Bachelor Thesis

Segment Routing Service Programming

Departement of Computer Science

OST - University of Applied Sciences

Campus Rapperswil-Jona

Spring Term 2021

Authors	Julian Klaiber Severin Dellsperger

Advisor	Prof. Laurent Metzger
Project Partner	Cisco EMEA *represented by* Francois Clad
External Co-Examiner	Laurent Billas
Internal Co-Examiner	Prof. Mirko Stocker

Abstract

In the last few years, the IT network domain has changed fundamentally. New approaches and technologies were introduced, which has changed and is changing the future of this area radically. The results are modern and dynamic networks that close the gap between networks, applications, and end-users. It permits creating applications that work closely with the underlying network and create a network that fulfills customer needs entirely. Network services like firewall systems or intrusion detection/prevention systems have become indispensable and are firmly anchored in computer networks. Nowadays, these services are not to assume away yet have also a massive disadvantage: they are consumed in a static manner. Service Programming is one of the outcomes in future networks and solves the problem of static service consumption. It allows configuring the network dynamically so that network services can process customer traffic according to their necessities. Following network services can be placed universally in the network - the service programming application will find the best services according to the traffic characteristics. Hence, networks with integrated service programming become more intelligent, economic and are prepared for future needs.

This thesis is a follow-up thesis from the Service Chaining Path Calculation thesis written in the autumn term of 2020, which introduced a way to calculate service chains in a Segment Routing network. This bachelor thesis aimed to find a solution that can help program so-called steering policies in the network to steer the traffic according to the needs of the customer networks over the most suitable services. In order to achieve this goal, the network protocol Segment Routing with the IPv6 data plane (SRv6) was used. The goal was to deliver an application that can calculate and program the best suitable path according to specified parameters from the customer. The application should react dynamically to changes in the connected network and deliver consistently the best policy that fits the altered topology. As a consequence, the user can always rely on the data on which he is working. Hence, the application always had to know the present network topology and has to be informed about network changes. An external system is used to get the actual topology data; the system aggregates and processes all the topology information, which can be used in so-called Segment Routing applications.

During the bachelor thesis, a complete Service Programming application could be developed. The application is developed entirely in a cloud-native way in order to be highly scalable and available. The application consists of different services, which communicate with each other over a dedicated messaging system. A polling service handles all the update messages from the topology and informs the backend service automatically about changes. The backend service uses the topology data to perform path calculations, deploy policies to the underlying network, and deliver the topology data so that the frontend can easily visualize the network and paths. The frontend was developed in collaboration with the Institute for Networked Solutions and provides the customer an easy-to-use way to create and manage policies over the backend.

Management Summary

Initial Situation

In modern networks, network services are required to secure and enhance the data flow between customer networks. However, the service handling, especially when several services are included, has ever been driven by static manner. As a consequence, many services had to be installed and maintained. Additionally, a service failure often implied also an outage in the delivery of data. Thus, the effort and costs were extremely high, whereas the output was not intuitive. This situation has changed now with the Segment Routing Service Programming application. The application serves as an easy-to-use interface, which allows an operator to create network rules naturally. These rules ensure that the traffic is steered through the most suitable services in the network. So, it is straightforward to steer traffic through one or several services according to the customer's desire. A significant advantage of this solution is that the services can be distributed in the network. The application will automatically find the best services and influence the traffic to flow through them. Furthermore, it reacts to network changes and adapts to these quickly. Outages caused by service failures are thus a thing of the past. As a result, the application delivers a dynamic, enhanced, and cost-efficient solution that is future-proof and is changing the service consumption of the future.

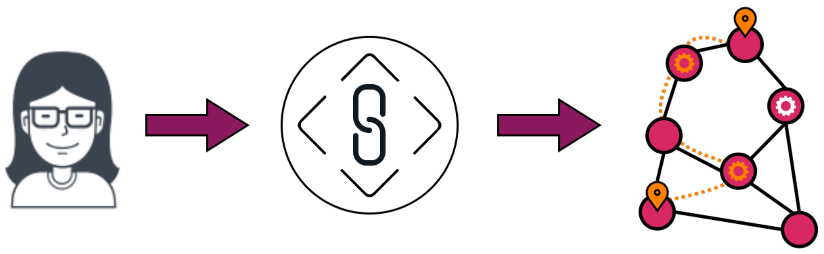


Figure 1: Our Vision

Procedure and Technology

In order to record all requirements in detail, all use cases were discussed with the project partner. So, it was ensured that the features could be prioritized for the development phase. In addition, various non-functional requirements are met so that the application would also meet the high demands in production.

The non-functional requirements then led to an intensive architecture and prototype phase, in which the basic architecture was thoroughly tested in a minimal application. That ensured from the very beginning, that the design delivers what it promised and will meet the non-functional requirements.

The most significant phase was the development phase, in which all features were developed and tested. In this phase, much research had to be done to determine, how exactly to configure the network, so that the traffic is routed according to the defined parameters. In this phase, two services were developed. One is the polling service, which was entirely developed in Python and is responsible for the whole update strategy of the application. The other is the backend service, which was also built in Python using the Django framework. The backend has several functionalities: Data provision, path calculation and policy management.

Results

In this bachelor thesis, a comprehensive Segment Routing Service Programming application could be developed.

The application allows the customer to manage the different policies from a central location. The granular permission structure allows the customer to control: who is allowed to perform and what activities. Constant manual adjustment of the various policies belongs to the past, thanks to automatic recalculation and redeployment. With the ability to dynamically route traffic through the various services, services such as a firewall or an intrusion detection/prevention system can now be utilized better and centrally deployed in the network.

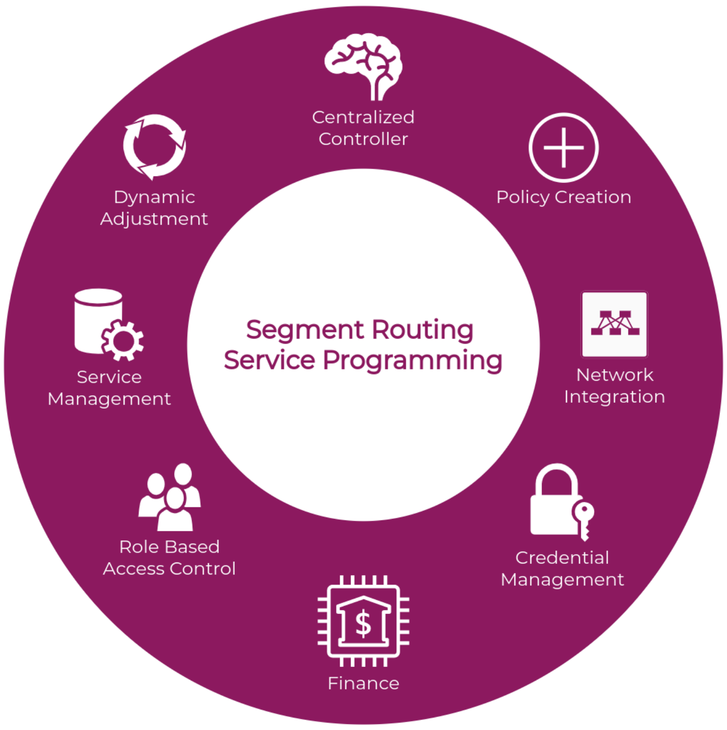


Figure 2: Customer Benefits

Through the standardized Application Programming Interface (API), all application functions can be controlled; a frontend, like the one developed by the Institute for Networked Solutions, can display the complete topology and inform the user dynamically about updates. The application can be seamlessly integrated into a cloud environment due to its cloud-native structure and can scale with the size of the network without any problems.

Outlook

During this thesis, a stable Service Programming application could be developed. In the future, the focus might be on adding more metrics and advanced options. A particular emphasis should be placed on the interaction with existing network automation solutions to integrate the application perfectly into a customer network. The goal in the future should be to improve the application and bring it to the next level. The explicit goal should be, that the application can prove itself in a provider network.

Acknowledgments

We want to express our gratitude to the following people who have accompanied us during this bachelor thesis:

Laurent Metzger We want to thank Laurent very much; as our supervising professor, he always believed in us and helped us better understand the domain of Segment Routing. Which further strengthened us to consolidate and educate ourselves in this field.

Francois Clad as our project partner and expert in Segment Routing deserves special thanks for always taking time for questions and helping us with our concerns.

Ahmed Abdelsalam we would like to thank him very much for his support and swift help with the Segment Routing Services. Without his preliminary work and the quick provision of a new Linux kernel, the services would not be operational.

Bruce McDougall we would like to thank him for his support with questions regarding the external software system Jalapeño. Despite the time shift, he always found time for us.

Judith Manhart we would like to thank her very much for correcting our work and checking the English grammar.

Sharon Moll as our study colleague, gets a special thanks again for proofreading our work and his enormous knowledge about various software architectures and distributed systems.

Contents

Glossary and Abbreviations

Ansible IT Automation Framework. 76

API Application Programming Interface. ix, 14, 15, 18, 29, 34–38, *see* Application Programming Interface

Application Programming Interface Programming interface into a software. Can be used and extended by other software components. 14

ArangoDB Open-source graph database. 88

ASGI Asynchronous Server Gateway Interface. ix, *see* Asynchronous Server Gateway Interface

BGP Border Gateway Protocol. ix, 8, 18, *see* Border Gateway Protocol

Border Gateway Protocol Exterior Gateway Protocol (EGP), which exchanges network information between different autonomous system (mostly on the internet). 8

CD Continuous Delivery. ix, *see* Continuous Delivery

Celery Simple, high-available, fast and flexible Python-based task queue. 63, 78, 84

channels_rabbitmq Django Channels with RabbitMQ backing store. 74

CI Continuous Integration. ix, *see* Continuous Integration

Cisco DNA Center Software-defined Access Solution by Cisco. 30

Cisco IOS-XR Service Layer API IOS-XR protocol stack API which can be used over GRPC. 38

Cisco Network Service Orchestrator Network automation and orchestration tool. 38

DDoS Distributed Denial-of-Service. ix, *see* Distributed Denial-of-Service

Dijkstra Mathematical algorithm which solves the problem of the shortest path for a given start-point. 20–23

Django REST Framework Django extension, which allows the development of web APIs. 69

django-redis Jazzband Django Redis. ix, 72, *see* Jazzband Django Redis

DRF Django REST Framework. ix, 69, *see* Django REST Framework

ECMP Equal-Cost Multi-Path Routing. ix, 7, 20, 30, 48, *see* Equal-Cost Multi-Path Routing

Egress Router The router where the packet exits the Segment Routing Domain. 5

Elasticsearch, Logstash and Kibana Stack out of different softwares, that enable log management and monitoring. 67

ELK Elasticsearch, Logstash and Kibana. x, 67, *see* Elasticsearch, Logstash and Kibana

Equal-Cost Multi-Path Routing Strategy which enables forwarding packets on several best paths with equal routing costs.. 7

Firewall Security system designed to prevent unauthorized access to network resources. 56

Floyd-Warshall All-pair shortest path in directed graph with weighted edges. 22

FW Firewall. x, 56, *see* Firewall

gRPC Modern, high-performance, scalable remote procedure call framework. 38, 75

Helm Package manager for Kubernetes, which allows to find, share and use software built for Kubernetes.. 93

HTTP Hypertext Transfer Protocol. x, 60, *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. 60

IDS Intrusion Detection System. x, 10, 56, *see* Intrusion Detection System

IGP Interior Gateway Protocol. x, 14, 32, *see* Interior Gateway Protocol

Ingress Router The router where the packet enters the Segment Routing Domain and the Segments are added . 5, 6

INS Institute for Networked Solutions. x, 12, 35, 63, *see* Institute for Networked Solutions

Institute for Networked Solutions Computer science institute at the University of Applied Sciences of Eastern Switzerland. 12

Interior Gateway Protocol A type of routing protocol used to exchange network information within an Autonomous System (AS). 14

Internet Protocol Version 6 Most recent version of the Internet Protocol (IP) which introduces new techniques and capabilities. 3

Intrusion Detection System Monitoring system, that detects suspicious activities and creates warning messages. 10

Intrusion Protection System Monitoring system, that detects suspicious activities and executes protection mechanisms against these. 10

IPS Intrusion Protection System. x, 10, *see* Intrusion Protection System

IPv6 Internet Protocol Version 6. x, 3, 32, *see* Internet Protocol Version 6

IS-IS Intermediate System to Intermediate System. x, *see* Intermediate System to Intermediate System

Jalapeño Software system developed by Cisco, which aggregates and processes the data of a segment routing network. 18, 32, 36, 38, 60, 88, 101

Jazzband Django Redis Full featured Redis cache and session backend for Django. 72

Johnson All-pair shortest path in directed graph with weighted edges. 22

JSON JavaScript Object Notation. xi, *see* JavaScript Object Notation

JSON Web Token Open industrie standard method for representing access tokens.. 15

JWT JSON Web Token. xi, 15, 36, 86, 87, 99, *see* JSON Web Token

Kafka Distributed event streaming platform. 38

Keda Kubernetes-based event driven autoscaler. 94

Kubernetes Open-source container orchestration system, which allows easy scaling and management of the deployed applications. 17, 92

LAN Local Area Network. xi, 59, *see* Local Area Network

Linux iptables Firewall included in the Linux operating system. 10

Local Area Network A network primarly spanned over a small geographic are. 59

Memcached Open-source in-memory key-value cache system. 72

micro segment A ultra-scale compressed Segment format with minimum MTU overhead. 32

Minimum Viable Product Version of a product which includes the customer's minimum requirements so that it can be used in production. 34

MPLS Multi Protocol Label Switching. xi, 3, *see* Multi Protocol Label Switching

Multi Protocol Label Switching Routing protocol which routes packets based on path labels and not network addresses. 3

MVP Minimum Viable Product. xi, 34, 42, *see* Minimum Viable Product

NAPALM Network Automation and Programmability Abstraction Layer with Multivendor support. xi, 76, *see* Network Automation and Programmability Abstraction Layer with Multivendor support

NETCONF Protocol for writing and receiving network device configuration.. 75

Netmiko Python library for CLI connections to network devices. 76

Network Automation and Programmability Abstraction Layer with Multivendor support Easy-to-use Python library that let the user interact with network devices. 76

Nornir Python automation framework. 76, 85

NSO Cisco Network Service Orchestrator. xi, 38, *see* Cisco Network Service Orchestrator

ORM Object-relational mapping. xi, *see* Object-relational mapping

OSPF Open Shortest Path First. xi, *see* Open Shortest Path First

Paramiko Simple and easy-to-use Python library for the SSH protoocl. 76

PE Provider Edge. xi, 22, 23, 48, 59, *see* Provider Edge

PEP8 Document providing guidelines and best practices for the Python programming language. 112

PmQm Project and Quality Management. xii, *see* Project and Quality Management

Provider Edge Interface between end customer and service provider. 22

RabbitMQ High-available and high-scale message queue system. 74

RBAC Role-Based Access Control. xii, 86, *see* Role-Based Access Control

Redis Object Mapper Python data modeling with support for Redis cache. 72

REpresentational State Transfer Standardized architecture for providing data on the web. 14

Request for Comments Standard document from the Internet Engineering Task Force (IETF). 3

REST REpresentational State Transfer. xii, 14, 15, 18, 29, *see* REpresentational State Transfer

RFC Request for Comments. xii, 3, *see* Request for Comments

Role-Based Access Control Approach for restricting system access to authorized users.. 86

rom Redis Object Mapper. xii, 72, *see* Redis Object Mapper

SaltStack Automation and Configuration Management Framework. 76

SDN Software Defined Networking. xii, *see* Software Defined Networking

Segment Identifier A identifier for a Segment in Segment Routing. 3

Segment Routing Source-based routing protocol that enables new approaches and optimizes traffic engineering, network protection and more. 2

Segment Routing aware services Services which can handle Segment Routing. 10

Segment Routing over IPv6 dataplane Source-based routing protocol over the IPv6 dataplane, that enables new approaches and optimizes traffic engineering, network protection and more. 3

Segment Routing over the Multi Protocol Label Switching (MPLS) dataplane Source-based routing protocol over the MPLS dataplane, that enables new approaches and optimizes traffic engineering, network protection and more. 3

Segment Routing Service Programming The name of this Bachelor thesis. 10

Segment Routing Traffic Engineering Influence and steer traffic according to own desire in a Segment Routing Domain.. 48

SerChio Service Chaining Path Calculation. xii, 2, 9, 10, 12, 14, 17, 18, 20, 57, 68, 70–72, 88–90, *see* Service Chaining Path Calculation

SerPro Segment Routing Service Programming. xii, 10, 16, 18, 19, 32, 38–40, 57, 60, 63, *see* Segment Routing Service Programming

Service Chain Concatenation of different services. 2

Service Chaining Path Calculation The name of the preliminary thesis. 2

SID Segment Identifier. xii, 3, *see* Segment Identifier

Simple Network Management Protocol Protocol which allows devices to communicate even when they are running on different hardware or software. 39

SNMP Simple Network Management Protocol. xii, 39, *see* Simple Network Management Protocol

SQL Structured Query Language. xiii, *see* Structured Query Language

SR Segment Routing. xiii, 2, 3, *see* Segment Routing

SR-aware services Segment Routing aware services. xiii, 10, 11, 13, *see* Segment Routing aware services

SR-MPLS Segment Routing over the Multi Protocol Label Switching (MPLS) dataplane. xiii, 3, *see* Segment Routing over the Multi Protocol Label Switching (MPLS) dataplane

SR-TE Segment Routing Traffic Engineering. xiii, 48, *see* Segment Routing Traffic Engineering

SRv6 Segment Routing over IPv6 dataplane. xiii, 3, 10, 14, 18, 28, 32, *see* Segment Routing over IPv6 dataplane

Standard Output Default file descriptor where a process can write output to. 122

stdout Standard Output. xiii, 122, *see* Standard Output

streaming telemetry gRPC based push mechanism to send data from network devices to a management system. 18

TE Traffic Engineering. xiii, 36, 49, *see* Traffic Engineering

TI-LFA Topology Independent Loop-Free Alternate. xiii, *see* Topology Independent Loop-Free Alternate

Traffic Engineering Technique used to control and steer traffic to optimize the network utilization and performance. 36

uSID micro segment. xiii, 32, *see* micro segment

Virtual Routing and Forwarding Virtual router which lives on a physical. Enables the coexistence of e.g. different customers on the same router. 59

VM Virtual Machine. xiii, *see* Virtual Machine

VNF Virtual Network Function. xiii, *see* Virtual Network Function

VRF Virtual Routing and Forwarding. xiii, 59, *see* Virtual Routing and Forwarding

WebSocket Communication protocl which uses the HTTP protocol to provide full-duplex channels for communication. 29, 36, 73

YANG Yet Another Next Generation. xiii, 75, *see* Yet Another Next Generation

Yet Another Next Generation Data modeling language for network device configuration. 75

Bibliography

[1] MIT OpenCourseWare. "6.046J / 18.410J Design and Analysis of Algorithms". In: *Lecture 11: All-Pairs Shortest Paths* (2015). URL: https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2015/lecture-notes/MIT6_046JS15_lec11.pdf (visited on 06/05/2021).

[2] Clarence Filsfils et al. *Segment Routing. Part 2.* Independently published, 2019.

[3] Severin Dellsperger and Julian Klaiber. "Service Chaining Path Calculation". Project Thesis. Eastern University of Applied Sciences Switzerland, 2020. URL: https://eprints.ost.ch/id/eprint/912/.

[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 (visited on 09/23/2020).

[6] *boost C++ library implementation of shortest path algorithm.* URL: https://www.boost.org/doc/libs/1_57_0/libs/graph/doc/dijkstra_shortest_paths.html (visited on 04/01/2021).

[7] Simon Brown. *C4 Model.* URL: https://c4model.com/ (visited on 11/03/2020).

[8] *Django's cache framework.* URL: https://docs.djangoproject.com/en/3.2/topics/cache/ (visited on 06/03/2021).

[9] Cloud Computing Foundation. *Cloud Native.* URL: https://cncf.io/ (visited on 10/11/2020).

[10] *graph-tool implementation of shortest path algorithm.* URL: https://graph-tool.skewed.de/static/doc/topology.html#graph_tool.topology.shortest_path (visited on 04/01/2021).

[11] *IETF Draft: Network Programming extension: SRv6 uSID instruction draft-filsfils-spring-net-pgm-extension-srv6-usid-10.* URL: https://datatracker.ietf.org/doc/html/draft-filsfils-spring-net-pgm-extension-srv6-usid-10 (visited on 06/10/2021).

[12] *IETF Draft: Service Programming with Segment Routing draft-ietf-spring-sr-service-programming-04.* URL: https://datatracker.ietf.org/doc/html/draft-ietf-spring-sr-service-programming-04 (visited on 06/10/2021).

[13] *IETF RFC4360 - BGP Extended Communities Attribute.* URL: https://datatracker.ietf.org/doc/html/rfc4360 (visited on 06/11/2021).

[14] *IETF RFC4741 - Network Configuration Protocol (NETCONF).* URL: https://datatracker.ietf.org/doc/html/rfc6241 (visited on 06/03/2021).

[15] *IETF RFC5512 - The BGP Encapsulation Subsequent Address Family Identifier (SAFI) and the BGP Tunnel Encapsulation Attribute.* URL: `https://datatracker.ietf.org/doc/html/rfc5512` (visited on 06/11/2021).

[16] *IETF RFC8200 - Internet Protocol, Version 6 (IPv6) Specification.* URL: `https://datatracker.ietf.org/doc/html/rfc8200` (visited on 06/10/2021).

[17] *IETF RFC8402 - Segment Routing Architecture.* URL: `https://datatracker.ietf.org/doc/html/rfc8402` (visited on 06/10/2021).

[18] *IETF RFC8754 - IPv6 Segment Routing Header (SRH).* URL: `https://datatracker.ietf.org/doc/html/rfc8754` (visited on 06/10/2021).

[19] *IETF RFC8986 - Segment Routing over IPv6 (SRv6) Network Programming.* URL: `https://datatracker.ietf.org/doc/html/rfc8986` (visited on 06/10/2021).

[20] Craig Larman. *Applying Evolutionary Use Cases.* URL: `https://www.craiglarman.com/wiki/downloads/applying_uml/larman-ch6-applying-evolutionary-use-cases.pdf` (visited on 09/23/2020).

[21] *Pickling Graph in graph-tool.* URL: `https://graph-tool.skewed.de/static/doc/quickstart.html?highlight=pickle#graph-i-o` (visited on 02/28/2021).

[22] *Programmability Configuration Guide for Cisco ASR 9000 Series Routers, IOS XR Release 6.4.x.* URL: `https://www.cisco.com/c/en/us/td/docs/routers/asr9000/software/asr9k-r6-4/programmability/configuration/guide/b-programmability-cg-asr9000-64x/b-programmability-cg-asr9000-64x_chapter_010.html` (visited on 03/17/2021).

[23] *Python static typing.* URL: `https://docs.python.org/3/library/typing.html` (visited on 03/06/2021).

[24] *Redis Publish/Subscribe.* URL: `https://redis.io/topics/pubsub` (visited on 06/13/2021).

[25] *Redis Sentinel Documentation.* URL: `https://redis.io/topics/sentinel` (visited on 03/12/2021).

[26] *Segment Routing Policy Architecture draft-ietf-spring-segment-routing-policy-13.* URL: `https://datatracker.ietf.org/doc/html/draft-ietf-spring-segment-routing-policy-13` (visited on 06/10/2021).

[27] *Serializing Python Objects with pickle.* URL: `https://docs.python.org/3/library/pickle.html#module-pickle` (visited on 03/28/2021).

List of Figures

List of Tables

Part I

Technical Report

Chapter 1

Introduction

1.1 General

This chapter contains an introduction to this thesis. It shows the essential techniques, concepts, and terminology that are valuable to understand according to comprehend the content of this thesis. The different elements are described as deep as necessary to understand the mentioned concepts in this thesis. The appropriate sources, where more detailed knowledge can be gathered, are linked respectively.

The different methods, especially the Segment Routing (SR) technology with all its advantages and the term Service Chain were introduced in the preliminary thesis Service Chaining Path Calculation (SerChio)[3]. It is assumed that the reader has a basic knowledge of the mentioned technologies and methods. These points are not amplified, and it is recommended to look at the previous thesis if the reader has no fundamental knowledge of these topics.

1.1.1 Thesis Structure

The thesis is structured into two parts. More detailed information can be found below.

Technical Report

The first part, the Technical Report, is composed of three chapters. The first chapter is the Introduction. It introduces the reader to the whole topic and ensures that the basic understanding is given to follow the topic. Therefore, the most important terms and techniques are described. The next chapter is the Results chapter. This chapter contains more detailed information about the Achievements, which enlists which results are accomplished during this thesis. Furthermore, it includes the Implementation section. This part contains a description of how the different approaches were realized. Thereby it concentrates on particular solutions which are noteworthy. The last chapter is called Conclusion. Firstly, it gives the reader a Retrospective of the thesis, which enlists a discussion about each use case. Secondly, it includes an Outlook on how the thesis could be developed further or improved. It has to be mentioned that all three chapters are directed to engineers in the field of computer science. Therefore, a basic level is required to understand these mentioned concepts.

Project Documentation

The second part, the Project Documentation, includes the necessary information about how the final result was achieved. Thereby, it concentrates not only on the technical but also on the non-technical methods. The first chapter, the Requirement Specification, includes the documentation about the different requirements collected at the start of this project. This chapter is divided into functional requirements – use cases – and non-functional requirements. The Domain Analysis, the subsequent chapter, introduces the reader to the domain and shows the domain-specific elements. Following, the Architecture and Design Specifications show the different characteristics of the application under development. It provides more information about the architecture, the used technologies and mentions design-specific thoughts and approaches. The seventh and last chapter includes information about the project management. Besides the scheduling, it included information about the milestones and iterations, risk management, and development characteristics and decisions.

1.2 Terms and Techniques

There is no doubt, that services are needed in today's networks. Nevertheless, the consumption of network services was not straightforward and often not optimized in the past. Hence, this should change in the future. The Segment Routing Service Programming should have a significant influence on that behavior. This section introduces the necessary terms and provides more detailed information about the used techniques. This section is based on the Segment Routing Architecture defined in RFC 8402[17] and furthermore on the following draft items and Request for Comments (RFC) documents:

1. IETF Draft: Service Programming with Segment Routing draft-ietf-spring-sr-service-programming-04 [12]
2. RFC 8986: Segment Routing over IPv6 (SRv6) Network Programming [19]
3. IETF Draft: Network Programming extension: SRv6 uSID instruction draft-filsfils-spring-net-pgm-extension-srv6-usid-10 [11]
4. IETF Draft Segment Routing Policy Architecture draft-ietf-spring-segment-routing-policy-13 [26]

1.2.1 Segment Routing over IPv6 (SRv6)

As mentioned in the preliminary thesis, SR can be used over the Multi Protocol Label Switching (MPLS) or the Internet Protocol Version 6 (IPv6) dataplane. If the IPv6 dataplane is used, one is generally speaking about Segment Routing over IPv6 dataplane (SRv6)

The general idea of Segment Routing over the Multi Protocol Label Switching (MPLS) dataplane (SR-MPLS) and SRv6 are the same. Nevertheless, SRv6 has some own characteristics, which are important to understand. These different concepts are outlined in the specific subsections below.

SRv6 SID Format

The first and probably most important point is that one Segment in SRv6 refers to one IPv6 address. The SRv6 Segment Identifier (SID) can be devided into three logical elements [19]:

Locator is used to route the packet to the appropriate node - this locator information has to be distributed in the network to ensure connectivity.

Function which should be executed on the appropriate node - the appropriate node can identify the behavior which is bound to the function (more information can be found here: `https://datatracker.ietf.org/doc/html/rfc8986#section-4`)

Arguments which give some additional information about the local behavior (function) which should be processed on the appropriate node

It is important to be mentioned, that the SRv6 Segment, like an IPv6 address, has a maximum length of 128 bit. The length of the locator can be chosen freely and there is no need for arguments. The structure of an SRv6 Segment can be seen in figure 1.1.

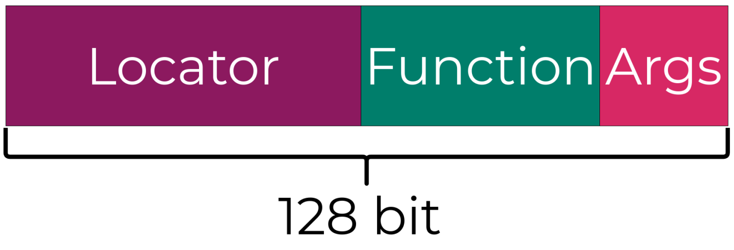


Figure 1.1: SRv6 Segment Structure

Segment Routing Header

In order to ensure that the IPv6 packets are steered through the desired path in a Segment Routing Domain, they have to be supplemented with the appropriate segments. The Segment List is added to the IPv6 header as soon as the packet enters the Segment Routing Domain. For this purpose, the so-called Segment Routing Header was invented. The Segment Routing Header, therefore, is built on the IPv6 Routing Extension Header [16, 18]. The structure of the Segment Routing Header can be observed at listing 1.1.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Next Header   |  Hdr Ext Len  | Routing Type  | Segments Left |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Last Entry   |     Flags     |              Tag              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
|            Segment List[0] (128-bit IPv6 address)             |
|                                                               |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
|                                                               |
                              ...
|                                                               |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
|            Segment List[n] (128-bit IPv6 address)             |
|                                                               |
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
//                                                             //
//         Optional Type Length Value objects (variable)       //
//                                                             //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Listing 1.1: Segment Routing Header

The delivery of a packet in a Segment Routing Domain can be simplified as follows: The header includes the different Segments which should be completed during the path traversal in the Segment List. The Segment, which should be executed first, corresponds to the last

Segment List entry. In addition, the header includes the `Segments Left` field, which contains the information, how many instructions still need to be executed. This field points to the last Segment List entry at the beginning. As soon as the first Segment has been completed, this value is decremented. Consequently, the field refers to the next Segment, which has to be processed further. Next, the Active Segment is copied into the destination address field in the IPv6 header. The packet is forwarded according to the destination address information to the appropriate node. If the packet had received an intermediate node, the packet is redirected according to its routing entries. As soon as the packet has reached the node, which is responsible for the Active Segment, the instruction can be finished. This whole process can now be repeated until no Segments are left. By these means, it can be ensured, that the packet takes the desired path, which is encoded in the Segment Routing Header.

