`), which the bus driver can type in its navigation device. The navigation device then leads the bus driver according to the best path to the next Bus Stop (`Interface`) belonging to a specific City (`Node`). The best path, of course, is similar to the list of Bus Connections. A Journey Calculation Result belongs to exactly one Journey Calculation (`ServiceChainCalculation`).

# Architecture and Design Specifications

---

## 6.1 General

The work in this thesis aimed to achieve the Minimum Viable Product. However, any architectural decisions were made in such a way that the application could be easily extended in the future. The scope of this thesis was clearly defined from the beginning. The thesis should address the complete backend application that takes care of all necessary steps to calculate a service chain. By contrast, the frontend was no part of the semester thesis and was developed by the Institute of Networked Solutions. The complete application there consists of two independent components: the front- and backend. This construct laid the foundation for additional extensions in the future and will be continued.

## 6.2 System Overview

The whole application should be able to be deployed independently of the infrastructure. Therefore the complete application is delivered in the form of Docker containers.

The goal was to abstract the actual backend from the network's actual data source (Arango database). This abstraction could be solved by introducing an own polling application, which fetches the data from the Arango DB and stores it in a Redis cache, detects changes, and send these only to the backend application. Also, the polling container allowed to take some load off the backend container. The periodically check if any new or changed data are available in the Arango graph database were moved entirely to the self-written polling logic. By this abstraction and the possibility of informing the backend about data changes, the backend can focus completely on its tasks, including data processing (for the front end) and the calculation of the different paths.

In order to describe the existing software architecture and give the readers a visual overview, several diagrams according to the C4 model[6] were created.

The following C4 system landscape diagram shows the different actors from the use case diagram, which can be found in figure 4.1. The landscape diagram also visualizes the two different software systems, the developed application *SerChio*, and the industry partner's external system.



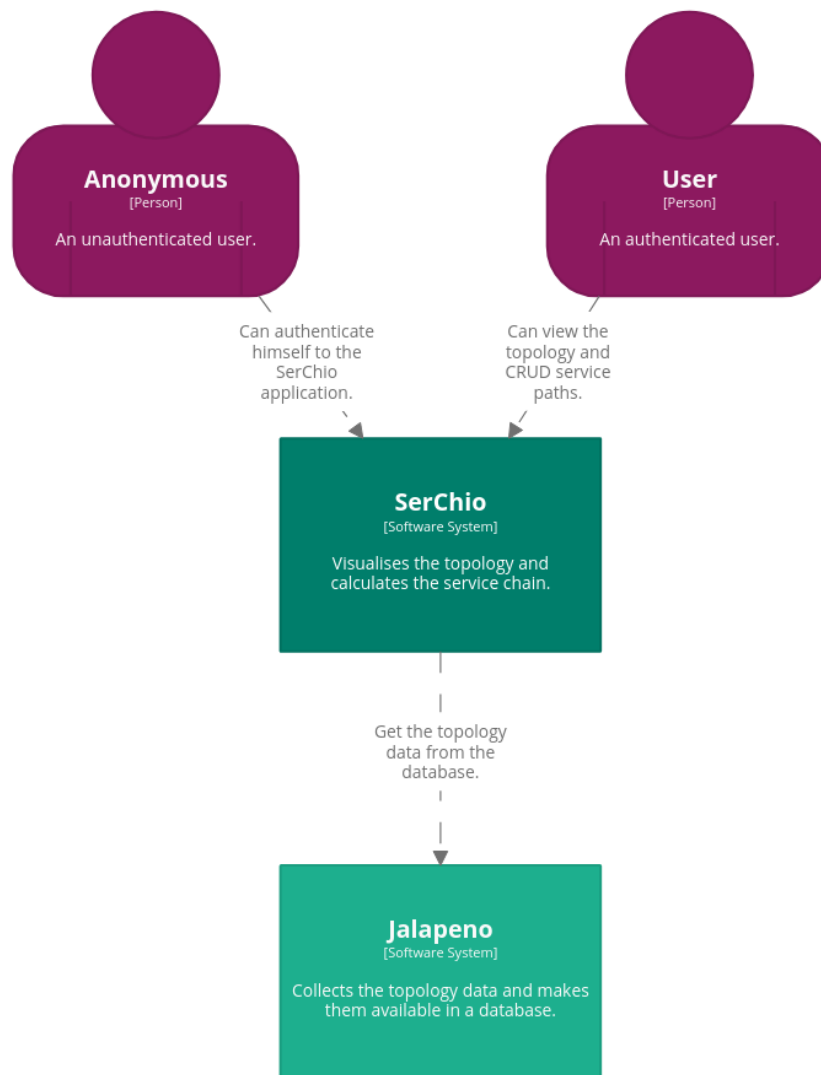


Figure 6.1: C4 System Landscape

The following C4 container diagram shows how the different containers interact with each other and which protocols are used. Additionally, the diagram shows how the user communicates with the software system. The external software system Jalapeno is not visualized in detail because the SerChio application is only getting the data from an Arango database inside the Jalapeño application.

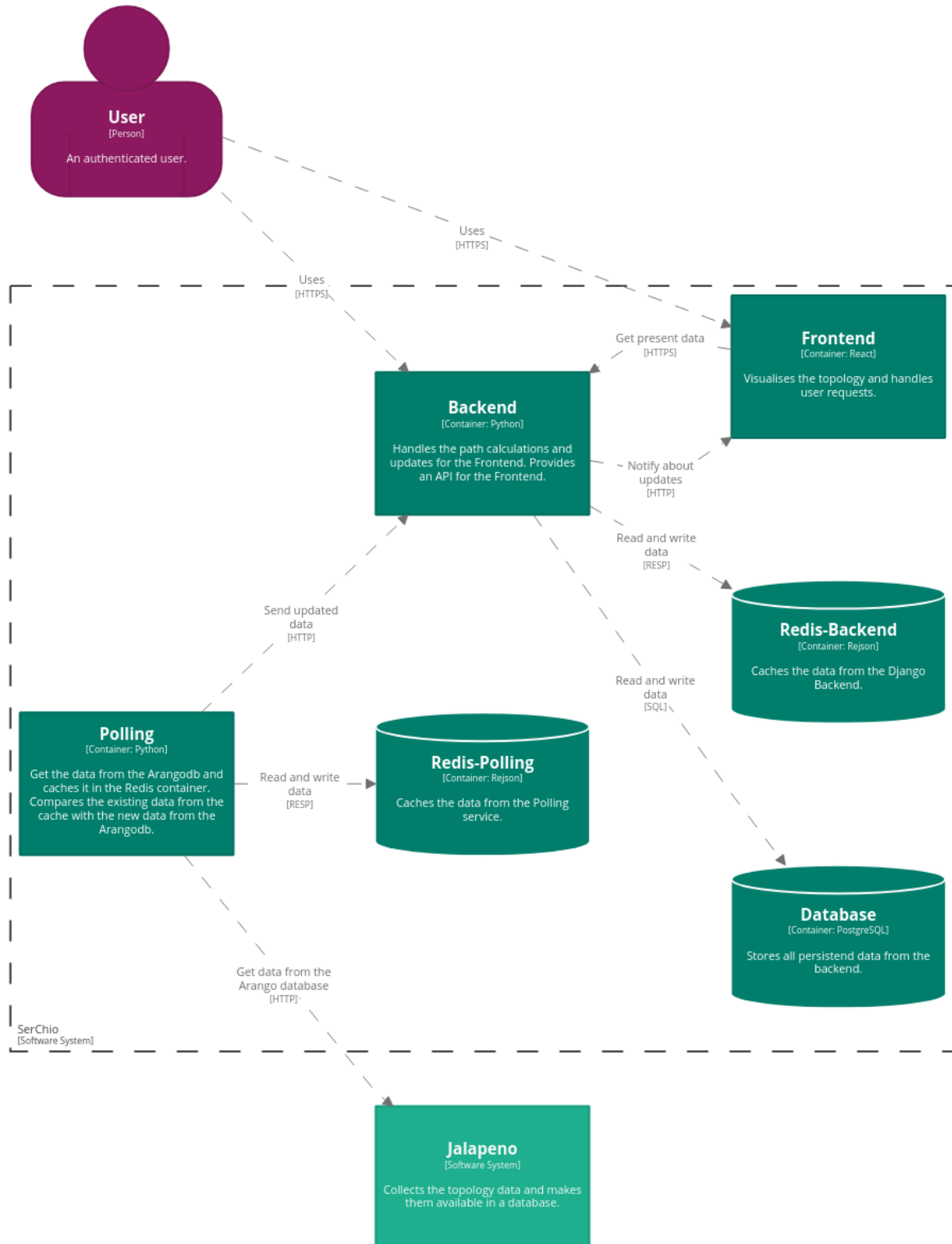


Figure 6.2: C4 Container Diagram

### 6.2.1 Jalapeño

Jalapeño is the external software system SerChio communicates with. The system collects data from productive networks to process them in a graph database. SerChio uses this external software system to get the necessary topology data for the graphs and calculations.

Jalapeño was not part of this work but is also a dependency covered in the risk analysis. The application was provided by the industry partner and deployed on an internal server at the Institute for Networked Solutions.

### 6.2.2 Frontend

A user who uses SerChio communicates directly with the frontend container. The frontend displays the topology graphically. The user can enter the desired parameters for the calculation via various input fields. After a successful calculation by the backend, the frontend displays the calculated path graphically in the topology.

Together with the supervisor, it was decided at the beginning of the work that the frontend should not be part of the thesis. With the requirement that the frontend should combine several segment routing applications in the future, the involved parties decided that the frontend would be entirely developed directly by the Institute for Networked Solutions. Due to the external development, an additional communication effort was created, which could be solved by a weekly meeting with the frontend developer.

### 6.2.3 Backend

The backend is the brain of the whole SerChio application, where all requests and calculations converge. The backend continuously receives new data from the external software system and processes them. The frontend receives all necessary data to display the topology directly from the API of the backend.

The main task of the backend is to perform the service chain calculations and process the data for the frontend. For this purpose the backend stores all data to be persistent in a database. To perform calculations as fast as possible the backend uses a Redis Cache in addition to the persistent database, in which it caches the graph, which is needed for the calculations.

The biggest challenge for the backend is that it must always maintain an up-to-date graph. As soon as new data from the polling service arrives at the backend, the backend has to update the graph and store it back in the Redis Cache. This allows a user to view and perform calculations on the most current topology without having to trigger manually an update each time.

Because all calculations and requests come together in the backend, it is the most heavily loaded component, apart from the frontend, which has to render the topology but which is not in the scope of this work. Therefore, a lot of attention was paid to performance.

### 6.2.4 Polling

To get another level of abstraction between the backend and the external software system Jalapeño, it was decided to introduce a polling service. The polling service is responsible for ensuring that the backend always has the most current data available.

The polling service continuously retrieves data from the graph database of the external software system. After the data is fetched, the service compares the checksums of the new data with the existing checksums in the Redis cache. If the service notices that data has been added, changed, or deleted, it informs the backend and sends an update.

By outsourcing the data processing, the load could be taken off the backend container. Due to the additional abstraction between the backend and the external software system, when the

data resource is changed, not the complete backend has to be adapted but only the polling service.

### 6.2.5 Database

Django works with models which each need their own database table. Therefore a relational database is needed for the backend.

To persist this data of the backend it was decided to use a postgres database. The backend can then communicate directly with the database instance using [Structured Query Language \(SQL\)](#) and store the data persistent.

In order to simplify testing and development, it was made sure that the backend can seamlessly support different database systems. For example, a SQLite database can be integrated for testing without the need for a separate container.

### 6.2.6 Redis

To use the full functionality of Django the backend needs a relational database. However, operations on such a database are usually relatively expensive and are therefore not an option for all data.

All data processed by the polling service are stored on a Redis cache. This data is not required persistently and can be easily reloaded from the external software system. The polling service stores the data in the cache to compare the different checksums.

The backend needs a place where to store the graph besides the relational database. A Redis cache was used there for performance reasons.

## 6.3 12-Factor Methodology

To ensure that the application is developed according to the Cloud Native Standard and can therefore be deployed in the Cloud as easily as possible, the 12-Factor [4] methodology should be used.

For this reason, a complete analysis of the project according to the requirements of the 12-Factor[4] methodology was carried out and evaluated.

### 6.3.1 Codebase

**There should be only one codebase, but multiple deployments can exist.**

GitLab was chosen as the version control system. The complete [Continuous Integration \(CI\)](#) and [Continuous Delivery \(CD\)](#) could be implemented by using GitLab. There are different repositories for the frontend and backend.

The [CI/CD](#) always builds a Production Container for each repository. The code is worked with directly during development.

### Conclusion

Although the same code base is used and there is also a distinction between production

and development deploys. However, to fulfil this factor, one would have to define different CI/CD stages, such as build, test, publish. On the whole, we believe that the first factor is not completely fulfilled. An additional Development Build Stage would have to be set up in the CI for this factor to be completely fulfilled.

### 6.3.2 Dependencies

**All dependencies should be explicitly declared and isolated.**

The application consists of two different components, the frontend, and the backend. Since the frontend is not within the scope of this work, only the backend is described here.

The backend component was implemented with Python and Django. Dependencies were thereby declared and isolated in this repository.

**Backend Component** The backend component developed with Python uses pip as a dependency manager. All dependencies are stored in a text file (requirements.txt) and can be easily installed with `pip install -r requirements.txt`. A virtual environment is used to isolate the dependencies, which is typical for Python. You can isolate all dependencies from the rest of the system with the virtual environment.

**Polling Component** As with the backend component, the polling component also works with Python and uses pip as dependency manager.

#### Conclusion

The project implements Dependency Isolation and Declaration in such a way that the second factor of the Twelve-Factor Apps concept is fulfilled. This means that no adjustments need to be made.

### 6.3.3 Config

**Configurations should be stored in the environment.**

The project uses environment variables in development and the productive version.

**Backend Component** The backend component works with environment variables, so in the finished containerized version of the backend, we can easily give the container the environment variables at startup.

**Polling Component** As with the backend component, the polling component works also with environment variables, therefore the environment variables can be injected at startup.

#### Conclusion

The third factor is fulfilled by using environment variables during development and in the productive system.

### 6.3.4 Backing services

**Backing services should be treated as attached sources.**

The project uses a PostgreSQL database and two Redis caches to store various data. There is no distinction between local and third-party services for Postgres and Redis cache. The resource is controlled in the backend via a config file which attracts environment variables.

The environment variables are given when the backend and polling container are started. The Postgres database can be easily replaced by using environment variables. Every time the backend component is started with a new database, the database tables are recreated.

### **Conclusion**

The Postgres database and the Redis Caches are treated as attached resources. Since environment variables are also used for the configuration of the two services, the fourth factor is filled.

### **6.3.5 Build, release, run**

**Use separate stages for build and run.**

The backend has 2 important stages in total. The first is the testing stage where automated tests are performed. The second is the build stage where the productive Docker Image is generated.

### **Conclusion**

The fifth factor is not yet met. To fulfill this factor, one would not only have to do CI but also CD. But the CD will only come into play once the application is used productively in the cloud.

### **6.3.6 Processes**

**The application should be stateless.**

According to the Twelve-Factor Apps concept, the processes of an app must be stateless. Stateless means that the application may not store and share any state. Data may be stored in databases. Locally stored data (on the hard disk or in memory) are only allowed for a short time.

### **Conclusion**

The application has been developed so that the processes are stateless. The use of the PostgreSQL database is excluded by the use of the attached storage. The Redis cache only stores data for a short time. Therefore a scaling is easily possible. Therefore, no adjustments, need to be made for the 6th factor.

### **6.3.7 Port binding**

**Make services only available via Port Binding.**

With the factor port binding, the application should not be exported through a web server such as IIS, Nginx, etc. The application should have an integrated port binding. The backend was developed with Python and Django. Django already has a web server called **gunicorn**. The backend is made available via this web server.

### **Conclusion**

By using the web server gateway interface **gunicorn** and not an external web server, the 7th factor is also fulfilled.

### 6.3.8 Concurrency

**The application should be scalable via the process model**

The project was built in the backend with Django and Rest, so the application offers a complete REST-API, which can be used to trigger all possible functions. The web process is independent, stateless, and has no persistence. Therefore the web service can easily be scaled. With a load balancer, the requests can be distributed to the different nodes. Because of the intention to run the application on a Kubernetes cluster, the Nginx load balancer already available on the cluster can be used.

#### Conclusion

The project does not require any adjustments regarding the eighth factor, it already meets it.

### 6.3.9 Disposability

**The robustness should be increased with a clean shutdown and quick start.**

The application should start within a few seconds, which would allow quick and easy scaling. The application should have the possibility to be shut down **graceful**. In doing so, it should still be able to process existing requests.

#### Conclusion

The application starts very quickly and is then available immediately. When the application is shut down, however, a new function would have to be developed to check that all requests are processed completely. At the moment it could be that a request is simply cancelled.

### 6.3.10 Dev/prod parity

**Development, Staging and Production should be as similar as possible.**

The tenth factor aims to keep the differences between the Development Environments and the Production Environment as small as possible. It refers to the time, personnel, environment, and tools used.

In table 6.1 a comparison to a conventional application that is not twelve-factor compliant can be found.

	Traditional App	Twelve-factor App	SerChio
Time between deploys	Weeks	Hours	Hours
Code authors vs code deployers	Different people	Same people	Same people
Dev vs production environment	Divergent	As similar as possible	As similar as possible

Table 6.1: Dev/prod parity comparison

## Conclusion

Since the backend components are already developed directly in the container, the difference to the production environment is kept to a minimum. The 10 factor is therefore fulfilled.

### 6.3.11 Logs

**Logs should be treated as event streams.**

Logs should not be stored in a log file, they should be written to the standard output. By this procedure, the logs could be forwarded to an external system like an [Elasticsearch, Logstash and Kibana \(ELK\)](#) stack.

All logs in this project are written directly with a logger in stdout.

## Conclusion

No changes need to be made to the application. The 11 factor is therefore fulfilled.