An example of this process is visualized in figure 1.2. The particular Active Segment respective destination address of the packet is marked with the orange star. The `Segments Left` counter can be found on the top right corner. The process starts after the packet has entered the Segment Routing Domain. The Ingress Router assembles the packet with the relevant Segment List. In this example, the packet should be steered via *R2* to *R4* and afterwards via the shortest path to *R10*. Therefore, the first instruction is executed, and the packet is sent to *R2*. There the Segment is completed, and the Segments Left field can be decremented. So, the new Active Segment can be copied into the destination address field. Next, the packet can be sent to *R4*. This process is repeated until the packet has reached the Egress Router *R10* and the packet can be forwarded outside of the Segment Routing Domain.

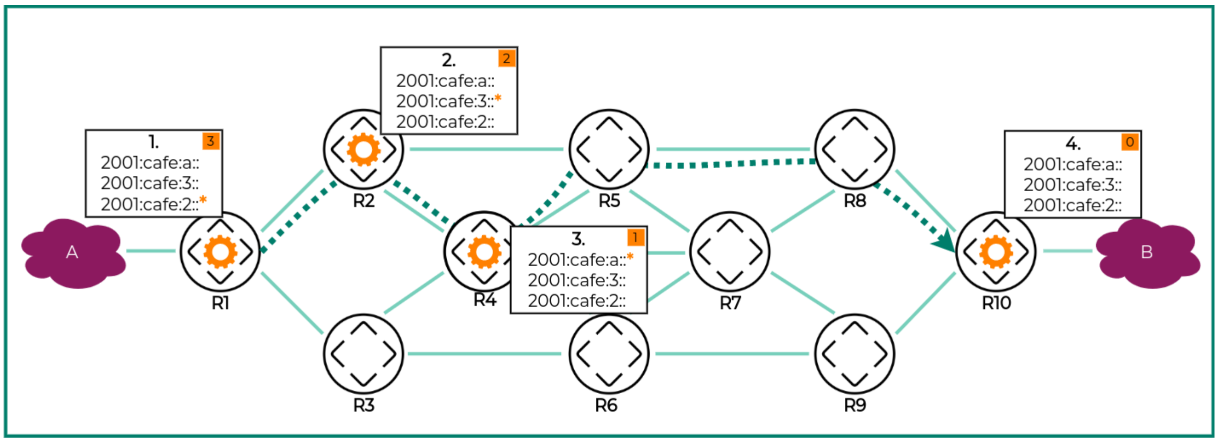


Figure 1.2: SRv6 Path Traversal

Micro SID

The introduction of the so-called SRv6 Micro SID, or short uSID, proposes a method to optimize the usage of the Segment Routing Header. Instead of including duplicated information into the Segment Routing Header multiple times, the duplicated information is only included once. This method can be used if different standard behaviors are used. [11]

The figure 1.3 shows a uSID format. If it is compared to the figure 1.2 it stands out, that the SRv6 uSID Block is only mentioned once and the three Segments are consilidate in one single SRv6 uSID Carrier. The functionality is similar to the standard: The foremost uSID represents the Active Segment. As soon as this instruction is completed, the uSID is removed from the SRv6 uSID Carrier. So, the next uSID becomes active. This process is repeated until no uSIDs are left. At this point, the packet has reached the destination router.

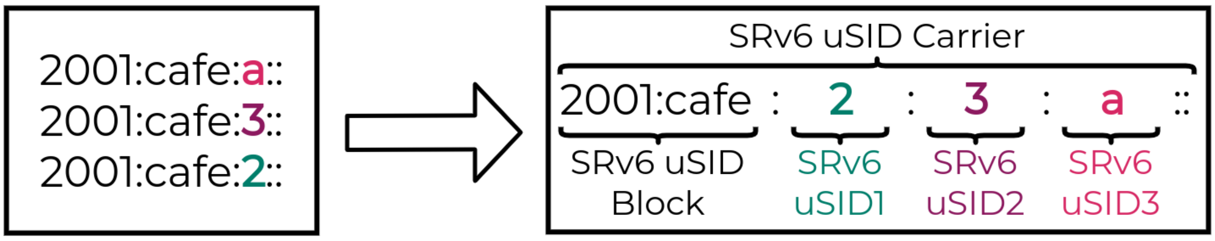


Figure 1.3: SRv6 Micro SID (uSID)

1.2.2 Segment Routing Policy

SR Policy Definition

An SR Policy is a framework that enables the instantiation of an ordered list of segments on a node for implementing a source routing policy for the steering of traffic for a specific purpose (e.g. for a specific SLA) from that node. [26]

As already mentioned, Segment Routing enables a new level of Traffic Engineering capabilities. It allows steering traffic according to the operator's desire and the packet characteristics through the network. These operations are possible, because the state is always available in the packet through the managed Segment List. In order for the packet to be associated with the correct Segments, and therefore the correct path traversal is ensured, the concept of SR Policies was introduced. An SR Policy can be seen as a configuration that ensures that packets with matching options are assigned with the appropriate Segments.

A SR Policy is identified by tree elements [2, 26]:

- **headend** is the IPv4 or IPv6 address of the node where the policy is instantiated (the Ingress Router)
- **color** is a numeric value to differentiate between different policies with same endpoint
- **endpoint** is the IPv4 or IPv6 address of the destination of the policy

It is possible to treat individual packets according to their characteristics differently with this technique. Each policy is therefore unique by the introduced tuple `(headend, color, endpoint)`. The figure 1.4 shows two different policies with different meanings. Both policies are introduced on the headend *R1*. The green policy should steer packets according to the minimal cost to the destination node *R10*. This policy is uniquely identifiable with the tuple `(R1, 10, R10)`, whereas the 10 is the numeric value for the intent of minimal cost. The second policy, the pink one, can be recognized by its unique tuple constraint `(R1, 20, R10)`. Its purpose is to steer traffic over the minimum delay path to the destination router *R10*.

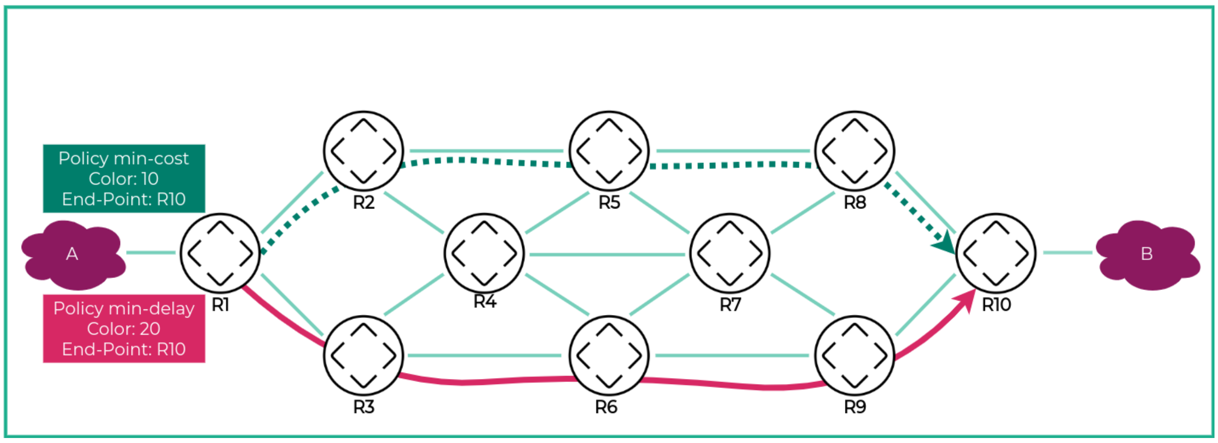


Figure 1.4: Segment Routing Policies

1.2.3 Explicit Path

There are many ways to steer traffic in the field of Segment Routing. However, probably the most basic one, which also is important for this thesis, is the Explicit Path method. A configuration is applied on the headend, which tells which explicit Segments should be applied to a packet. Therefore, the router adds this information to the appropriate packets. This technique has the advantage that a path can be calculated externally and then be configured on the headend. The router, at this point, does not know the intent behind this policy. It simply executes the associated instructions. So, it is possible to craft the path manually and without considerable effort, which a packet should take. Thereby not the whole path has to be mentioned. Segment Routing takes care that the appropriate Segments existing in the packet header are executed.

The figure 1.5 exemplifies this behavior. On the headend *R1* an Explicit Path Policy was instantiated. It ensures that the explicit path is taken care of. In the header, there are mentioned the Segments of the Router *R4, R5, R7, R10*. Therefore, Segment Routing ensures automatically that the best Equal-Cost Multi-Path Routing (ECMP) path from the source to the destination node is found. In this particular case, equality is shown between the first and the last passage, whereas between *R4* and *R7*, only one path is considered as the most suitable one.

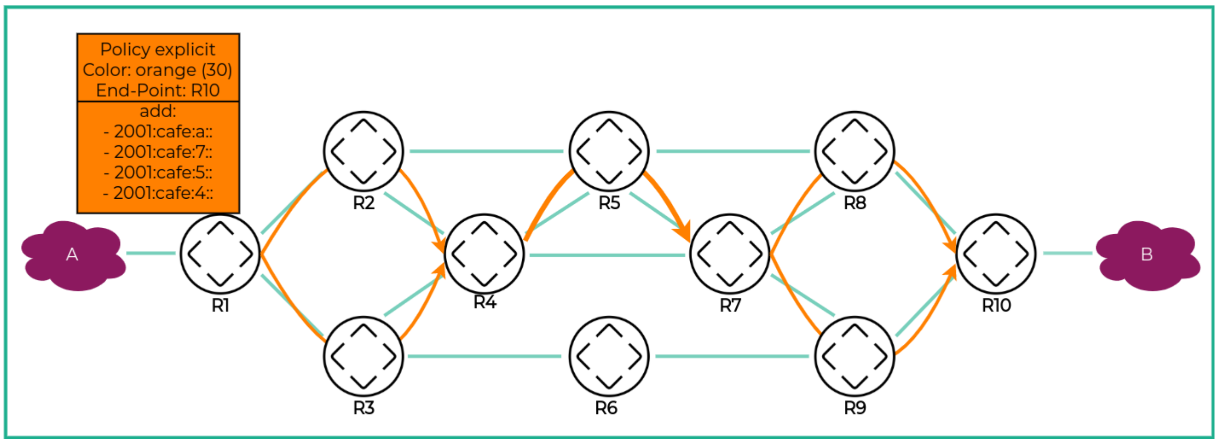


Figure 1.5: Explicit Path Policy

1.2.4 Automated Steering

Automated Steering is another crucial concept, that is essential for steering traffic in a Segment Routing Domain. It is the solution, that allows to steer traffic automatically into SR Policy based on the service route. Simplified, it is the technique for the headend to map the traffic automatically with the correct policy. The policy then ensures the correct treatment and steering according to its configuration. Like already mentioned, a Policy can be identified by the unique tuple `(headend, color, endpoint)`. The color is the concept to differentiate between different policies with the same endpoint. Furthermore, it has another crucial functionality: It provides the routes additional information into which policy the traffic has to be steered. Thus, the color can be seen as the mapping between the policy and the traffic, which should be processed. [2]

An example of colorized routes can be observed in listing 1.2. It shows the two routes *170.0.0.11/32* and *171.0.0.11/32* which were colorized with the color values *16* and *17*. The different colors can be identified with the syntax `C:<color-value>`.

```
Status codes: s suppressed, d damped, h history, * valid, > best
              i - internal, r RIB-failure, S stale, N Nexthop-discard
Origin codes: i - IGP, e - EGP, ? - incomplete
   Network            Next Hop            Metric LocPrf Weight Path
Route Distinguisher: 10.10.10.10:1 (default for vrf VPN1)
*> 170.0.0.10/32      0.0.0.0                  0         32768 ?
*>i170.0.0.11/32      2001:db8:bbbb::b C:16    0    100      0 ?
*> 171.0.0.10/32      0.0.0.0                  0         32768 ?
*>i171.0.0.11/32      2001:db8:bbbb::b C:17    0    100      0 ?
```

Listing 1.2: Colorized Routes IOS-XR

In order that Automated Steering can work, a network route must have additional color information. This color information is an additional property connected to a route used to characterize the network route. This particular color information is later used in the headend to define the special steering treatment. Because each network route has to be colorized to enable the Automated Steering functionality, this configuration must be made on the destination router, where the network is announced. As soon as the route is colorized, this information is distributed in the network and made available for the potential headends. The distribution of this information is made via Border Gateway Protocol (BGP) protocol. The color attribute is specified by an extended community attribute in the BGP packet[13, 15] Once the headend node receives the color information, the appropriate policies can be created, and the Automated Steering enforced. A process of this technique is visualized in figure 1.6.

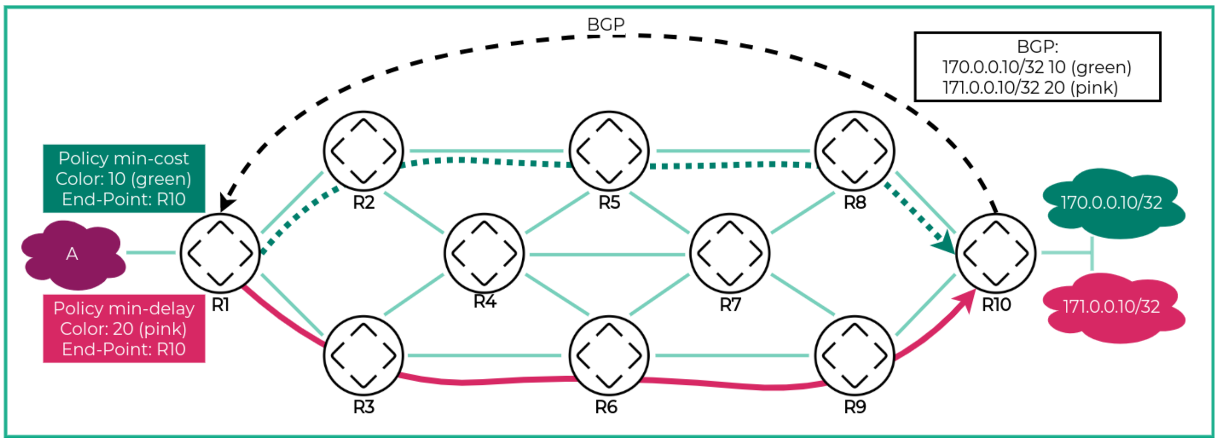


Figure 1.6: Automated Steering

1.3 Aims and Objectives

This section is intended to provide an introduction to the problem as well as the associated solution.

1.3.1 Problem

The problem was already analyzed in the preliminary work SerChio. Therefore, the problem is only discussed briefly again here. However, more attention should be paid to what has not yet been achieved in the preliminary work and the problems that still need to be addressed.

The fundamental problem is that traffic, which should flow through defined services, can only be configured by static intervention. This problem has already been discussed in the preliminary work but still has not been solved completely. However, an essential so-called service chain could already be calculated. This service chain has laid the foundation for this continuing bachelor thesis.

A solution, that can enforce this service chain in the network and route the traffic according to this service chain, has to be found. Moreover, this solution must not only steer the traffic according to this service chain; it has to adapt the service chain to topology changes automatically. That is the only way to ensure that traffic can always take the most suitable route via the correct services.

1.3.2 Solution - Service Programming

The solution for the discussed problems is Service Programming. The term Service Programming was first established in the book *Segment Routing Part II - Traffic Engineering*. Its meaning can be summarized as follows: Service Programming is a solution to enforce traffic to be steered and processed through network services on its path traversal from source to destination. Because Service Programming is more intelligent, dynamic, and therefore more enhanced than simple service chaining, it can be seen as its successor. Service Programming aims to introduce a technique to program network paths with included network services like a modern programming language. The path should be expressed with single instructions and thus defined uniquely. Therefore, this very dynamic approach addresses the customer needs of the future and aims to deliver a method to be as adaptable as possible.[2]

Service Programming and Segment Routing

Segment Routing is the protocol that makes intelligent Service Programming possible. It is the optimal approach on which a Service Programming solution can be based on. Thanks to the instructions saved in the packet header, the Service Programming can be realized without the need to start a completely new technique. The whole approach is based on the traffic engineering and network programming approach defined in RFC 8986, which allows to specify the packet processing in the Segment List in the IPv6 header.[19] The need for that solution is that a service can be assigned with a segment in order to be fully integrated into the Segment Routing Domain. So, as soon as this happens, the Service Programming solution can benefit from the powerful traffic engineering concepts of Segment Routing. Simplified, criteria that must be ensured are: First, the packet must be steered to the service. This step should be possible without any significant innovations. The second step is to ensure that the service understands the segment and knows which instruction has to be executed. More details about this current state can be found in section SR-aware Services.

SR-aware Services

In order that a service can be integrated into a Segment Routing Domain, the appropriate service has to deal with Segments.[12] The exact functionality and implementation go beyond this thesis. However, is has to be mentioned, that a few services are aware of the Segment Routing technology. These services are called Segment Routing aware services (SR-aware services) in this thesis. To introduce a complete Segment Routing Service Programming, the solution has drawn a profit from the following SR-aware services:

- **SERA** stands for *SEgment Routing aware firewall* and extends the Linux iptables functionalities to handle Segment Routing packets. More information can be found under `https://www.segment-routing.net/open-software/SERA/`
- **SR-aware Snort** is an extended version of the well-known Intrusion Detection System (IDS)/Intrusion Protection System (IPS) snort. More information can be found under `https://www.segment-routing.net/open-software/SR-Snort/`

Concept

This thesis aims to develop an application to lay the foundation of Segment Routing Service Programming and thus solve the discussed problems. This section provides more detailed information, what general concepts and different techniques are applied to implement the needed logic.

The Segment Routing Service Programming (SerPro) can be seen as the successor of the SerChio application. The results, which were found and discussed in the preliminary thesis, are used to enhance the application in this thesis. The new application is predominantly based on the calculation basics, which were found in the previous work. In order to implement a future-proof solution, this thesis has to adapt the findings to the SRv6.

Furthermore, this thesis aims to enforce traffic engineering in the network. The goal is to ensure that traffic flows over the programmed path that has been defined by a user in the application. Thereby the packets should not only be steered from source to destination but, furthermore, sent via the most suitable services in the network. The appropriate services should process the packets and execute the necessary actions according to their configuration. The previous thesis managed to find a solution for identifying the best service or the best service chain.

So, that packets can be processed by services on its path traversal SR-aware services have to be used. The services are assigned with appropriate Segment information in order that the application can handle network services. The service installation and configuration are not part of this thesis. However, it is necessary to introduce a way to bridge the services in the network to the application. As soon as the services are fully integrated within the Segment Routing Domain and configured properly, a way should be found to steer traffic over these.

In order to ensure that the calculated best paths can be enforced in the network, the introduced technique Automated Steering is used. The application has to guarantee that the following tasks are included, so that the mechanic works:

1. The application has to ensure that the appropriate routes are extended with a color value and that this information is distributed in the Segment Routing Domain. Therefore the application has to change configuration aspects on the endpoints.
2. The application has to ensure that the path is programmed and enforced. Therefore, the application has to create Segment Routing Policies with an Explicit Path and the appropriate Color information. This way the traffic can be directed to the services with their Segment information.

An overview of the different concepts and the functionality of the application can be found in figure 1.7.

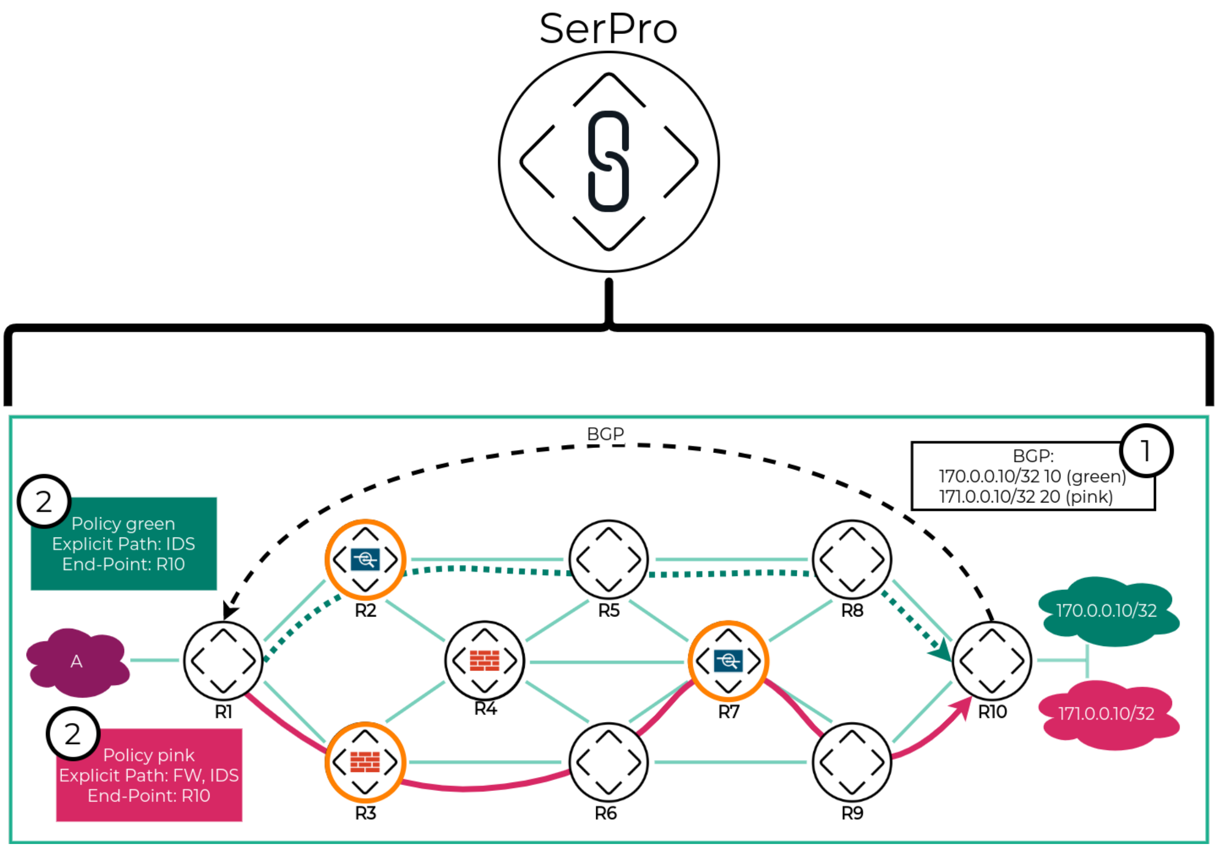


Figure 1.7: Application Concept Overview

Chapter 2

Results

This chapter is intended to show what could be achieved in this bachelor thesis. Therefore, a clear delineation should show what has been developed in this thesis, what has not been developed by the authors, and what has already been developed in the preliminary work SerChio. This delimitation can be found in section Distinction. In the following section Achievements, the achievements will be discussed. Section Implementation gives then a technically more intensive view of various aspects of the work.

2.1 Distinction

In order to avoid any misunderstandings when reading this work, this section will discuss what has been developed and achieved and what has not.

The frontend mentioned several times in this thesis for completeness was developed in cooperation with the Institute for Networked Solutions (INS) and is not part of this bachelor thesis. Together with the advisor it was decided, that there was time-wise no room for the frontend in this work. Due to the advisor's wish to implement a frontend, it was determined that the INS would do the complete development work. The two authors of this thesis would hold joint sessions with the INS developers.

As this bachelor thesis continues the project work (SerChio) from the fall semester of 2020, several aspects from this thesis have been adopted, adapted, and improved. Although this thesis is a further development of the project work, it can occur that elements have been taken over and not wholly redeveloped; for such elements, it was ensured that they were cited correctly in the documentation.

2.2 Achievements

This section is intended to address the different achieved results. The different use cases will be briefly addressed individually, and it will be shown what has been achieved. More information about the implementation can be found in section 2.3. Different points are illustrated with pictures of the frontend to make them more precise. That was not part of this bachelor thesis (see Distinction) but is used to display the backends possibilities graphically.

2.2.1 View Topology

The authors of this bachelor thesis have worked intensively with the INS's frontend developers to provide the customer with the result that is as intuitive as possible. The topology is automatically adapted in case of updates, and the frontend is informed so, that it can load the new topology automatically. That allows the user to have a well-rounded result and always to be sure to see the latest topology. Through the additional linking with the various SR-aware services, which are also displayed in the topology, customer success is reaching another level, allowing a direct overview of what services are deployed in the network.

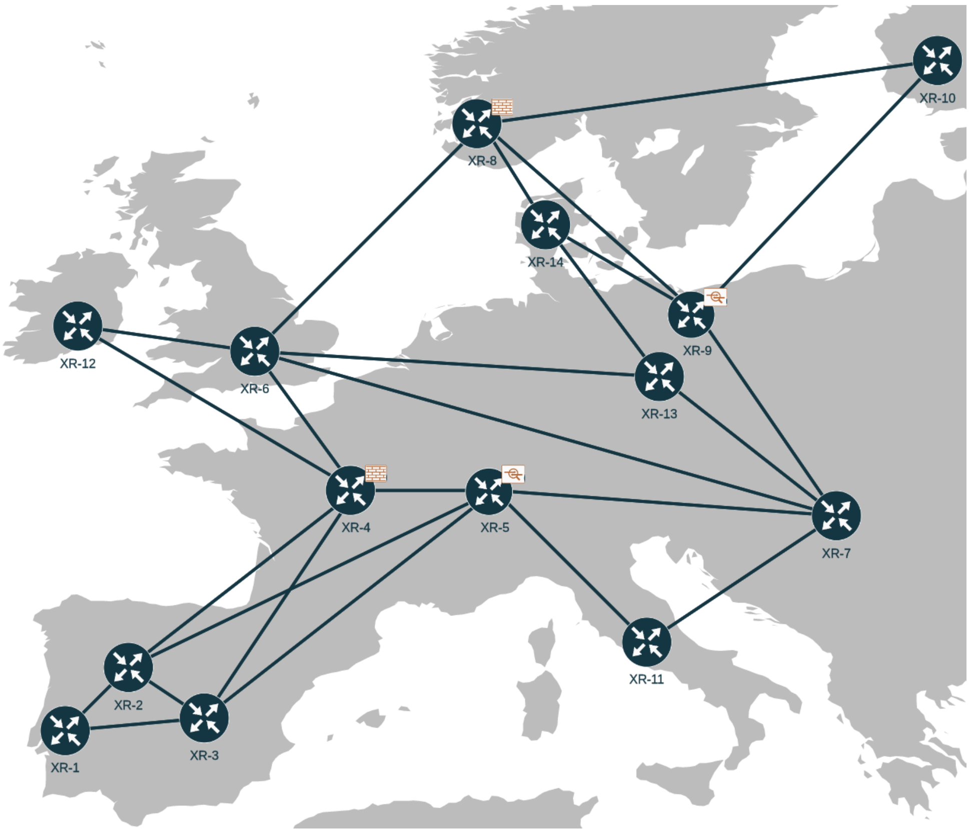


Figure 2.1: Frontend Topology View

2.2.2 CRUD Segment Routing Traffic Engineering Policy

Policies can be created and validated directly through the API. The frontend offers a modern and simple form and communicates directly with the backend to give the user the right suggestions for the different parameters. The frontend and backend validate all inputs. Policies can also be edited after being created; various regulations have been made to allow the edits only under some conditions. For example, policies that are in a deployed state may not be changed before they are reverted. After creation, the policy is validated and created by the backend. The user can then view the result directly via the frontend. The example result can be seen in figure 2.3, and the policy create form in figure 2.2.

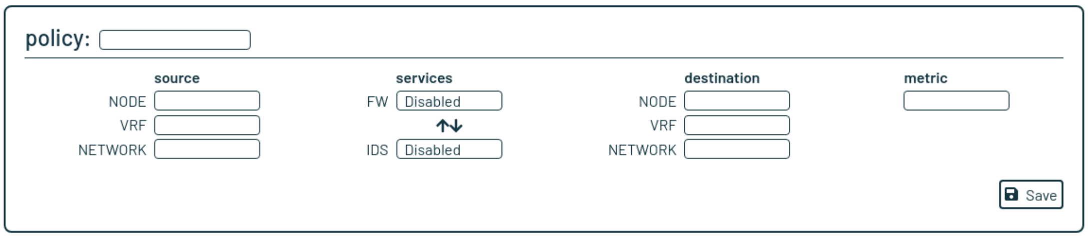


Figure 2.2: Policy Create Form

2.2.3 Structure Data

The backend should provide all data in a structured format so that the frontend can retrieve these and use them without enormous logic. That is made possible by a standardized REpresentational State Transfer (REST) Application Programming Interface (API) which the backend provides.

For larger topologies, the policy results can quickly become confusing. Therefore, a clustering approach was implemented which can display the topology in a minimized form. The backend should take over the complete logic. The frontend should only have to plug the nodes and links together. Such a clustered topology can be seen in figure 2.3. More information about the clustering can be found in section 2.3.6.

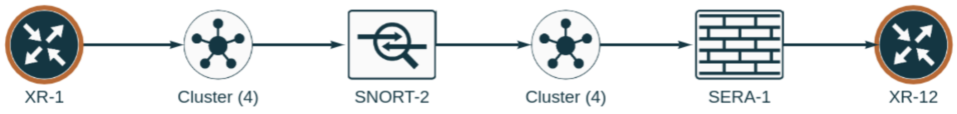
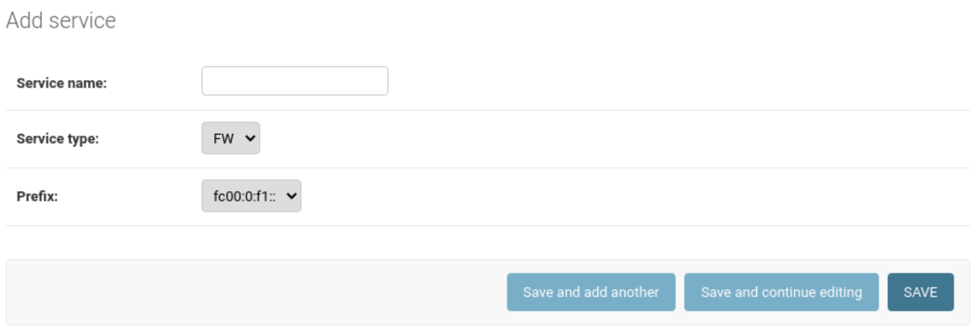


Figure 2.3: Clustered Result Example

2.2.4 Define Service

Services in the topology are detected by special Interior Gateway Protocol (IGP) prefixes so called service prefixes and communicated to the application. The user gets the possibility to link service instances with these service prefixes. The entire configuration of these service instances happens directly in the admin portal and can only be done by authorized users. Through this procedure, service prefixes can be easily created in the network. The application then recognizes these prefixes. The user is thus allowed to define a service instance. All service instances are immediately visible in the topology after creation and can be used for policies. More about, on how the service prefixes are created can be found in section 2.3.7.