### 6.3.12 Admin processes

**Administrative tasks should be carried out as one-off processes.**

The last factor is about the one-time tasks that have to be executed on a productive application. These tasks can be for example database migrations, console commands or one-time scripts. The tasks should be executed on an identical system if possible.

## Conclusion

The entire project is containerized, so such tasks are not even needed. In case of changes or adjustments, the affected container is simply redeployed. This also has the advantage that the complete application can be tested on a test system before it is used on the productive system.

## 6.4 Technology Decisions

In this project different technologies and libraries were used together. Table 6.2 should give a short overview and the following sections an explanation how the technology decisions were made.



Component	Technology
Backend	Django based on: <ul style="list-style-type: none"> <li>• Python</li> </ul> With the following libraries/extensions: <ul style="list-style-type: none"> <li>• Django REST Framework</li> <li>• drf-extensions</li> <li>• Django Cache Framework</li> <li>• django-redis</li> <li>• Django Channels</li> <li>• channels-redis</li> <li>• drf-yasg</li> </ul>
Polling	Python with the following libraries: <ul style="list-style-type: none"> <li>• pyArango (CRUD accesses for ArangoDB)</li> <li>• asyncio (async task management)</li> <li>• schedule (job scheduling)</li> <li>• websockets (websocket management)</li> <li>• rejson (CRUD accesses for Redis)</li> <li>• reachability (socket management)</li> </ul>
Database	<ul style="list-style-type: none"> <li>• PostgreSQL for development and productive usage</li> <li>• SQLite for unittests</li> </ul>
Cache	<ul style="list-style-type: none"> <li>• Redis cache with JSON module</li> </ul>

Table 6.2: Technologies

### 6.4.1 Programming Language

It was decided to choose Python as a programming language. Python became more and more popular in the last years and is especially in the network automation area not to be imagined without.

As both project members mainly use Python for programming it was the first choice when it came to choose the programming language.

### 6.4.2 Frameworks

**Django** (<https://www.djangoproject.com/>)

Because it was decided to use Python as the core language, the framework had also to be working with Python. Because Python is such a powerful programming language, there also exist many web frameworks: Pyramid, Flask, Django, Tornado, to just name a few from a long list. For this project, Django was chosen because it has different aspects which fit perfectly for the application and also made the way easier to develop the software:

- Django is the most common web framework written in Python. It delights over a vast community. Therefore, if there exist bugs or problems, they will be found and solved quickly.
- Many well-known applications, like Youtube, Instagram, and Spotify, are written with Django. These companies show what is possible with this web framework.

- Django has many built-in features and does help the developer to creates things fast. Besides, there exist many extensions, which makes it even more powerful.
- Django has an easy to understand and massive, detailed documentation and API, which makes it easy to develop new things.
- The Institute of Network Solutions and also the editors of this thesis knows the web framework. This was most basically the biggest reason to choose Django.

As mentioned above, Django gets even more potent with its extension. The most important libraries and extensions are described below, which significantly impacts developing the **Se-Chio** app.

**Django REST Framework** (<https://www.django-rest-framework.org/>) **Django REST Framework (DRF)**, is a Django extension that quickly allows the development of web APIs. Since it was decided to implement a REST API in the application, Django Rest Framework was considered immediately. The reasons are obvious; besides the vast customer base of DRF like Mozilla, Red Hat, or Heroku, DRF allows the perfect mix of easy to use standard features and extensibility. The result is a feature-rich REST API with enhanced features. More information about the developed REST API you can find in the chapter 6.5.

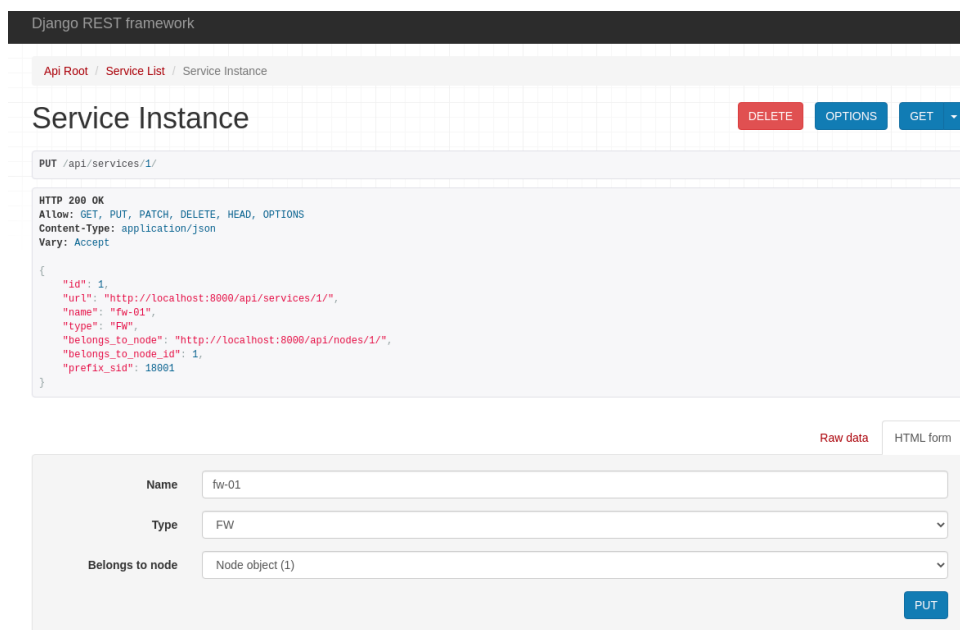


Figure 6.3: DRF built-in Web API

**drf-extensions** (<https://github.com/chibisov/drf-extensions>) **drf-extensions** is a module, which allows extending the functionality of the Django Rest Framework. Because the application needs a nested API, which can not be implemented with the DRF's standard built-in functionalities, it was decided to cover this feature with drf-extensions. Other modules also implement nesting. The choice fell on drf-extensions because it includes several features that can be considered for the future extensions of the application. More information about the API and the nesting functionalities can be found under the chapter 6.5.3.

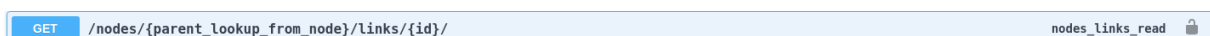


Figure 6.4: Example Endpoint with Nesting

**Django Cache Framework** (<https://docs.djangoproject.com/en/3.1/topics/cache/>) The Django Cache Framework is a built-in extension of Django. It allows caching data easily and brings fast access to this information. During the development process, it was decided to use the Django Cache framework because it can be integrated seamlessly into a Django environment. Also, it delivers the necessary functionality and delivers the necessary functionality to save the graph information, which is essential for the Service Chain Path Calculation. The graph is always maintained in the memory of the backend application and saved to a Redis cache. This implementation has a significant advantage: if the backend application is restarted, the graph has not to be recreated but can only be read out of the cache. This concept allows making calculations available with lightning speed after a restart or breakdown of the backend application.

**django-redis** (<https://github.com/jazzband/django-redis>) Django-Redis is a module, which allows using a Redis cache within a Django application. Actually, Django Cache Framework is using Memcache as their default cache server. Because the application already uses a Redis Cache in the Polling application, the decision was made to use Redis in the backend application as well. Besides the standard functions of the Django Cache Framework, this module extends the features to make caching even more powerful. Nevertheless, this module is developed by [Jazzband](#), which is highly recognized for their phenomenal work in the Python open-source community.

**Django Channels** (<https://channels.readthedocs.io/en/stable/>) Django Channels is an admitted extension of Django, which helps extend Django's features for supporting WebSockets and other asynchronous protocols. It's possible to support using Websockets besides the standard technology of Django with this code extension. Also, the extension allows running some background processes besides the Django process. Because the updates should be sent over a WebSocket to the frontend applications, it was decided to use Django Channels. Furthermore, Django Channels brings the feature to communicate between different application instances and implement low-latency queues. This functionality helps, for example, to notify each WebSocket that updates are present. This implementation is also a look ahead because, with this solution, it would also be able to communicate between applications if several backend containers would exist. An impressive overview can be found under [django-channels-overview](#)

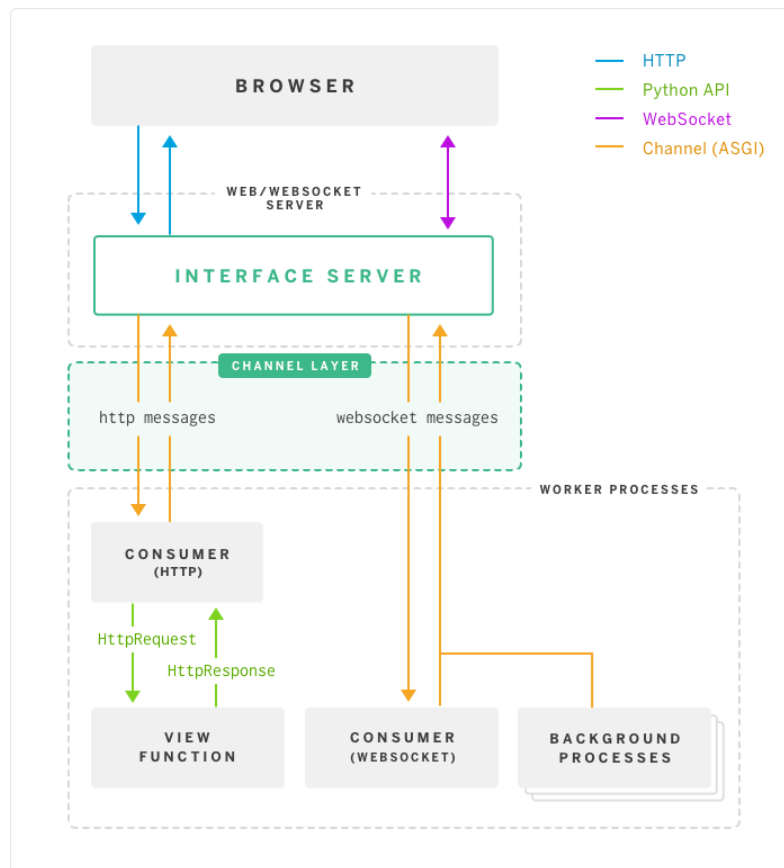


Figure 6.5: Heroku Overview Django Channels

Source: <https://blog.heroku.com>

**channels-redis** ([https://github.com/django/channels\\_redis/](https://github.com/django/channels_redis/)) Django channels recommend using Redis as its backing container, where the different messages, which should be exchanged, are saved. The basis for this concept provides the library channel-redis. With this approach, the messages could be saved on the Redis container, which also exists to save the network graph. Because of reusability reasons, since a Redis Container is also present, and the suggestions of Django Channels, the decision to use this module was made.

**drf-yasg** (<https://github.com/axnsan12/drf-yasg/>) drf-yasg, Yet another **Swagger** generator, is a Django Rest Framework extension that expands the functionality by automatically creating a **Swagger** and **ReDoc** documentation of the application. The decision to use this extension was made because it supports the most common REST API documentation tools, Swagger, and ReDoc. Another reason for introducing the tool was because a standardized API Documentation (OpenAPI 2.0) without code changes was out of the box available. More information can be found under [6.5.1](#).

### 6.4.3 Graph Library

Calculating a service chain is tightly related to maintaining a graph with vertices and edges and do shortest path calculations. In the end, a service chain calculation is tied with a graph calculation. A graph in the **SerChio** context describes the network with vertices and edges. Therefore a relation has to be made between Segment Routing nodes and vertices plus network links and edges.

There exist different approaches to maintain a graph and do calculations on it. The develop-

ers of this application had to decide at the beginning of this project which option is the most suitable to calculate a service chain in the future. The developers took three slightly different implementations and compared them with each other to answer this question. Besides functionality, speed was also a factor that was important for the implementation. Below each approach is shortly described with its advantages and disadvantages.

**NetworkX** (<https://networkx.org/>) NetworkX is probably the most known graph library within the Python domain. It allows maintaining graphs and directed graphs (digraphs) and multigraphs (graphs with multiple edges between vertices). Advantages are a broad community, good documentation, and that it comes with several well-known standard graph algorithms. Also optimistic is that almost each Python objects can be used as nodes. The latter also leads to disadvantages. Because the core library is written in Python, it's slow. A comparison to the other approaches can be found under Figure 6.6 Also, the shortest path calculation, which can be done with **Dijkstra**, returns the nodes and not the edges, which is also needed for the Service Chaining Path Calculation. All in all, the editors could relatively fast decide that NetworkX can not be used for future implementation.

**Arango Database** (<https://www.arangodb.com/>) Because the network information is stored in an Arango database, Julian Klaiber and Severin Dellsperger thought about performing the calculations on the existing graph database. The single performance measurements, exhibited in Figure 6.6, showed that simple shortest-path calculation is fast. On closer inspections, it was detected that it is impossible to execute several shortest path calculations at once and get the result that could be processed further. Simplified, this means that each shortest path calculation had to be delivered to the graph database via an HTTP request. Following 1000 shortest path calculations would result in 1000 requests and responses. This approach was undue and did not scale well. Therefore, the calculation directly on the Arango database was avoided.

**graph-tool** (<https://graph-tool.skewed.de/>) graph-tool is an impressive Python library to maintain graphs because the core data structures and algorithms are implemented in C++. Simplified graph-tool is a Python wrapper around a C++ application. It's also noticeable in the performance comparison. The approach with this module is a notch above the other options. More detailed information about the performance can be found in Figure 6.6.

Besides that, the library is high-speed, is also has a massive amount of features, which made this library an optimal decision for the future. For example, graph-tool can calculate all shortest paths from a given source and destination node and return all equal-cost multi-pathing edges. Another significant advantage is the documentation is very well maintained and serves all information that is needed. The only disadvantage is the installation because the library's core is written in C++; the installation is not as easy as with other libraries, but even this small difficulty is well documented and can therefore be neglected.

Because graph-tool delivers the needed features and more, the editors have decided to use this approach in their implementation. The feature-rich and enormous enhanced library make the most sense to use this module in the Service Chaining Path Calculation application.

# nodes / vertices	arangodb	networkx	graph-tool
100 / 300	~ 0.0037s	~ 0.34s	~ 0.0015s
1000 / 3000	~ 0.023s	~ 0.025s	~ 0.018s
3000 / 9000	~ 0.05s	~ 0.093s	~ 0.036s
10000 / 30000	~ 0.14s	~ 0.28s	~ 0.093s

Table 6.3: Calculation Performance Comparison

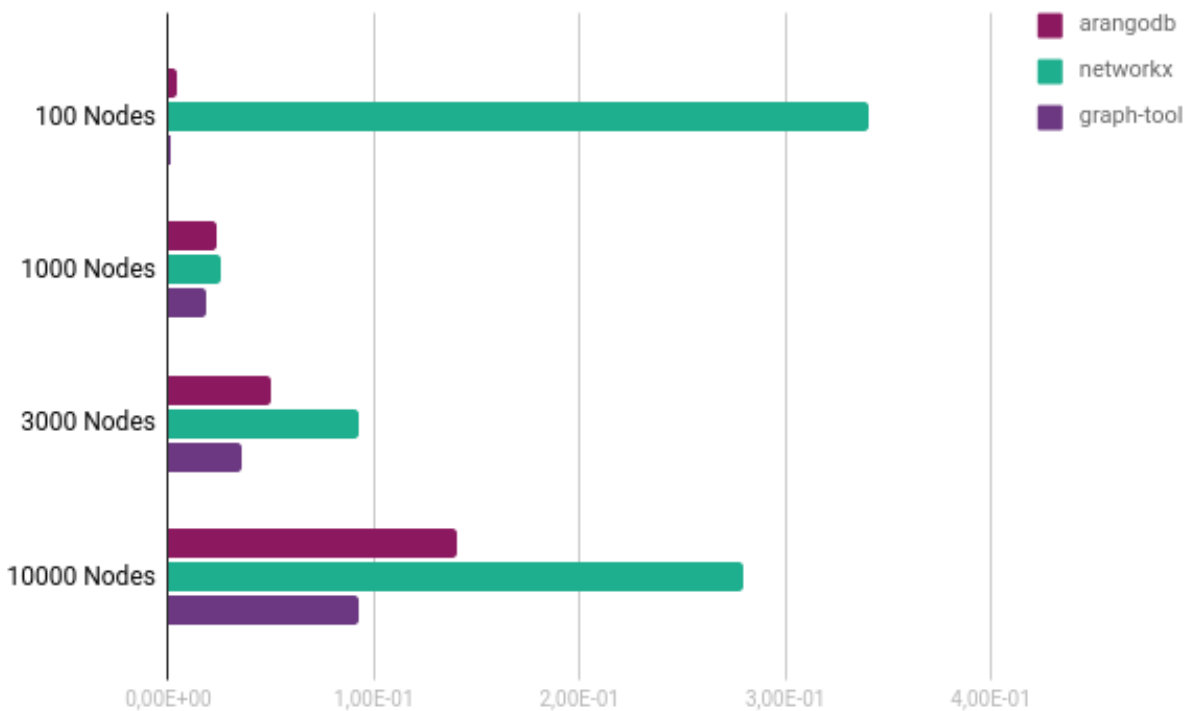


Figure 6.6: Calculation Performance Diagram

#### 6.4.4 Cache

For the cache system, the most popular one was chosen, which also has the best support for the chosen programming language.

Redis is the most popular cache system on the market. As open-source software, it has a huge community and very good support of the programming language Python.

#### 6.4.5 Database

The decision for a relational database was made depending on the popularity and compatibility with the programming language and frameworks used.

PostgreSQL was chosen as the relational database. PostgreSQL is one of the most popular relational databases worldwide and is supported by many programming languages and frameworks, as well as Python and Django. It should be emphasized that PostgreSQL harmonizes perfectly with Django's [Object-relational mapping \(ORM\) Mapper](#), which also speaks

for the use of PostgreSQL.

## 6.5 REST API

The application, which was developed during the time of the project thesis, includes a unique API. The API has some non-standard features, which are outlined in this chapter.

A REST API has the advantage that it can be used with each client. This means that the frontend can be exchanged easily, or several frontends for this application could exist.

The API was designed with the help of some API guidelines. Especially the guideline of the Zalando API (<https://opensource.zalando.com/restful-api-guidelines/>) was heavily used. It was tried to pick out the best things of the guideline and implement it into the **SerChio** API.

### 6.5.1 Documentation

The complete REST API was documented with **Swagger** and **ReDoc**. With this documentation even outsiders without knowledge of the code can easily understand and test the API.

The Redoc documentation can be found under the endpoint `/api/redoc`, a sample output can be found in Figure 6.7. The Swagger documentation can be found under the endpoint `/api/swagger` a extract of this type of documentation can be found Figure 6.8.

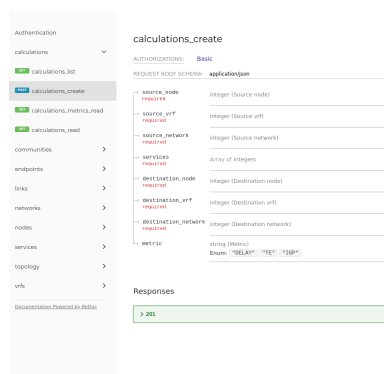


Figure 6.7: API Redoc

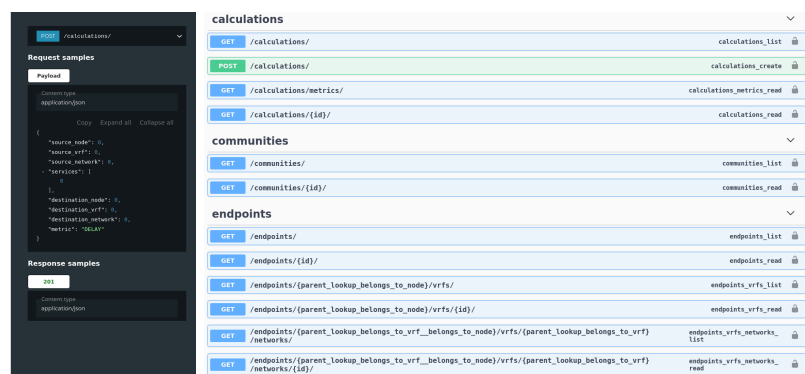


Figure 6.8: API Swagger

### 6.5.2 Pagination

Many endpoints return a list of elements. Therefore, the list of objects could be huge and could have a significant impact on network traffic. To address this problem, paging was introduced. Pagination describes a process or feature in which the list of objects is split into several pages. This concept has the advantage that not the whole object list has to be loaded at once. Instead, parts, the so-called pages, are fetched step-wise. Paging makes sense if one request is too big to be loaded at once.

There exist different strategies. In this chapter, it will explain the implemented strategy in the **SerChio** API and how it works. The other strategies go beyond the scope.

The introduced paging strategy is called **LimitOffsetPaging**. As the name mentions, this strategy implements a limit and an offset option. These options are mostly submitted via query parameters. The limit parameter limits the number of the maximum items which should be

included in the response. The offset parameter indicated the starting position of the first element (offset = 0 → first element overall). It is noteworthy that the elements have to be ordered to prevent, that no object is returned twice. In the `SerChio` API, the sorting happens over the unique object id. For more information about the id's, take a look at the persistency diagram in chapter 6.9. The API's response also introduces a next and previous element, which shows the next or previous  $n$  entries.

The pagination is available on all endpoints on the `SerChio` API. Furthermore, the limit parameter can be chosen freely by the asking party. It should be clear that if the parameter is too large, the pagination strategy loses its advantage.

The listing 6.1 shows a response example with included pagination. The request was made to the URL `http://localhost:8000/api/nodes/?limit=2&offset=2`.

```
GET /api/nodes/?limit=2&offset=2
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
{
  "count": 10,
  "next": "http://localhost:8000/api/nodes/?limit=2&offset=4",
  "previous": "http://localhost:8000/api/nodes/?limit=2",
  "results": [
    {
      "id": 3,
      "url": "http://localhost:8000/api/nodes/3/",
      "name": "XR-3",
      "router_id": "0.0.0.3",
      "asn": 64075,
      "node_sid": 16003
    },
    {
      "id": 4,
      "url": "http://localhost:8000/api/nodes/4/",
      "name": "XR-4",
      "router_id": "0.0.0.4",
      "asn": 64075,
      "node_sid": 16004
    }
  ]
}
```

Listing 6.1: Example Response with Pagination

### 6.5.3 Nesting

Nesting describes a method to introduce hierarchical relations between objects. Instead of delivering standard endpoints, which return a list of all elements, the different endpoints become nested and just return the elements which fulfill the constraints given by its hierarchical relations.

With a little example, it should be more precise. In the first example, which can be observed in the listing 6.2, the response includes all available links.

```
GET /api/links/?limit=100&offset=0
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
```



```

Content-Type: application/json
Vary: Accept
{
  "count": 36,
  "next": null,
  "previous": null,
  "results": [
    \\ all links available
  ]
}

```

Listing 6.2: Example Request without Nesting

The application programming interface gets more enhanced if nesting is introduced. Moreover, the creation of nested endpoints results in other advantages: Nested API routes create readability and automatically implement filtering. Like in listing 6.3 the URL automatically expresses the relationship of the node and the links.

```

GET /api/nodes/1/links/?limit=10&offset=0
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
  "count": 5,
  "next": null,
  "previous": null,
  "results": [
    // Just the links which belong to node with id 1
  ]
}

```

Listing 6.3: Example Request with Nesting

With nesting, it is also possible to introduce relationships over a few levels. The listening 6.4 shows an example that delivers the networks from a specific VRF which belongs to a unique endpoint/node.

```

GET /api/endpoints/10/vrfs/9/networks/?limit=10&offset=0
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
  "count": 1,
  "next": null,
  "previous": null,
  "results": [
    {
      "id": 9,
      "url": "http://localhost:8000/api/endpoints/10/vrfs/9/networks/9/",
      "network": "170.0.0.10/32"
    }
  ]
}

```

Listing 6.4: Example Request with enhanced Nesting

Nesting delivers a easy to understand and clear URL. A non-nested API often works with complicated filtering, which does not show the relation between the objects and can be overwhelmed. A non-nested example would like following:

```
http://localhost:8000/api/networks/?limit=10&offset=0&endpoint=10&vrf=9
```

In the SerChio API Filtering is needed for other purposes, for more information take a look at the chapter 6.5.5

## 6.5.4 Hyperlinking

### From the book The Missing Link

*"A change in recent years has been the realization that the use of links brings a significant improvement to the usability and learnability of all APIs, not just those that are designed to be consumed"[1]*

The SerChio API was developed with the book *Web API Design: The Missing Link from apigee, Google Cloud*. This book called attention to the use of hyperlinking into APIs. Hyperlinking describes a method to include Hyperlinks and not just keys for referencing other objects into API responses. This concept has a significant advantage. With the inclusion of hyperlinks in each response, the context is straightforward, and it also delivers a more comfortable way to access the resource for the client. Instead of just returning a key, which forces the frontend to add the base URL to the key, the URL is directly returned in the response.

In listening 6.5 we could see its benefit. In the response, the element `belongs_to_node` directly returns the element to which the service instance belongs. If the frontend has to load information, this URL can be requested, and the necessary information will be delivered. If there is only the key returned, the base URL has to be added to the key. This process is a unnecessary and can be avoided by using hyperlinking.

```
GET /api/nodes/3/services/?limit=10&offset=0
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
  "count": 1,
  "next": null,
  "previous": null,
  "results": [
    {
      "id": 2,
      "url": "http://localhost:8000/api/nodes/3/services/2/",
      "name": "ids",
      "type": "IDS",
      "belongs_to_node": "http://localhost:8000/api/nodes/3/"
    }
  ]
}
```

Listing 6.5: Example Request with Hyperlinking

The concept of hyperlinking has emerged and is now implemented in many high-quality APIs like:

**Google Drive API** <https://developers.google.com/drive/api/v3/reference>

**GitHub Repository API** <https://developer.github.com/v3/repos/>

### 6.5.5 Filtering

Filtering describes the method to filter results by a specific criterion. In the SerChio API, filtering is implemented by query parameters. The filtering is essential for the Service Chaining Path Calculation. With this concept, it is, for example, achieved that only calculations between nodes and networks can be made, which have connectivity. To check if connectivity is present, intelligent filtering by `export_tag` is used. Another example would be to filter after a specific service type.

By looking at the listening 6.6 we can observe that only networks with the `export_tag` 1:170 and 1:171 are shown. This indicates that the source network exports the tag 1:170 and 1:171. Therefore the source can only communicate with the destination node if the network list is not empty. This concept is central to deliver the frontend only the correct data, allowing the frontend to send calculation requests that have no validation errors.

```
GET /api/endpoints/2/vrfs/?export_tag=1:170,1:171
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept
[
  {
    "id": 1,
    "url": "http://localhost:8000/api/endpoints/2/vrfs/1/",
    "name": "VRF-1",
    "ext_comm": [
      "1:170"
    ]
  },
  {
    "id": 2,
    "url": "http://localhost:8000/api/endpoints/2/vrfs/2/",
    "name": "VRF-2",
    "ext_comm": [
      "1:171"
    ]
  }
]
```

Listing 6.6: Example Request with Filtering by `export_tag`

## 6.6 Backend Architecture

This chapter contains an overview of the present backend architecture and shows the necessary communication between the different constructs. A visualization of the backend architecture can be found in figure 6.9.

It follows a short description of the different elements needed in the central part of the application. The backend application is composed of five different Django applications. Django apps are like sub-applications, which have separated logic. It can be compared with Python packages, which have a unique set of features, reused in other packages or even other Python/Django applications. Each unique application is marked with *Django Application*. The characteristics can be found in the description of the individual application. The differentiation of

the applications leads to design goals: loose coupling and high cohesion. Besides, the different applications stand for a different purpose and therefore fulfill the single responsibility principle.

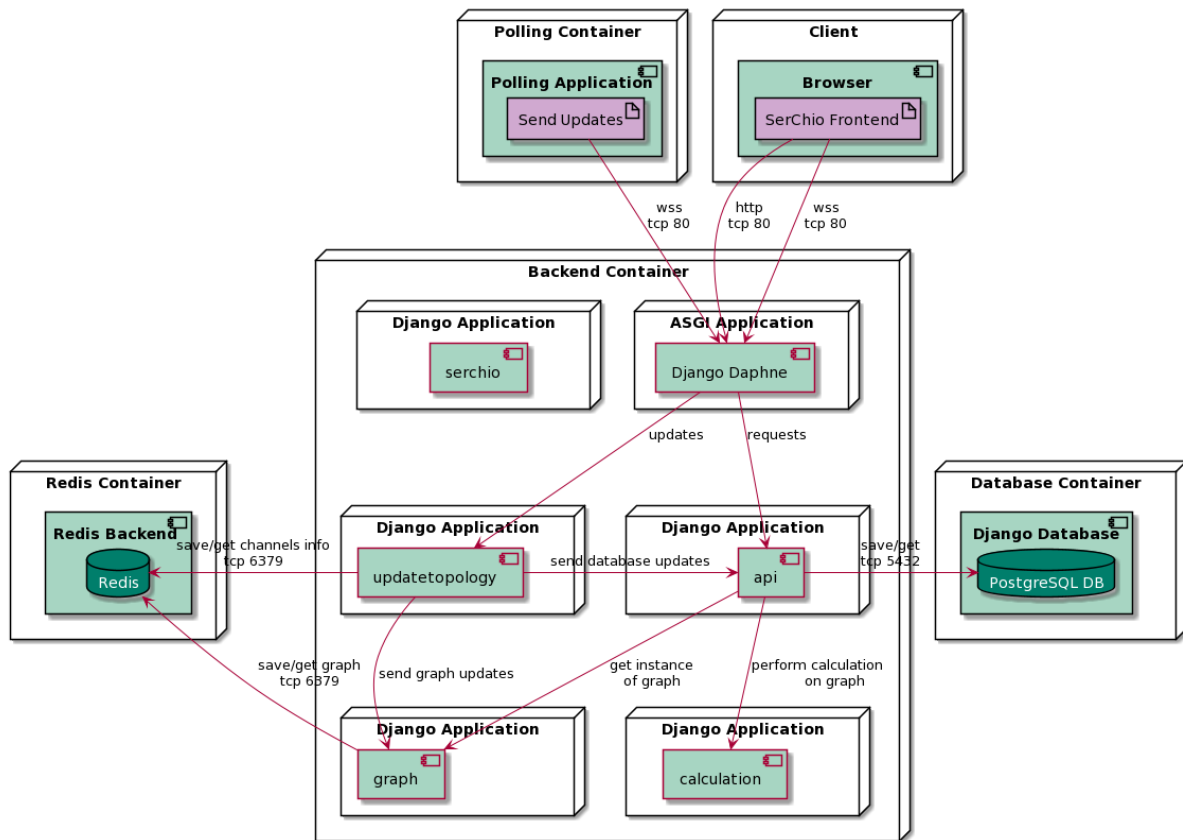


Figure 6.9: Backend Architecture Overview

### 6.6.1 Django Daphne

By default, Django is using a so-called web server gateway interface, which handles the requests synchronously. Because this application uses Django Channels, which extends the application's functionality with asynchrony, an [Asynchronous Server Gateway Interface \(ASGI\)](#) has to be introduced. Django Daphne is the default [ASGI](#) protocol server recommended by Django Channels, which can handle WebSocket and HTTP responses. Therefore, this instance is needed to terminate the HTTP requests sent by the frontend and forward it to the API application. Additionally, the WebSocket with the update information is terminated here and is forwarded to the updatetopology application, which handles the protocol data.

### 6.6.2 serchio

The serchio application is responsible for loading and applying the default settings to the other Django applications. It can be seen as the management application of the whole backend. It is also the application that loads the ASGI-application and includes different routes and URLs. Besides, it has the responsibility to load the other four applications' configuration and direct these settings to the individual one.

### 6.6.3 updatetopology

The updatetopology Django application includes the whole logic to process the topology updates. It gets information about changes in the network. After the reception, the updates will be processed. First, the information will be sent to the api application, which will trigger the writing of the database changes. Second, the changed data will be sent to the graph application, which refreshes the graph.

An additional part of the application is to save and read the Django Channels messages. It has the role to enable communication between different application instances. This task can be summarized as follows: it enables all connected WebSockets to get the information, and that updates in the topology exist. To implement this feature, the channel information has to be saved into the Redis instance.

### 6.6.4 graph

In the graph application, all is about creating and maintaining the graph instance. As soon as the application is started, a graph instance is created. For this purpose, it is first checked if a graph instance is present in the cache. If there is no graph instance present in the cache, this indicates that the application is started the first time. Therefore a new graph instance is created and maintained as soon as the first graph update information is received from the updatetopology app. If the application detects a present graph, the instance from the graph is restored and reused. This behavior can happen if, for example, the application was restarted.

Each time a calculation is performed, the api application asks for the current graph instance. The graph application delivers the current instance, which enables the calculation.

### 6.6.5 api

The api application, as the name mentions, includes the whole application programming interface, which is a central point for interacting with the program. It is responsible for offering different API endpoints. These endpoints can be used from an outside client retrieve as well as create elements in the backend. For example, it allows to get all topology endpoints or create a service instance.

It also has to be the interface between the PostgreSQL database and the Django application to retrieve and save the information. It contains the logic which allows translating Python objects for saving these to the database and vice-versa. This code is needed when topology changes are received, which were sent from the updatetopology app.

As mentioned above, another significant role is to enable the function to create things like a calculation. In this particular case, the graph application is asked for the current instance. The actual graph objects are together with the calculation request's information, are forwarded to the calculation application. After the calculation is finished, the api application has to save and return the calculation result.

### 6.6.6 calculation

The calculation application includes the logic to perform a calculation on a given graph. To do this, it offers methods which accept a graph instance and also additional calculation parameter. After this method is called, the calculation will be executed internally, and the result will be given back to the api application. More detailed information about the implementation can be found in the [chapter 2](#).

## 6.7 Sequence Diagrams

Sequence diagrams were created and described in the following sections to better visualize and explain the most important processes in the SerChio application.

### 6.7.1 Polling Service

The backend needs the data from the external software system Jalapeño. The backend gets this data from the developed polling service.

The polling service gets the data directly from the ArangoDB which is located in the external software system. After the data is loaded from the ArangoDB it is compared with the data in the Redis Cache. This procedure guarantees that the Redis Cache always contains data from previous executions and can then be easily compared with the new data. This way the polling service can only send the backend an update about which data has changed, which takes the load off the backend and increases performance.

This process can be viewed graphically in Figure 6.10. Internal processes have been omitted for better visualization.