Add service

Service name:	
Service type:	FW
Prefix:	fc00:0:f1::

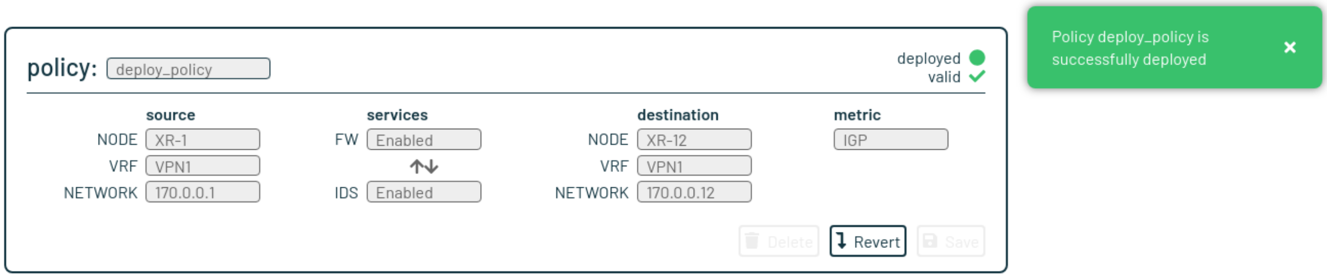
Figure 2.4: Admin Page Add Service

2.2.5 Calculate Paths

The complete calculation was adapted to the new segment routing protocol SRv6 compared to the previous work SerChio. However, it was already noticed in the SerChio application that the calculation can still be improved in terms of speed. Therefore, a preprocessing was introduced, enabling the application to return the correct result to the user in the shortest possible time. More information about the calculation and the preprocessing can be found in section 2.3.3.

2.2.6 Deploy Link Metric Algorithm Policy

Created policies can be deployed to the routers via the frontend or directly via the REST API. The whole deployment is controlled by a separate module which asynchronously outsources the deployment jobs. The users are informed dynamically about the status of the job. For example, in figure 2.5, a deployed policy can be seen with the message indicating that the deployment was successful.



policy: deploy_policy — deployed ● valid ✓

Policy deploy_policy is successfully deployed

source		services		destination		metric
NODE	XR-1	FW	Enabled	NODE	XR-12	IGP
VRF	VPN1		↑↓	VRF	VPN1	
NETWORK	170.0.0.1	IDS	Enabled	NETWORK	170.0.0.12	

Figure 2.5: Policy Deployment Example

2.2.7 Login

Due to the requirement to deploy policies directly to the network, a solution had to be developed to deny access to unauthorized users. Developing an access system with JSON Web Token (JWT) in the backend allows the frontend to deliver a login page that requests the JWT from the backend and renews it before the token expires. Thus, unauthorized users will be denied access to the application already at the login page. Furthermore, access to the admin portal is already secured by using the Django framework. In addition, only specially authorized users can log in to the admin portal, which increases further security. More information can be found in section 6.7.

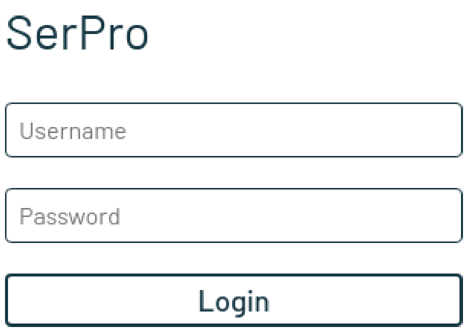
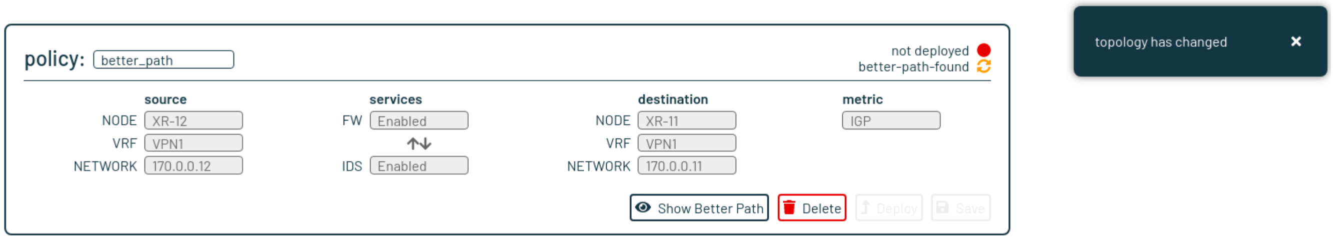


Figure 2.6: Frontend Login Page

2.2.8 Manage Recalculations

The application registers and controls all topology changes automatically. As soon as changes that change the topology are detected (for example, a new node is added, or a link metric is changed), the application checks all policies that could be affected. Affected policies are automatically recalculated and communicated to the user. For example, in figure 2.7, a policy can be seen set to a better path that has to be confirmed manually by the user.



policy: better_path

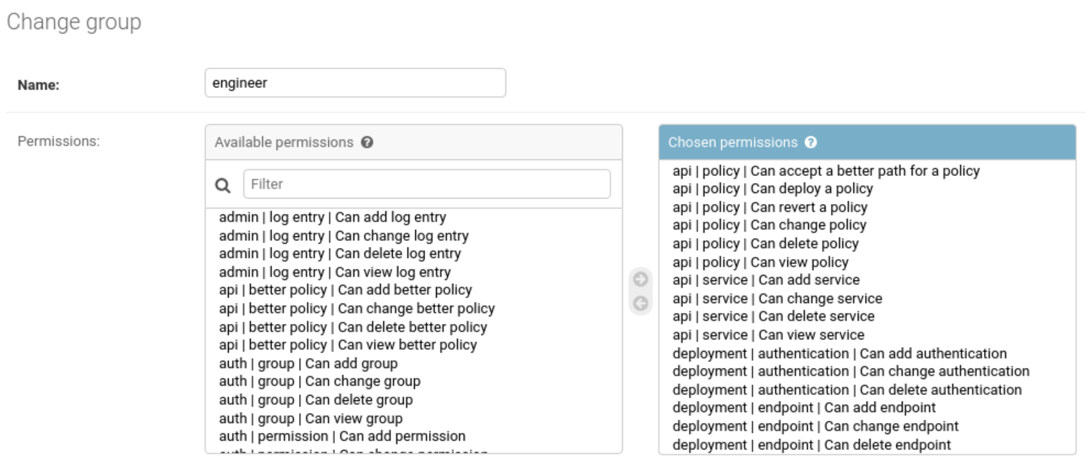
not deployed ●
better-path-found

	source	services	destination	metric
NODE	XR-12	FW Enabled	XR-11	IGP
VRF	VPN1		VPN1	
NETWORK	170.0.0.12	IDS Enabled	170.0.0.11	

Figure 2.7: Policy Recalculation Example

2.2.9 CRUD Roles and Users

Using the Django framework in the backend service, the logic for user and group management already existing in this framework could be adopted. Authorized users can manage groups and users via the admin portal (see figure 2.8). By adding various additional permissions, the existing permission system could be easily adapted and tailored to the needs of the SerPro application.



Change group

Name: engineer

Permissions:

Available permissions

admin | log entry | Can add log entry
admin | log entry | Can change log entry
admin | log entry | Can delete log entry
admin | log entry | Can view log entry
api | better policy | Can add better policy
api | better policy | Can change better policy
api | better policy | Can delete better policy
api | better policy | Can view better policy
auth | group | Can add group
auth | group | Can change group
auth | group | Can delete group
auth | group | Can view group
auth | permission | Can add permission
auth | permission | Can change permission

Chosen permissions

api | policy | Can accept a better path for a policy
api | policy | Can deploy a policy
api | policy | Can revert a policy
api | policy | Can change policy
api | policy | Can delete policy
api | policy | Can view policy
api | service | Can add service
api | service | Can change service
api | service | Can delete service
api | service | Can view service
deployment | authentication | Can add authentication
deployment | authentication | Can change authentication
deployment | authentication | Can delete authentication
deployment | endpoint | Can add endpoint
deployment | endpoint | Can change endpoint
deployment | endpoint | Can delete endpoint

Figure 2.8: Admin Page Group Creation

2.2.10 Handle Permissions

In order to allow only authorized users access to the functions permitted to them, a logic was implemented in the backend which takes care of this function. This function is mainly based on the user and group management of Django, which was already mentioned in section CRUD Roles and Users.

2.3 Implementation

In the following sections, the most critical aspects of implementing the SerPro application will be discussed in more detail. The basic approaches that have been used in this work will be referred to. Technologies are not examined in detail in this section; these can be viewed in section Technology Decisions.

2.3.1 Architecture

Since Segment Routing is nowadays mainly used in large networks, especially in provider networks, the application must handle extremely large topologies and many users. Due to this fact, the application had to be scalable and highly available. Therefore, a cloud-native approach came to the fore, which was also used in the preliminary work SerChio. The goal was to deploy the application on a Kubernetes cluster, allowing the application to scale quickly and even allowing the possibility to activate autoscaling functions.

Figure 2.9 shows the complete architecture in a very abstract view. This abstraction makes it relatively easy to show how the architecture roughly looks like and how the various elements interact.

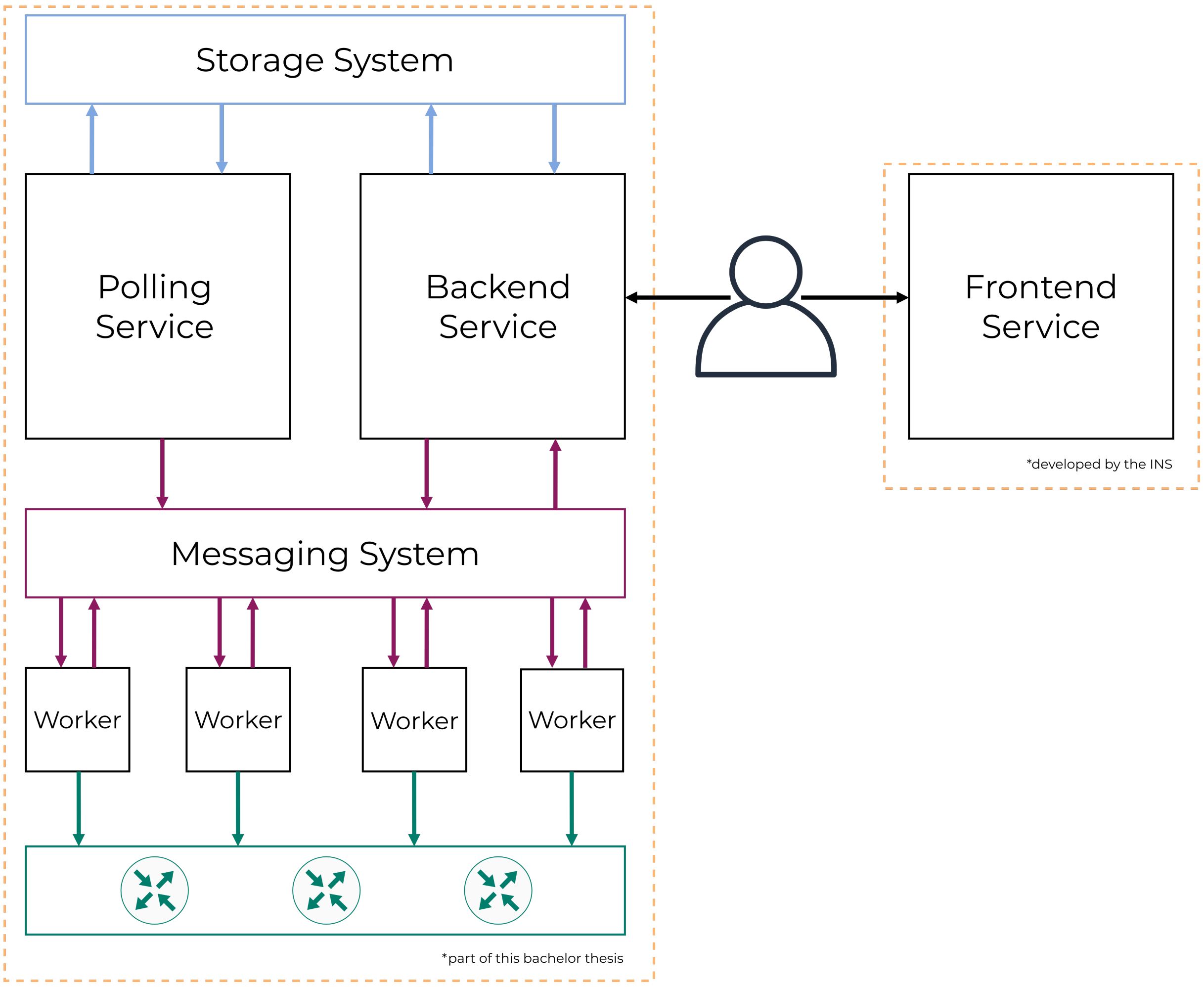


Figure 2.9: Abstract Architecture Overview

Storage System

A storage system consisting of a Redis cluster and a PostgreSQL database was deployed. The combination of cache and database has the advantage that the data stored in the cache is available extremely quick. Data that needs to be persisted or is sensitive is stored on the relational database. This combination has proven to be a very stable and fast solution during the entire bachelor thesis. More on the reasons for this division can be found in section Caching.

Messaging System

The messaging system is the foundation for ensuring that the application can be deployed in a stable and scalable manner. The messaging system is used for complete communication between the different services. For example, updates, WebSocket messages, job tasks, or job results are stored and transmitted over the message system. A RabbitMQ cluster was deployed to ensure that this central element of the application is always available. Since messages are now always stored in a queue, they can no longer be lost on the way between services, which makes the entire communication much more stable and trustworthy. The ability to consume messages in a controlled manner also provides the basis for scaling the entire SerPro application.

Backend Service

The backend is the most critical service in the whole SerPro application. As the brain of the application, it is responsible for all data delivery via an API. For this purpose, a standardized REST API was implemented, making it as easy as possible for the frontend service to query and change data.

In addition to the data provided, the backend is responsible for the entire calculation, administration, and keeping updated all policies. Policies should be created as efficiently as possible via the backend. These policies should be always up-to-date, so they must react to changes in the network and be adapted accordingly. How this was implemented can be seen in sections Calculation and Preprocessing and Policy Verification. In addition to the entire policy treatment, the backend is also responsible for all notifications for the clients. It sends update and status messages to all connected WebSockets, so that they can adapt dynamically and withoud delay.

Polling Service

In order to continuously provide the whole application with the latest topology data, a dedicated service had to be developed that fetches and processes the latest data from the external software system Jalapeño.

Jalapeño, as an external system, integrates and processes all BGP link state messages and streaming telemetry data from the Segment Routing network. This system was developed by the project partner Cisco and provided for this bachelor thesis.

The polling service fetches the required data from the Jalapeño system, detects the changes using various developed detection mechanisms, and then sends only the updates intended for the backend to the messaging system via update messages. The polling service was already developed and used in the preliminary work SerChio. However, that was converted entirely to the new SRv6 and adapted to the new messaging system in the bachelor thesis.

Workers

The workers are responsible for the complete deployment of the different routers in the network. The workers enable the backend to commission deployment jobs asynchronously. Due to this asynchrony, the backend is not blocked, and the job is entirely outsourced to these workers. All deployment jobs are placed directly from the backend into the messaging system and picked up by the workers. With the autoscaling possibility on the Kubernetes cluster, the workers can be automatically scaled up or down, depending on how many jobs there are in the messaging system.

Frontend Service

As already described in the Distinction, the frontend is not part of this bachelor thesis. However, for the sake of completeness and a better presentation, it is mentioned again here.

The frontend is responsible for the presentation of the data and functions provided by the backend. The frontend provides the end-user with a graphical user interface that allows them to use all the backend functionalities in a structured and straightforward way. The frontend communicates via the REST API, which is provided by the backend. In addition to the regular API communication, the frontend offers the end customer the possibility to establish a WebSocket connection with the backend. Through this WebSocket, the customer's browser can dynamically display messages and changes without delay.

2.3.2 Enabling / Disabling

Modern networks are very dynamic and can face a huge number of changes every day. Many changes are planned, such as adding a new node, alternating a link, or network maintenance. Some others are not planned, like outages caused by hardware faults, human errors, or similar mistakes. Generally, networks are planned to survive many mutations in the network. Therefore, the SerPro application should also be designed to handle these shifts in the network.

During the early development phase, it was considered that alternations in the network could lead to problems in the application. Especially the removing of central elements like nodes and services leads to faults in the application because the policies are directly related to such components. Consequently, after the deletion of specific elements, the policies were related to non-existing elements. This situation has lead to severe imperfections. Therefore a solution had to be found to solve these problems and hence deal with changes of any kind.

A solution was found and implemented. The solution was to introduce an *Enabling/Disabling* strategy. Instead of deleting an element in the application, the appropriate elements were disabled. Hence, as soon as a remove message was received, the element was disabled. Logically, also the opposite action has to be introduced. As soon as an add message about a known element is received, the element can be re-enabled. This introduced logic is the fundament of many further implementations and has a few considerable advantages:

- Disabling an element has no harmful impact on the application; the element is still accessible but marked as disabled.
- It can be reacted very fast to network changes by masking the disabled elements.
- An enabling of an element is done almost instantly because the information is already present in the system.
- Policies can automatically be verified with a validation check (see also chapter Policy Verification)
- Instead of changing the maintained graph thoroughly, the calculation can be made on a masked graph, which consists only of the enabled nodes.

2.3.3 Calculation and Preprocessing

This section primarily contains information about how the calculation can be done in an optimized way. It includes information about the drawbacks of the previous application and delivers an answer on how the disadvantages were improved in this thesis. It shows the most critical concepts to give an understanding of how the implementation is done.

General

The calculation of the most suitable paths is a crucial part of the application. It not only should deliver a correct result, but it also has the requirement to do it within an affordable time frame. During the previous work, a way to calculate the service chain and the appropriate paths was found and implemented. For more details, see the preliminary work SerChio.

The difficulty of the calculation lays in the structure of a complete path. A complete path contains several sub-paths depending on the number of service instances in the path. So the calculation has to analyze all sub-paths to decide if the whole path is the most suitable one. If, for example, the path has only one service instance in it, it has to consider two sub-paths: The first path goes from the start node to the service. The second sub-path runs from the service node to the destination node. A similar example is illustrated in figure 2.10.

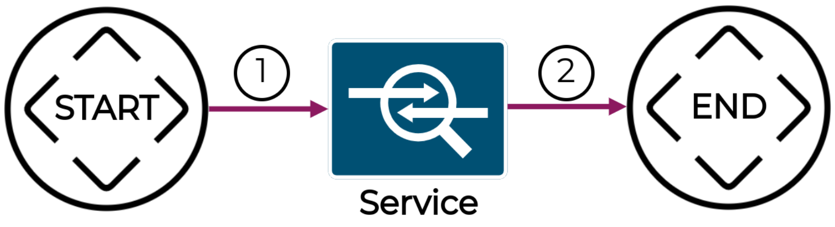


Figure 2.10: Calculation Sub-Paths

Previous Drawback

The problem of the previous calculation was the following: it calculated a whole shortest-path with the help of Dijkstra for each sub-path. Therefore, it also calculated the exact ways for paths, which finally were not appropriate because they lead over services, which are not the most suitable ones. Consequently, these non-optimal paths were finally not considered in the result. Indeed, this process was optimized by cancelling a path calculation if the path already had a higher total cost than the current best one. However, it was overhead to calculate the whole path information continuously.

Figure 2.11 serves as an example for this suboptimal behavior. The previous implementation first calculated all ECMP paths for the dashed path and saved the result. Next, it calculated all ECMP paths for the green path compared it to the current best path and noticed that the path was worse. Therefore, the second calculation was fully processed, altought it was not considered as suitable in the end. This was a major drawback which should be improved in this thesis. Further information about how the problem was solved in the next section.

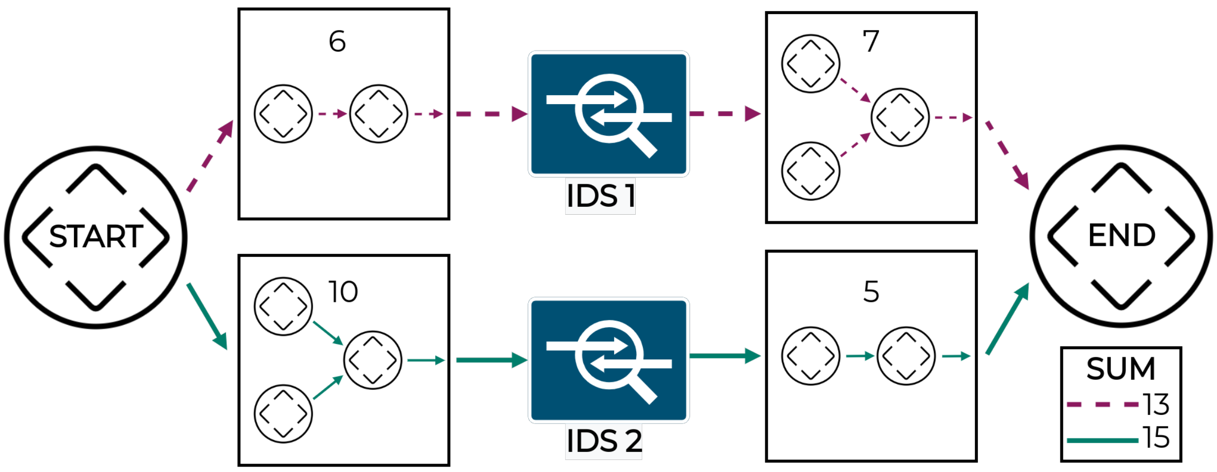


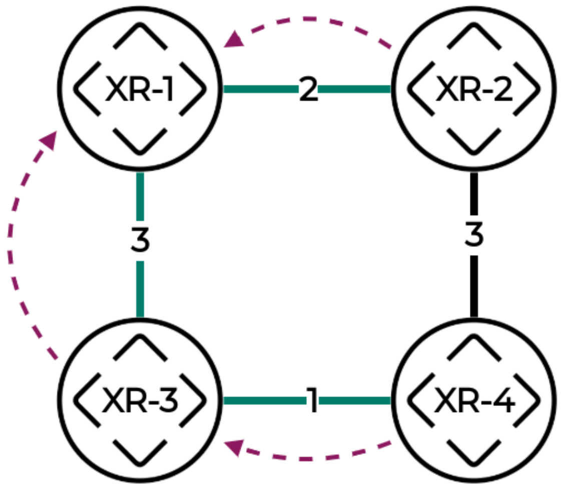
Figure 2.11: Previous Calculation Implementation

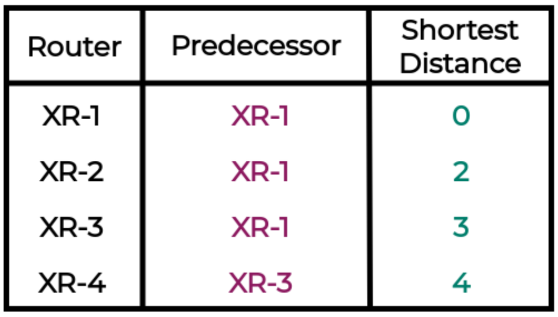
Preprocessing

Besides implementing the calculation method in the previous project thesis, appropriate research delivered the bottleneck of it. In addition, it delivered an idea on how to improve the performance in the future. The answer to the problem could be found possibly with a valuable and specific preprocessing. The preprocessing should help to improve the calculation efficiency. The idea behind this is straightforward: Before a calculation is triggered, information relevant to the calculation should be collected, that could be reused later by the calculations and thus enhance the calculation.

The first step was to identify which calculation-relevant characteristics should be calculated in the preprocessing to improve the calculation later on. Usually, without preprocessing, the Dijkstra shortest path algorithm calculates the shortest path between two edges. By inspecting the algorithm, this process can be optimized: The preprocessing has to deliver a map containing the information of the shortest distance of a node and its preprocessor. As soon as the mapping has been calculated, the algorithm can use this information to calculate the shortest path efficiently. The shortest distance can be observed directly in the mapping table. The path can be observed by following the preprocessor entries until the preprocessor is the start node. [10] [6]

Figure 2.12 shows an example of this procedure.





Router	Predecessor	Shortest Distance
XR-1	XR-1	0
XR-2	XR-1	2
XR-3	XR-1	3
XR-4	XR-3	4

Figure 2.12: Preprocessed Values

After the preprocessed values could be identified, an approach had to be found to execute the preprocessing. This process was related with research about possible algorithms. There are algorithms, which calculate the all-pairs shortest paths. So, these algorithms calculate the shortest path from each vertex to each other vertex in a graph. Johnson and Floyd-Warshall are such algorithms, that could be considered for the preprocessing. However, it has been shown that these algorithms do not have a smaller time complexity than execute different shortest-path calculations with Dijkstra. [1]

Further, it was recognized that an all-pairs shortest path algorithm is a calculation overhead because not every node could be a potential start node in a calculation. Especially in the particular use case of the SerPro application, it was figured out that the Dijkstra approach is faster than an all-pair shortest path algorithm. As already mentioned, the calculation of the best service chain path has to be divided into sub-parts (compare figure 2.10). A start node has to be a Provider Edge (PE) router or a node with a service trailed in these sub-parts. Therefore, the preprocessing could be optimized with this insight: a preprocessing has only to be calculated for each potential starter node. Because the number of starter nodes is smaller than all vertices in the graph in a common provider network, the calculation with the Dijkstra algorithm is more performant than calculating the all-pair shortest paths.

Regarding the example network in figure 2.13 it should be apparent that the mentioned insight has a considerable advance. On the other hand the preprocessing without an optimization would calculate all shortest paths for all routers in the shown network graph. Whereas the optimized solution only calculated the necessary information for the possible start nodes. The optimization averts the unnecessary calculation of the shortest distance information for the intermediate nodes *XR-3*, *XR-4*, *XR-5* which are not possible in the shown network.

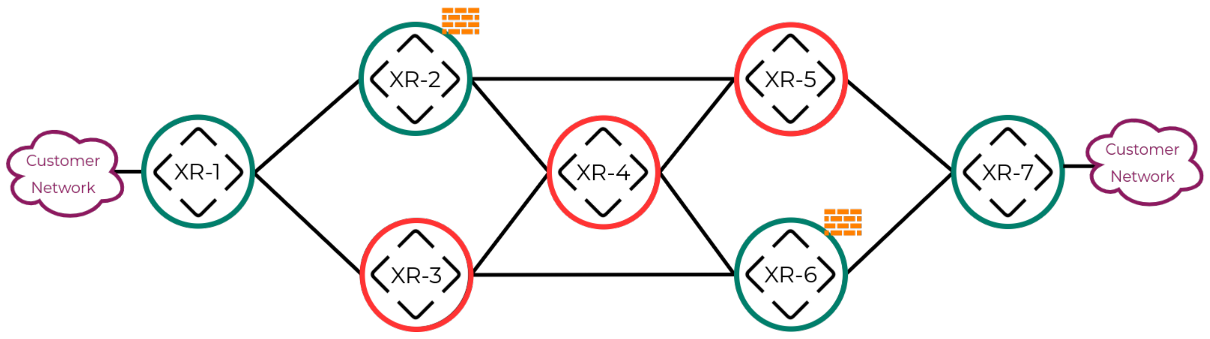


Figure 2.13: Preprocessing Optimization

So, it was decided to implement a preprocessing, which calculates the shortest distance and the preprocessor information for each node of interest with the help of Dijkstra. Like mentioned before, possible starter nodes are regarded as these nodes of interest. Starter nodes in our application are PE routers, which have at least one customer network attached, or nodes, which have at least one service connected.

Obviously, the preprocessing has to be triggered as soon as the application is started and the initial loading of the network information is done. Furthermore, the network can be altered so that a new preprocessing is necessary to deliver the correct result once again after a topology change.

A new message was introduced to address this and similar problems, which signals that the altered information was carried over. The so-called *finish message* indicates that it is the last message of a polling iteration and is structured like listing 2.1.

```
{
  "type": "info",
  "status": "finished",
  "message": "polling iteration finished",
  "data": {}
}
```

Listing 2.1: Finish Message

As soon as this message is retrieved, the backend service can check if new preprocessing is necessary. The decision if a preprocessing is necessary depends on the changes since the last *finish message*. If any changes are included that have influenced the network graph maintained in the application, a new preprocessing execution is necessary. Otherwise, the preprocessed information is still correct and doesn't need to be changed. An example of an update that triggers an adjustment in the graph is a link addition, whereas for example a change of the router name does not change the graph behind the application. As soon as such a significant change has happened, a flag is saved in the cache, indicating that a new preprocessing is necessary. By saving the flag in the cache, it is also possible to detect changes in the network if a backend instance would crash. This flag can be examined after the *finish message*. If it is set, the preprocessing has to be executed. This mechanism ensures that the preprocessing is only executed on demand – a further optimization step included in the application.