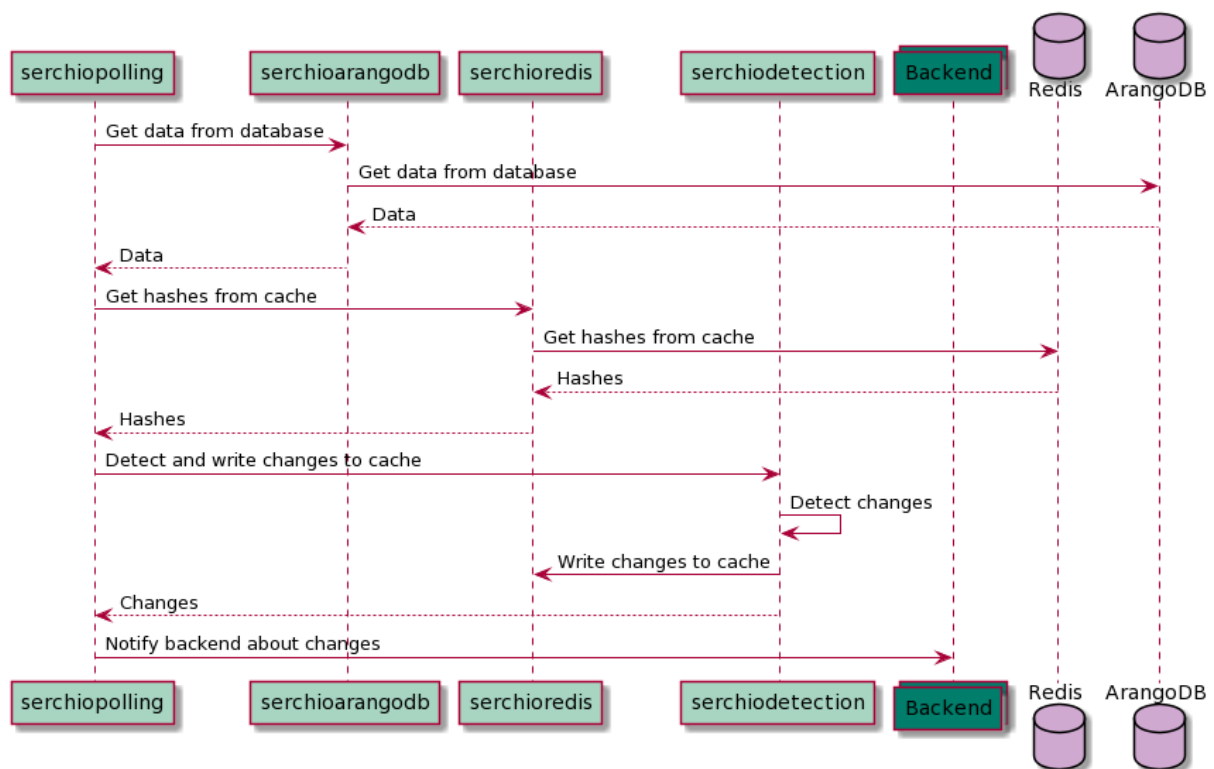


Figure 6.10: Polling Service Data Update

### 6.7.2 Get Data from API

A user or the frontend can retrieve data directly from the API. The process within the backend from request to response is always structured in the same way and is specific to Django.

As an example of the sequence diagram, the request GET topology was used, which should symbolically stand for the other requests.

When the API is requested, all requests go first to the `serchio` module (see [serchio](#)), from there they are forwarded to the `api` module (see [api](#)). In the `API` module, a so called `ViewSet` processes the request and sends a request to get the data from the database. So-called `models` are then responsible for the specific SQL statements in the database. After the data has been fetched from the database, it must be serialized so that a user or the frontend can work with it. So-called `serializers` are responsible for the serialization. Finally, the serialized data is returned to the user or the frontend.

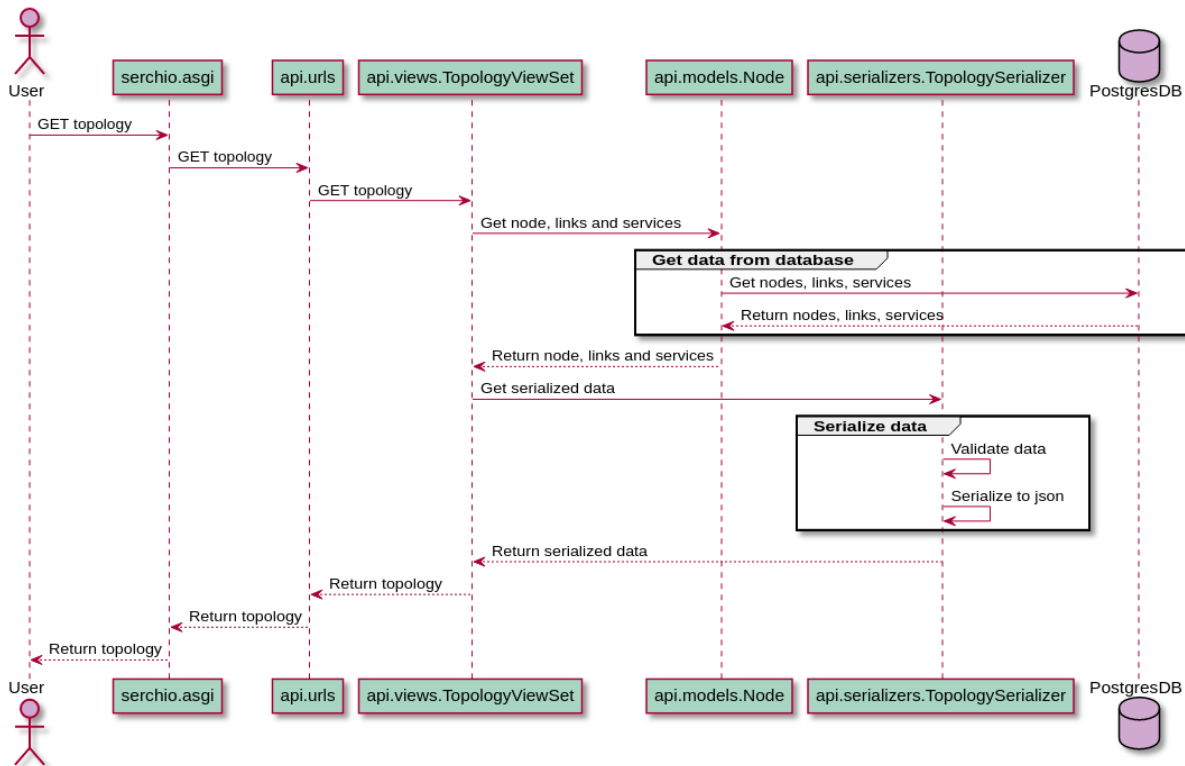


Figure 6.11: GET topology from API

### 6.7.3 Backend Data Update

The backend receives the data directly from the polling service via a WebSocket.

After the backend has received the data, it is responsible for maintaining the graph and the database with the newly deleted or updated data. For this purpose, the backend has a consumer which is responsible for forwarding the data to the Update Manager when data arrives via WebSocket. The Update Manager is responsible for distributing the data to the appropriate update listeners.

There are two different Update Listeners, each of which is responsible for one part of either the database or the graph. The listeners will then call the respective functions needed to update the database or graph.

This process can be viewed graphically in Figure 6.12. Some calls have been grouped together for better visualization.

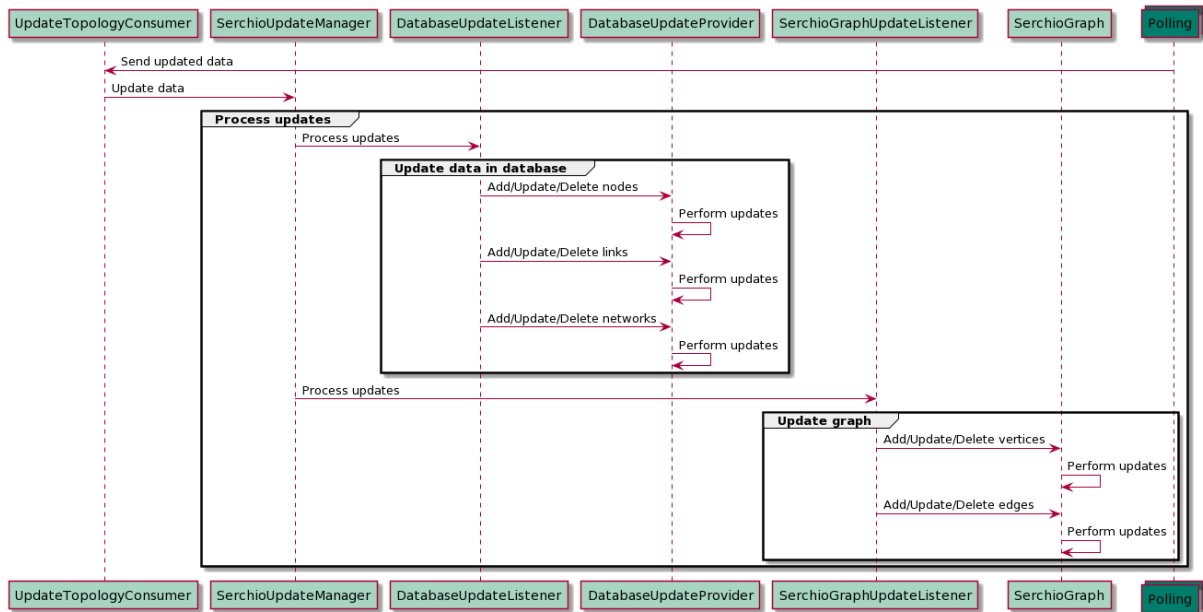


Figure 6.12: Backend Data Update

#### 6.7.4 Backend Calculation

The calculation of the service chain is the main task of the backend. Through a POST request, the backend receives the data it needs for the calculation and the request to perform the calculation.

For the calculation, the backend receives the source and destination parameters as well as the services and the metric. Via the CalculationViewSet the Calculation Model gets the task to load the required data from the database. This data is then used by an optimized serializer to calculate and create the new data in the database. The calculated path is then returned to the frontend or to the user.

This process can be viewed graphically in Figure 6.13. All internal calls for the calculation have been omitted for better visualization.



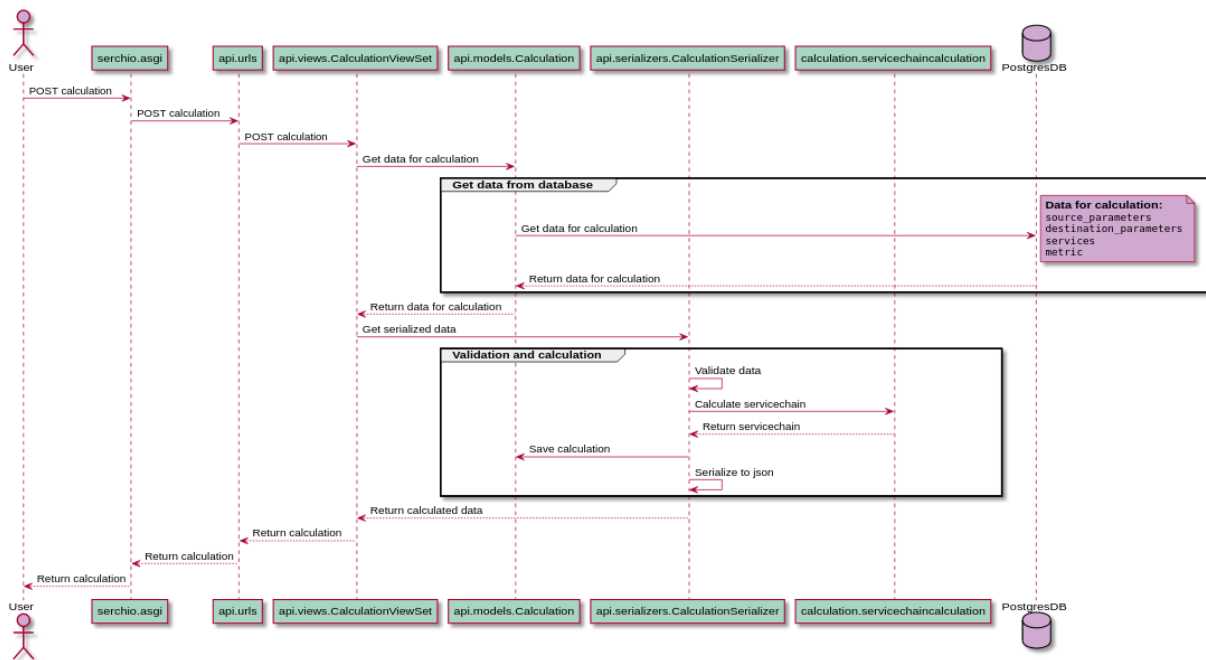


Figure 6.13: Backend Calculation Sequence

## 6.8 Deployment

The complete deployment of the Serchio application should be done using Docker containers. The containers for this are already created in the pipeline and are divided into development containers and productive containers. The productive containers are much lighter due to a multi-stage build and are therefore better suited for a productive environment.

As we made sure to develop as Cloud Native compliant as possible, the goal is to deploy the application accordingly. However, since the work is being continued in a bachelor thesis, the final deployment on the Kubernetes cluster could be outsourced to it.

### 6.8.1 Current Deployment

The focus in this work was to lay a foundation for the follow-up work. For this reason, great importance was attached to ensuring that the containers would be able to withstand the demands of maintainability, simplicity, and safety.

Since the Kubernetes deployment will not take place in this work, Docker Compose was used to create a clean deployment. The following deployment diagram gives an overview of the services and the connections between them.

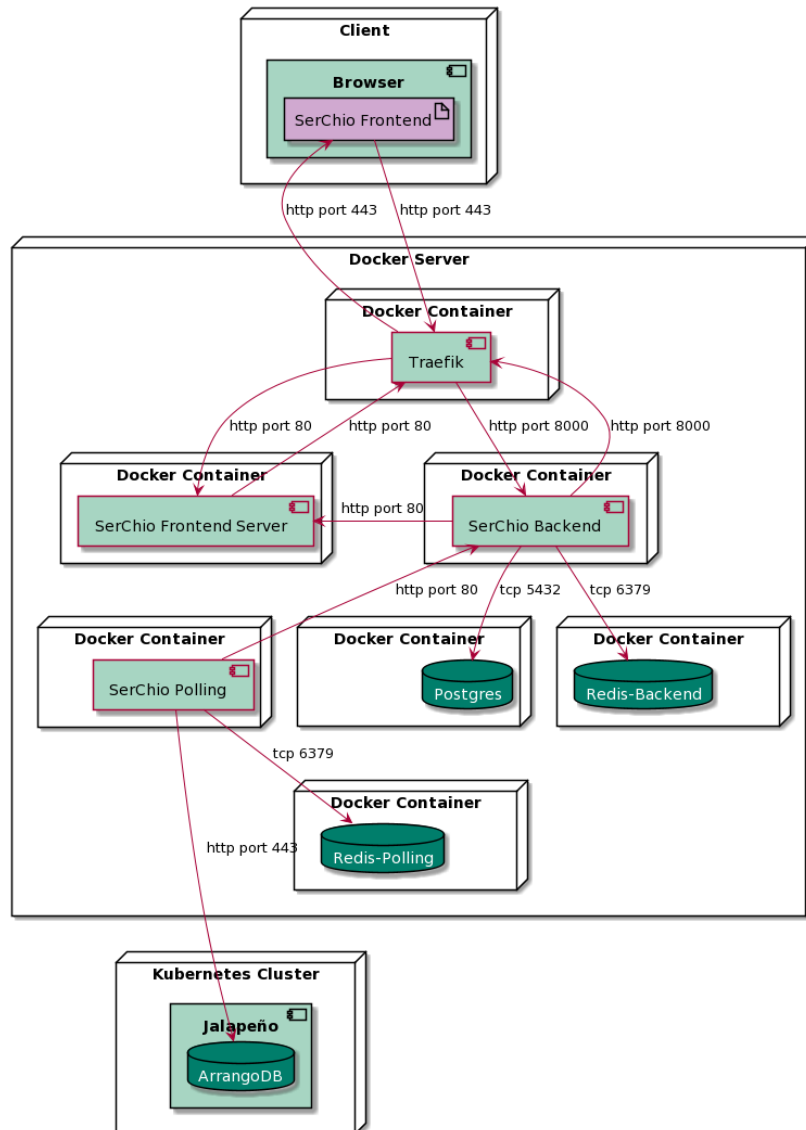


Figure 6.14: Docker Single Host Deployment Diagram

## 6.8.2 Planned Deployment

Due to the Cloud Native development of the entire project, the application should be deployed on a Kubernetes cluster in the future. The Kubernetes deployment will be tackled in the continuation of the project, i.e. in the bachelor thesis.

The following deployment diagram shows the intended final deployment on the Kubernetes cluster.

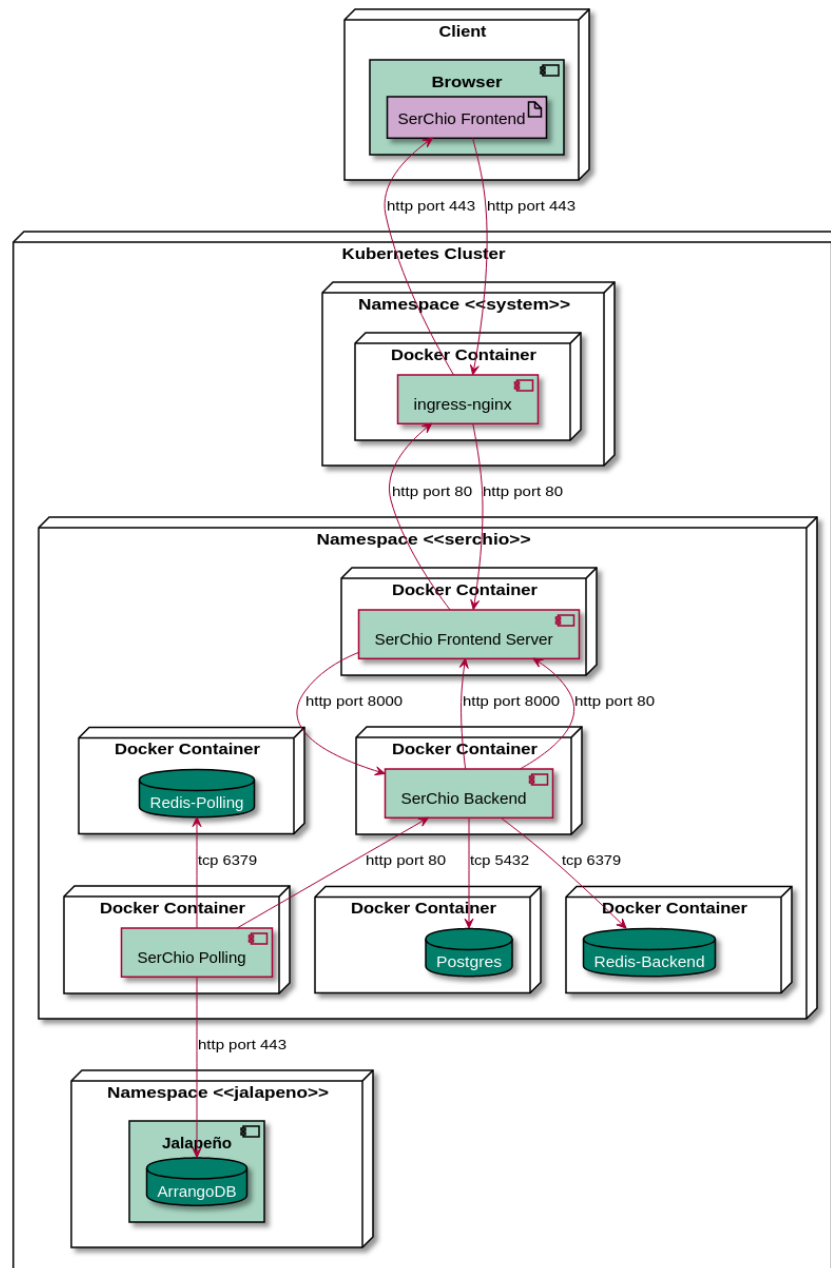


Figure 6.15: Kubernetes Deployment Diagram

## 6.9 Persistence

The following represents the persistence diagram with the single tables and the different associations that are responsible for the whole data retention.