Calculation

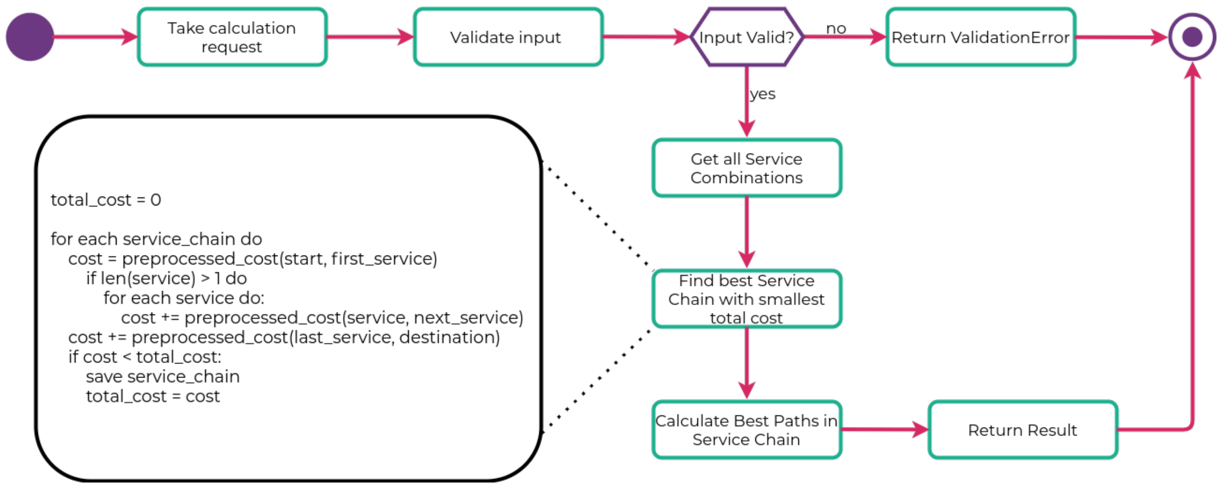
The calculation could be optimized with the help of the introduced preprocessing method. The workflow of a calculation is visualized in figure 2.14.

As soon as a calculation is requested, the different input parameters are validated. Supposed there is a non-viable input a `ValidationError` is returned. An example would be a destination VRF with no export tag with the source VRF in common and hence no reachability.

If the input parameter is valid, the next step is to create all possible combinations of the services. Like in the previous work, a combinatory approach is implemented with the help of the Cartesian Product, which returns all possible service combinations.

Once all the service combinations were created, the actual calculation can be started. The first step is to get the best service chain with the lowest possible total costs. This step is fundamental because the preprocessed smallest cost information could be used to get the service chain, which is the best one. It has to be mentioned that if several equal service chains exist, the service chain is considered as the best one, which was found first.

Until this point, no shortest path calculation has occurred thanks to the intelligent preprocessing strategy. However, the best service chain with its total cost has already been found. The only missing part is the correct path from the source to the destination thtough the best service combination. So, the subsequent step is to get the best path information out of the service chain. The path can be calculated with the help of the shortest path calculation and Dijkstra. Here it has to be mentioned, according to the exact amount of included services, at least two shortest path calculations have to be executed (see 2.10). This process could also be optimized thanks to the predecessor information, which was collected previously. Hence, the actual paths do not have to be calculated but have to be looked up in the predecessor map. As soon as this step is finished, the result can be returned.


Take calculation request
Validate input
Input Valid?
no
Return ValidationError
yes
Get all Service Combinations
Find best Service Chain with smallest total cost
Calculate Best Paths in Service Chain
Return Result

```
total_cost = 0

for each service_chain do
  cost = preprocessed_cost(start, first_service)
    if len(service) > 1 do
      for each service do:
        cost += preprocessed_cost(service, next_service)
  cost += preprocessed_cost(last_service, destination)
  if cost < total_cost:
    save service_chain
    total_cost = cost
```

Figure 2.14: Calculation Workflow

2.3.4 Policy Verification

General

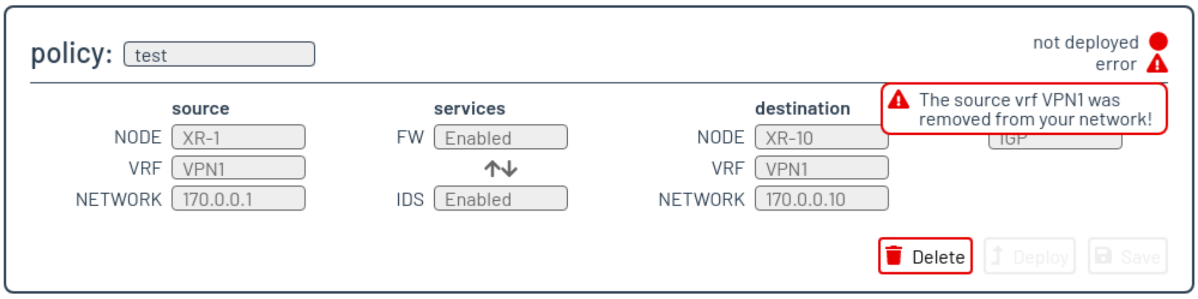
The policy verification is a central part of the application. It allows to react and trigger the correct handling after a change has occurred in the network. The goal is to verify the policies and ensure that the policies have a correct status in the network and the application according to the latest network situation.

Verification has, like the preprocessing, to be done after the network has converged. After the *finish message* was received, the preprocessing has to be made to ensure that the calculation data is up-to-date. More information about this process can be found in section Preprocessing. After the preprocessing is done, the verification of the policies has to be executed. The verification is a complex process, which goes over different levels. The different stages are outlined in the appropriate subsections.

A mechanism was introduced to decide which verification-relevant parts have changed since the last *finish message*. As soon as a significant update message is processed, the information is updated in the cache. Subsequently, the verifier can use checks if there have been any changes of interest. It works similarly to the check if a preprocessing was necessary. If, for example, a router name has changed, there is no need to verify the policies. However, if a VRF was removed from a router, this action can heavily influence existing policies and have to be examined thoroughly by the verification process.

Policy Validation

After a network change has happened, the appropriate policies have to be checked if the changes influence their validity. If a policy is in a valid state, a network change could lead to a defective state. In the case of an incorrect state, the verifier has to detect this defect, which has to react accordingly: the policy has to be put in an error state. This task is included in the application and will be triggered if there have been any changes, leading to error policies. The verifier checks if the source, destination and service information are still correct to create a valid policy. An example of a policy in an error state is shown in figure 2.15



policy: test

not deployed ●
error ⚠

⚠ The source vrf VPN1 was removed from your network!

source		services		destination	
NODE	XR-1	FW	Enabled	NODE	XR-10
VRF	VPN1			VRF	VPN1
NETWORK	170.0.0.1	IDS	Enabled	NETWORK	170.0.0.10

Figure 2.15: Policy Error after Network Change

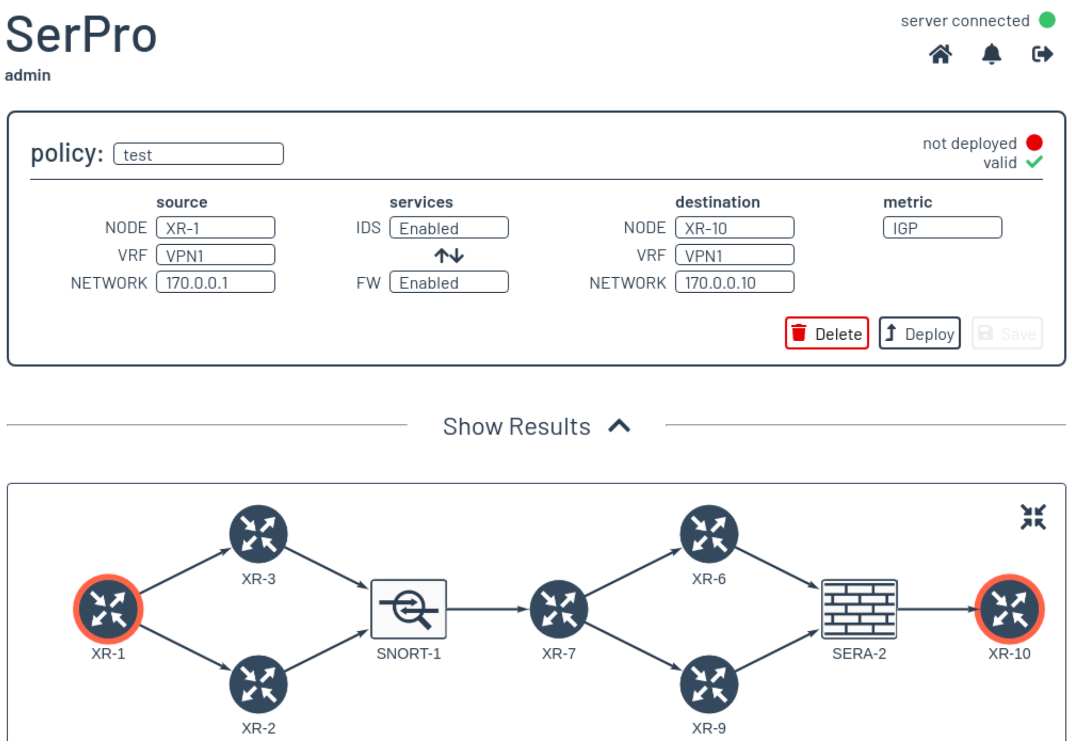
In addition, the reverse check is included in the application. As soon as a network change has occurred, which reenables elements, the reverse check must be triggered. This investigation has the task of checking if the policy can be put into a valid state again. So, if the procedure can classify the whole policy data as correct, the policy can be reenabled and put into a valid state again.

An example of such a case is the situation, if a router is reloaded. Once the router loses its connection, the application is informed, and the node is disabled in the application. The policy verifier check will detect automatically all policies that have installed this exact node as a source or destination node. Consequently, these policies will be in an error state until the node is enabled again. When the node is enabled again, the policies will be back in a valid state if all other elements are accepted too.

Policy Recalculation

Besides the policy validation, which ensures the correct state of all the policies at any time, it is also essential to always hold the correct result. This task is also included in the verification process.

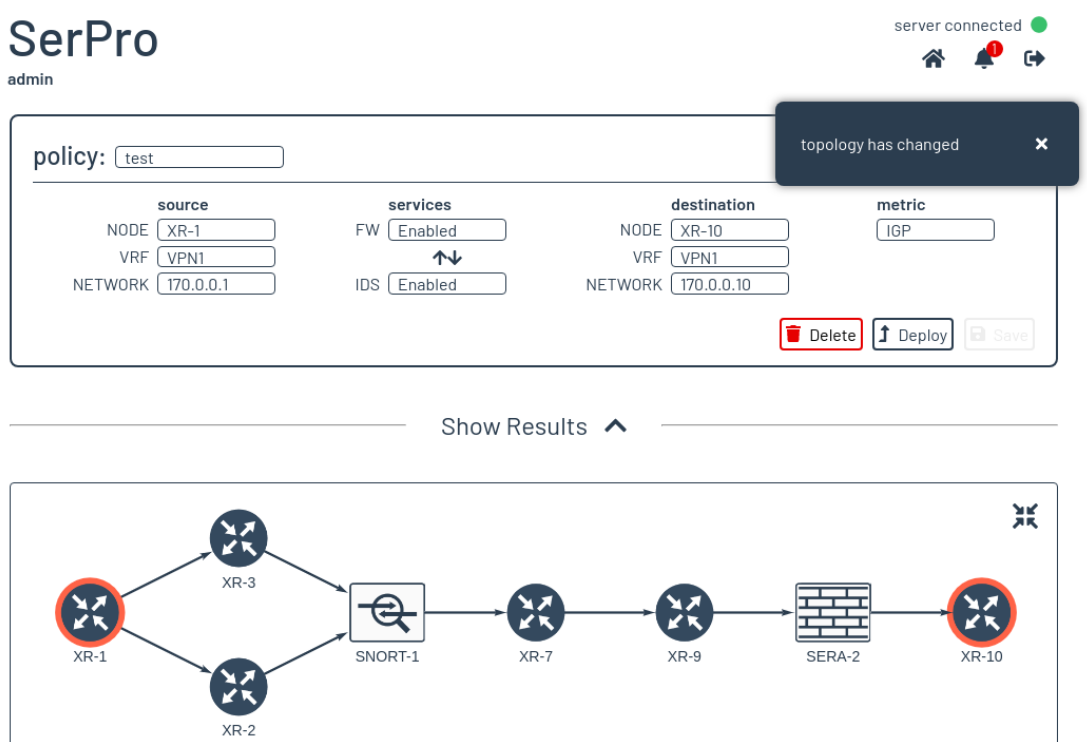
After the validation process is completed, the application detects automatically if there are any policies to recalculate. So, if any major changes happen in the network, the application will react to these and recalculate automatically the affected policies. This has several positive effects. The first positive point of the intuitive recalculation is that it ensures that the result of each policy is always correct. Hence, this means the user can always observe the up-to-date policy with the latest result information - including network path, total cost, and the correct SID list. An example of this process is visualized in figure 2.16 and 2.17. A link failure between *XR-7* and *XR-6* is detected immediately. The policy result is recalculated automatically, and hence the correct up-to-date version presented to the user.



policy: test

source		services		destination		metric
NODE	XR-1	IDS	Enabled	NODE	XR-10	IGP
VRF	VPN1			VRF	VPN1	
NETWORK	170.0.0.1	FW	Enabled	NETWORK	170.0.0.10	

Figure 2.16: Policy Result before Topology Change


SerPro
admin

server connected

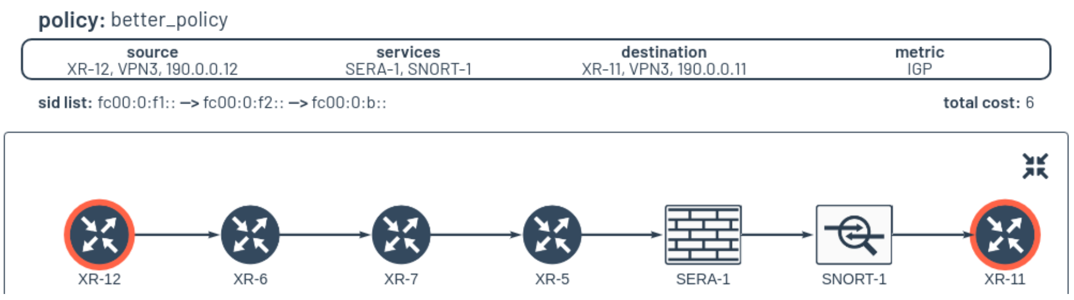
topology has changed

policy: test

source		services		destination		metric
NODE	XR-1	FW	Enabled	NODE	XR-10	IGP
VRF	VPN1			VRF	VPN1	
NETWORK	170.0.0.1	IDS	Enabled	NETWORK	170.0.0.10	

Figure 2.17: Policy Result after Topology Change

The notification about better paths is another advantage of this recalculation logic. After a topology change, the user can be notified about better path options. This status distinguishes that a new service combination is found, which has a smaller total cost than the current one. The user can now observe the changes and decide to accept the better path options. This case indicates that the old service chain is not the best anymore, but it is still working. Such a situation could happen if, for example, a new service is added to the network, which reduces the total cost of the policy. An example of this case is shown in figure 2.18 and 2.19

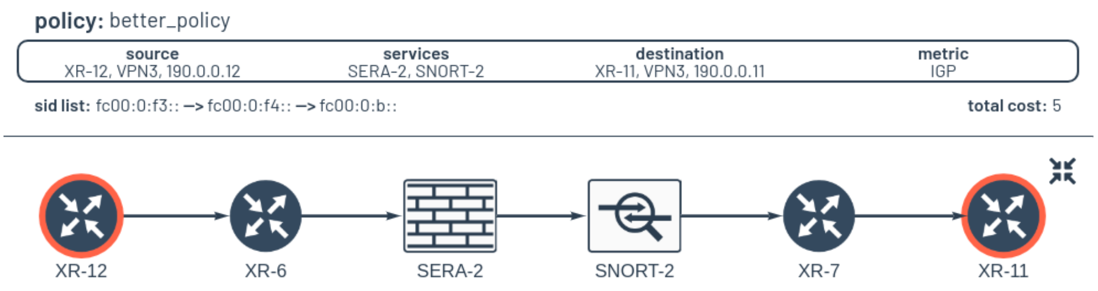

policy: better_policy

source	services	destination	metric
XR-12, VPN3, 190.0.0.12	SERA-1, SNORT-1	XR-11, VPN3, 190.0.0.11	IGP

sid list: fc00:0:f1:: –> fc00:0:f2:: –> fc00:0:b::

total cost: 6

Figure 2.18: Non-optimal Policy Result



policy: better_policy

source	services	destination	metric
XR-12, VPN3, 190.0.0.12	SERA-2, SNORT-2	XR-11, VPN3, 190.0.0.11	IGP

sid list: fc00:0:f3:: –> fc00:0:f4:: –> fc00:0:b::

total cost: 5

Figure 2.19: Better Policy Result

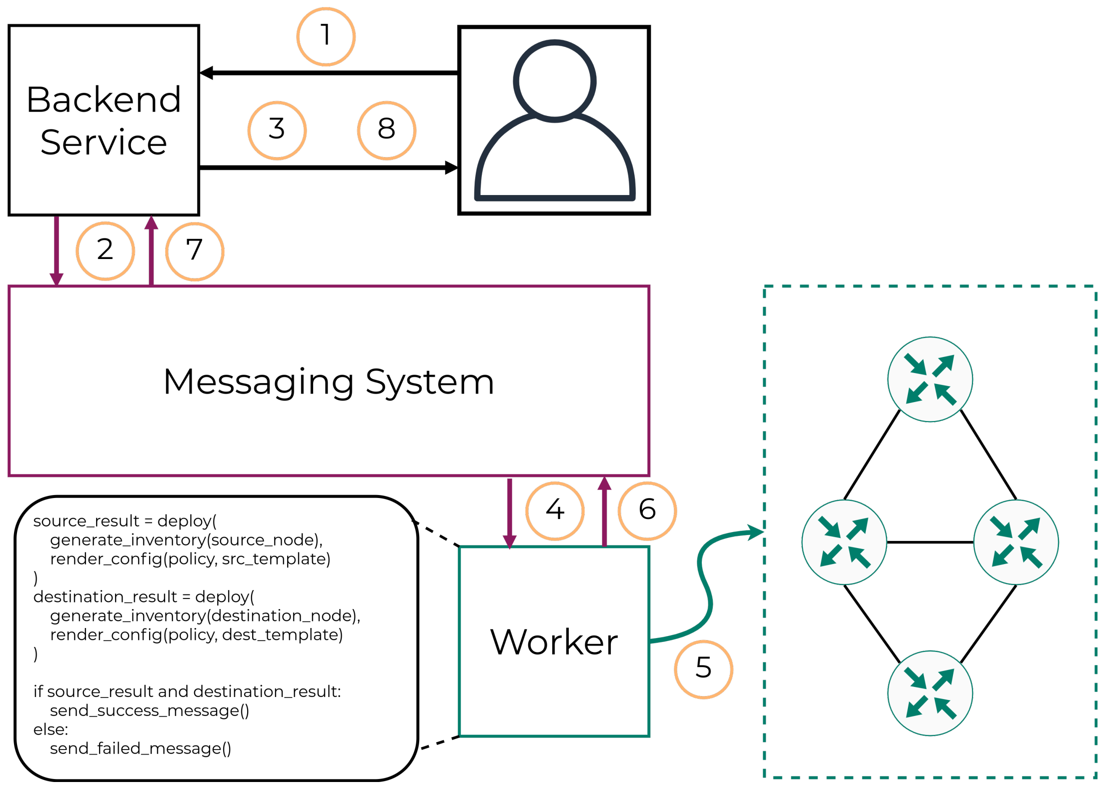
Thanks to this implemented logic, it is also possible to survive network outages. The application finds automatically a possible path if there is any. If, for example, the most suitable firewall system is disconnected, the next best firewall system is found automatically. The user has, therefore, always a working result, if there is any. This step was even improved: if the policy was deployed in the network and an outage influences the service, the better policy will be calculated and redeployed automatically. This behavior is necessary to ensure that traffic always gets the treatment, which the customer wants. It has to be mentioned that if the customer likes to support this feature, the different systems have the same configuration. If, for instance, the firewall system don't have the same rules configured, this could lead to traffic interruptions.

2.3.5 Policy Deployment

As already written in section 2.2.6, the created policies can also be written to the correct routers. For this purpose, a completely separate module was developed in the backend service. The complete process can be seen in figure 2.20. The individual steps are then briefly explained below.

ℹ Router Configuration

The entire policy configuration that is written to the routers is based on the automated steering of SRv6. However, since various traffic engineering options are not yet publicly available, we are forced not to publish the configuration in this thesis.



```
source_result = deploy(
    generate_inventory(source_node),
    render_config(policy, src_template)
)
destination_result = deploy(
    generate_inventory(destination_node),
    render_config(policy, dest_template)
)

if source_result and destination_result:
    send_success_message()
else:
    send_failed_message()
```

Figure 2.20: Deployment Workflow

The individual numbers marked with a yellow circle in figure 2.20 are examined and described in more detail below:

1. A customer can deploy a created policy via an explicit REST API action.
2. The backend service creates a job task to perform the full deployment asynchronously. By offloading the deployment, the backend is no longer blocked and can continue its work. Furthermore, the created job is stored in the messaging system and thus made available to the worker.
3. Once a deployment job is created, the backend sends a WebSocket message to the client that the policy deployment is being executed.
4. As soon as a job is in the dedicated queue in the messaging system, the worker is informed and can pick up and start its job.
5. The worker now creates an inventory for the source and destination node, which it needs to connect to the correct router. It also renders the respective config template for the source and destination router. Once, it has rendered the config, the worker deploys the config to the two routers.
6. Depending on the result of the deployment process, the worker sends a success or failure message to a dedicated result queue in the messaging system.
7. As soon as a new result is available in the result queue, the backend is informed and can process this result.
8. The backend now sends the user a processed result message via the WebSocket, allowing the customer to see whether the deployment worked or not.

2.3.6 Clustering

The former thesis proved that the best suitable paths can be quite complex and therefore badly comprehensible. A severe difficulty was detected if the connected network was extensive and there were not many services distributed. Therefore, it was decided that this problem should be solved at the beginning of the bachelor thesis. So, a method should be implemented to visualize the best available path in a more clear variant.

Similar to existing solutions it was observed to get inspiration on how to solve the problem. One solution, the Cisco DNA Center, implements a method to visualize graphs cleanly and therefore, serves as a model for the searched clustering solution. Such an example visualization can be observed in figure 2.21.

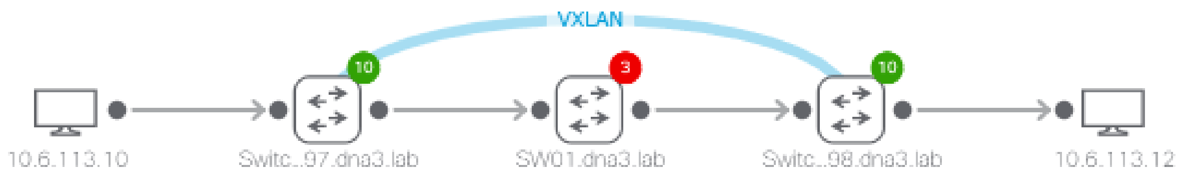


Figure 2.21: Graph Visualization in Cisco DNA Center

A clustering solution was introduced in this thesis, which visualizes the result path in a clean way. It was possible to maintain a graph that contains a simple design and omits more detailed information. Therefore, the intermediate nodes are consolidated into a cluster. The reason behind such a graph is to get a fast overview of the result. Especially, the most suitable service instances, which are included in the result could be detected effortlessly. Hence, the users have the possibility to detect rapidly if they are satisfied with the represented result. An example of such a graph representation can be observed in figure 2.22.

Moreover, the user also gets the chance to expand the graph and examine the entire graph. This type of graph includes the information about each node and the connection between them. It also visualizes the ECMP paths in a lucid form. So, a user can verify the full graph from source to destination and track the paths the packet will take if the policy is deployed into the network. An example of the entire graph is shown in figure 2.23.

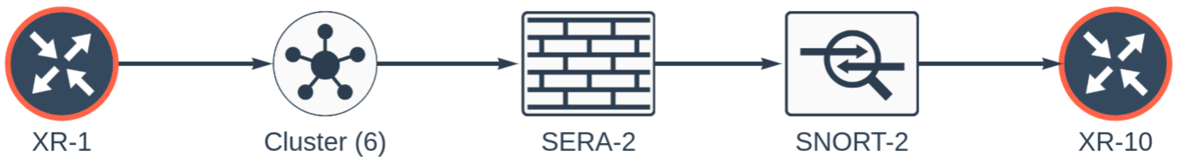


Figure 2.22: Clustered Graph Example

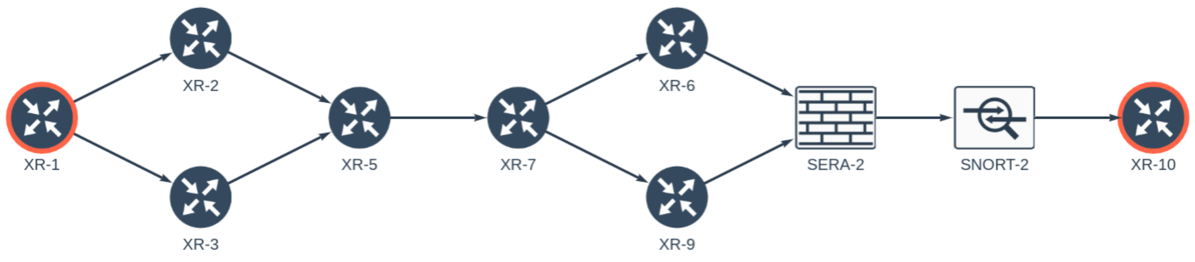


Figure 2.23: Expanded Graph Example

This difference between the two graphs not only helps the user to get a rapid or more detailed view. But it also allows the user to identify a non optimized service placement. If the services are not placed in strategic locations or the user has selected specific non-optimal service instances, it could lead to unclear graphs. Mainly, the reason is acyclic graphs, which are challenging to draw obviously. The clustering algorithm has therefore be improved to answer this hurdle. The large graph was linked with the information about the different sub-graphs. This addition helps to follow the paths better and highlight the different sub-paths. More information about the sub-paths can be found in section Calculation and Preprocessing whereas the sub-path highlighting is visualized in figure 2.24.

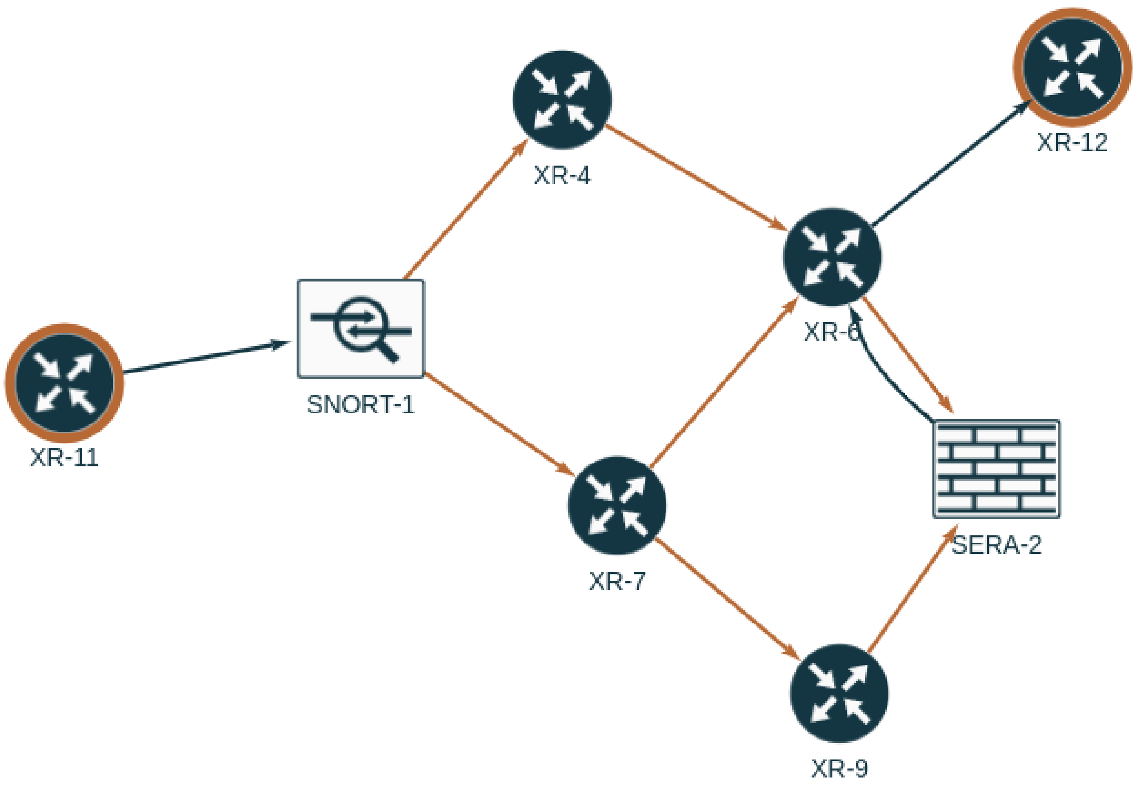


Figure 2.24: Highlight Sub-Paths in Graph

2.3.7 Service Management

General

The preliminary thesis revealed that the application has no information about the service instances of the network. Therefore the connection between the application and the service instances was completely manual driven and hence not optimal. Each service had to be manually introduced, updated, and deleted. It was mentioned that an improved service management technique should be introduced to maintain the existing service instances in the application in a more dynamic way. Thus, this new service management technique should focus on adding, updating, and deleting of services in the network and should be as automized as possible. In this bachelor thesis, an approach was introduced which takes care of these requirements. Subsequently, the method is described in more detail.

Implementation

The idea was to introduce a solution that is backed by the existing tools and techniques. Therefore, the best possible solution should operate with the help of the external software tool Jalapeño and the help of the applied IGP protocol. The basic idea was simple: announce the information over the IGP protocol and therefore make it available in the external system Jalapeño. As soon as the information is available in the graph database, retrieve the data with the help of the developed polling service and send it to the SerPro application. The significant advantage of this solution is that the application is notified automatically if there have been any changes. Hence, the solution should be able to identify new additions, updates, and also deletions of these related service prefixes and react to them.

The solution and the needed steps can be described in a few lines. The simplified process is visualized in figure 2.25.

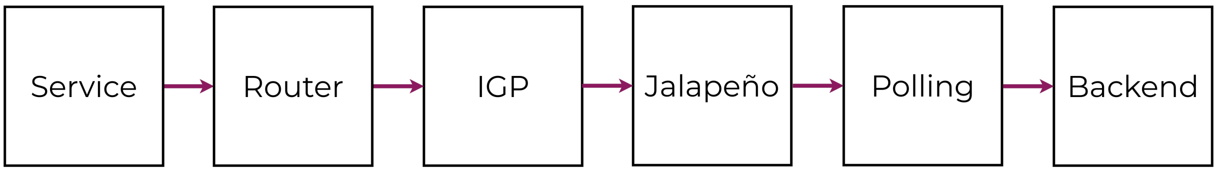


Figure 2.25: Service Announcement Chain

1. First, the service has to be installed and configured. The service must be configured with a special IPv6 prefix. These IPv6 prefix is later used as SRv6 micro segment (uSID) and therefore make the service treatment possible. In addition, the polling service is configured to retrieve these unique service prefixes and send them further to the backend service. It is to mention that it makes sense to introduce a well-planned address scheme for the different service prefixes announced in the network.

2. So that Segment Routing packets arrive at the service, the service prefix has to be reachable. Therefore, the router connected to the service must introduce a route to make the service prefix reachable in the network. So, the next following step is to create a static route to introduce reachability to the service.

3. In order that the service prefix is reachable in the whole network and also is available in the external system Jalapeño the static route has to be redistributed into the IGP. As soon as the service prefix is announced in the applied IGP protocol, the prefix information is made available in the graph database in Jalapeño. From there, the information can be

retrieved by the polling service and sent to the backend service. The backend service can save this information and make it available to the user.

4. To create a service instance in the application, the user must add necessary information to the service prefix. The announced service prefix can be assembled with the service name and the location information. The location information is given by adding the node/router over which the service is available. As soon as this information is added to the service prefix, the service can be created and be applied. The service is now installed in the network and available in the application.

Chapter 3

Conclusion

In the following chapter, a review of the achieved use cases and a critical discussion about things that could have been done better will be given. In addition, there is also a digression about further improvements in the future.

3.1 Retrospective

The following sections discuss the various use cases that have been achieved and those that have not. Afterwards, there will be a discussion about possible improvements and things that could have been done better.

3.1.1 Use Cases

To better distinguish the different use cases, different colors and symbols are used. All Minimum Viable Product (MVP) use cases are displayed in green. The optional non-MVP use cases are shown in purple. In addition, use cases that could be achieved are shown with the following symbol ✓ in contrast to use cases that could not be achieved, shown with the following symbol ✗. Each use case begins with the user story from the section 4.1 and is marked with an *"italic font"*.

✓ UC01: View Topology

"As User, I overview the topology of the network connected to the application."

Due to the structured data provided by the API, the frontend can display the topology without any problems. This use case could be achieved entirely. Due to the standardized API, which was kept as simple as possible, changes can be introduced without significant adjustments.

The user is automatically informed about updates in the topology; through the update messages, the frontend can automatically load the latest topology, allowing the user always to overview the actual topology.

✓ UC02: CRUD Segment Routing Traffic Engineering Policy

"As a User, I manage an SR-TE Policy to steer the traffic according to its characteristics on the most appropriate path through the network and enable the most optimal service treatment."

The backend API allows the user to create a policy; the backend validates all entries for correctness and gives the user a message about what was not correct. The frontend can provide the user a form, which is dynamically updated through API requests, this allows them to create a policy as smoothly as possible; the frontend can make various dynamic requests based on the user's input to the backend. That allows the user to see all available options automatically.

After creating a policy, the user can view the result in a clustered or expanded view. This use case was achieved and allows the user to create, read, update and delete policies dynamically.

✓ UC03: Structure Data

"As User, I need structured data, in order to have an appropriate presentation of the topology and to create Segment Routing policies."

The backend provides an API that has already implemented the complete logic. That allows the frontend to make queries without having to implement any logic itself. Furthermore, the topology displayed is wholly created in the backend as a result and divided into a clustered and full graph. That allows the backend to assemble the graph graphically.

This use case could be completely fulfilled. Intensive discussions were held with the INS's frontend developers to determine the most significant complexity and how the backend can help transfer it from the frontend to the backend.

✓ UC04: Define Service

"As User, I define a service instance and its topology location, in order to use this service instance in traffic engineering policies."

Services can be defined directly in the admin portal by authorized users. The detected service prefixes will be listed there, so that a user can only link the service prefix with the type and the name of the service, which he wants to declare in the topology. When the service is defined, it will be displayed in the topology and can be used for creating policies.

Using the admin portal to create the services, the user gets a fast and straightforward solution to declare services. Thus, this use case is fulfilled and completed by this option.

✓ UC05: Calculate Paths

"As User, I get the most appropriate path after generating a Segment Routing policy."

By introducing preprocessing, the calculated paths can be returned even faster. By the possibility to scale the backends, many requests can be answered at the same speed. The calculation of paths in a network with thousands of nodes succeeds through the preprocessing even faster and more stable.

Through the use of preprocessing, this use case is more than satisfied.

✓ UC06: Deploy Link Metric Algorithm Policy

"As User, I deploy the created Segment Routing policy, which includes a link metric, on the appropriate nodes."

Policies can be deployed directly to the router. The deployment is outsourced from the backend, so that it can be executed asynchronously. The user will be informed about every status change and will know, if the deployment was successful or not. A deployed policy can, of course, also be reverted again. For the configuration, the link metric algorithm is used.

The deployment use case for the link metric algorithm could be fully covered. Policies can be deployed reliably and also reverted again.

✓ UC07: Login

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

Unauthenticated users do not get access to the application. Instead, a login procedure secures all-access points including frontend, API, admin portal, and the WebSocket connection.

By introducing an authentication option with JWT, this use case is fully met.

✓ UC08: Manage Recalculations

"As User, I check the affected policies after a topology change to take action and to bring the policy to a desired state."

During development, this use case proved to be the most complex use case of the entire application. All policies created are continuously checked for validity after each topology change. Policies that are no longer valid are set directly to an error state, and an error message is sent to the user. Policies that are still valid, but the path has changed, are automatically updated if the segment list has not changed. Policies that are still valid but have a new best segment list are put into a status, that tells the user that a better path could be found. The user can then manually view this path and accept it, if he wants to. If policies are affected, that are deployed, then these are automatically redeployed if this is possible.

This use case could be wholly fulfilled and makes the whole application completely dynamic, as it can now react automatically to topology changes and take measures.

✗ UC09: Deploy Delay/TE Algorithm Policy

"As User, I deploy the created Segment Routing policy, which includes a delay or TE metric, on the appropriate nodes."

This use case could not be completed. That is, because the current software version of the external software system Jalapeño does not provide delay/Traffic Engineering (TE) metric information. However, the application could handle this additional deployment type without significant changes.

Although this use case could not be fulfilled, the application can support this in the future without any problems. Furthermore, changes in the application to support this use case are possible without major adjustments.

✓ UC10: CRUD Roles and Users

"As Admin, I create, read, update, and delete the roles and users of the application."

The user gets the possibility to create and customize the different users and groups via the admin portal. Various fundamental groups and permissions are already created at the start of the application if they do not already exist.

This use case can be fully covered. Through the Django framework, which already brings a group and user management possibility, the authorized users can adjust all authorization levels.

✓ UC11: Handle Permissions

"As User, I only perform actions for which I am authorized."

The backend checks all API requests to see if this user is authorized to execute them. In order to achieve this, the backend uses the tools already provided by the Django framework and extends them to meet the unique requirements.

Users can only use the functions of the application if they are authorized to use them. This use case is, therefore, achieved.

3.1.2 Discussion

In the following sections, different topics of the bachelor thesis will be discussed critically.

Data Retrieval

Polling data from a database at a fixed interval is not an optimal solution. The polling service has shown to be very stable and reliable in this work. But using a push mechanism would improve the overall handling of the updates. The polling service would only have to poll the data once during the initial load using a push mechanism. Afterwards, it would then receive a notification about updates via push, which it could then retrieve individually from the database. That would increase the general performance of the polling service, and updates could be detected and processed faster.

Configuration Management

Policies are currently written directly to the affected routers; this works perfectly assuming that no one deletes manually a policy on these routers. If a manual intervention on this policy configuration would occur, then the application would no longer know the status of the policy. Consequently, it would lead to inconsistent state. This problem is not only with this application but in general, when one starts to automate things. In the future, one would have to make sure that the application could recognize manual interventions on the routers and adjust the policies accordingly.

Backend Framework

During the project, it turned out that the Django backend framework does not meet the exact requirements. Django offers many functions out of the box, which is often a relief and can save much work. However, because Django provides so much and does it itself, the developer also has less control of what he can do himself. Through this mass of functions that the framework offers, the whole application is also inflated a bit. In the future, it should be considered whether a lighter framework might be better to improve the application even further and give developers more control over the code.

3.2 Outlook

This section is intended to give an outlook on things that could be improved in the application and features that could be added.

3.2.1 Improvements

In the following, various improvements that can be made to the application, will be discussed in detail. Most of the improvements arose from the points mentioned in the section Discussion.

Update Processing

The data should no longer be fetched from the external system using the polling method. It would be possible to hook into an already existing Kafka topic in the external system and thus be notified, if there are updates. The other possibility is to change the polling service to communicate with a planned Jalapeño API gateway via gRPC. This gateway is not yet developed but is currently in the planning phase.

Deployment

The deployment module should be supplemented with additional options. As an example, the users should be able to choose, which deployment type they prefers. Furthermore, the deployment module should be adapted to support the Cisco IOS-XR Service Layer API. In addition, Cisco Network Service Orchestrator (NSO) should also be supported since this tool is already in use by many customers; therefore, it makes sense to run all configurations of the routers via one tool.

Microservice Architecture

The backend has grown a lot in this work, and many different individual modules have been developed. Since these modules can function individually, it would make sense to extract them into isolated microservices. That would increase the overall maintainability. In addition, individual microservices could be scaled more efficiently, making the application more stable and available.

Service Management

Service management is to be improved. In consultation with the developers of the external software system Jalapeño, a new processor should be written, that writes automatically services to a dedicated collection in the graph database of Jalapeño. That would allow SerPro to query this collection directly and not find a way to get the service data itself.

3.2.2 Innovations

Subsequent innovations can take the application to the next level and provide the user with an even better experience.

Versioning

Since policies have a significant impact on the traffic, that network A takes to network B, versioning should be introduced in which a user can see when and where changes are made which affects a policy. Additionally, it should be possible to restore earlier versions if they are still valid. That would give the user more control when the application is used in a productive environment.

Path Selection

The users should decide which path a policy should take, if there are several paths with the exactly the same costs. That would give the users even more control over the policy and allow them to look at all the policies in advance and decide which path the policy should take.

Diversity

A new policy character trait should be introduced. The diversity metric should show how many policies flow over a single service instance and help to utilize the services evenly. A user could use the diversity metric to determine on which path the services are less used and configure this path. That would make the entire application even more intelligent and support the user in choosing the best policy.

Advanced Policy Options

In order to be able to control the policy even better, different policy properties are to be added. For example, the user should decide that the policy should not be routed via explicit countries. Further options would be that the policy should only be routed via encrypted links or that new metrics should be used. Also, those policies should enforce a load-balancing between two services. This innovation would be a good addition and would give the users even more options to control the policies exactly to their wishes.

3.2.3 Further Thoughts

During the development of the SerPro application, other ideas came up, that would add value. These ideas are not explicitly related to this bachelor thesis but are ideas which would give additional value in the area of Segment Routing. In the following, a few of these ideas are briefly explained.

Location Tags

During the development of the work, the question came up time and again on how one could place the different routers in the frontend in a meaningful way. The answer was relatively straightforward; one would need a location where it would be possible to place the router exactly on a world map. The problem is, that the used router images (IOS-XR) do not offer any location configuration options. The idea came up, that this could be done with Simple Network Management Protocol (SNMP), because there a location is supported. In order to make this data available in the external software system, a new processor would have to be written, which reads this location information out of the SNMP packages. The data would then be available in the graph database of the external system and could be fetched and used by applications such as SerPro. That would be a relief, especially in the frontend when topologies with several thousand routers are to be displayed.

Central Service Management

All SR-aware service instances currently in use are entirely independent and therefore also configured individually. Central management would simplify the complete configuration and would also allow the synchronization of two service instances. That would have the significant advantage of then doing a load balancing of the traffic over two different service instances. Furthermore, it would assure that a second service could take over that service configuration in the event of a service failure.

Part II

Project Documentation

Chapter 4

Requirement Specification

This chapter includes the entire requirement specification, which was defined together with the industrial partner as well as the supervisor of this thesis. It is composed of two parts.

The first part outlines the different use cases, which have been adapted from the earlier project thesis. Particular emphasis was placed on defining the use cases to get the maximal benefit for a potential customer and that the solution could be used in a production scenario.

The second part lists the non-functional requirements which define the rules, which the application should meet.

Since this Bachelor thesis is a follow-up of a project thesis, the use cases were partly taken over and adapted for the sake of completeness. The original use cases can be found in the thesis Service Chaining Path Calculation[3].

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 MVP only refers to the Main Scenario within a Use Case. The extensions are always optional and will be implemented if enough time will be available.

4.1.1 Actors

Actor	Description
Anonymous	The Anonymous role stands for an actor which is not yet authenticated in the system. The entity has to prove its identity to the system to be authenticated and use the application functionalities.
User	The User role stands for an actor which is authenticated in the system. The entity can perform actions which it is authorized for.
Admin	The Admin role stands for a user, which has, compared to the non-admin user, additional permissions to perform supplementary operations within the application.

Table 4.2: Actor Description

4.1.2 Use-Case-Diagram

The following use case diagram should provide an overview over the tasks which should be accomplished in this thesis. The follow-up sections should be considered for further information about the specific use cases .

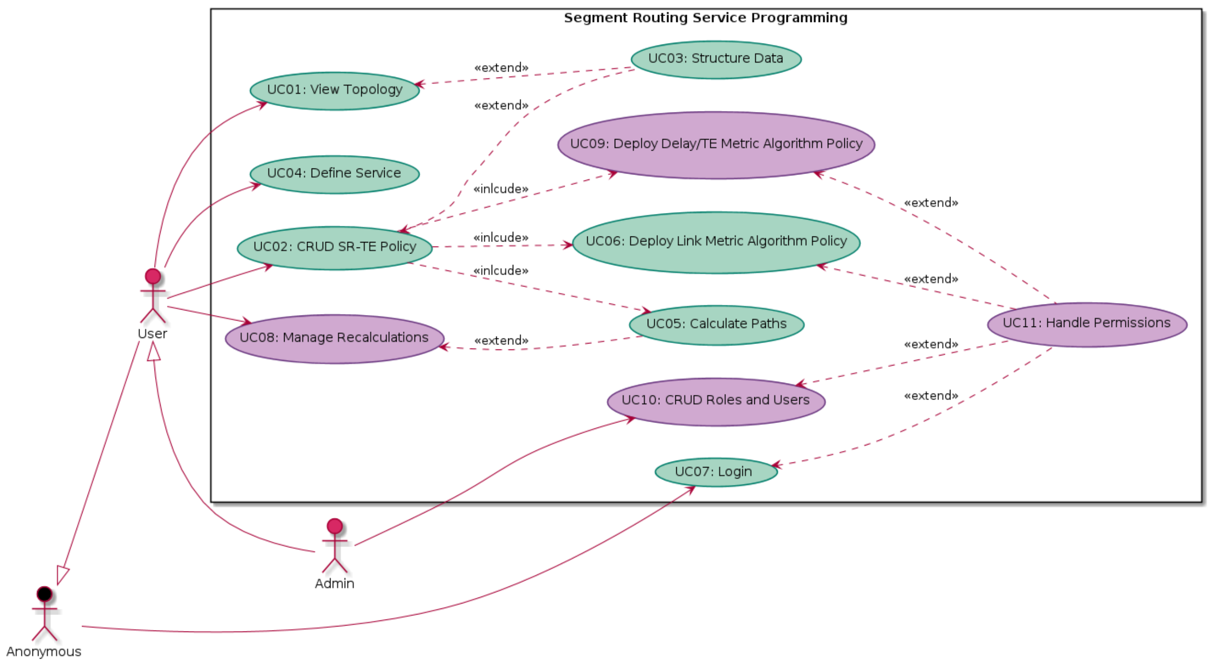


Figure 4.1: Use-Case-Diagram

4.1.3 Use-Case-Description

All the use cases are described in user stories according to the defined template found below. The user stories have always the same format, which makes it easier to read and understand. The motivation is omitted when it does not give the reader an additional value.

To declare the different use cases in more detail. Work accroding to Larman[20] was done. Through the use of this standardized way, the use cases descriptions follows international standards for the functional requirement engineering.

User-Story 00 - User Story Template

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

UC01: View Topology

User-Story 01 - View Topology

As User, I overview the topology of the network connected to the application.

Primary Actor	User
Overview	As a User, I look at the present network topology with its nodes/routers, services, and connections.
Stakeholders and Interests	User: wants to get the information of the interconnections and arrangements of the network nodes and services.
Preconditions	• The User is logged into the application, and has the required permission to do so.
Postconditions	• The User can regard the network topology in the form of a graph. The vertices are either nodes or services, and the edges the connection between the network nodes. The distinction between the node types is made with separate icons.
Main Success Scenario	1. The User wants to overview the topology of the present network. 2. The User examines the network topology.
Frequency of Occurrence	As often as required

Table 4.3: UC01: View Topology - Fully Dressed Description

UC02: CRUD SR-TE Policy

User-Story 02 - CRUD Segment Routing Traffic Engineering Policy

As a User, I manage an SR-TE Policy to steer the traffic according to its characteristics on the most appropriate path through the network and enable the most optimal service treatment.

Primary Actor	User
Overview	**Create and Update** As User, I select the necessary parameters to create or update an SR-TE Policy to steer the traffic according to its characteristic on the most appropriate path through the network and enable the most optimal service treatment. Those parameters are: • Source/Ingress Information (Node, VRF, Network) • Destination/Egress Information (Node, VRF, Network) • Metric/Algorithm Information • Services (spefic or general) **Read** As User, I read the parameters and the result of an already created SR-TE policy **Delete** As User, I delete an existing SR-TE policy to revoke the SR-TE policy settings and impacts.
Stakeholders and Interests	User: Wants to manage SR-TE policies to influence the traffic steering and service handling within its network: Network packets are steered from source to destination through the most appropriate path and are treated by the desired and most suitable service instances on their way.
Preconditions	• The User is logged in the application. • The User is authorized to do at least one of the create, update, read or delete SR-TE Policy operation. • The necessary data is available - especially at least one service has been added to the application • Action Update/Delete: The policy is not deployed in the network

Postconditions	**Create** • The User gets the result, which shows the best path – according to the input parameters – taken through the network and includes the information on which services handle the traffic through its travel. Furthermore, specific information like the Segment (ID) list or the total cost is retrieved. • The information about the SR-TE Policy is saved in the application. • If the use cases *UC06* and/or *UC09* are realized, a button in the UI is shown, which allows writing the configuration to the appropriate router. As a consequence, the traffic is steered and processed by the services as desired. **Update** Same like in *Create*, but with the updated values. **Read** The User gets the input and result information from a specific SR-TE Policy. **Delete** The User receives the notification that the deletion was successful. The corresponding SR-TE Policy is removed from the application.

Main Success Scenario	**Create** 1. The User wants to create an SR-TE policy to steer traffic from the source to the destination on a specific path in which the most suitable services process the packets. 2. All possible source nodes are shown. A node is considered as a possible source node if the router is PE and has customer networks assigned (L3VPNv4 Prefix). The user selects the source node from the available list by selecting the router's hostname. 3. All possible source VRF's, which have been configured on the previously selected source node, are shown. The user selects the source VRF from the available list by selecting the VRF name. 4. All possible source networks, which belong to the previously selected source VRF, are shown. The user selects the source network from the available list by choosing a specific network address 5. The user defines the service chain by selecting general service types (FW or/and IDS), services instances (e.g., SNORT-1), or a mixture of both (FWs and SNORT-1). 6. All possible destination nodes are shown. A destination node has the same requirements as a source node PE and L3VPNv4 Prefix. Besides, the destination has to possess at least one VRF where one export tag matches the selected source VRF's export tags to be valid. The user selects the destination node from the available list by selecting the router's hostname. 7. All possible destination VRF's, which are configured on the previously destination node and meet the necessary criterion, are shown. Only the VRF's with at least one export tag with the source VRF in common are valid and shown. The user selects the destination VRF from the available list by selecting the VRF name. 8. All possible destination networks, which belong to the previously selected destination VRF, are shown. The user selects the destination network from the available list by choosing a specific network address. The source and destination network don't have to be the same. 9. The user selects IGP as a metric, which is used for the calculation of the most suitable path. 10. After the calculation, the first found path with the smallest metric is returned. The result includes the links, nodes, and services involved in the most suitable ECMP from source to destination, which steers traffic through the appropriate service instances. Furthermore, SR-TE relevant information like the Segment list and the total path cost is returned. 11. The user can inspect the result, which shows the way through the network from the source to the destination over transit nodes as well as service instances. Besides, the user can examine the Segment Routing Traffic Engineering (SR-TE) relevant information like the Segment List and the total cost of the path.

	Update 1. The User wants to adjust an existing SR-TE policy. 2. The User selects an existing SR-TE Policy and adjusts the input parameters mentioned in the *Create* scenario steps 2 - 8. 3. The User triggers a recalculation of the SR-TE Policy and thus an update. 4. The User does the same action as mentioned in the *Create* scenario steps 10 - 11. **Read** 1. The User wants to receive information about an existing SR-TE policy. 2. The User selects an existing SR-TE policy and examines input as well as result information of the appropriate policy. **Delete** 1. The User wants to delete an existing SR-TE policy. 2. The User selects an existing SR-TE policy . 3. The User confirms the deletion of the SR-TE policy. 4. The SR-TE policy is removed from the application. If use case *UC06* and/or *UC09* are implemented, the relevant configurations will be removed on the relevant nodes.
Extensions	*a During the actions, an error has occurred. 1. A faulty action has lead to an application error. 2. The application returns the error with a meaningful description. 3. The User can retry the action. 9a The user selects TE as a metric, which is used for the calculation of the most suitable path. 9b The user selects Delay as a metric, which is used for the calculation of the most suitable path. 10 There is more than one result with an equivalent metric. 1. The calculation has returned several most suitable paths with equivalent metrics. 2. The User examines all results, including the appropriate paths, which have all the same metric count. 3. The User selects and examines its desired path from the best paths and takes further actions.
Frequency of Occurrence	As often as required

Table 4.4: UC02: CRUD SR-TE Policy - Fully Dressed Description

UC03: Structure Data

User-Story 03 - Structure Data

As User, I need structured data, in order to have an appropriate presentation of the topology and to create Segment Routing policies.

Primary Actor	User
Overview	The User wants to check the topology visualization and CRUD the policies. Therefore he needs data in an appropriate form in the frontend application, to achieve this, structured data is required from the backend for the frontend.
Stakeholders and Interests	User: The User wants to CRUD the Segment Routing policies and overview the network topology.

Table 4.5: UC03: Structure Data - Casual Description

UC04: Define Service

User-Story 04 - Define Service

As User, I define a service instance and its topology location, in order to use this service instance in traffic engineering policies.

Primary Actor	User
Overview	As a User, I define the different service instances and their topology location.
Stakeholders and Interests	User: Wants to define different service instances and their topology location, in order to use this instances to create new policies.
Preconditions	• The User is authenticated within the application. • The User has the necessary permission to define a service instance. • The services have to be configured and deployed on the network to influence the traffic with the service application.
Postconditions	• The new service instance is present in the application and can now be used to create new policies.
Main Success Scenario	1. The User wants to define a new service instance and its topology location. 2. The User can select the topology location to which node the service instance is connected and configure service instance-specific configuration data to use the application for the policy generation.
Frequency of Occurrence	As often as required

Table 4.6: UC04: Define Service - Fully Dressed Description

UC05: Calculate Paths

User-Story 05 - Calculate Paths

As User, I get the most appropriate path after generating a Segment Routing policy.

Primary Actor	User
Overview	The User gets the most appropriate path after generating a Segment Routing policy.
Stakeholders and Interests	User: The User wants to get the most appropriate path for the generated Segment Routing policy.
Preconditions	• The User is authenticated within the application. • The creation or update of a Segment Routing policy was successful and supplies appropriate data.
Postconditions	• The most appropriate path is calculated.
Main Success Scenario	1. The User wants to get the most appropriate path according to a previous generated Segment Routing policy. 2. The most appropriate path will be calculated and returned.
Extensions	2a The return of the most appropriate path is influenced by the diversity of the different services instances. 1. The most appropriate possible paths will be calculated. 2. When there are more paths with equivalent metric numbers, the diversity of the different service instances will be considered. The diversity metric means, that all Segment Routing policy paths will be considered and it will be calculated how many paths go through each service instances. 3. For each potential best path the diversity value of all affected service instances will be calculated. 4. The path that has the least total diversity value will be returned.
Frequency of Occurrence	As often as required.

Table 4.7: UC05: Calculate Paths - Fully Dressed Description

UC06: Deploy Link Metric Algorithm Policy

User-Story 06 - Deploy Link Metric Algorithm Policy

As User, I deploy the created Segment Routing policy, which includes a link metric, on the appropriate nodes.

Primary Actor	User
Overview	The User can deploy the created Segment Routing policy, which includes a link metric, on the appropriate nodes.
Stakeholders and Interests	User: The User wants to deploy the created Segment Routing policy, in order to steer the traffic accordingly.
Preconditions	• The User is authenticated within the application. • The User has the necessary permission to deploy a traffic engineering policy. • The Segment Routing policy was successfully created.
Postconditions	• The Segment Routing policy is deployed on the appropriate nodes. • The traffic is steered according to the deployed policy.
Main Success Scenario	1. The User wants to deploy the previous created Segment Routing policy to the appropriate nodes. 2. The User can trigger manually the deployment of the policy. 3. The policy will be automatically deployed on the appropriate nodes. 4. The policy deployment was successful 5. A message is shown, that the policy is successfully deployed. 6. The policy is marked as deployed.
Extensions	4a The policy deployment was not successful. 1. An error message is shown, that the policy was not deployed successfully. 2. The policy is set into an error state and marked to be resolved manually.
Frequency of Occurrence	As often as required.

Table 4.8: UC06: Deploy Link Metric Algorithm Policy - 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 functionalities.

UC08: Manage Recalculations

User-Story 08 - Manage Recalculations

As User, I check the affected policies after a topology change to take action and to bring the policy to a desired state.

Primary Actor	User
Overview	The User can check the policy, which was affected after a topology change, and can take further actions to bring the policy to a desired state.
Stakeholders and Interests	User: The User wants to have the opportunity to check the affected policy after a topology change, and take further actions to bring the policy to a desired state.
Preconditions	• The User is authenticated within the application. • The User has the permission to take action on an affected policy. • The application has found some non-optimal Segment Routing policies after a topology change.
Postconditions	• The Segment Routing policy is in the desired state.
Main Success Scenario	1. The User wants to approve manually the affected policy. 2. Icons and colors indicates an affected policy. 3. The User can regard the new and old result in the topology. 4. The User can approve manually the policy with the new result. 5. If the policy was deployed in the network, the new policy is redeployed on the devices.
Extensions	*a The user will not approve the better policy. 1. The application will not make any changes to the affected policy. 2. The affected policies will stay marked as affected until they have been approved.
Frequency of Occurrence	After each topology change, which leads to an affected policy.

Table 4.9: UC08: Manage Recalculations - Fully Dressed Description

UC09: Deploy Delay/TE Algorithm Policy

User-Story 09 - Deploy Delay/TE Algorithm Policy

As User, I deploy the created Segment Routing policy, which includes a delay or TE metric, on the appropriate nodes.

UC10: CRUD Roles and Users

User-Story 10 - CRUD Roles and Users

As Admin, I create, read, update, and delete the roles and users of the application.

UC11: Handle Permissions

User-Story 11 - Handle Permissions

As User, I only perform actions for which I am authorized.

4.2 Non-Functional Requirements

For the Non-functional Requirements the FURPS+[5] principle was used.

4.2.1 Functionality

Suitability

By using a login, the application should be protected from unwanted use by non-authorized entities.

Accuracy

The application should work with the latest data from the Arango database at all times. The up-to-date data always allows each policy to be created or deployed using the correct data.

Interoperability

The application should work with the Cisco developed application Jalapeño (version-hash: `f951290185a6b5b8244f7fce100e5369cffa0dbe`). The topology data used comes from Cisco IOS XR devices on which SRv6 is configured. Configurations are written to the same devices which are in the same topology.

4.2.2 Usability

Understandability

Existing policies should be easy to view and understand. Using colors and icons (e.g., like a traffic light system) a policy's status should be available at a glance.

Operability

Through the targeted use of descriptions and icons, a user should be able to edit, create or delete a policy without further assistance.

4.2.3 Reliability

Availability

The application should be available 99% of the time.

Recoverability

The complete application should be developed according to the Cloud Native[9] standard. By correctly applying this standard and deploying on a Kubernetes cluster, redeployments after a failure should be possible without further manual intervention.

Fault Tolerance

Incorrect input values should not affect the functionality of the application. If incorrect values arrive at the backend, they should be intercepted directly and processed with error handling.

4.2.4 Performance

Capacity

The application should be able to handle a network topology of up to 1000 routers. Overall, it should be designed for two different service types (Firewall (FW) and IDS).

Time behavior

Once the Arango database has processed topology changes, it should take no longer than one minute for the application to fetch the new data, process it, and make it available to the various services. This characteristic only obtains topology changes and not the initial loading of the topology data at the application startup.

4.2.5 Scalability

The application should be utterly scalable through the usage of Cloud-Native development rules. If more or fewer pods are needed, it should be easy to scale them up or down.

4.2.6 Maintability

Analysability

The application should include a customizable log level. The log should be written to the standard output by applying Cloud-Native development.

4.2.7 Traceability

All code changes and container deployments should be tracked using the CI and versioning tool GitLab. Versioning should ensure easy traceability and documentation.

Chapter 5

Domain Analysis

This chapter serves as an entry point into the domain. It analyses the present domain of the Segment Routing Service Programming environment. Therefore a Domain Model is included which visualizes the different logical elements in the area. The Administrative Concepts explains further the different elements and explains their purpose.

In the previous work SerChio a complete analogy was created to understand the concept of Segment Routing and introduce the reader into the domain. Furthermore, the project thesis also includes an introduction to the Segment Routing technology. The reader is suggested to look into these sections if there is no fundamental understanding of Segment Routing.