There are a few things to consider so that the diagram can be read correctly.

- **Primary keys** are marked underlined and bold. They can be found in the first position in the table.
- Foreign keys are underlined and follow directly after the primary key at position two and following.
- *Uniqueness* is highlighted with an italic font.
- **Not Null** constraints are emphasized with a purple color.

- Fields can also have multiple options.  
For example, the *name* of a Node has to be unique and may not be Null/empty.

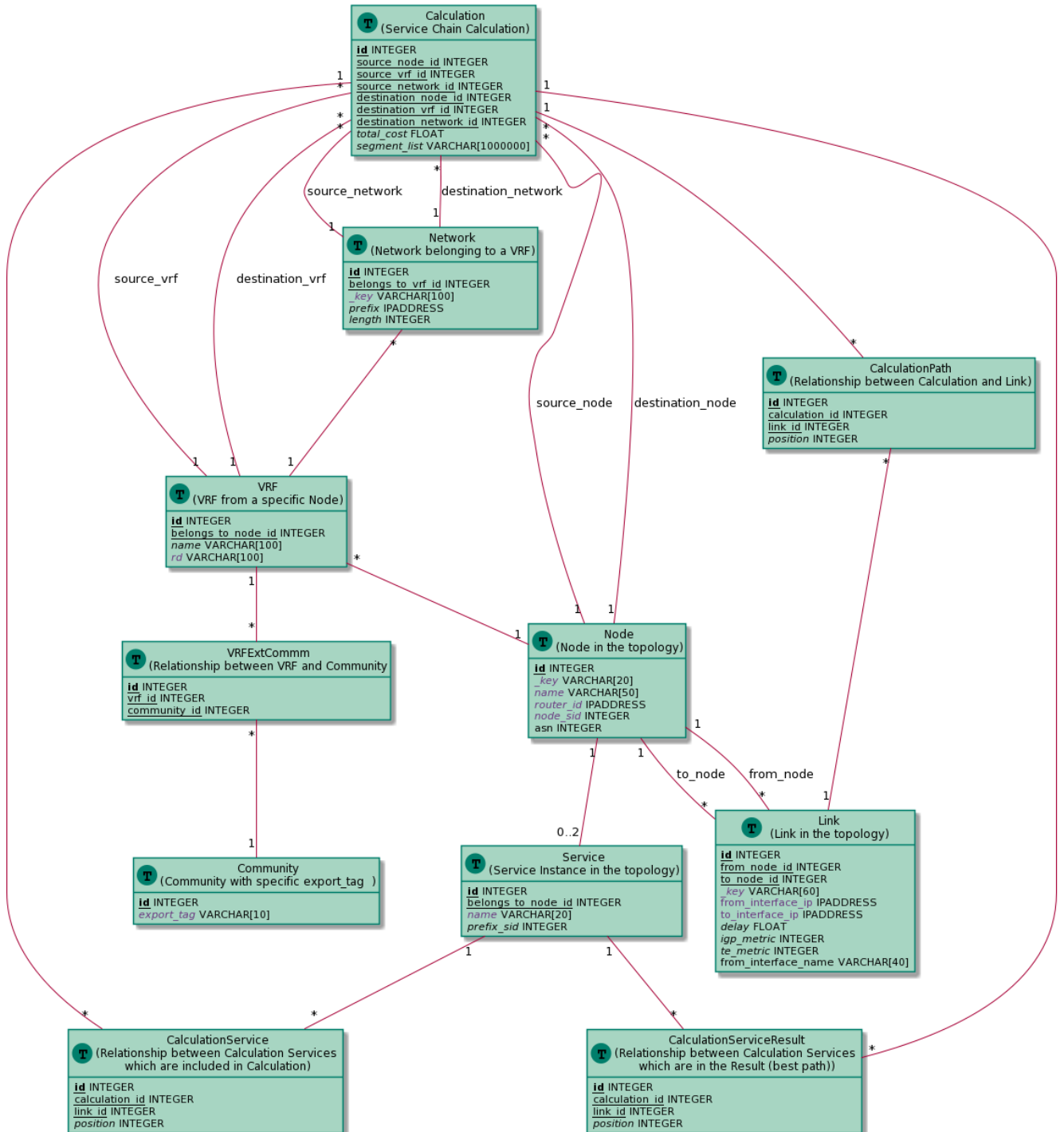


Figure 6.16: Persistence Diagram

The single tables have the following characteristics:

**Node** is the table which contains all Segment Routing nodes from the network. This table is one of the central parts of the application. Therefore it's connected to several other tables. With the `_key` a relation can be made to the Arango database documents and the

SerchioGraph vertices.

**Link** is the table which contains the connections between nodes. This table is directly related to the Nodes and the CalculationPath and therefore indirectly with the Calculation. With the `_key` a relation can be made to the Arango database documents and the SerchioGraph edges.

**VRF** is the table that contains the VRF information of nodes. This table is directly related to the Nodes and the VRFExtComm and therefore indirectly with the Community table.

**VRFExtComm** is the intermediate table between VRF and Community. It connects the different Communities, more specifically the different `export_tags`, with a concrete VRF.

**Community** is the table that contains a unique `export_tag`. The `export_tag` is central information in the whole application because the connectivity between VRFs and finally between endpoints is ensured with this information.

**Network** is the table which contains all network information. It has to belong to a VRF and is related to the Calculation table.

**Service** is the table that contains services configured in the Segment Routing Domain. A service has to be attached to a Node. Also, it has a connection to the CalculationServiceResult and the CalculationService.

**Calculation** is the table that contains information about performed calculation. The intermediate table CalculationPath is used to connect the Calculation table with the Link table. This relation has to be ensured to store the `best_path` in the calculation. There exist two different relations to the Service table. The first one, CalculationService, is used to determine which Services were taken into account in the calculation. The second one, CalculationServiceResult, is used to store the `service_result`, which services are used according to the most suitable best path.

**CalculationPath** is the table that connects the Calculation with the Link table. A singularity is the `position` field, which has to be introduced to order the best path.

**CalculationService** is the table that connects the Calculation with the Service table. It is used to maintain the information, which service instances were included in the calculation. The `position` element is vital, according to ensure the correct order in the service.

**CalculationServiceResult** is the table that connects the Calculation with the Service table. It is used to maintain the information, which service instances were included in the result of the calculation according to the best path. The `position` field is vital, ensuring that the information is stored, which service instance is used at which position of the service chain.

## 6.10 Packages and Classes

The following sections describes the different packages and classes in the two components backend and polling.

### 6.10.1 Backend

#### Packages

Like mentioned in the chapter. 6.9, the central part of the backend application consists of 5 different packages. The serchio app is the configuration app and has, therefore, limited dependencies to the other packages. Each Django application uses specific other ones. The overview of the package communication can be obtained in the figure 6.17

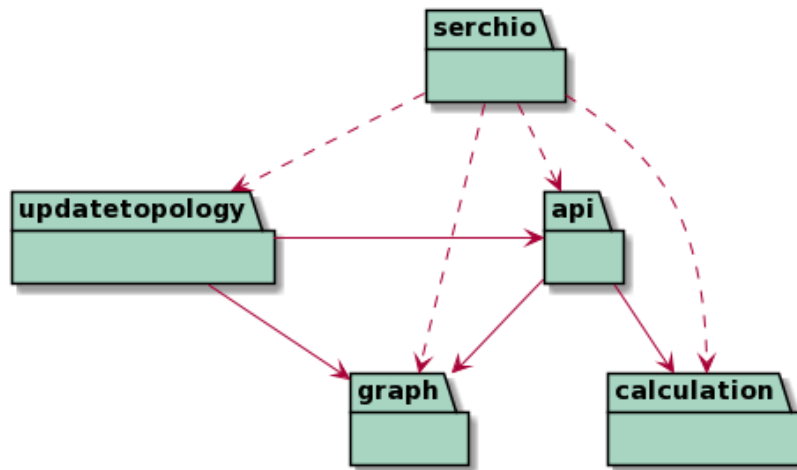


Figure 6.17: Package Diagram Backend

## Classes

In the figures below, the essential packages and classes of the backend application can be found. For reasons of clarity a too much in detail illustration was omitted. A more detailed class diagram of the central api package can be found under ??

In the backend, the development approach *develop against an interface* was used. With this concept, the code includes several classes that extend from a general abstract class, which is the substitution of interfaces in the Python language. An example of this approach can be seen in the package `updatetopology.services.listeners` in figure 6.19.

This approach helped improve the code concerning dependencies and testing. It is possible to create classes with the same interface as the actual classes and write test cases really fast with this concept paired with dependency injection.

In the code, a meaningful naming was introduced. The naming should help to understand the responsibility of each class. For example, all abstract class names extend from the Python Abstract Base Class (short ABC) and end with the noun *interface*. All abstract classes in the figures below are also marked with an *A*. The general classes are stand out with a *C*

Especially to emphasize is the class `SerchioGraph`, which is a Singleton class. The graph class was written in such a way that always only one graph can exist. The class can be found in figure 6.20 and is marked with a *S* and unique string *«Singleton»*.

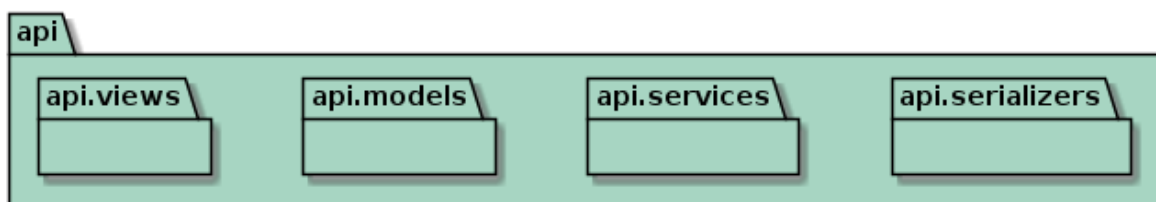


Figure 6.18: Class Diagram api

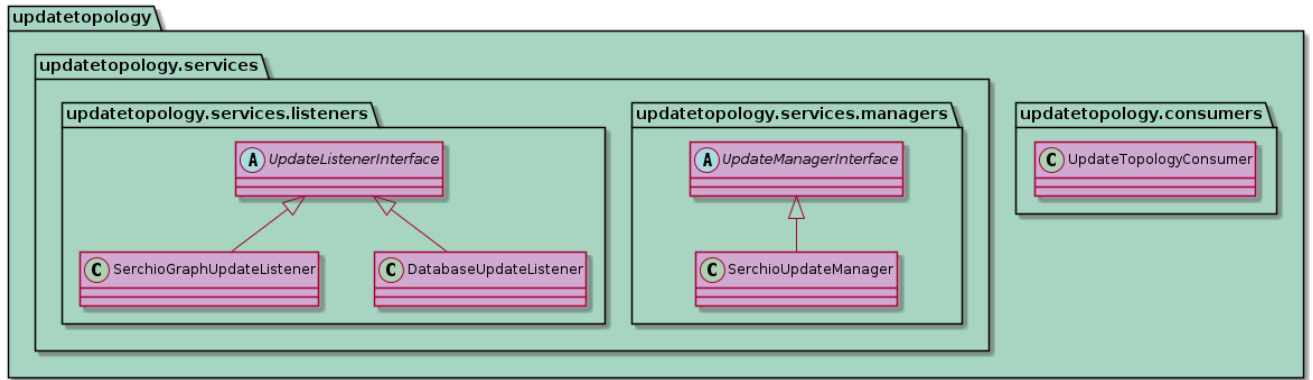


Figure 6.19: Class Diagram updatetopology

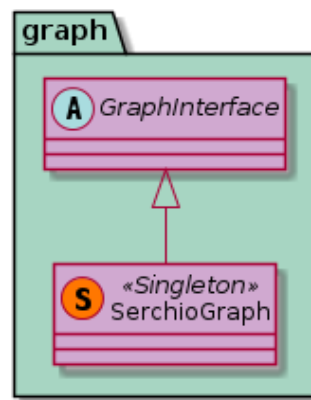


Figure 6.20: Class Diagram graph

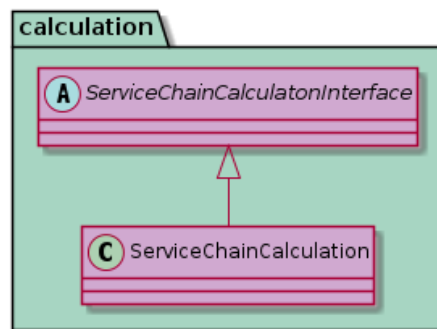


Figure 6.21: Class Diagram calculation

### 6.10.2 Polling

With the polling service, as with the backend, care was taken to program against the interface. Since Python does not know interfaces, abstract classes were used for this. The advantage of this approach is that Dependency Injection can be used to easily inject different mocks, which simplifies testing extremely.

By strictly using Dependency Injection, the dependencies between the classes could also be reduced. As you can see in Figure 1, the SerchioPolling class has the task to call the other

classes, so this class is the hub for the whole polling service.

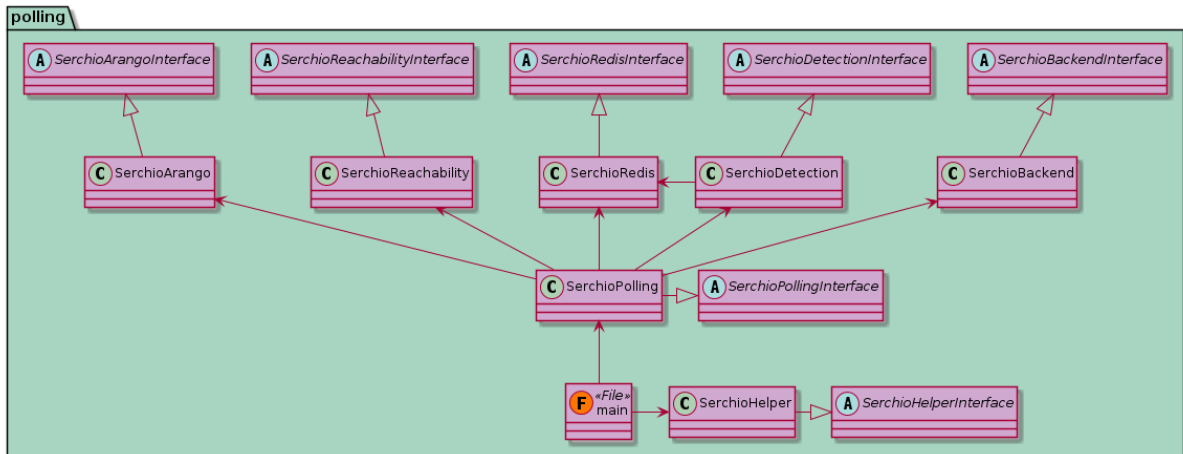


Figure 6.22: Class Diagram polling



## Chapter 7

---

# Project Management

---

In this chapter, the whole project management specific documentation can be found. The time management, milestone, sprints, developer concepts, and risk analysis are documented here. This chapter is continuously updated during the project. All changes can be tracked in the table [7.1](#).

Date	Version	Changes	Author
09.12.2020	1.6	Update sprint planning (sprint 7).	sdellsperger
25.11.2020	1.5	Update sprint planning (sprint 6).	sdellsperger
11.11.2020	1.4	Update sprint planning (sprint 5). Update risk management: R2 - service military - can be annulled because service is ended Update risk management: R5 - calculation - can be deleted because calculation is implemented	sdellsperger
28.10.2020	1.3	Update sprint planning (sprint 4).	jklaiber
14.10.2020	1.2	Update sprint planning (sprint 3). Update risk management: Define that Jalapeño data will be mocked	jklaiber
30.09.2020	1.1	Update sprint planning (sprint 2).	jklaiber
22.09.2020	1.0	Initial project plan was created.	jklaiber sdellsperger

Table 7.1: Version History Project Mangement

## 7.1 Project Management

As a project management method, we use Scrum Plus, which is a combination of Scrum and Unified Process. Scrum Plus is mainly taught at the University of Applied Science of Eastern Switzerland and, therefore, mostly used in projects like this. This project management method enables us to achieve an agile approach with fixed milestones in the various phases of the Unified Process (Inception, Elaboration, Construction, Transition).

## 7.2 Scheduling

The eight credits allocated to this module result in a workload of 240 hours per person, which should be complete. Because we are a team of two, we have 480 hours available for this project.

Those 480 hours should be used up ultimately. Since we have chosen an agile approach, either functionality is omitted if there is not enough time, or new functionality is worked on with the remaining available time.

Below you can find an actual timeline of the project. All sprints are planned for a duration of two weeks. Issues that cannot be closed in a sprint are moved to the next sprint. A detailed description from the sprints can be found at the section [7.2.1](#), and for the milestones at the section [7.3](#).