5.1 Domain Model

The figure 5.1 shows the different logical objects and how they are related to each other. It should help to give an overview of the different parts of the SerPro application. The individual elements are then described in more detail in the section Administrative Concepts.

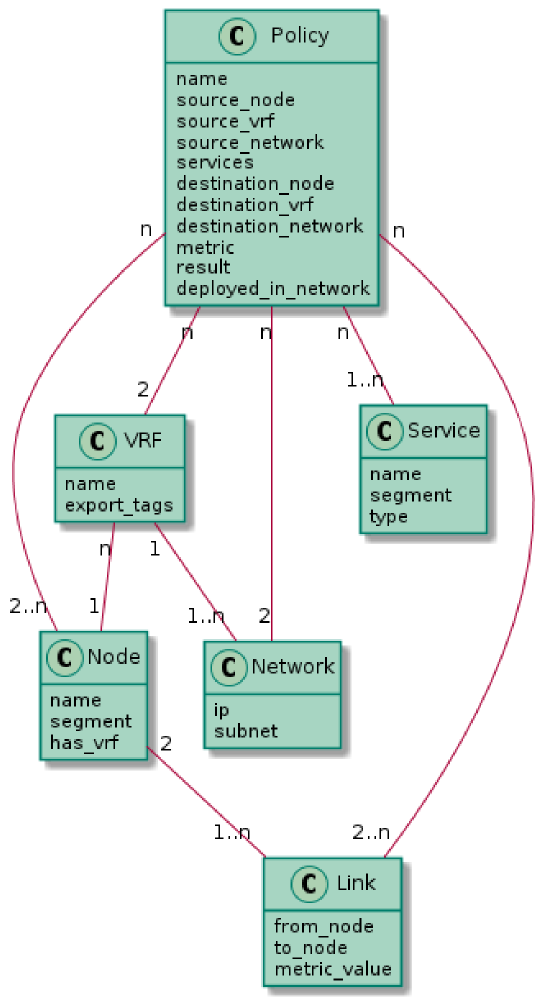


Figure 5.1: Domain Model

5.2 Administrative Concepts

This section describes the different parts of the Domain Model in detail. An understanding of the administrative concepts is assumed to understand the different parts of the developed application.

5.2.1 Policy

The *Policy* is the central element in the domain. The *Policy* contains all the information responsible for creating a steering Policy in the network and thus steering the traffic according to the user's requirements. A *Policy* includes different elements: A `name` is needed to differentiate between the other policies in the system. Furthermore, information about the origin is necessary. The `source_node`, `source_vrf`, and `source_network` refer to the appropriate elements of the domain. These elements describe where the traffic is derived from, which should be steered. The opposite information is found in the `destination_node`, `destination_vrf`, and `destination_network`. These elements describe the target of the traffic to steer. Because a *Policy* element always needs a source and destination information, the following relations can be formed: a *Policy* always includes exactly two *VRFs* and two *Networks* and at least two *Nodes*. Since the goal is to steer traffic over the most suitable `services`, at least one *Service* is associated with a *Policy*. Each *Policy* contains `metric` information. This information describes which `metric` was used to calculate the most suitable path for the *Policy*. A *Policy* is always assembled with the calculated `result`. The `result` includes information about the best paths like the actual way, the total cost, and necessary Segment Routing information. Therefore it includes information about which links are used and which intermediate nodes the traffic is flowed through. Hence, the relations have to be made to the *Node* and the *Link*. Additionally, the result information is used to create automatically a *Policy* in the network which enforces the steering. The `deployed_in_network` information informs if the *Policy* is written down to the nodes or not.

5.2.2 Node

A *Node* refers to an essential waypoint in the connected Segment Routing network. In this domain, the nodes are represented by Cisco IOS-XR routers. Each router has a `name` assigned to differentiate between the other *Nodes* in the domain. Also each *Node* has a unique `segment` assigned which is needed for the *Policy* result. This domain distincts between intermediate nodes and PE nodes. The PE nodes can serve as a source or destination node in the policy and have a *VRF* assigned. Nodes, which have not set the `has_vrf` are called intermediate nodes and do not fulfill the PE characteristics. Therefore they are only needed to forward the packets to the next waypoint in the path. A *Node* can occur in zero or more *Policies*. Additionally, a *Node* has one or more *Links* assigned to have connectivity in the Segment Routing domain.

5.2.3 VRF

A Virtual Routing and Forwarding (VRF) is an essential construct to enabling connectivity between customer networks. A *VRF* is identified by a `name` and has one or several `export_tags`. The `export_tags` are crucial to ensure the reachability to another *VRF*. At least one of the `export_tags` have to match with another *VRF* to ensure that the attached *Networks* have connectivity. So, a VRF has at least one *Network* assigned. Besides, a *VRF* is configured on exactly one *Node*. In a *Policy* there belong exactly two *VRFs*: the source and the destination VRF.

5.2.4 Network

The *Network* represents the customer Local Area Network (LAN). It contains an `ip` address and the according `subnet`. It is the most granular element in a *Policy* and is needed twice. Once as a source and once as a destination network. Besides, a *Network* belongs always to exactly one *VRF*.

5.2.5 Service

A *Service* is another essential element in this domain. These *Services* must be known to the application in order to controll traffic specifically via defined *Services*. Therefore, the `segment` is required to direct traffic specifically to this `segment` and thus also through this *Service*. The `name` and `type` are mainly intended for easy recognition and use of the *Service* in the application.

5.2.6 Link

A *Link* is of central importance for the complete application. That is because a *Link* is always directed from a node (`from_node`) to another node (`to_node`). Which is especially important if such a *Link* has different metrics (`metric_value`) between two nodes in the respective direction. This `metric_value` is then included in the path calculation and influences the *Policy* result significantly. As a consequence, a *Link* can be included in a Policy but does not have to.

Chapter 6

Architecture and Design Specifications

6.1 General

The goal of this thesis was to extend and improve the application developed in the project thesis. As in the project thesis, the complete backend architecture is part of this thesis. The frontend is not part of this thesis but is still listed in the following chapters for the sake of completeness to show the entirety of the application.

The complete architecture is based on different cloud-native services, which fulfill their own requirements and tasks. A more detailed view of the architecture can be found in the following chapters.

6.2 System Overview

The following sections should give an overview of the application developed in this bachelor thesis. Among other things, various C4[7] diagrams were created for this purpose.

The complete application consists of a software component called SerPro. This software component accesses another software component called Jalapeño, which was developed by Cisco Systems. The software system SerPro obtains topology data of a physical network from the software system Jalapeño.

Figure 6.1 provides an overview of the relationship between these two software systems. The complete SerPro application is not coupled with the external system. All connections which take place to this system can be exchanged over runtime variables. Thus, both applications can be deployed at different locations and need only Hypertext Transfer Protocol (HTTP) access to each other.

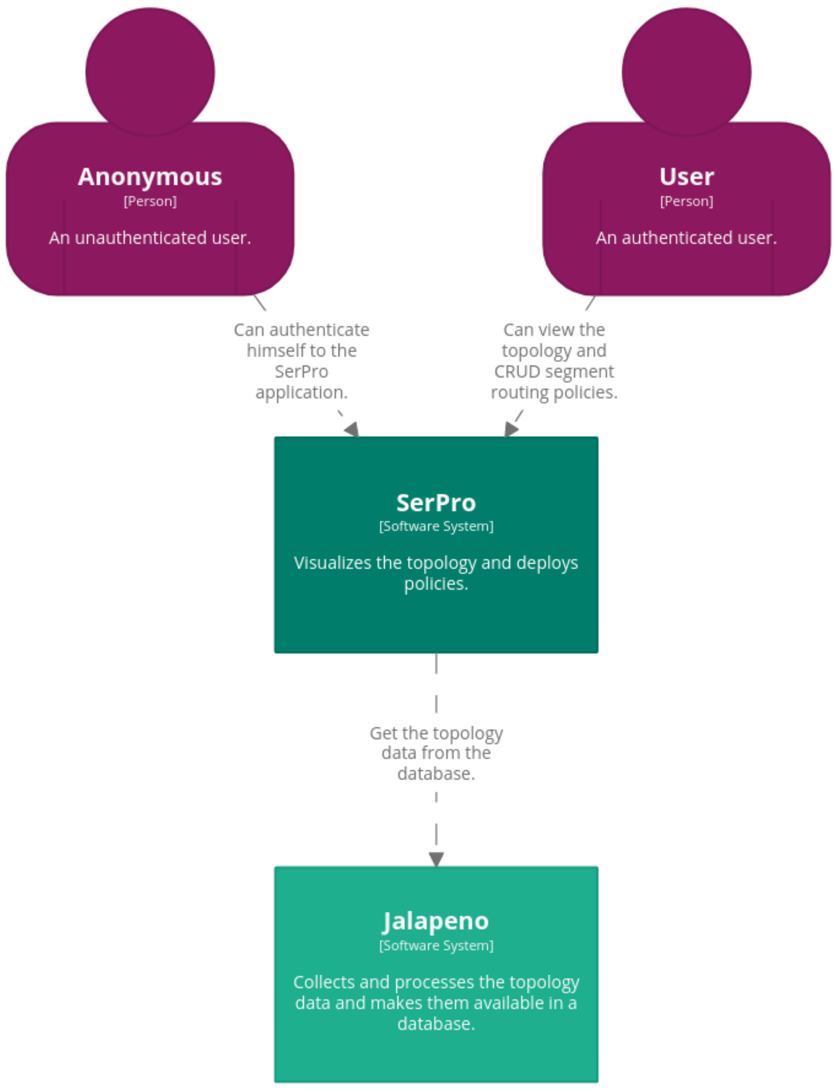

Anonymous
[Person]
An unauthenticated user.
User
[Person]
An authenticated user.
Can authenticate himself to the SerPro application.
Can view the topology and CRUD segment routing policies.
SerPro
[Software System]
Visualizes the topology and deploys policies.
Get the topology data from the database.
Jalapeno
[Software System]
Collects and processes the topology data and makes them available in a database.

Figure 6.1: C4 System Landscape

The SerPro software system consists of several different components which are all deployed using containers. The user communicates with two containers only, the frontend, and the backend. All other containers are never directly addressed by the user and remain in the background.

The various components are described in more detail in the sections below. Figure 6.2 should provide a rough overview of SerPro's container infrastructure. The exact communication processes are described in more detail in section Service Communication.

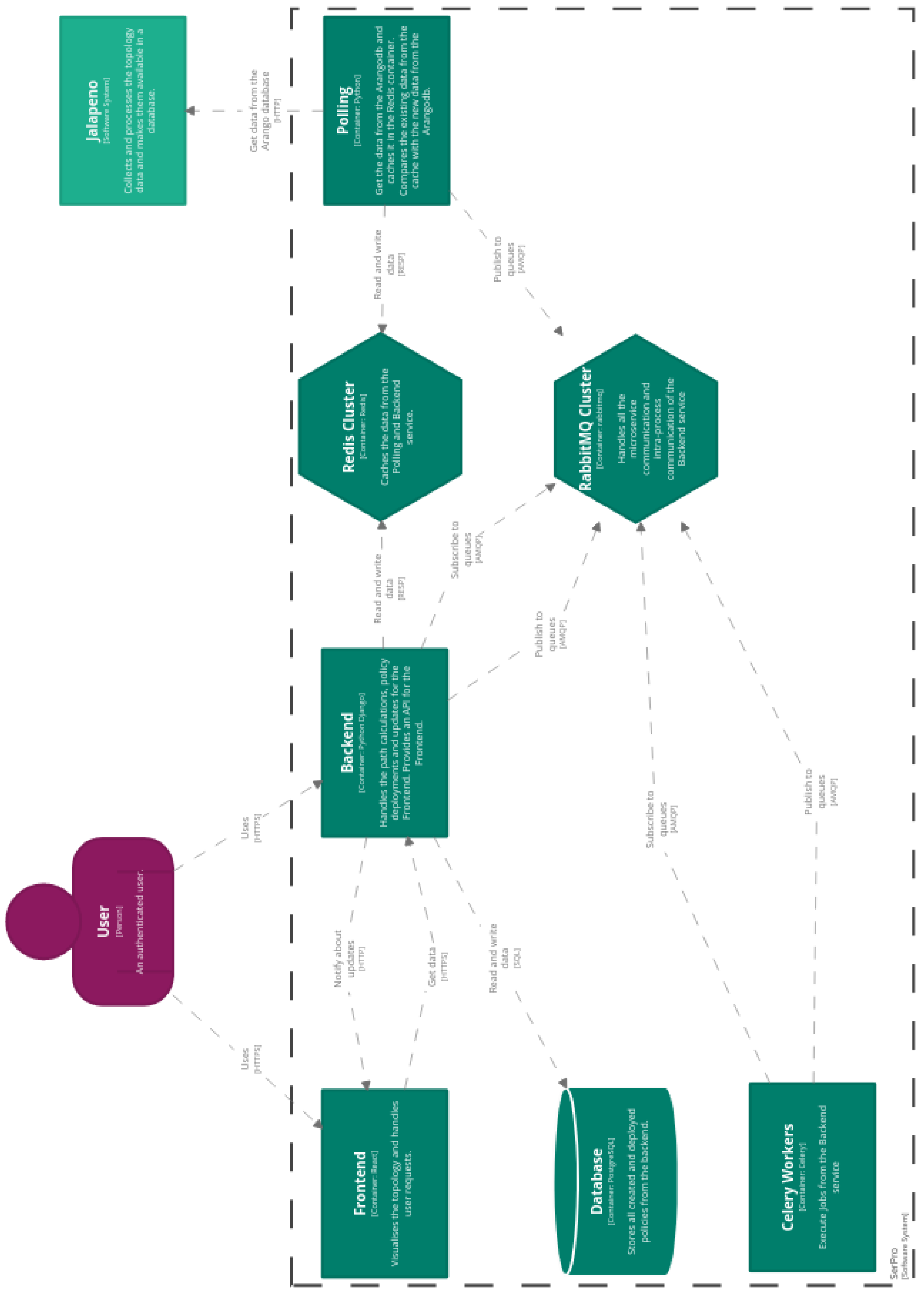

Jalapeno
[Software System]
Collects and processes the topology data and makes them available in a database.
Get data from the Arango database [HTTP]
Polling
[Container: Python]
Get the data from the Arangodb and caches it in the Redis container. Compares the existing data from the cache with the new data from the Arangodb.
Read and write data [RESP]
Publish to queues [AMQP]
Redis Cluster
[Container: Redis]
Caches the data from the Polling and Backend service.
RabbitMQ Cluster
[Container: rabbitmq]
Handles all the microservice communication and intra-process communication of the Backend service
Subscribe to queues [AMQP]
Read and write data [RESP]
Publish to queues [AMQP]
Backend
[Container: Python Django]
Handles the path calculations, policy deployments and updates for the Frontend. Provides an API for the Frontend.
User
[Person]
An authenticated user.
Uses [HTTPS]
Notify about updates [HTTP]
Get data [HTTPS]
Read and write data [SQL]
Subscribe to queues [AMQP]
Publish to queues [AMQP]
Uses [HTTPS]
Frontend
[Container: React]
Visualises the topology and handles user requests.
Database
[Container: PostgreSQL]
Stores all created and deployed policies from the backend.
Celery Workers
[Container: Celery]
Execute Jobs from the Backend service
SerPro
[Software System]

Figure 6.2: C4 Container Diagram

6.2.1 Design Goals

The precise design and architecture goals were to ensure that the application could be deployed in a scalable and highly available manner. These goals were achieved by using a messaging system (see section Messaging) and the targeted adherence to the 12-Factor Methodology.

6.2.2 Backend

The backend is the brain of the entire SerPro application. It is responsible for processing and providing all topology data which are received from the Polling service.

The backend is responsible for distributing the different messages to the clients and creating different jobs (deployment, reversion, and redeployment). In addition, the services and the complete access data for the endpoints can be managed via the backend. The main task, however, is the management and calculation of the various policies. That also includes keeping them dynamically up-to-date; more about this can be found in section Calculation and Preprocessing.

6.2.3 Polling

The polling service is responsible for keeping all topology data in the application up-to-date. For this purpose, the polling service connects to the external system Jalapeño and pulls the data to detect the updates and forward them then to the backend. By using a messaging system, it can be ensured that no updates are lost. See section Service Communication for more information on how the messaging between the services works precisely.

6.2.4 Worker

Workers are responsible for processing deploy, revert and redeploy jobs. They use the same container image as the backend but start the Celery worker process only. The workers allow the backend to submit the jobs completely and process them asynchronously, so the backend is not blocked and can accept and process further requests. For more detailed information about the technology in the workers, please refer to section Message Format.

6.2.5 Frontend

The frontend is responsible for providing the customer with a graphical interface. The frontend uses the API provided by the backend to give the user the ability to create and modify policies. By using additional WebSockets, the frontend allows the user to work dynamically and asynchronly with the backend.

The frontend was not part of this bachelor thesis. It was developed by the INS in consultation with the authors of this bachelor thesis.

6.2.6 Messaging

A RabbitMQ cluster is used for all messaging between the different services of the SerPro application. How the messaging works exactly is described in section Service Communication. The exact technology description and why exactly RabbitMQ was used can be read in section Messaging.

6.2.7 Caching

The caching system is used for data storage and fast access to the various backend instances. A complete graph is stored on it, which is used for the calculations of the various paths. The caching system is a central point for the whole application since both backend and polling services use this system.

The caching system was implemented using a Redis cluster. More details can be found in section Caching.

6.2.8 Database

Specific data should be stored persistently on a relational database, including all policies, services, and credentials. For this data, a PostgreSQL database is deployed, which has a persistent volume.

6.3 12-Factor Methodology

The goal of this thesis was to develop a scalable and highly available application. For this purpose, the 12-Factor[4] method was used, as in the previous work SerChio[3]. Because both works are based on the cloud-native approach, there may be overlaps between the different points. All points in the 12-Factor[4] method were therefore adapted and expanded for the sake of completeness.

By using the 12-Factor[4] method, the cloud-native approach can be followed very well and also controlled. This method ensures that the application can be deployed and used in a cloud environment.

1. Codebase

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

Both services backend and polling have an entirely own codebase as well as an utterly own repository. By building containers on each push to the repository, there are different containers for each application state.

Conclusion

This factor is ultimately fulfilled through the different codebases and the possibility of deploying different application releases on different Kubernetes clusters.

2. Dependencies

All dependencies should be declared explicitly and isolated.

Both the backend and polling service uses the Python dependency manager `pip` to install and manage all dependencies. The dependencies are declared in a so-called `requirements` file. In order to isolate production and development requirements, an additional development requirements file was created to define the dependencies needed only in the development environment.

Conclusion

The two services meet the requirements for dependency isolation. Only the graph-tool extension, which is needed in the backend, must be installed outside the Python dependency manager `pip`. That is because the Graph Tool has a C++ core and only has a wrapper for Python.

3. Config

Configurations should be stored in the environment.

The complete application with all services backend and polling can be configured using runtime variables.

Conclusion

Using runtime variables and the possibility to dynamically deploy the application using Helm Charts on the Kubernetes cluster, makes it possible to launch any number of versions of the application with different configurations on different clusters. Therefore, the third factor is fulfilled completely.

4. Backing services

Backing services should be treated as attached sources.

Both polling and backend services require access to backing services such as RabbitMQ, PostgreSQL, or Redis. All these services can be configured dynamically via runtime variables. That means, that no distinction is made between local and external backing services.

Conclusion

By using runtime variables, backing services can be exchanged quickly and dynamically. The customer can easily decide whether he wants to use a local, internal or external service. That point is therefore achieved without any problems.

5. Build/release/run

Use separate stages for building and running.

Both GitLab pipelines for the backend and polling are identical. The detailed descriptions can be found in section Continuous Integration. The pipelines contain a test, build, and analysis stage. In the test stage, all automated unit tests are executed. In the build stage, either the production or the development Docker images are built. The analysis stage is then responsible for generating a Sonarqube report and uploading it to the Sonarqube instance.

Conclusion

This factor is not yet fulfilled for the time being. To fulfill it, one would undoubtedly have to adapt the software versioning, for example, with timestamps or incrementing version numbers. With minor changes, however, this point should also be achievable in the future.

6. Processes

The application should be stateless.

All data that should be persistent is outsourced to so-called backing services such as RabbitMQ, PostgreSQL, or Redis. The actual processes should not have their state. The application uses different backing services to store or transfer different data persistently.

Conclusion

All services are currently entirely decoupled from each other and use only backing services across the board to persist or transmit data. This approach makes the services scalable without any problems. Through this solution, this point is also achieved.

7. Port binding

Make services only available via Port Binding.

The application should not be published by an external web server such as IIS, Nginx, or Apache2. An integrated web server should be used. The polling service does not need to have a port open and thus does not need port binding. The backend service uses the `runserver` provided by Django, which starts the application and binds a predefined port.

Conclusion

By using the webserver provided by Django, the seventh factor is fulfilled.

8. Concurrency

The application should be scalable via the process model.

While only one process runs in the polling container, the backend container requires several processes, which all run independently on the container. By deploying on a Kubernetes cluster, the scaling options already integrated into it can be used.

Conclusion

The fact that the application can be deployed on a Kubernetes cluster and scaled without problems means that the eighth factor is fulfilled.

9. Disposability

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

No data should be lost in the application in the event of a shutdown or crash. The use of message queues ensures that messages are not deleted from the queue until they have been processed completely. That means that even if a backend crashes, no messages are lost, as they are not acknowledged and can therefore be processed by another backend instance. The same principle is applied to the polling service. That ensures that no updates are lost when the polling service is no longer available.

Conclusion

The individual service containers can be started within a few seconds. Since messages are only deleted when they have been processed completely, it can be assumed that no data is lost in the event of a crash. The ninth factor is therefore fulfilled.

10. Dev/prod parity

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

The development should be done with the same services that run in production. That minimizes errors caused by differences between the productive and developer services. Because the entire application was developed entirely cloud-native directly in the Kubernetes cluster (see section Development Environment), the same services could be used in the production as in the development environment.

Conclusion

Due to the utterly cloud-native development of the entire application directly in the Kubernetes cluster, all services could be used as in the production environment. That ensures that the application runs as productively as possible. The tenth factor is fulfilled by ensuring that the differences between the production and development environments are as minimal as possible.

11. Logs

Logs should be treated as event streams.

The logs in all developed services are standardized and written in `stdout` (see section Logging). That allows the logs to be easily read from the command line or sent to an external logging system such as an Elasticsearch, Logstash and Kibana (ELK) stack.

Conclusion

The eleventh factor is entirely fulfilled by writing the logs to `stdout`.

12. Admin processes

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

Admin tasks should always be performed on an identical system that is also running in production. Because all services are containerized, administrative tasks are performed automatically when the application is started.

Conclusion

Due to the containerization, all administrative tasks are already executed when the application is started and can also be executed separately on a staging or testing system. This factor is therefore also achieved.

6.4 Technology Decisions

This section lists the different libraries and technologies which were used to implement the application. Moreover, it contains a detailed section about the motivation and the reason behind the technology decision for this application. In the section, it is also mentioned which other technologies were encountered to be used. Significant decisions were influenced highly by the vision to create a high-scalable and -available software.

Several technologies were inherited from the previous project thesis. More detailed information about these technologies can be found in the SerChio thesis.

6.4.1 Technology Stack

The following table contains an overview of the most important technologies which were applied in the software.

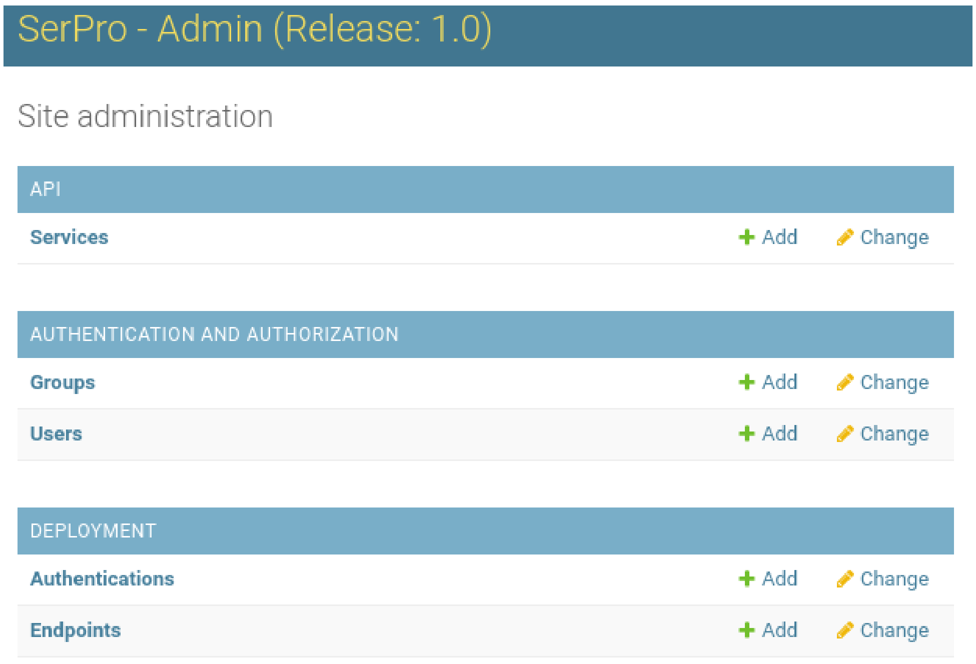
Component	Technology
Backend	• Programming Language: Python • Framework: Django • Framework Extensions: Django REST Framework • Additional Libraries: – django-redis (Caching) – djangorestframework-simplejwt (Authentication) – nornir (Policy Deployment) – channels_rabbitmq (Websockets) – carehare (RabbitMQ Messaging) – celery (Worker Tasks) – graph-tool (Graph Implementation) – rom (Redis Object Mapper - Caching)
Polling	• Programming Language: Python • Additional Libraries: – pyArango (Graph DB Access) – asyncio (async Task Management) – schedule (Job Scheduling) – pika (RabbitMQ Messaging) – rejson (Redis Cache Access)
Database	• PostgreSQL for development and productive usage • SQLite for unittests
Messaging	• RabbitMQ
Cache	• Redis cache with JSON module

Table 6.1: Technologies

6.4.2 Programming Language and Framework

In the previous project thesis, the polling service was implemented with Python. Because it has proven to be stable and accurate, the service was further improved on the existing codebase.

Python Django, together with the Django REST Framework (DRF), was already a proven design decision in the backend service during the previous work, and therefore it was adhered to. It has been proven that this design decision was appropriate because some features, like authentication and authorization, could be implemented swiftly with the help of the Django Framework.


SerPro - Admin (Release: 1.0)

Site administration

API		
Services	+ Add	Change

AUTHENTICATION AND AUTHORIZATION		
Groups	+ Add	Change
Users	+ Add	Change

DEPLOYMENT		
Authentications	+ Add	Change
Endpoints	+ Add	Change

Figure 6.3: Output Django Admin Site

According to the programming language and actual code writing, one of the most significant changes was the introduction of static typing [23]. The development team had decided to improve the code further and use the static typing feature, making the code less error prone and more readable. Static typing adds type information to the code and helps the developer to give additional information about variable types, return values, and more. In the listing 6.1 an old code snipping is shown, which has no type hints included. On the opposite, the listing 6.2 shows the same method with static type information. It can be observed that the parameter `object_hashes_from_cache` has to be a Python Dictionary, and the return value is a boolean value (`bool`).

```
def object_cache_is_empty(self, object_hashes_from_cache):
    return object_hashes_from_cache is None
```

Listing 6.1: without typing

```
def object_cache_is_empty(self, object_hashes_from_cache: Dict) -> bool:
    return object_hashes_from_cache is None
```

Listing 6.2: with typing

More detailed information about the technical decisions according to the programming language, the web framework and its extensions can be found in the thesis SerChio.

6.4.3 Database

In the pre-work, called project thesis, the decision was made for PostgreSQL as a database system. This database system has proved a perfect fit for the application to persist data and works perfectly together with the introduced frameworks. Therefore PostgreSQL was inherited and also applied in the bachelor thesis.

6.4.4 Caching

General

A cache instance was mainly used to cache data for the polling service in the pre-bachelor thesis. The cached data were used to detect network changes and notify the backend about them. This implementation showed that the cache has ultimate speed, and the cache could and should be used for more intended uses.

So, at the beginning of the bachelor thesis, it was decided to stick with the Redis cache technology and extend its usage. The goal was to rely highly on Redis for the development and production of the whole software and thus, enhance the whole architecture. Redis should not only be used in the polling service to detect changes but furthermore, enable caching in the backend service.

The permanent storage could be identified as a massive bottleneck for data accessing in the previously done work. Writing specific data that were already held in the persistent graph database in the cache instead of in the PostgreSQL, should help to raise the whole application performance. This design decision was central for building a high-scalable, high-available, and distributed software.

Because the Redis Cache System has such a central value in the complete software architecture design, it was decided to build a high-available Redis caching system with the help of the widely used technology Redis Sentinel described in the following section.

The reason why Redis was used in the previous work, can be found in the documentation of the SerChio thesis.

Redis Sentinel

Redis Sentinel

Redis Sentinel provides high availability for Redis. In practical terms this means that using Sentinel you can create a Redis deployment that resists without human intervention certain kinds of failures.[25]

The caching system is one of the core components of the whole software. It enables the delivery of information in no time. Because the whole application counts on the caching system's functionality and reachability, it was decided to enable a high-available solution with the solution called Redis Sentinel.

Redis Sentinel is a Redis solution that enables high availability for the caching solution. Redis follows a master-slave principle, which is called master and replica in Redis terms. The Sentinel has the task of monitoring the status of the master and replica instances. As soon as the master malfunctions, the Sentinel starts a failover process, when a new master is elected, and the other Replicas are informed about the new master instance. This process is visualized in figure 6.4.

Another essential responsibility of the Redis Sentinel is the master discovery and updating of the configuration. Redis Sentinels acts as a mapping service: Because the Redis Sentinel always has the connection to the current master, clients can ask a Redis Sentinel instance for the current master address. So, it can be ensured that the clients always get the connection information of the current master.