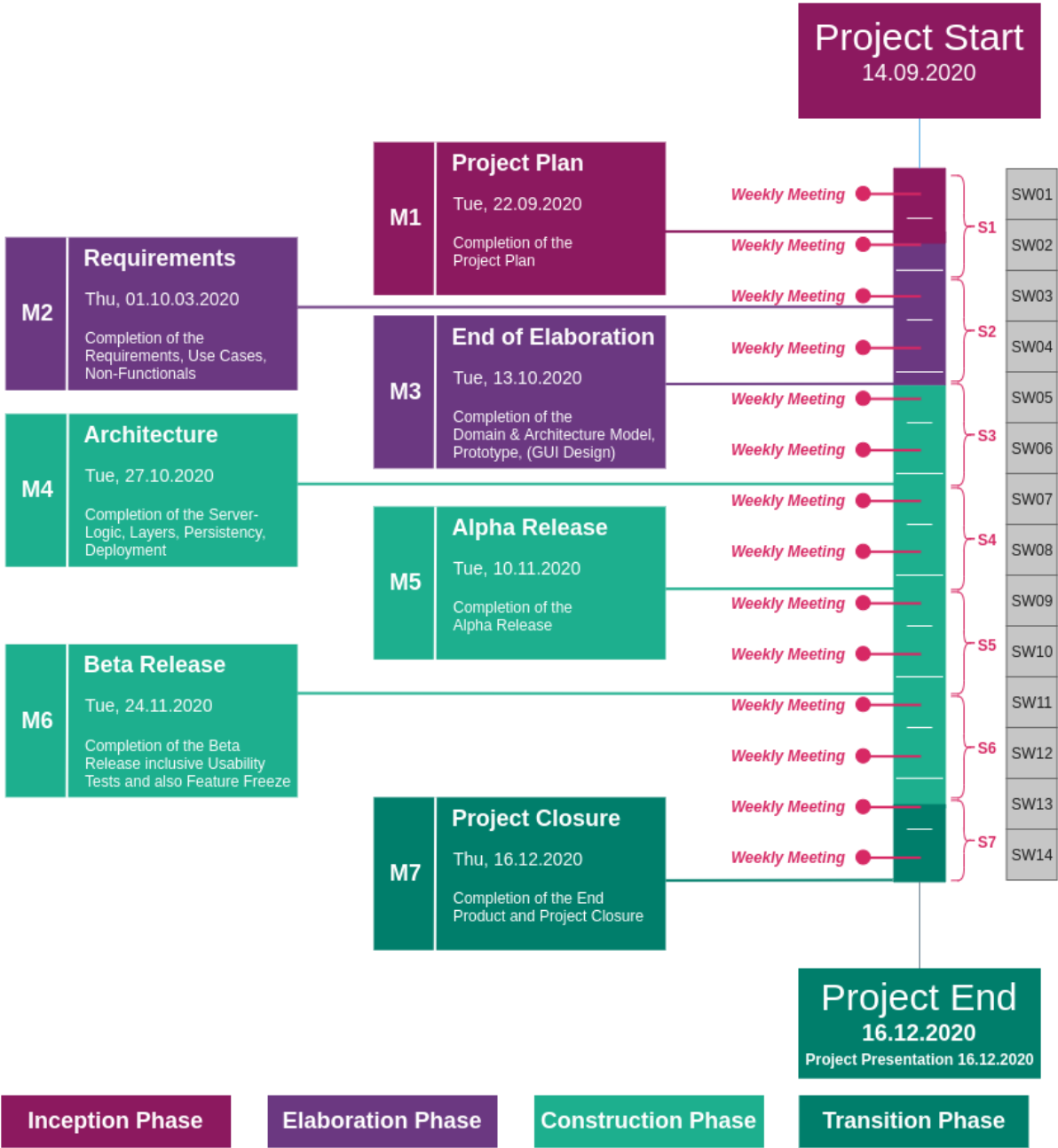


Figure 7.1: Project Timeline

7.2.1 Iteration Planning

The fact that an agile approach has been chosen led that there are no fixed work packages within a Sprint. At the beginning of each Sprint, it is decided which work packages will be included in a Sprint. Therefore, table 7.2 is continuously updated during the project.

Inception		
<b>Sprint 1</b>	15.09. - 29.09.	KickOff meeting with industrial partner Creation of the project plan
Elaboration		
<b>Sprint 2</b>	30.09. - 13.10.	Architecture and domain models API Specification for the frontend First simple prototype
Construction		
<b>Sprint 3</b>	14.10. - 27.10.	Creation of the CI/CD Django model creation Creation of the topology endpoints for the frontend
<b>Sprint 4</b>	28.10. - 10.11.	Update API endpoints with nested links Graph calculation and maintenance
<b>Sprint 5</b>	11.11. - 24.11.	Calculation with different metrics (IGP, TE, Delay) Creation segment list Code optimizations
<b>Sprint 6</b>	25.11. - 08.12.	Finish testing Bug fixing
Transition		
<b>Sprint 7</b>	09.12. - 16.12.	Finish thesis documentation

Table 7.2: Iteration Planning

### 7.2.2 Estimates

Estimates are recorded directly in the YouTrack tool on the respective cards. This allows us to easily track time via the dashboard or different time reports.

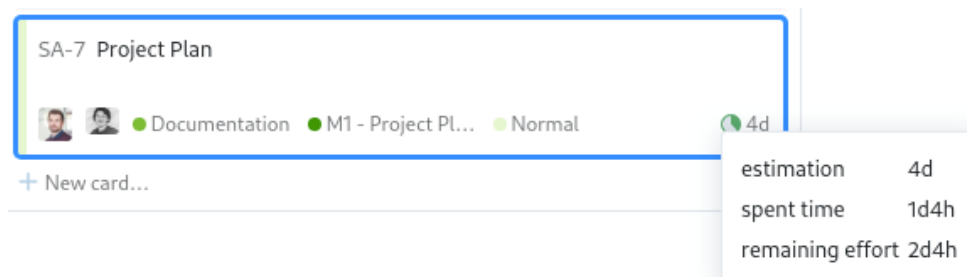


Figure 7.2: YouTrack Estimates

### 7.2.3 Time Evaluation

The complete time recording, as well as time evaluation, is carried out with the project management tool YouTrack from JetBrains. This tool also allows us to export visual graphics with the help of so-called reports.

## 7.3 Milestones

We defined a total of seven milestones to track our progress during the project. The milestones are each closed after a weekly meeting to be able to react to suggestions for improvement from the supervisor and the industry partner.

Between the different milestones, which, as already mentioned, only serve to control the progress of the project, work is done as agile as possible.

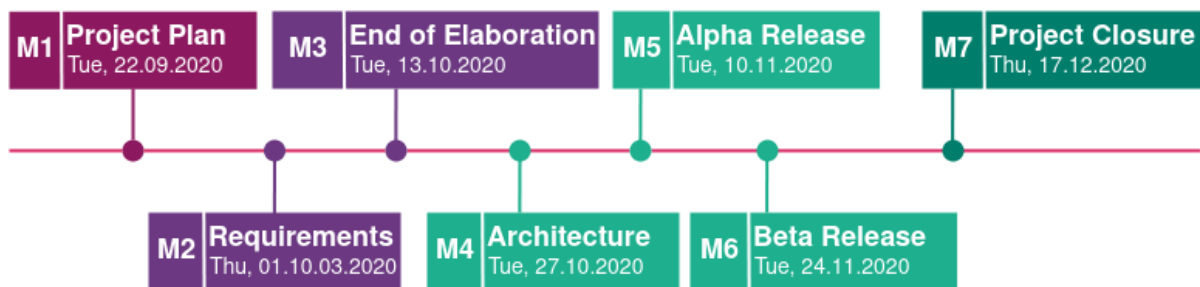


Figure 7.3: Milestone Overview

Milestone	Due Date	Goals
<b>M1 - Project Plan</b>	22.09.2020	The goal is to complete the project plan and initialize the project management tool (YouTrack) and LaTeX template.
<b>M2 - Requirements</b>	01.10.2020	The goal is to create the requirements, use cases and non-functional requirements, which fulfill the needs of the industrial partner.
<b>M3 - End of Elaboration</b>	13.10.2020	The goal is to create a first minimal prototype over all layers of the application. Also the domain and architecture model should be finished. All risks should be minimized.
<b>M4 - Architecture</b>	27.10.2020	The goal is to create the server logic, layer, persistency and deployment diagram which fulfill the needs of the architecture model.
<b>M5 - Alpha Release</b>	10.11.2020	The goal is to have a first working release.
<b>M6 - Beta Release</b>	24.11.2020	The goal is to have a release which can be used in production. Besides, there is a feature freeze.
<b>M7 - Project Closure</b>	17.12.2020	The goal is to finish the project. This means all source code is documented. All documentation is completed and the whole project, including all enclosures, is delivered.

Table 7.3: Milestone Description

## 7.4 Meetings

The meetings' main aim is to identify problems at an early stage and communicate the status to the other project participants, including the industry partner. The conference will take place on-site if possible. Additionally, the industry partner presented by Francois Clad will join the meeting via Microsoft Teams. Due to the Corona crisis, it is also essential to comply with the local COVID-19 regulations. The appointment will take place weekly on Wednesday morning.

## 7.5 Responsibilities

The two authors Severin Dellsperger and Julian Klaiber are responsible for the backend and the polling application. The front-end is assigned to the Institute for Networked Solutions, after consultation with the supervisor.

For the backend and the polling application, the distribution of the work was based on the interests and experiences of both authors. Severin Dellsperger concentrated more on the implementation with Django and Julian Klaiber on the whole cloud-native work like the docker containers and the whole GitLab workflow.

In the end, both authors worked on all the components, and with the responsibilities, they were able to use the synergies perfectly.

## 7.6 Repositories

Due to the fact that in this project work only one backend is developed, it was not necessary to create several repositories. To limit the access rights, a separate GitLab group was created in which the repository is located.

The repository SerChio contains in this case the complete implementation of the backend for the project Service Chaining Path Calculation.

README.md	Project description
.gitlab-ci.yml	Pipeline definition
docker-compose.prod.yml	Productive Compose file
docker-compose.yml	Development Compose file
.env*	Various environment files for Dev and Production
backend	Backend directory
Dockerfile	Development Dockerfile
Dockerfile.prod	Multistage Dockerfile for production
entrypoint.sh	Development Docker entrypoint script
entrypoint.prod.sh	Productive Docker entrypoint script
requirements.txt	Python requirements for the backend
manage.py	Django management service
api	Backend API module
updatetopology	Backend updatetopology module
serchio	Backend main module
calculation	Backend calculation module
common	Backend common module
benchmark	Benchmarking for module decisions
benchmark.py	Benchmarking service
requirements.txt	Python requirements for the benchmarking
data	Data for the benchmarking
mocking	Topology mocking
mocking.py	Topology mocking service
l3vpn_prefixes	L3VPN mocked data
lsnode	LSNode mocked data
lsv4_topology	Mocked LSv4 topology data
polling	Polling service
testing	Directory with all Polling related tests
mocks	Directory with all Polling related mocks
Dockerfile	Development Dockerfile
Dockerfile.prod	Multistage Dockerfile for production
entrypoint.sh	Docker entrypoint script
main.py	Main Polling class
helper.py	Polling service helper class
i*.py	Polling related interfaces
serchio*.py	Polling related classes
requirements.txt	Python requirements for the polling service

Table 7.4: Project Structure

## 7.7 Infrastructure

A virtual machine in the INS network was set up especially for the semester thesis. The VM runs various docker services such as the GitLab Runner and Sonarqube. Furthermore, microk8s is installed which is needed for the test environment of Jalapeño.

The official GitLab instance was used as a versioning tool and for the CI. As an issue tracker,

You Track from JetBrains is used which is freely available to students.

## 7.8 Development Concept

### 7.8.1 Definition of Done

To keep the project's quality as high as possible and to close issues only when it is finished, care is taken to always adhere to the Definition of Done.

The general definition of done contains the following points:

- [Code Style Guidelines](#) are met.
- The [Continuous Integration](#) run successfully without errors.
- The documentation is updated.
- All unit tests run successfully.

### 7.8.2 Code Style Guidelines

In order to obtain a source code that is as uniform as possible, care is taken to work according to the [PEP8](#) style guidelines. PEP8 is the de facto standard for Python code style guidelines.

### 7.8.3 Development Workflow

The classic Git Workflow, which can be seen in [Figure 7.4](#), is used for development. The pipeline and review by another developer ensure code quality.



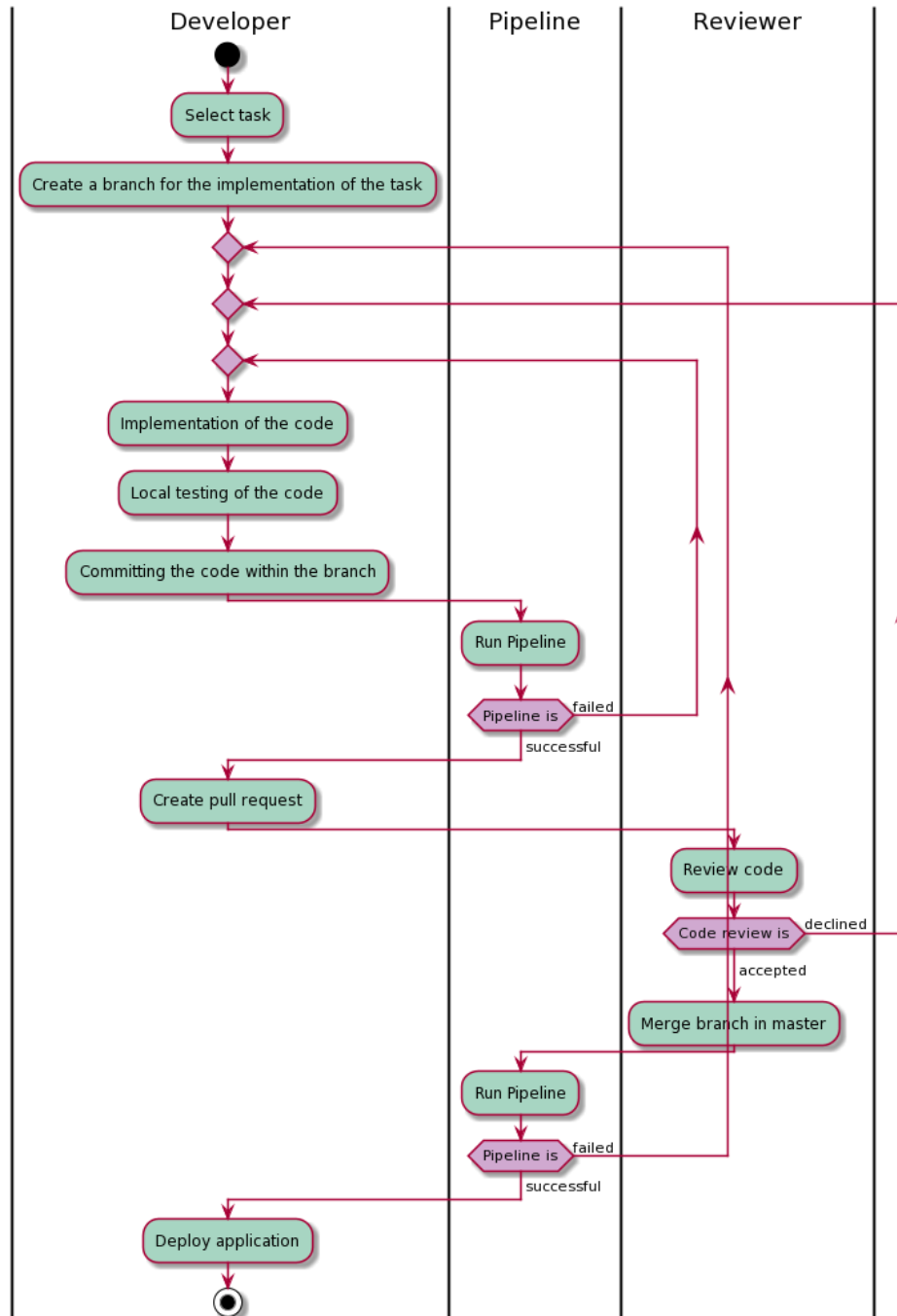


Figure 7.4: Development Workflow

## 7.9 Continuous Integration

The complete integration is implemented using the GitLab pipeline. The different stages are described below in Table 7.5.

Stage	Description	Execution
1. <code>init_testing</code>	Prepares the virtual environment for testing purpose.	On every commit
2. <code>testing-backend</code>	Tests the backend with the django built-in testing tools.	On every commit
3. <code>testing-polling</code>	Tests the polling service with the unit and coverage module.	On every commit
4. <code>dev-build</code>	Builds the non-productive Docker images	On every commit exclude the master branch
5. <code>sonar</code>	Uploads the generated coverage XML files to Sonarqube.	When merging in the master branch
6. <code>prod-build</code>	Builds the productive Docker images from the multistage Docker-files.	When merging in the master branch

Table 7.5: Continuous Integration Stages

The workflow in Figure 7.5 is intended to show graphically when which stages of the pipeline start and how the flow is in between.