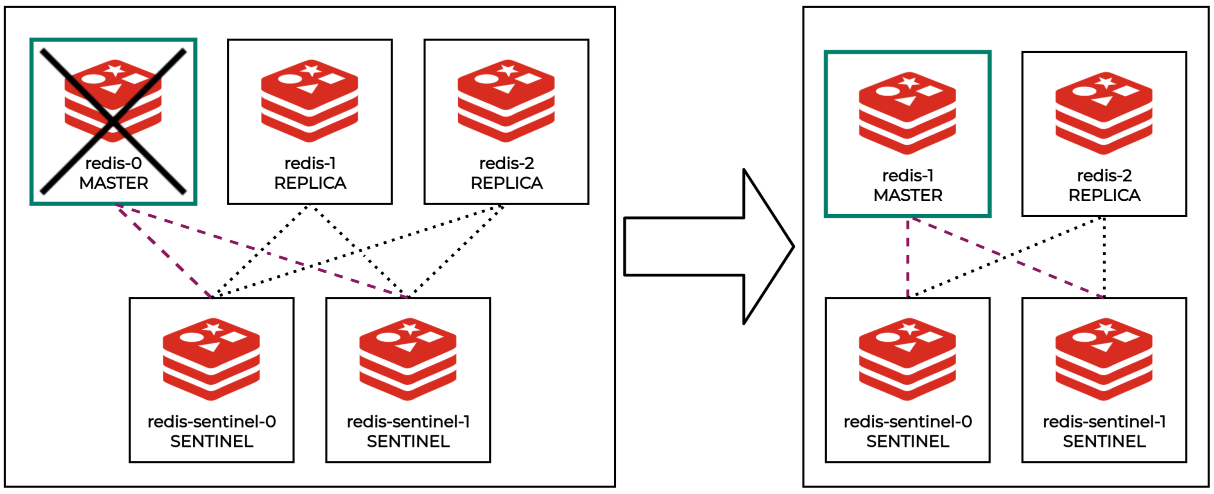


Figure 6.4: Redis Sentinel Failover

Applied Application Libraries

rejson (`https://github.com/RedisJSON/redisjson-py`) rejson is a library that allows a performant write and load of JSON objects into a Redis database. One of the most significant advances is the on-the-fly (de)serialization of objects into valid JSON. This library is used in the polling service to implement data caching. This library was already used in the previous project thesis and, therefore, not replaced.

django-redis (`https://github.com/jazzband/django-redis/`) Django comes with an implemented framework for caching data [8]. It supports officially Memcached as their primary cache system. Because we already had a Redis caching system, we decided to use it with a community project called Jazzband Django Redis (django-redis). This library not only enables the same functionality as the default caching system moreover, it also supports saving things with an infinite timeout. This feature is one thing that the application relies upon. Besides, it also enables the automatic (de)serialization of Python objects, making it effortless to save things permanently and enable fast access. Early in the project, it was decided that this library should be used for storing and retrieving calculation and graph information from the Redis cache. On the one hand, it should ensure that the current, complete graph with its characteristics is saved into the cache. On the other hand, it should be used to store preprocessing information like shortest distance information about the graph in the cache. So, with the help of this library and the cache system, it should be ensured that each backend instance disposes of the latest data and therefore can calculate the correct policy results.

Redis Object Mapper (`https://github.com/josiahcarlson/rom`) In the previous work Ser-Chio, the performance was identified as there is need to improve in the future. This crucial concern was addressed in the early designing phase. A way should be found to minimize the slow database accesses. The solution to this problem was found in the Redis Object Mapper (rom). rom allows the creation of database-like models and saves them in the more performant Redis cache. Hence, not only the performance problem should be solved. In addition, the information, which is already persisted in the Arango graph database, could be easily cached in the Redis cache and therefore has not to be saved in a persisted database twice. Another advantage to the non-comparing performance of the cache models, is the easy syntax to use. The syntax for the model definition (listing 6.3) and the data query (listing 6.4) is quite similar compared to the Django ORM ones (listings 6.5 & 6.6). This fact also supported the decision to rewrite the existing appropriate Django ORM models from the previous work to Redis Object Mapper models in this thesis.

```
class Link(rom.Model):
    _key = rom.Text(required=True, unique=True)
    local_node = rom.ManyToOne("Node", on_delete="cascade")
    remote_node = rom.ManyToOne("Node", on_delete="cascade")
    local_link_ip = rom.Text(required=True, unique=True)
    remote_link_ip = rom.Text(required=True, unique=True)
    igp_metric = rom.Integer(required=True)
    enabled = rom.Boolean(index=True, default=True)
```

Listing 6.3: ROM Model Example

```
Link.query.filter(enabled=True)
```

Listing 6.4: ROM Query Example

```
class Service(models.Model):
    FIREWALL = "FW"
    IDS = "IDS"
    SERVICE_CHOICES = [(FIREWALL, "FW"), (IDS, "IDS")]
    name = models.CharField(max_length=20, null=False, unique=True)
    belongs_to_node = models.IntegerField(null=False, unique=True)
    service_type = models.CharField(
        max_length=3, choices=SERVICE_CHOICES, default=FIREWALL, null=False
    )
    service_prefix = models.IntegerField(null=False, unique=True)
    service_suffix = models.CharField(max_length=64)
    srv6_sid = models.CharField(max_length=128, unique=True)
    enabled = models.BooleanField(default=True)
```

Listing 6.5: Django ORM Model Example

```
Service.objects.filter(enabled=True).all()
```

Listing 6.6: Django ORM Query Example

6.4.5 Messaging

General

It was indicated in the project thesis that the communication approach needs to be improved in the future. Especially the messaging via WebSocket between the polling and the backend service was not a future-proof solution. It did not scale well, needed complex error handling, and was not optimal for several backend containers. Nevertheless, the WebSocket should not be removed from the backend application because it seems the best solution for notifying the frontend dynamically about existing changes in the network. So, because the WebSocket support should not be dropped and Django Channels is the only known library to support WebSockets in the Django framework, it was clear that Django Channels was also needed in the bachelor thesis.

The vision of using message queues between the polling and the backend application was clear from the beginning. It seemed like the perfect solution to implement a reliable, scalable, and fast communication channel between polling and the backend services. However, the exact solution and technique were utterly unknown at the start of the bachelor thesis.
The functionality of a message queue is quite manageable. A message is put into a message

queue, and the message queue acknowledges its receptions. The message queue then delivers the message to all receivers, which are subscribed to the appropriate queue. After all, subscribers have acknowledged their reception, the message will be deleted, and the following message is processed in the same manner. This process is visualized in figure 6.5.

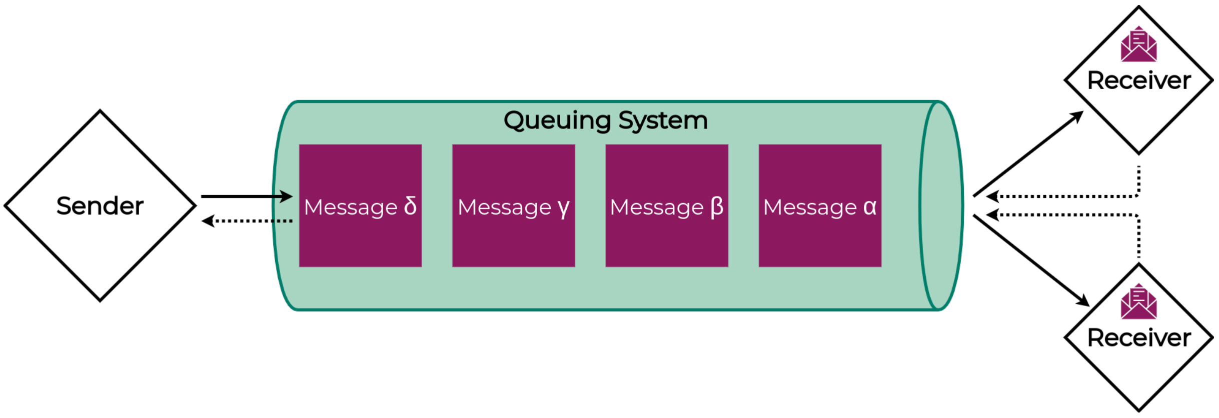


Figure 6.5: Message Queue Functionality

First, it was tried to find a message queue solution with the already determined Redis cache system. Redis cache system implements a publish-subscribe feature, which serves as a simple message queue solution. [24] It seemed like the most suitable solution first, but after more detailed research, it was discovered that Django Channels did not support Redis Sentinel. This design decision would avoid the implementation of WebSocket communication between the front- and backend. Because the vision included a dynamic application with WebSocket support and a high-available caching solution with Redis Sentinel support, this design decision was avoided, and another solution was examined.

The goal was to find a solution supporting high availability and was supported by Django Channels. The answer was given by inspecting RabbitMQ as a message queue solution. RabbitMQ not only delivers an excellent, high-available and -scalable solution, but there is also a community-driven library for using Django Channels together with RabbitMQ. Hence, this solution met the necessary requirements. Therefore, it was decided to implement a RabbitMQ cluster to communicate between polling and backend services, the WebSocket messaging, and additional parts like, for example, the Celery communication.

More detailed information about the messaging implementation can be found in section Messaging.

Applied Application Libraries

channels_rabbitmq (`https://github.com/CJWorkbench/channels_rabbitmq/`) The channels_rabbitmq supports the standard Django Channels interface to enable WebSocket communication. The difference between the regular Django Channels library lays in the backing store. The conventional Django Channels use Redis as their backing store, whereas this library saves and distributes the WebSocket messages over a RabbitMQ system. Therefore, the library is used to notify the different frontend clients with the update messages.

carehare (`https://github.com/CJWorkbench/carehare/`) The carehare library is a RabbitMQ client that allows receiving messages queued on a RabbitMQ system asynchronilly. The library is developed from the maintainer of the channels_rabbitmq and is also needed in the same library. This library was applied, because it is an excellent addition to the existing channel's library and enables the asynchronous receiving of the messages without blocking. The library is used to receive update messages from the polling service as well as notification of deployment and reversion tasks.

pika (`https://github.com/pika/pika/`) Pika is a RabbitMQ client which sends messages synchronically. Because the polling does not need asynchronous code, it was decided to use pika to send messages into the RabbitMQ system.

6.4.6 Graph Library

In the previous thesis, the library graph-tool (`https://graph-tool.skewed.de`) was selected according to its ultimate speed. Another advantage is the (de)serialization possibility which is implemented with the standard pickle module[27] in the library. The library, therefore, allows to save and restore the whole graph with its characteristics [21]. Besides the vast performance, this feature was one of the main reasons to stick with the library. It allows using the congruent graph in more than one backend instance.

6.4.7 Policy Deployment

General

The policy deployment was a new and unknown part at the beginning of the bachelor thesis. Neither the configuration for the policies was available nor the way how to configure the appropriate routers. So, the goal was to find a way to write Segment Routing Traffic Engineering policies automatically down to the present Cisco IOS-XR routers. The solution should be as reliable, easy-to-use, and -maintain as possible. Therefore, this design decision was also a crucial part of the Elaboration phase of this bachelor thesis.

Configuration Interface

The first part was to declare which programmatic configuration interfaces were given in the existing devices to configure these. An overview of the different techniques can be found in table 6.2.

IOS-XR systems support two programmatic interfaces besides the usual CLI [22]. The first interface is NETCONF, which enables retrieving or writing configurations via an SSH connection and is standardized in RFC 6241[14]. The second interface is gRPC, which connects to the routers via an HTTP(S) connection. Both approaches belong to the so-called model-driven programmability methods. Simplified, this means that they need data models in order to retrieve or configure device information. Data models in the field of network programmability are written in Yet Another Next Generation (YANG).

After some investigations and the help of the industry partner, it was concluded that the solution has to be based on the CLI because there are no YANG models for all the needed configuration parts present. Therefore, the developer team decided to find a possibility to automate the CLI mode.

	NETCONF	gRPC	CLI
Protocol	SSH	HTTP(S)	SSH
Python library	✔	✔	✔
Data Models needed	✔	✔	✖
Encoding	XML	Google Protobuf	-

Table 6.2: Comparision Configuration Interfaces IOS-XR

Tool

The second step was to decide which tool could be used to configure devices automatically. The restrictions were clear: it should be able to connect to the device, execute the commands, and disconnect after with the tool.

The first thoughts were about using plain SSH libraries like Netmiko or Paramiko to connect to devices and execute the commands. However, this proposal was adverted because there are frameworks available, which already have implemented SSH-connection handling and additional features like connection management and error handling. These frameworks are often built on top of these mentioned libraries and therefore take much work off.

One of these libraries, which is based on the mentioned libraries, is Network Automation and Programmability Abstraction Layer with Multivendor support (NAPALM). NAPALM seemed a perfect fit because for the maintainers of the bachelor thesis, it enables an easy-to-use interface with the demanded functions. Additionally, it includes impressive features like the detection of failed configuration results and more. Because the editors of the bachelor thesis were familiar with the NAPALM API, they decided to use this library.

In most cases, NAPALM is not used directly but instead accessed by frameworks on the top. Established frameworks, like Ansible or SaltStack, are such examples. These mentioned frameworks were also considered but were dropped because they are often not as extensible as the solution should be.

The solution was found in a pure Python library called Nornir. Nornir is a framework which is specialized in automation tasks. It can be used with Python, has an easy-to-use API and vast performance and is extensible. So, it has the advantage of being integrated easily into the existing backend service. Moreover, it allows managing efficiently an inventory of the different devices and their characteristics. Another part was the noteworthy support of different plugins, including NAPALM, which fit perfectly overall. So it was decided to use the Nornir framework, which is built on the desired stack of tools:

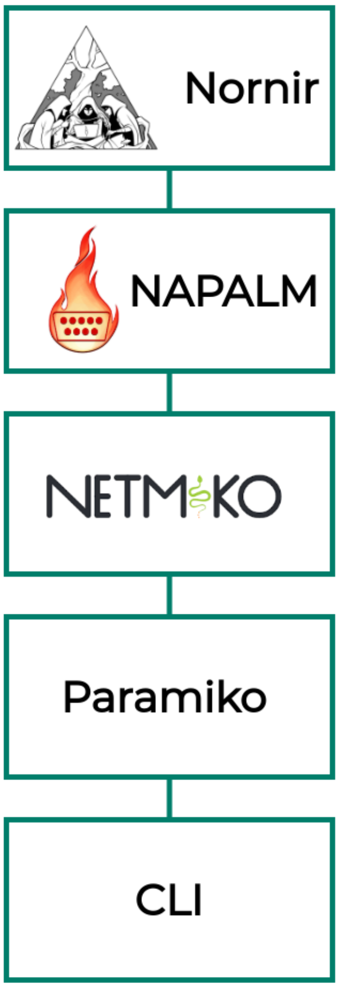


Figure 6.6: Nornir Stack

Automation

The goal was to introduce a technique that does not block the whole API during a deployment or reversion process. A solution had to be found, which could take on these tasks and fit perfectly into the application's ecosystem. For this operation, it was decided to introduce a task queue. A task queue is basically the same as a message queue, but instead functions are put into the queue. So-called workers or worker nodes then process these functions. After the function is executed, the result is returned to the caller, assigning the task initially. An overview of this process can be seen in figure 6.7.

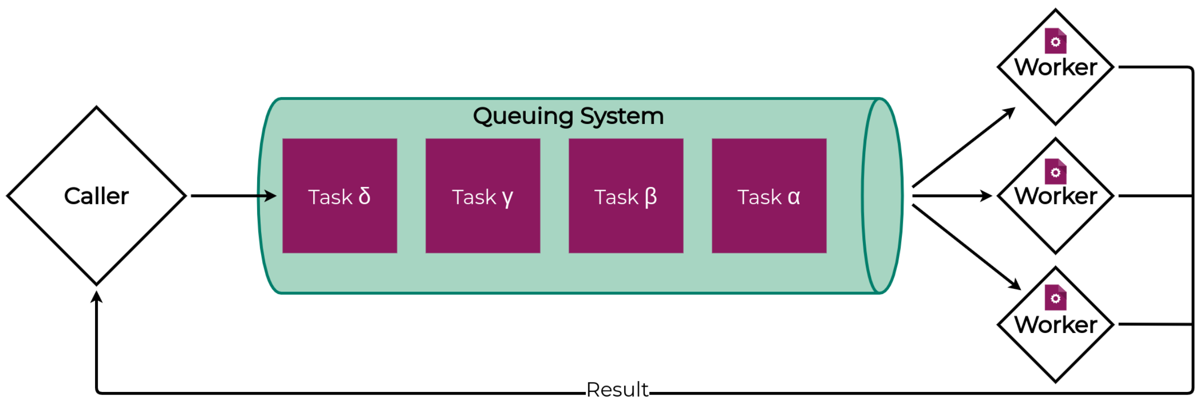


Figure 6.7: Task Queue Functionality

A task queue that suits perfectly in the new architecture could be found quickly. The well-known task queue Celery not only has excellent support for Django, but it also allows to use of RabbitMQ as a queuing system. The advantages of Celery are its simplicity, high availability and performance. After a first test, it was ensured that Nornir tasks could be automatized with the help of Celery and therefore, could be included in the application's ecosystem.

More information about the policy deployment with the help can be found in section 6.6.4

6.5 Messaging

This chapter contains further and more detailed information on how the messaging is solved within the application. It includes information about the different queues created in the existing RabbitMQ cluster and their purpose. Furthermore it also examines the bi-directional communication between the backend and the frontend of the application. The last part of this capture is about general message format and describes why a common message format was introduced.

6.5.1 Service Communication

The central system for all messaging between services is the RabbitMQ cluster. All the different queues, that connect the backend, polling, and workers run on this cluster. The individual queues and the messages, that are exchanged in them are described below. The whole communication can be seen graphically in the backend blueprint in figure 6.8.

changes The polling service sends all update messages directly, to the changes queue. The backend has a service subscribed to this queue and therefore, it gets the messages directly when the polling publishes them.

channels All WebSocket messages are stored in the channel's queue. That allows the various backend instances to forward the WebSocket messages to the clients, who are connected to them.

celery The celery queue is used for jobs (deploy, revert, redeploy). The backends publish the jobs in this queue and the workers react as soon as a new job is placed there.

workerresult The workers send their job results to the workerresults queue which the various backend instances are subscribed to. By this procedure, the backend instances do not have to request the results but get the result as soon as the workers have finished a job.

As shown on the blueprint in figure 6.8, the RabbitMQ cluster is the hub for the entire communication of the application. By using a RabbitMQ cluster, the availability of the messaging system can be improved significantly.

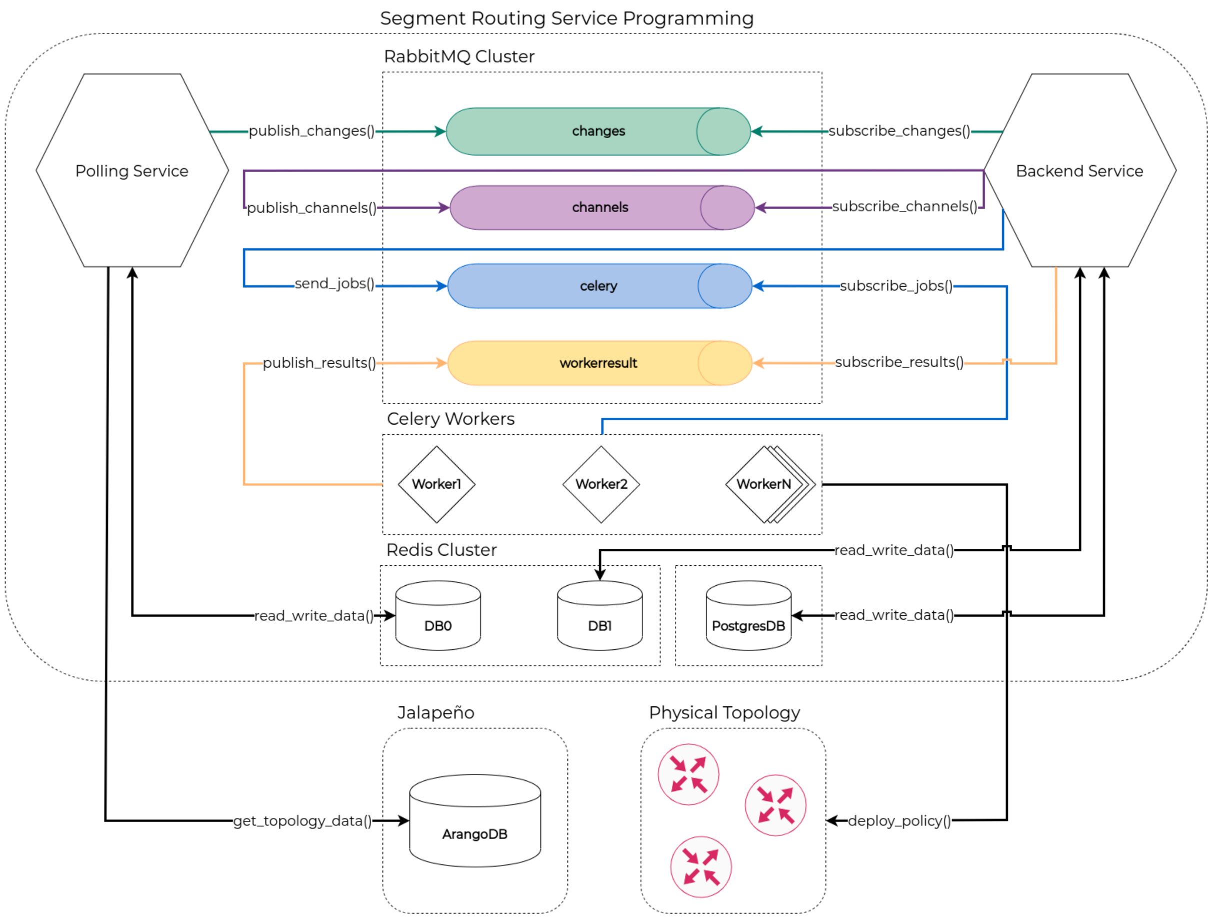


Figure 6.8: Backend Architecture Blueprint

6.5.2 Backend - Frontend Communication

The frontend makes the web application available in the browser to the client. The web browser then communicates with the API of the backend. The browser also establishes a WebSocket connection with a backend instance. Through this WebSocket connection, the browser receives real-time updates directly from the backend.

Figure 6.9 shows this communication graphically.

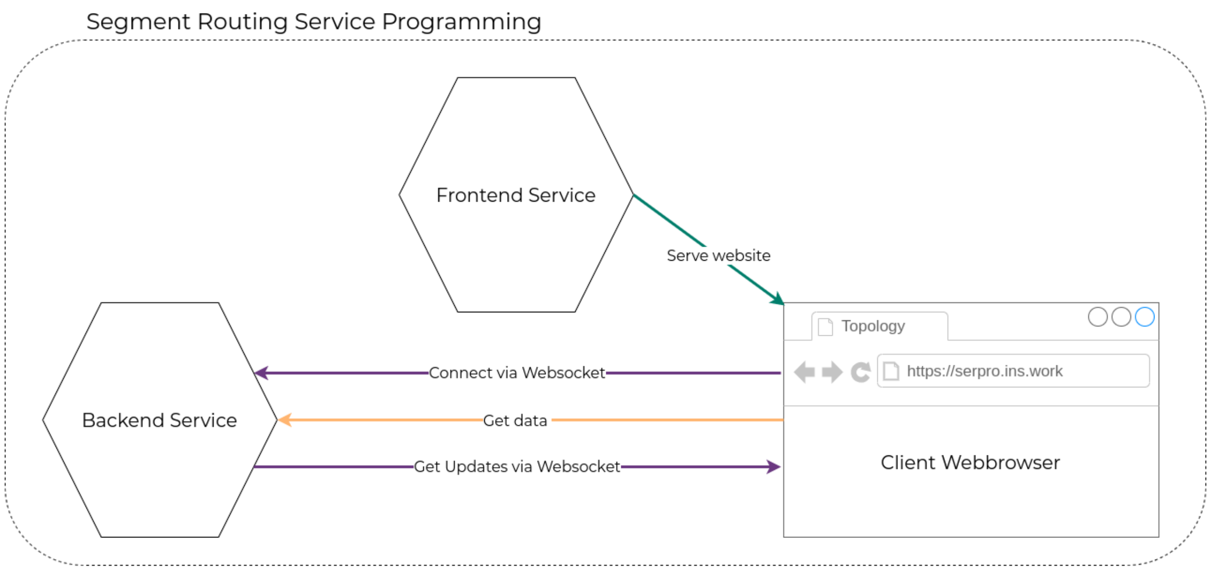


Figure 6.9: Frontend Architecture Blueprint

6.5.3 Message Format

At the beginning of the construction phase, a global consistent message format was introduced. It should standardize the skeleton of the messages which are sent over the RabbitMQ cluster. The message format was therefore adapted for each communication which was mentioned in the section Service Communication. It was decided to use JSON as a common message format from the polling, over all other services to the frontend. There were some reasons which backed this decision:

1. The polling service retrieves the network information from the graph database in the JSON format and caches it in the same format.
2. The frontend expect JSON from the API as well as from the WebSocket.
3. JSON is easy to-read and has great Python support

So, each message which is sent over the message queue has to adhere to the following format:

Key	**Value**
type	Contains information about the type of the message
status	Contains information about the status of the message
message	Contains a description of the message
data	Contains the effective payload of the message

Table 6.3: Message Format

After receiving a message in the standard message format, it is easy to detect its purpose. The combination of the type and the status information give the necessary information to process the message further. The additional information in the message and the data part delivers the necessary data to adapt to each message. Thus, the different messages can be adapted precisely to their purpose. Whereas a polling message delivers information about updates in the network (listing 6.7), the WebSocket message is adopted to deliver the necessary information to the frontend (listing 6.8).

```
{
  "type": "lsnode",
  "status": "add",
  "message": "incoming changes",
  "data": {
    "md5_checksum": "1ed4454d3e534f6ff4ef8c2b4308506b",
    "_key": "2_0_0_0000.0000.0004",
    "igp_router_id": "0000.0000.0004",
    "area_id": "49.0001",
    "name": "XR-4"
  }
}
```

Listing 6.7: Example Message Polling - Backend

```
{
    "type": "creation",
    "status": "successful",
    "message": "Policy testpolicy is successfully created",
    "data": {
        "id": 1,
        "name": "testpolicy"
    }
}
```

Listing 6.8: Example Websocket Message

The advantage is not far to seek: all messages have an equivalent format at each part of the application. The whole communication was more straightforward to develop and comfortable to maintain because the message format was always the same. Another benefit is the easy-to-understand format, which allows reacting quickly to changes. So an introduction of a new status or message type was done without circumstances. Thus, the messaging is prepared for the future with the help of this generalization.

6.6 Policy Treatment

The policies are the central elements in the application. This chapter delivers more detailed information about different policy treatments in different situations. Before a calculation can be created, the input must be validated; more detailed information can be found in this section. Furthermore, it contains a separate section on each of the following topics: policy creation, policy update, policy deployment and policy reversion, which deliver more explicit information.

6.6.1 Input Validation

The input validation is a crucial part of the application and the fundament for all the policy actions. It has the purpose to only let correct input enter the application. Therefore, it averts later problems in other backend processes like the calculation, the network configuration or deployment, and more. Furthermore, it permits security incidents of the backend service. Hence, before the actual calculation can happen, the correctness of the input has to be validated. As soon as the input is validated, a policy creation can proceed (see also 2.14 and Policy Creation).

As a solid validation automatically leads to a more secure and stable application, many different validation rules were created. As soon as one rule can not be adhered to, a `ValidationError` will be raised, which is translated automatically into a `HTTP 400 Bad Request` response code. It ensures that the request is declined and the user is notified about, what was wrong with the input. The listings 6.9 and 6.10 are some examples of validation errors, which fulfil different purposes. The listing 6.9 notifies that the `source_vrf` does not belong to the `source_node` what literally can never happend. The listing 6.10 hints that the name had characters in it, which are not permittable. If this validation did not happen, it would lead to an error in the policy deployment afterward. Such flaws are automatically averted using the created validation logic.

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

{
    "source_vrf": [
        "The source vrf with id 3 does not belong to the node with id 10! Please
    provide a valid source vrf!"
    ]
}
```

Listing 6.9: Source VRF Validation Error

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

{
    "name": [
        "The name can only include follwing characters [(a-z)(A-Z)(0-9)_-]"
    ]
}
```

Listing 6.10: Name Validation Error

6.6.2 Policy Creation

A policy creation, whose sequence is visualized in figure 6.10, starts with a `POST` request to the API route `/api/policy`. The request with its attached data is further redirected until its JSON data are translated into Python native types. This process is called deserialization. Afterwards, the data can be validated as described in more detail in section Input Validation. If the *PolicyValidator* cannot allocate the input data, an error is raised which is returned as an `HTTP 400: Bad Request` to the user. An example can be seen in figure 6.9. If the data are valid, the *Calculator* can be accessed, which calculates the result and returns it. Next, these result values are used to create and save the new policy in the database. As soon as this step is finished, the result can be serialized, translated into valid JSON, and returned. The user receives the result together with a `HTTP 201: Created` code, which indicates the success of policy creation.