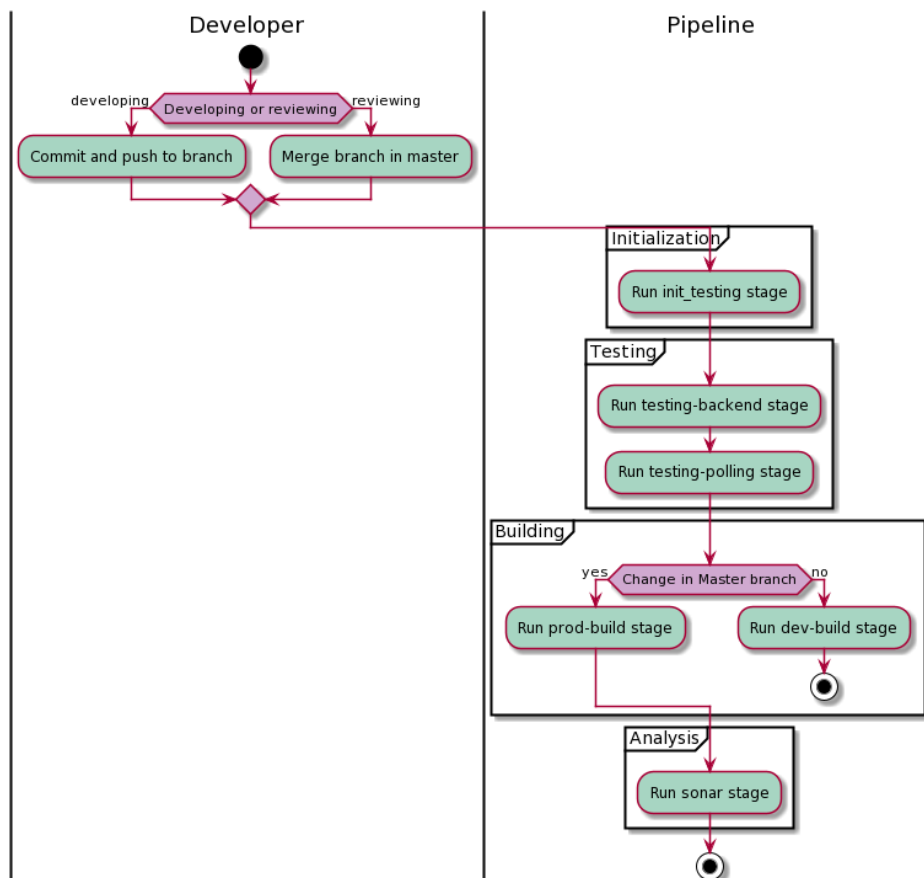


Figure 7.5: Pipeline Workflow

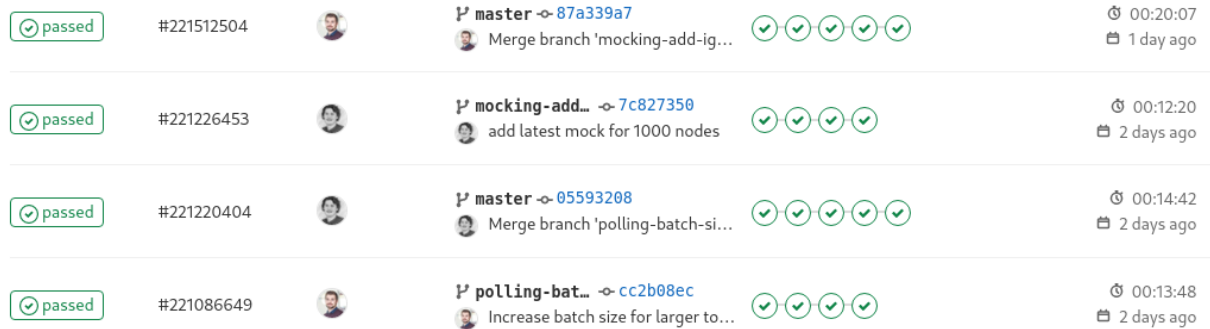


Figure 7.6: Executed Pipeline Example

## 7.10 Code Metrics

A dedicated Sonarqube instance was deployed on an internal server at the Institute for Networked Solutions to analyze the entire code. Sonarqube can analyze an enormous number of different metrics from different programming languages. In the CI Stage `sonar`, which was described in table 7.5, the collected code metrics are transmitted directly to Sonarqube using the Sonar Scanner and are analyzed.

Sonarqube includes a visual dashboard (see figure 7.7) that displays all code metrics in time history. In addition to the dashboard, you can also define your quality gates to be met.

In addition to the code metrics that can be read from the Sonarqube dashboard, each test stage in the pipeline (see table 7.5) will also output the metrics directly to `stdout` in the pipeline. This allows you to read the metrics for each commit directly from the pipeline.

The detailed quality attributes for this project can be found in the section 7.12.

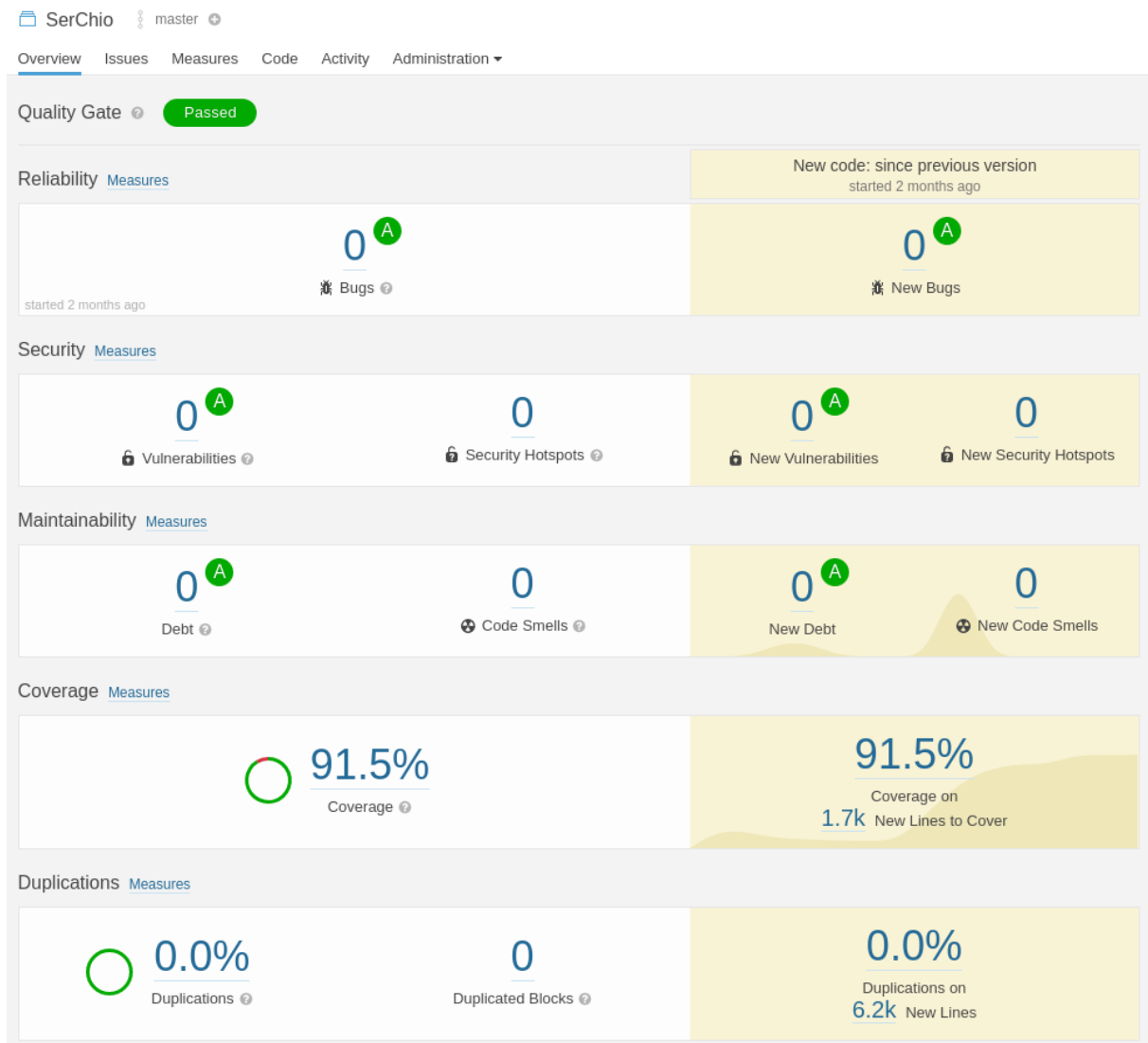


Figure 7.7: Sonarqube Dashboard

## 7.11 Risk Management

The following chapter contains an overview of the most crucial project risks. An outline of the various risk can be found in figure 7.8, a more detailed description of each risk can be found in the table 7.6.

The most significant risks are directly related to the Jalapeño application, which aggregates and provides data from the network. Since the Jalapeño application is not well-engineered yet, the authors and the supervisor have decided that it's best to fake the information delivered from the external application. Indeed this decision led to the continuation of the thesis but has to be dealt with in the future.

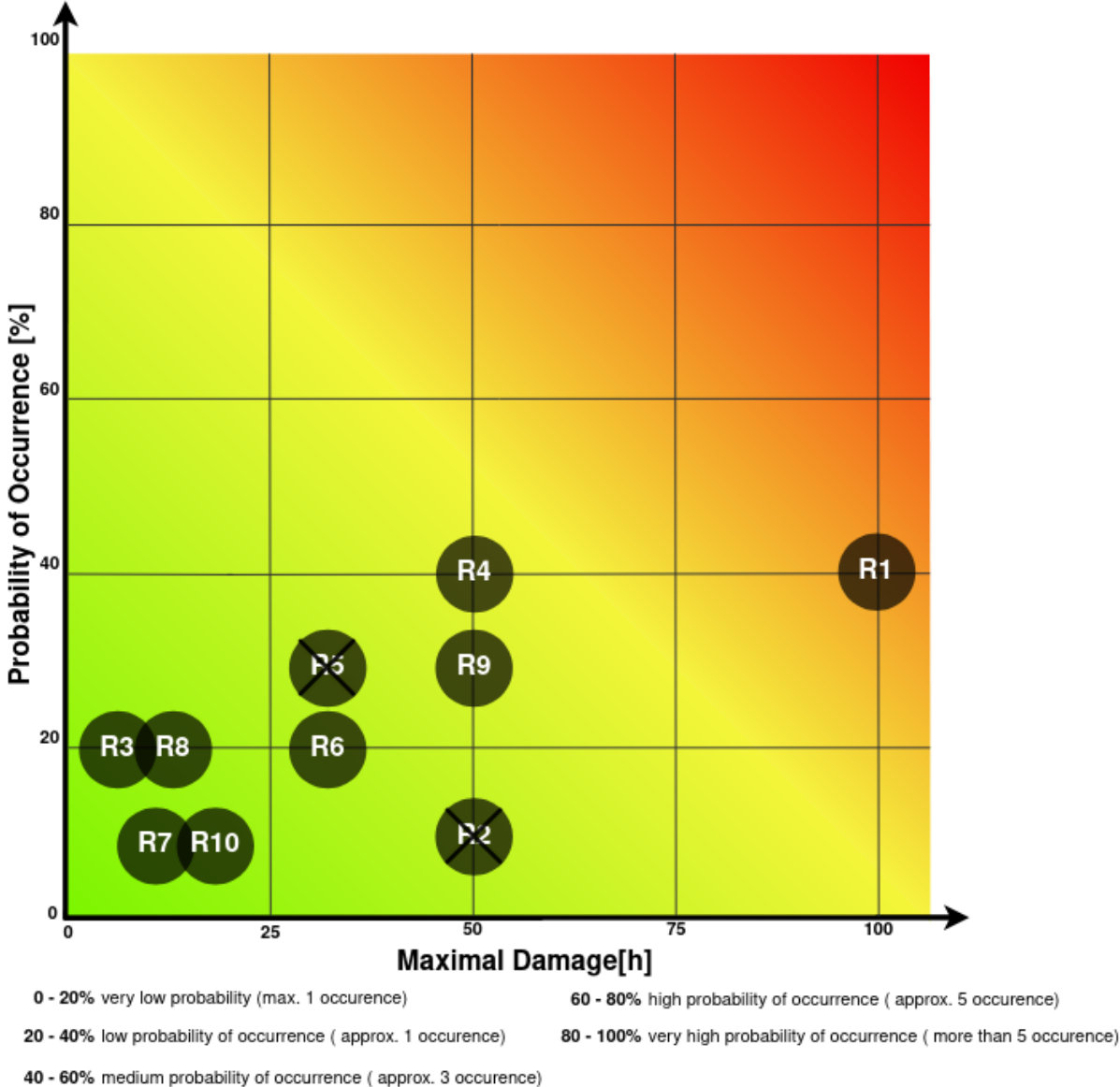


Figure 7.8: Risk Overview

#	Risk Description	Damage [h]	Probability	Weighted Damage
R1	The Jalapeño application does not deliver the information which is needed to implement the application logic.	100	40	40
R2	Severin Dellsperger has no option to work on the thesis during his service in the Swiss Armed Forces.	<del>50</del> 0	<del>40</del> 0	<del>5</del> 0
R3	There is a second lockdown in Switzerland or one of the team members is forced to go into the quarantine.	8	20	1.6
R4	A team member has to suspend working due to illness (most likely COVID-19) or an accident.	50	40	20
R5	The team is unable to implement the calculation of the total best path between the selected source and destination network.	<del>30</del> 0	<del>30</del>	<del>9</del> 0
R6	Extracting the segment list from the calculated most suitable path turns out to be more difficult and time-consuming than expected.	30	20	6
R7	The discovery of the different services and the various services nodes is not as easy as awaited and takes more time as expected.	10	10	1
R8	The coordination between the backend team and the front end developers is time-consuming and thus leads to delays.	16	2	3.2
R9	Topology changes lead to wrong paths because the update of the best path is not working correctly or the time is missing to implement this feature.	50	30	15
R10	Used software components like tools, libraries, and frameworks include unknown bugs or faults, causing delays in the project.	20	10	2

Table 7.6: Risk List

### 7.11.1 Dealing with Risks

The risk analysis carried out above in table 7.6 enabled various measures to be defined in table 7.7 to minimise or prevent the risk. Risks that have been crossed out have been eliminated during the project so that their occurrence is no longer possible.

#	Prevention	Measures
R1	The team members maintain close contact with the development team of Jalapeño and try to identify and communicate problems early.	If information is missing in the Jalapeño application, the developer will be contacted directly and will try to fix the problem together. If the Jalapeño application can not deliver the necessary data, it will be mocked to prevent the disturbance of the project's continuation.
R2	Severin Dellsperger is in permanent contact with his commander to be always up to date.	Severin Dellsperger will try to seek dialogue to find an individual solution with the military.
R3	The two team members adhere to the prescribed COVID-19 regulations from the BAG and avoid infection with the virus.	Through the first lockdown, many experiences with working remotely have already been gained. This risk can be minimized by continuous documentation and close contact between the two team members.
R4	By adhering to hygiene measures, both team members try not to contract a disease.	In case a team member falls ill, the situation is analyzed with the supervisor and depending on the duration of the absence, functionality is then removed.
R5	Different algorithms and procedures are considered in order to find a satisfying solution	If the calculation cannot be completed, the supervisor will be asked to find a solution as soon as possible.
R6	A paper is used to describe an algorithm that describes how to extract the segment list from the best path.	If this algorithm cannot be implemented as desired, the supervisor will be asked to contact the industry partner who provided the paper.
R7	Fixed segment areas are used to locate and identify the various services. The supervisor prepares the configuration to avoid misconfigurations.	The supervisor and the Jalapeño development team are asked to help localize the services. If the Jalapeño application can not deliver the configured service locations, the necessary data will be mocked to continue the project.
R8	Good communication and planning between the two teams should minimize the risk.	If this risk occurs, an extraordinary meeting with the supervisor will be organized to resolve the problem as quickly as possible.
R9	Continuous testing should enable incorrect paths to be detected and corrected as early as possible.	If this problem occurs, direct contact with the supervisor is requested to find the best possible solution for this functionality. If no solution is found, this feature must be omitted.
R10	The use of well-known components and tools like Visual Code as IDE, python3 as the primary language, Django as the framework, and Django REST Framework as an add-on minimize the risk score overall.	If there is a problem with the utilized software components, the Institute of Networked Solution employees will be asked for help. They contain considerable knowledge in this area.

Table 7.7: Dealing with Risks

### 7.11.2 Implications

If one or more risks occur, the project's scope can be reduced, and functionality can therefore be omitted. However, this should not be basic functionality. In such a case, the supervisor will be consulted as early as possible to discuss further procedures together with the team.

## 7.12 Quality Attributes

By using sonarqube as a code analysis tool, we also limit ourselves to the main metrics used by this tool. The most important metrics are described below and our goals for these are defined.

Metric	Description	Goal
Bugs	Bugs are errors in the code which can break the execution of the application.	0
Security Hotspots	Code which you have to check manually to make sure that there is no security hole.	0
Code Smells	Code which is hard to understand or to read.	0
Coverage	Coverage describes what percentage of the code is covered by tests.	min. 80%
Duplications	Duplications describes what percentage of the code is duplicated.	max. 1%

Table 7.8: Quality Attributes

## 7.13 Exception Handling

The backend continuously validates all incoming data and provides feedback to the frontend and backend. The validation is especially important for the calculation of the service chain, otherwise calculations are performed which do not make sense.

```
HTTP 400 Bad Request
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
  "source_node": [
    "The source_node has to have VRF configured (must be PE)"
  ]
}
```

Listing 7.1: Backend Validation Example

The polling service has many external dependencies in the progress of this work care was taken to ensure that the service is very stable. Therefore a mechanism was developed which stops all jobs in case of an error and waits until the external dependencies are working again without problems, only then the job will be executed. This handling has the advantage that the polling service is running continuously and is not crashed by errors in a dependency.



## 7.14 Logging

By trying to develop the project according to the [12-Factor Methodology](#), the logs are written to `stdout` as described in section [6.3.11](#). This approach has the advantage that the logs can also be extracted into a logging system and are thus centrally located in one place.

During the development, we paid attention to the fact that the different debug levels should provide the information you need. The time invested in this work pays off at the latest when you have to debug something.

Especially the polling component has a lot of external dependencies and therefore a lot of time has been invested in this component to enable a clean logging which does not give out too much information at a low log level. Especially with these components which have many dependencies it is very helpful to have a logging which provides the needed information in an appropriate measure.

In the listing [7.2](#) an example log output from the polling container can be found.

```
| -----FINISH UPDATING INGRESS/EGRESS CACHE-----
| 2020-12-01 09:31:29,091 |      INFO |
|
| -----BEGIN SENDING CHANGES TO BACKEND-----
| 2020-12-01 09:31:29,096 |      INFO | No Changes exist - not necessary to send
|      something to the Backend
| 2020-12-01 09:31:29,096 |      INFO |
|
| -----FINISH SENDING CHANGES TO BACKEND-----
| 2020-12-01 09:31:29,097 |      INFO |
|
| -----FINISH UPDATING CACHE-----
```

Listing 7.2: Polling Container Example Log Output

## 7.15 Testing

Two types of tests were performed on the application. On the one hand, unit tests were carried out in which the modules of the same name were used consistently. A total of 150 unit tests were written, all of them without errors. All tests are automated in the pipeline (see [Continuous Integration Stages](#)).

Name	Stmts	Miss	Cover
api/__init__.py	0	0	100%
api/admin.py	1	1	0%
api/filters.py	9	0	100%
api/migrations/0001_initial.py	7	0	100%
api/migrations/__init__.py	0	0	100%
api/mocks/dbdataprovidermock.py	82	5	94%
api/models.py	78	77	1%
api/serializers.py	290	18	94%
api/services/dataprovider/__init__.py	1	0	100%
api/services/dataprovider/dbdataprovider.py	170	12	93%
api/services/dataprovider/idataprovider.py	38	12	68%
api/urls.py	23	0	100%
api/views.py	94	1	99%
calculation/__init__.py	0	0	100%

calculation/exceptions.py	6	0	100%
calculation/iservicechaincalculation.py	5	1	80%
calculation/servicechaincalculation.py	107	14	87%
graph/__init__.py	0	0	100%
graph/igraph.py	26	8	69%
graph/mocks/serchiographmock.py	51	4	92%
graph/serchiograph.py	102	2	98%
serchio/__init__.py	0	0	100%
serchio/settings.py	29	29	0%
serchio/urls.py	4	0	100%
updatetopology/__init__.py	0	0	100%
updatetopology/consumers.py	27	4	85%
updatetopology/migrations/__init__.py	0	0	100%
updatetopology/mocks/updatedatabaselistenermock.py	11	0	100%
updatetopology/mocks/updateserchiographlistenermock.py	11	1	91%
updatetopology/services/listeners/__init__.py	0	0	100%
updatetopology/services/listeners/iupdatelistener.py	5	1	80%
updatetopology/services/listeners/updatedatabaselistener.py	41	0	100%
updatetopology/services/listeners/updateserchiographlistener.py	35	0	100%
updatetopology/services/managers/__init__.py	0	0	100%
updatetopology/services/managers/iupdatemanager.py	5	1	80%
updatetopology/services/managers/serchioupdatemanager.py	11	0	100%
-----			
TOTAL	1269	191	85%
-----			
Ran 95 tests in 5.262s			
OK			

Listing 7.3: Backend Testreport Output from Pipeline

Beside the unit tests also system tests have been done. The protocols can be found in the appendix (see [Test Protocols](#)).

Usability tests were not done because the frontend was not part of this work.

**Part III**  
**Appendix**

Appendix A

---

**Task Description**

---

## Description of the thesis: SR-App / Service chaining path calculation

A network operator may want the traffic from A to B to always get a certain "treatment".

- It may be required that all the traffic should be filtered by a FW or inspected by an IDS/IPS in order to stop and to detect cyberattacks.
- It may be required that all the traffic should be mirrored and should go through a network switch for port mirroring because all the data going from A to B should be copied for compliance reasons.

With a stateless service program traffic-engineering policy, it is possible to enforce a path through a service in an easy way. A service can reside anywhere in the SR domain, that is to say for example close to the access or within a DC. The same services are spread all over the network.

The best path to a service can be the closest from a geographical point of view, but not necessarily. It depends on the intent that is expressed for the path. There are three possible intents:

- Shortest metric with or without additional constraints
- Shortest TE metric with or without additional constraints
- Shortest delay with or without additional constraints

The additional constraints can be the exclusion of some links or/and the disjointness with another SR-TE Path.

The calculation of the path will find to best path from A to B with the existing constraint and that transit over a FW. It does not matter which FW the packets are going through as long as the traffic is transiting through one of them.

(Example: Find to best path from A to B with the lowest delay and that transit over a FW)

The result of the best path calculation is a Segment-List that is going to be sent via an API call to the SR-TE PCE or directly to the head-end router to be used for the forwarding of packets.

The following use cases should be addressed by the application:

Use Case	optional/mandatory
UC1: Viewing the topology The User can overview the topology in the form of graphs and vertices.	mandatory
UC2: Service chain The user gets the path with the smallest metrics of the selected metric type, which flows from source to destination and includes the instances of the selected services.	mandatory
UC3: Data definition The data should be presented to the frontend in a specific format agreed with the developer of the frontend (INS)	mandatory
UC4: Service Discovery The user observes the different service types and nodes to get an overview of which various services are present and where they are located in the network.	mandatory
UC5: Path calculation The user gets the most appropriate path after selecting the service chain.	mandatory
UC6: Path approval The path is re-calculated path after a topology change to verify if the optimal path has changed. The user can approve or reject a the new path.	optional
UC7: Login A non-authenticated user has access to some functions in the application and there is a log in to use some application's additional functionalities.	optional
UC4: application administrator role The application administrator role can create users and assign roles.	optional

Rapperswil, 16.12.2020

Prof. Laurent Metzger



Figure A.1: Task Description

Appendix B

---

# Class Diagrams

---

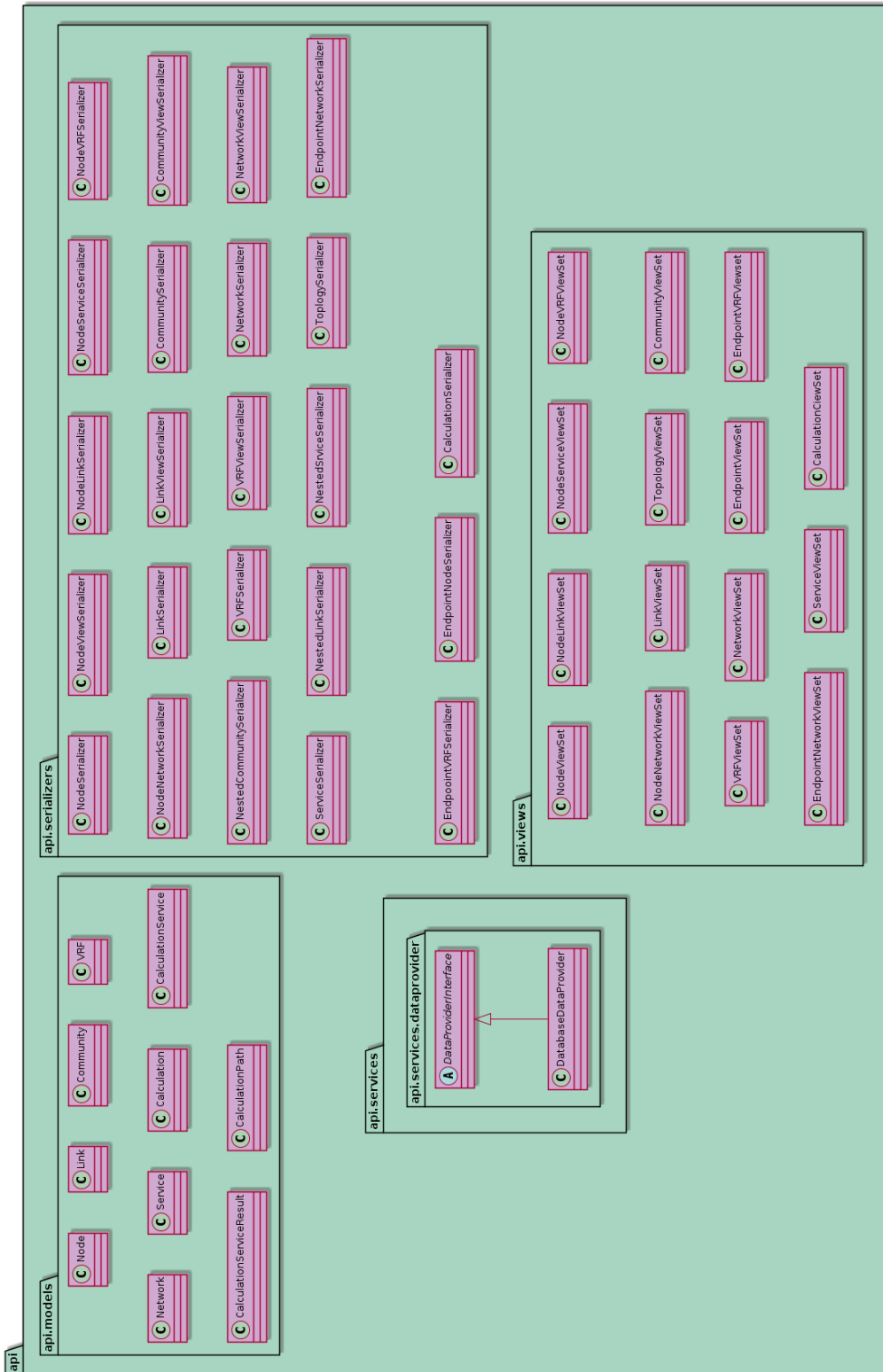


Figure B.1: Class Diagram Backend api

## Test Protocols

---

### C.1 System Tests

The System Tests were made with the popular API tool Postman (<https://www.postman.com/>). With the help of this tool, several requests were made to test the whole system's behavior. Besides, manual testing has also been done, mainly to test the action between the graph database Arango, the polling service, and the backend application. An extract of the system tests can be found in figure C.1.



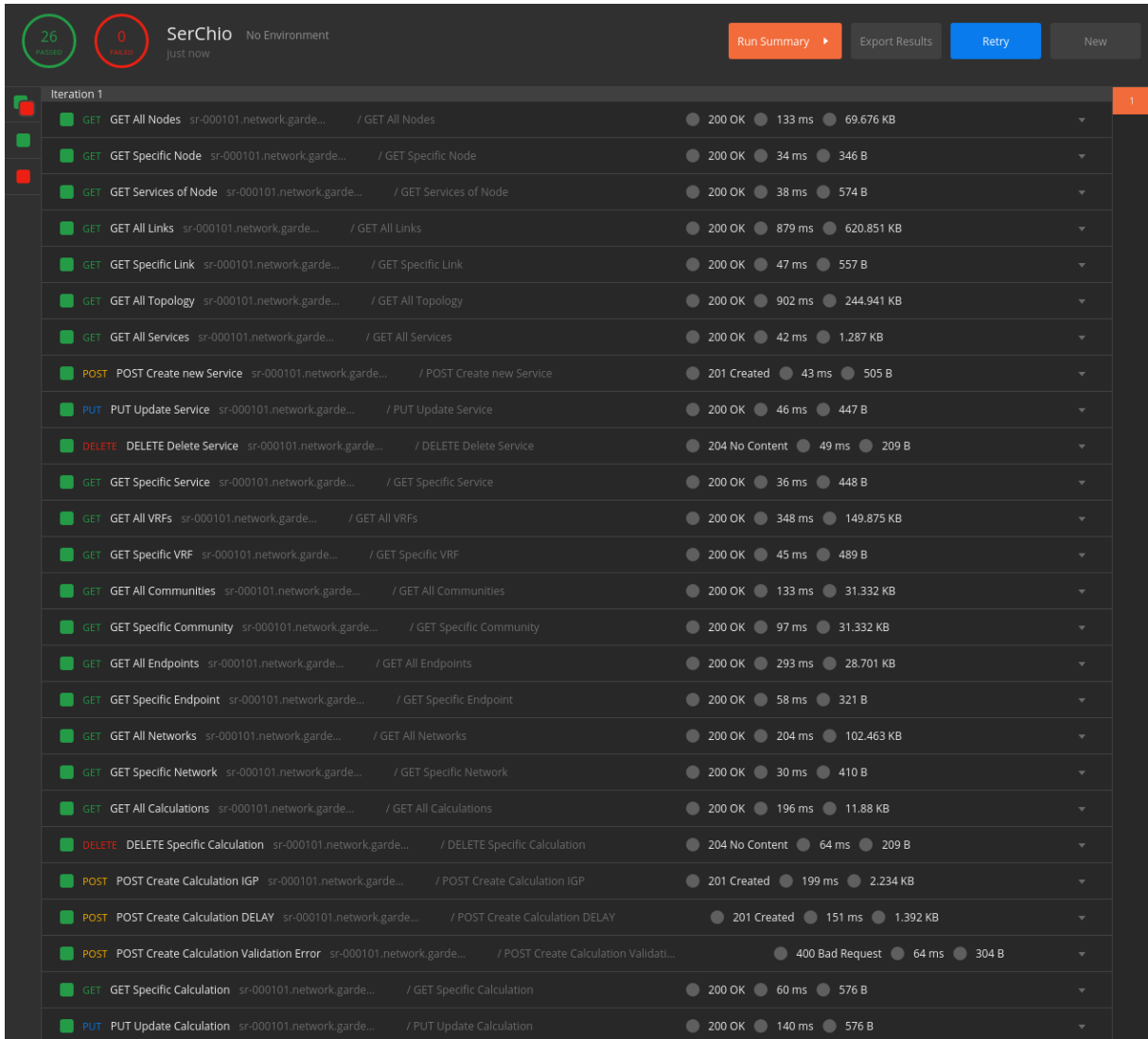


Figure C.1: System Tests Postman

## Appendix D

# Metrics

In the process of this work, a total of 5'5025 lines of code were written, which are distributed over two different components. The backend component is represented with 3'579 lines of code and the polling service with 1'446.

	Lines of Code	Bugs	Vulnerabilities	Code Smells	Security Hotspots	Coverage	Duplications
SerChio							
backend	3,579	0	0	0	0	91.9%	0.0%
polling	1,446	0	0	0	0	91.1%	0.0%

Figure D.1: Sonar Lines of Code

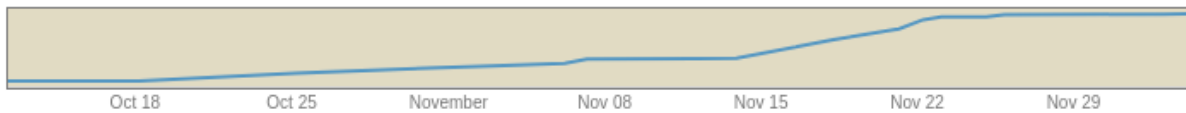


Figure D.2: Sonar Lines of Code Timeline

All metrics were created using Sonarqube and can be viewed in figure [D.3](#). Best values were achieved in all categories and thus also passed the quality gate of Sonarqube.

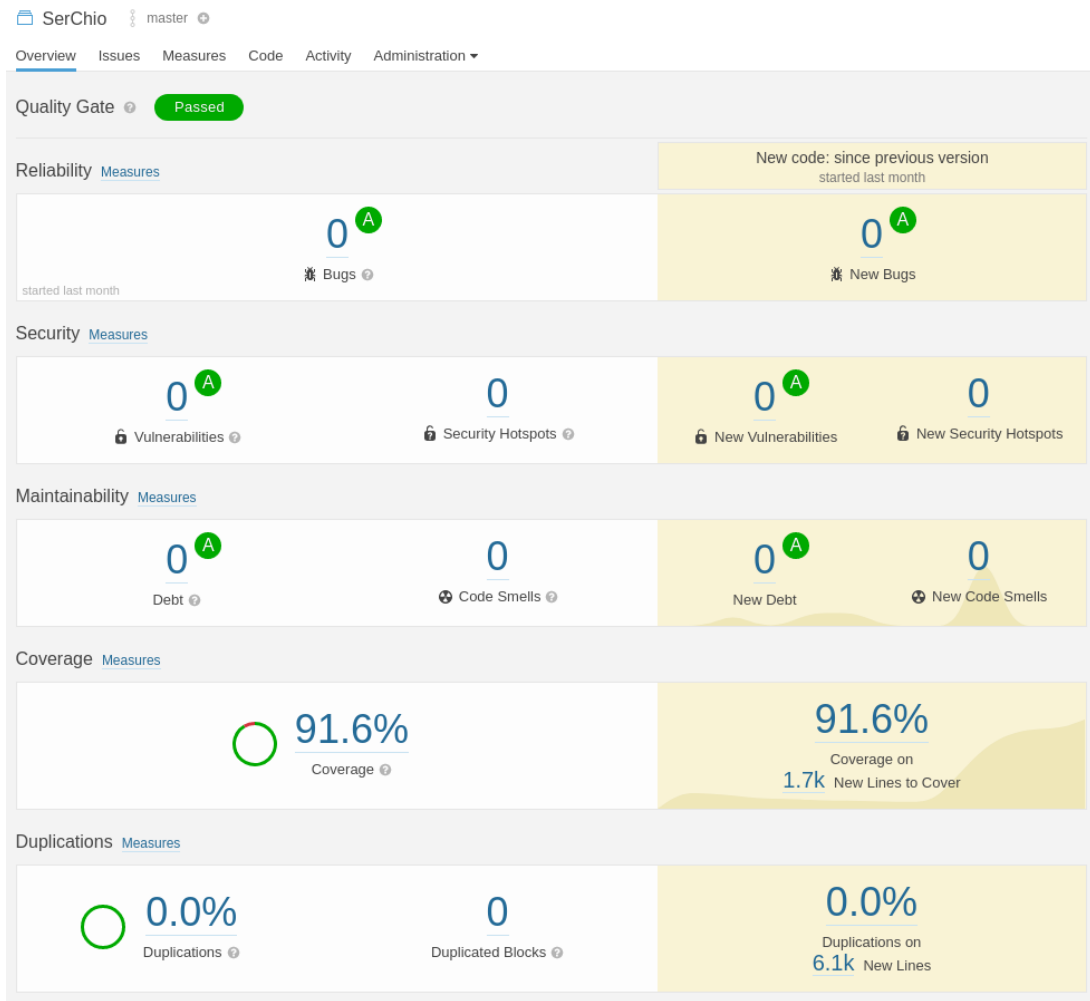


Figure D.3: Sonar Metrics