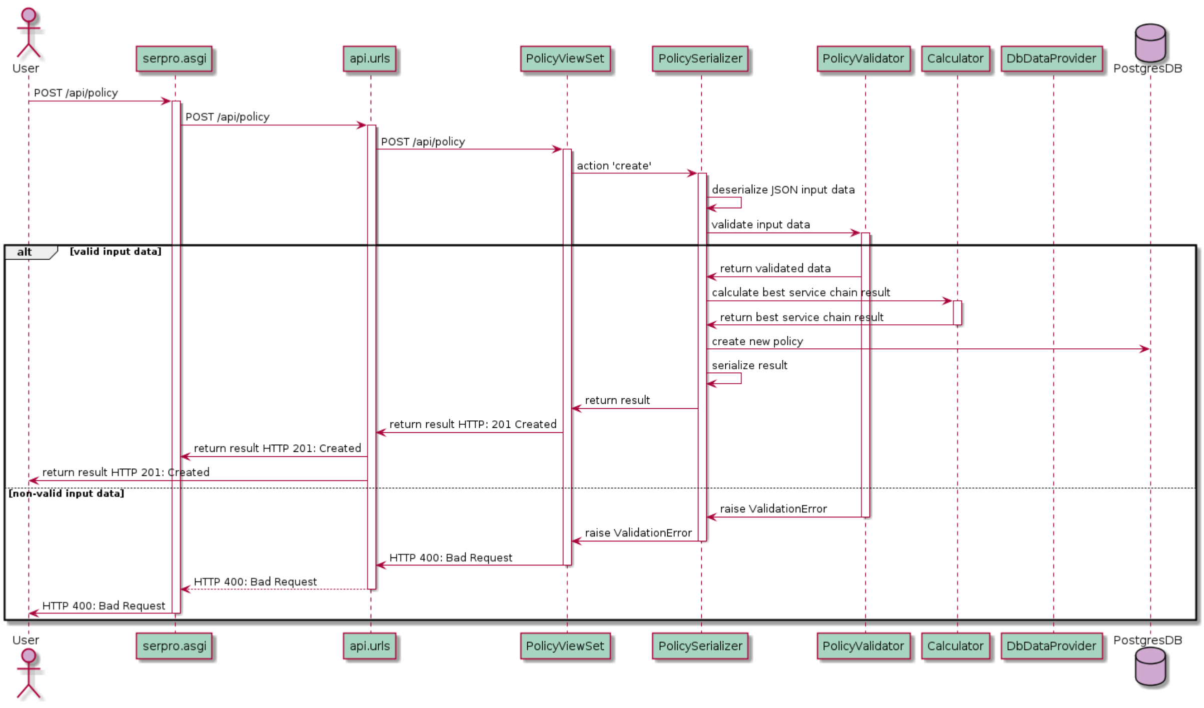


Figure 6.10: Policy Creation

6.6.3 Policy Update

The whole update procedure is visualized in figure 6.11. A policy update process is similar to the Policy Creation activity with slight differences. The request is triggered with an HTTP `PUT` method. If the input data are not correct or the policy is not editable, a `ValidationError` is raised. A Policy is only editable if the policy is not deployed in the network. As soon as a policy configuration is written down to the router, the policy is locked for changes. If the input parameters are valid, the update process is started. The application detects automatically if a new calculation of the best result is necessary and triggers if necessary. It is the case, if another parameter as the policy name has changed. In each case, the policy can be updated with the new values and can be returned. A policy update is converted automatically in a `HTTP 200 OK`, which indicates that the policy was updated successfully.

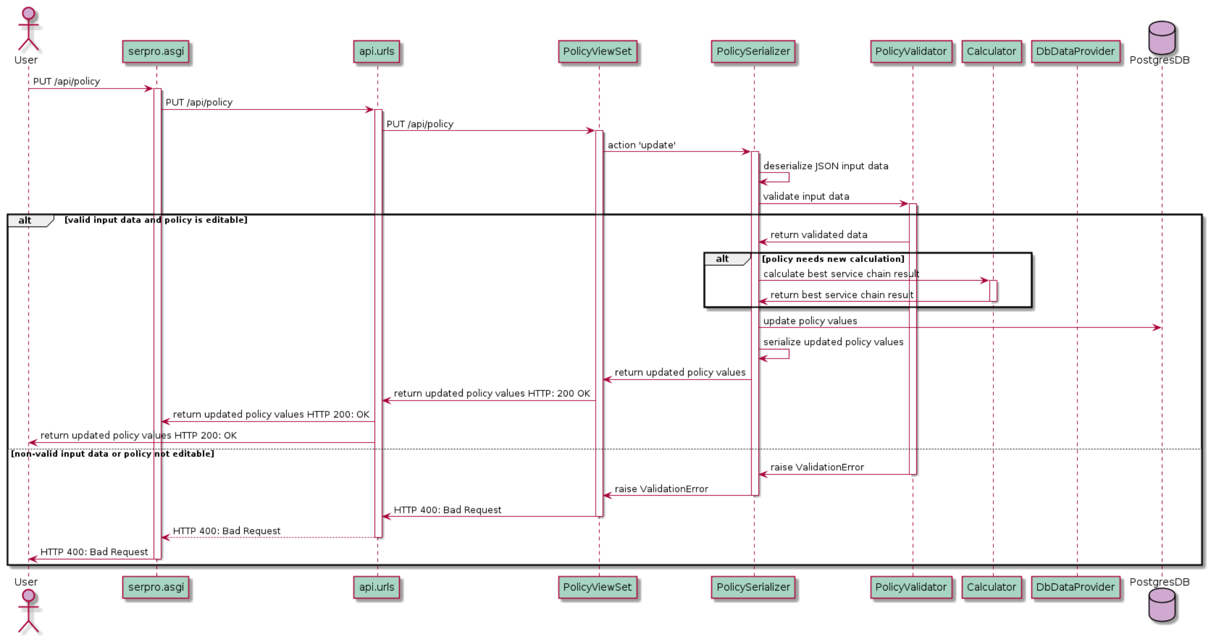

User
serpro.asgi
api.urls
PolicyViewSet
PolicySerializer
PolicyValidator
Calculator
DbDataProvider
PostgresDB
PUT /api/policy
PUT /api/policy
PUT /api/policy
action 'update'
deserialize JSON input data
validate input data
alt
[valid input data and policy is editable]
return validated data
alt
[policy needs new calculation]
calculate best service chain result
return best service chain result
update policy values
serialize updated policy values
return updated policy values
return updated policy values HTTP: 200 OK
return updated policy values HTTP 200: OK
return updated policy values HTTP 200: OK
[non-valid input data or policy not editable]
raise ValidationError
raise ValidationError
HTTP 400: Bad Request
HTTP 400: Bad Request
HTTP 400: Bad Request

Figure 6.11: Policy Update

6.6.4 Policy Deployment

As soon as the customer triggers a deployment via the dedicated action in the API module, the backend creates a so-called job using Celery, which then lands in a dedicated job queue. Celery was used here to execute the jobs asynchronously so that the backend is not blocked. By creating these jobs, the backend task is done for the moment, and the workers can start their work. All workers are subscribed to this job queue and are notified as soon as a new job is in the queue. Figure 6.12 describes the job creation process graphically using a sequence diagram.

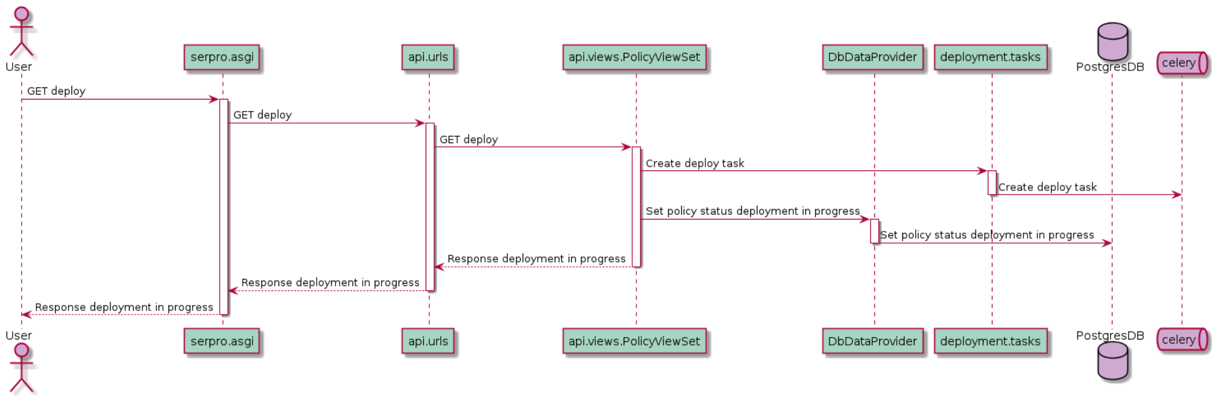

User
serpro.asgi
api.urls
api.views.PolicyViewSet
DbDataProvider
deployment.tasks
PostgresDB
celery
GET deploy
GET deploy
GET deploy
Create deploy task
Create deploy task
Set policy status deployment in progress
Set policy status deployment in progress
Response deployment in progress
Response deployment in progress
Response deployment in progress

Figure 6.12: Policy Job Creation

Celery Workers use the same code base as the backend. This allows all functions to be executed, that the backend can also execute. The Celery process running inside the workers subscribes them to the dedicated job queue. As soon as a job appears in this queue, the first available worker receives it and can start processing the job. The workers can be scaled up or down automatically based on the jobs in the job queue. More about this can be found in the Autoscaling section.

The deployment jobs will then first create a new inventory from which they know on which devices they have to store the configuration. After creating the inventory, the deployment template must be rendered with the correct data. Nornir will then finally store the configuration on the routers and store the result internally. For the result to get back to the backend from the workers, it was decided that the workers should store the result in a dedicated result queue. This way, the result message can already be created on the worker, and the backend can react accordingly as soon as the result has arrived.

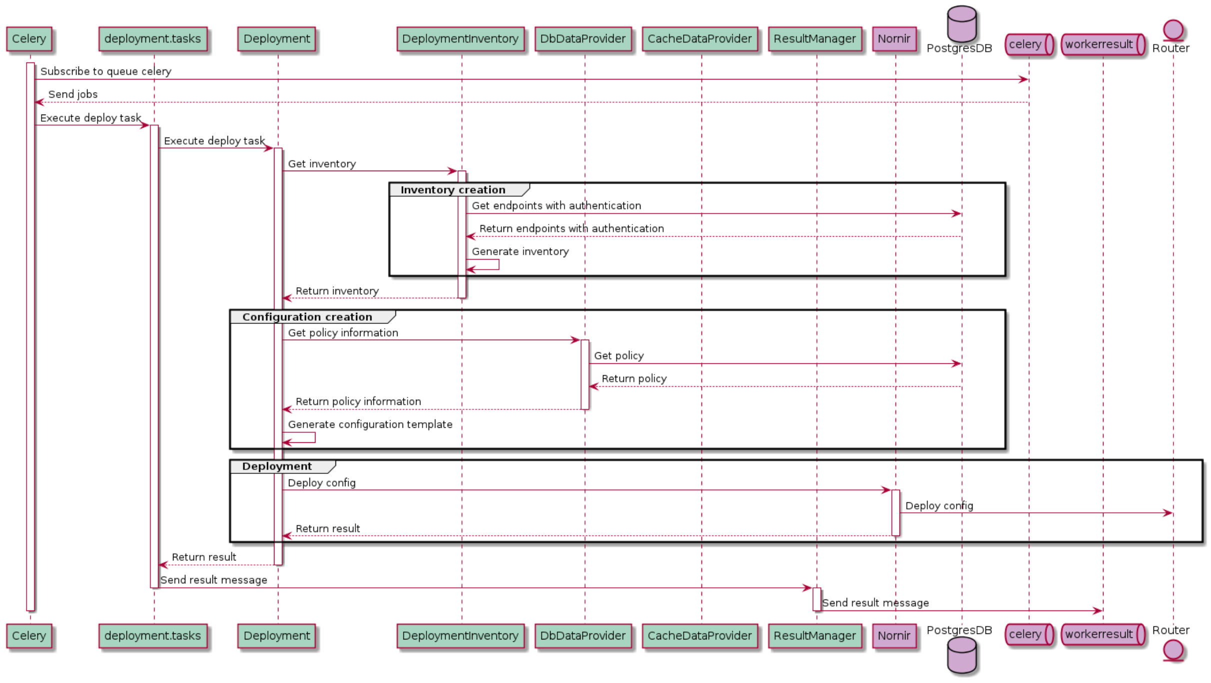


Figure 6.13: Policy Deployment

In order to be able to inform the customer directly about a deployment after it has been completed or has failed, a solution was sought in which the workers automatically inform the backend about the deployment result. After finishing the deployment, the workers put their results in a dedicated queue. Because the backend is subscribed to this queue, the backend is informed as soon as a new result is available and can then forward it asynchronously to the clients via a WebSocket message. The sequence diagram in figure 6.14 describes this process graphically.

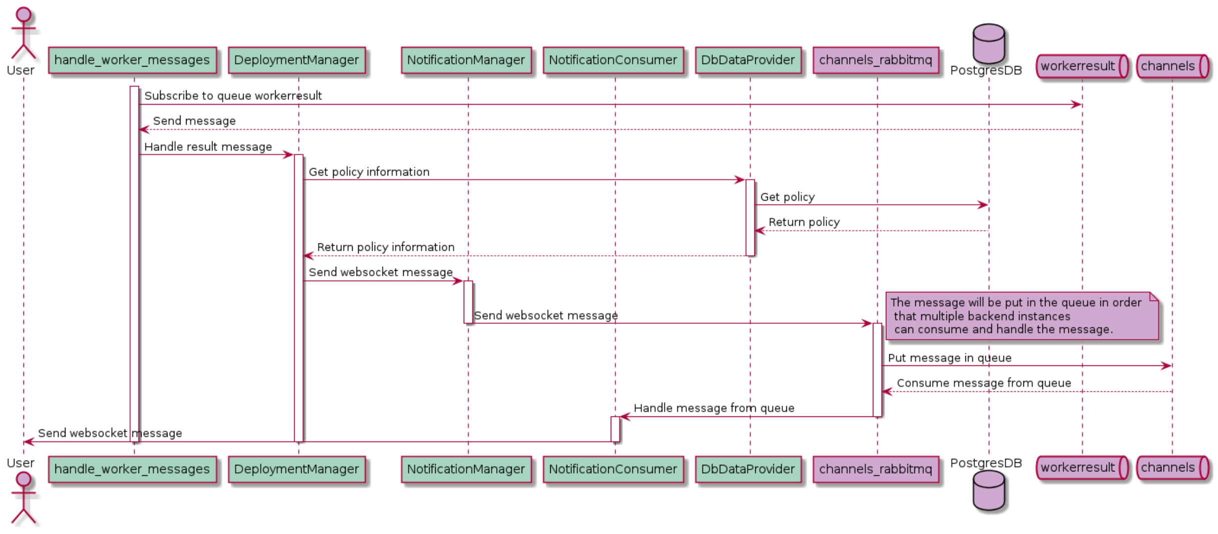


Figure 6.14: Policy Result Handling

6.6.5 Policy Reversion

The policy reversion is made analog to the Policy Deployment, but contrary to the deployment, this process reverts the policy configuration on the network devices

6.7 Role Based Access Control

Since the application can make changes in a physical network, it was necessary to control access to the application and allow the customer to create different access levels.

The access control, already prepared in the Python Django framework, was used to create a so-called Role-Based Access Control (RBAC). This makes it possible to create different groups and users out of the box and assign permissions.

In order to use a modern and secure authentication, the entire authentication for the API and the WebSocket was implemented by using the standardized and proven authentication method with JWT. Using JWT, two new routes had to be made available to the user: `authentication/token` and `authentication/refresh`. By calling `authentication/token`, the user receives `access` and `refresh` tokens to connect to the backend. After a certain period, the access token expires, and the user has to renew his access token with the `refresh` token. The exact procedure which happens when a user requests a new token can be seen in figure 6.15.

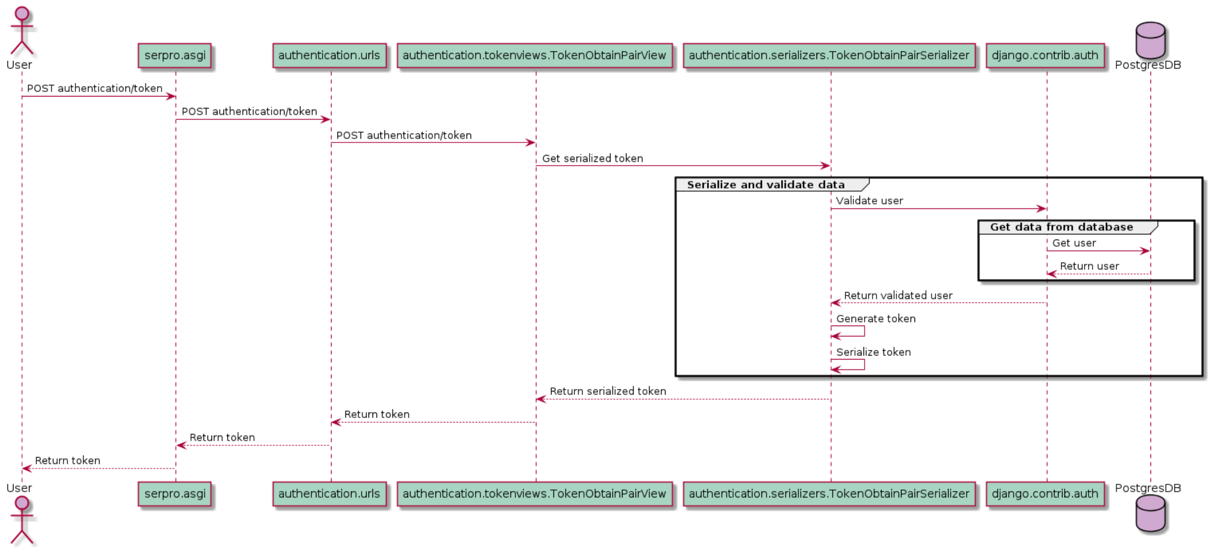


Figure 6.15: Authentication Sequence Diagram

The API will respond with `HTTP 403 Forbidden` when a user is not authenticated within the application. For the websocket, a separate middleware was written that checks, whether the user is authenticated or not when a WebSocket connects to enable authentication with JWT on the WebSocket. With this middleware, the connection can be authenticated already during the connection setup. In listing 6.11, the URL structure for the WebSocket connection can be seen, to mention is here that the `access` token is delivered with a query parameter.

```
wss://serpro.stud.pilota.ch/ws/?token=<jwt-access-token>
```

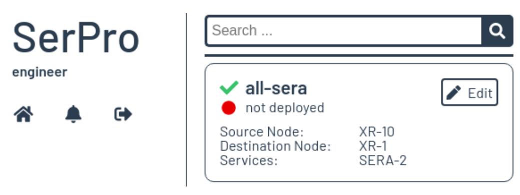
Listing 6.11: Websocket URL

It was decided that the JWT should also contain the group name to enable the frontend to show and hide buttons dynamically and adjust forms so that the frontend knows exactly which group a user is assigned to and which permissions, and therefore what buttons or forms should be displayed. An encoded example of a JWT can be seen in listing 6.12. In this example, it can be seen that the JWT also contains the group name.

```
{
  "token_type": "access",
  "exp": 1622622793,
  "jti": "8b6f3a04858649ce929e6f3a85f743ee",
  "user_id": 2,
  "username": "engineer",
  "group": "engineer"
}
```

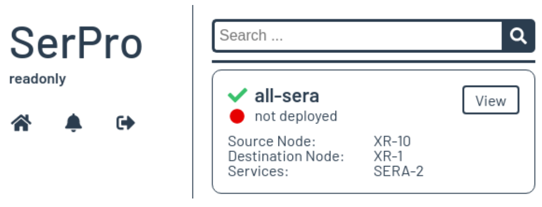
Listing 6.12: Decoded JWT

Because the frontend now knows about the group membership through the customized JWT, the design can be customized based on the user's permission. In figures 6.16 and 6.17, there can be seen the difference between a user in the `readonly` group and a user in the `engineer` group. The exact group permissions are described in table 6.4 and should show which group has permission in the application.



all-sera
not deployed
Source Node: XR-10
Destination Node: XR-1
Services: SERA-2

Figure 6.16: User in engineer group



all-sera
not deployed
Source Node: XR-10
Destination Node: XR-1
Services: SERA-2

Figure 6.17: User in readonly group

All permissions are created automatically when the application is started. However, these can also be adjusted afterwards without any problems.

anonymous	• Cannot log into the application.
read_only	• Can view all data.
operator	• Can view all data. • Can CRUD policies.
engineer	• Can view all data. • Can CRUD policies. • Can deploy policies. • Can revert policies. • Can CRUD services. • Can CRUD authentications. • Can CRUD endpoints.
superuser	• Can do anything.

Table 6.4: Group permissions

6.8 Update Processing

During the development of the entire application, the goal was to ensure that a user can always work with the latest data. That is very important because the application makes policies based on the current network status. These policies are then deployed directly to the network if desired. The network traffic is routed accordingly. If new calculations are not done on the most current data, it could be that a network has no connection to the rest of the topology. It was, therefore, crucial to have a functioning and, above all, very stable update processing.

Design-wise, the goal was not to have the backend to be responsible for detecting updates too. Therefore, it was decided to adopt and adapt the idea of the polling service, which was already created in the preliminary work SerChio. How the exact process in the polling service looks precisely can be examined in section Polling Service Update Processing.

6.8.1 Polling Service Update Processing

The application needs the data from the external software system Jalapeño. The polling service plays an essential role. It is responsible for the primary polling and detection of updates from the external software system.

The data is fetched directly from the external systems graph database ArangoDB. All these data have a checksum that allows detecting if data have been modified. The checksums of the new data are compared to the checksums of the data of a previous query. That allows the polling service to detect what data has been changed.

The big difference from the previous work (SerChio) is that the updates are no longer sent directly to the backend via a WebSocket. The data is now put into a queue on the RabbitMQ cluster via individual messages and stored individually in the cache. By processing these changes separately, it can be ensured that the checksum of the change is only stored in the cache when the update is already in the RabbitMQ queue. So if the polling service has a problem, then no updates would be lost, but it would instead send the updates a second time.

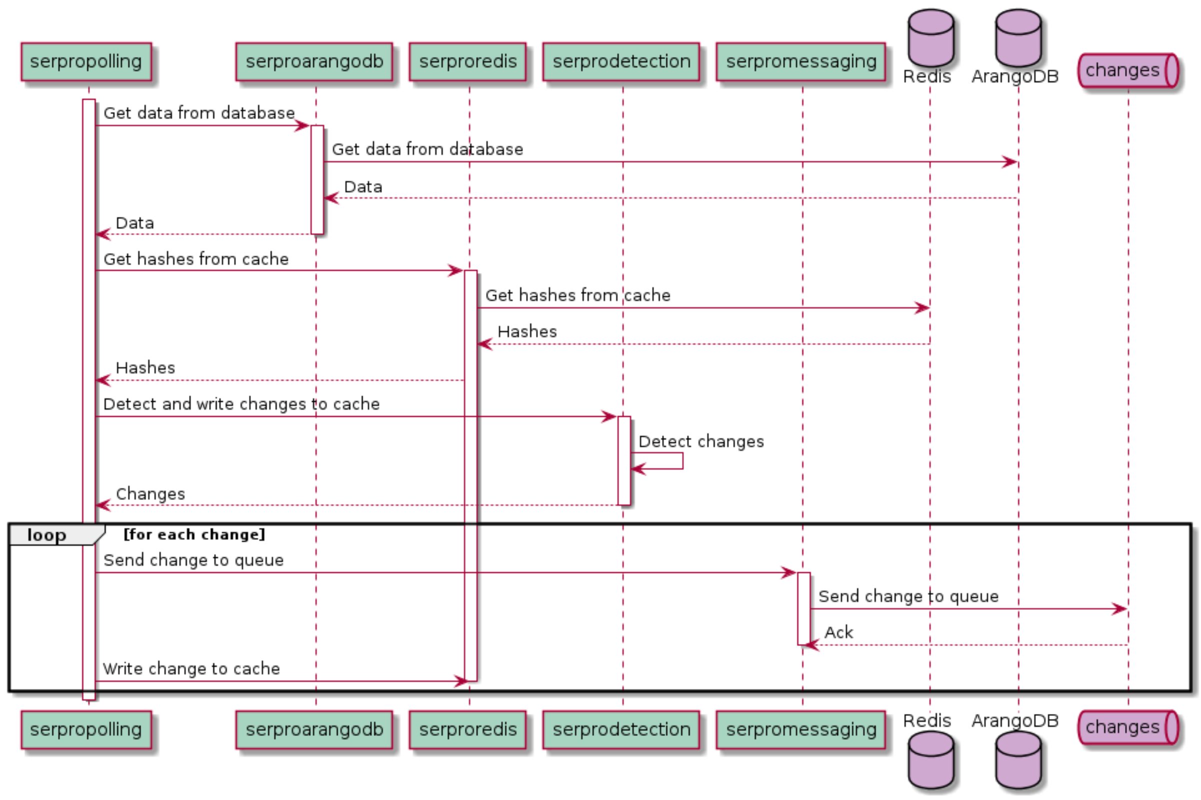


Figure 6.18: Polling Service Data Update

6.8.2 Backend Service Update Processing

The backend is responsible for keeping the data in the cache up-to-date; besides, it also has to update a complete graph of the topology on which the policy calculations happen. The backend gets all update messages directly from a dedicated queue on the RabbitMQ cluster. To avoid race conditions due to multiple scaled backends, the queue that transmits the update messages was declared a `single-active-consumer` queue. This way, only one backend instance gets the update messages, and the other instances are in standby mode. That ensures that all messages are processed in the correct order. Furthermore, the individual messages are only acknowledged at the queue and deleted from the queue when individual updates have been completed. This guarantees, that if a backend has a problem and cannot finish processing the message, the message can be treated by the next backend.

The entire process can be inspected graphically in figure 6.19.

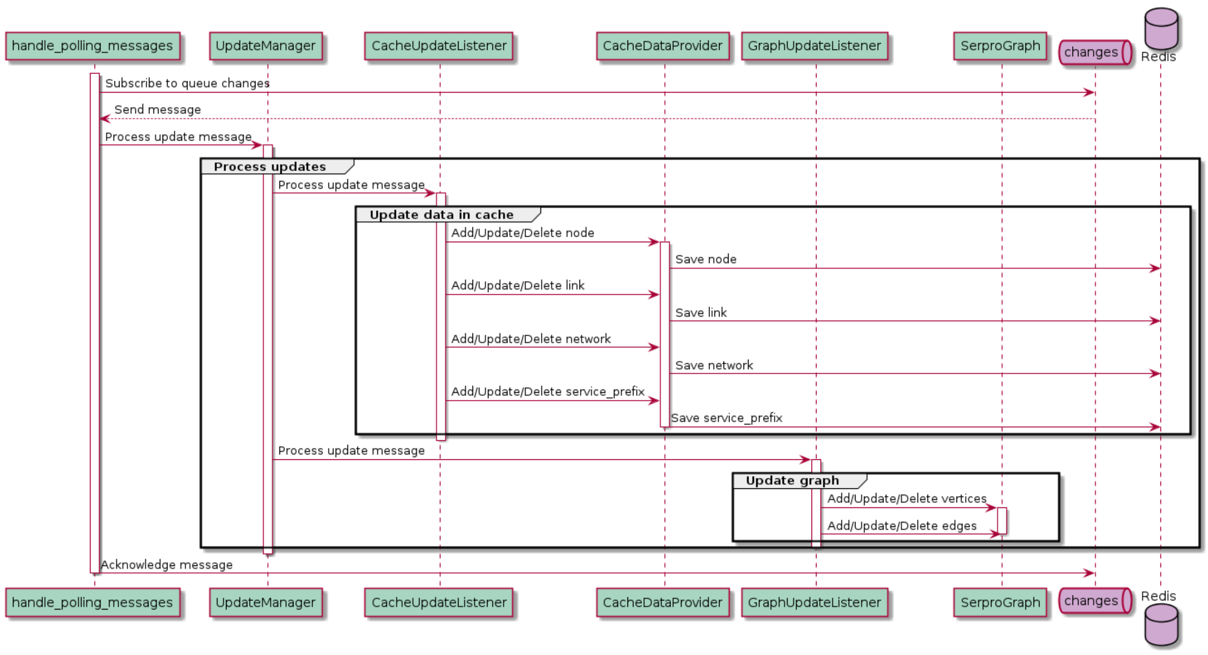


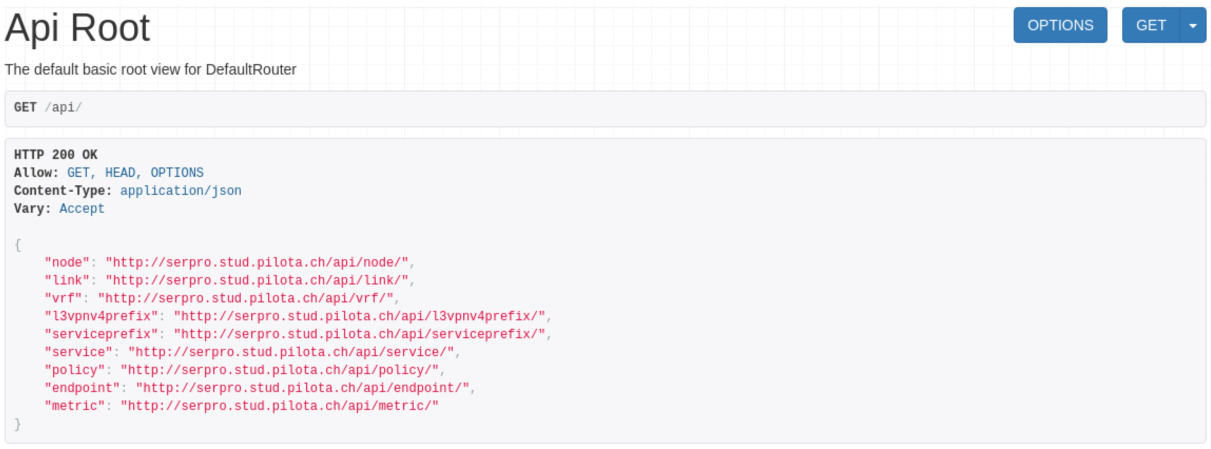
Figure 6.19: Backend Update Processing

6.9 REST API

6.9.1 General

Already in the preliminary work SerChio, an API was introduced. The same approach was followed in this bachelor thesis.

After consultation with the frontend developers, however, it was ultimately decided to revise and simplify the API. Fewer routes should be available, but these should be as simple as possible to get to the required information. The hyperlinking aimed at in the preliminary work was replaced by a classic API design. Features like pagination were not implemented, yet because they are not needed and can be upgraded without problems by using the Django Rest Framework.



Api Root

The default basic root view for DefaultRouter

```
GET /api/
```

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

{
    "node": "http://serpro.stud.pilota.ch/api/node/",
    "link": "http://serpro.stud.pilota.ch/api/link/",
    "vrf": "http://serpro.stud.pilota.ch/api/vrf/",
    "l3vpnv4prefix": "http://serpro.stud.pilota.ch/api/l3vpnv4prefix/",
    "serviceprefix": "http://serpro.stud.pilota.ch/api/serviceprefix/",
    "service": "http://serpro.stud.pilota.ch/api/service/",
    "policy": "http://serpro.stud.pilota.ch/api/policy/",
    "endpoint": "http://serpro.stud.pilota.ch/api/endpoint/",
    "metric": "http://serpro.stud.pilota.ch/api/metric/"
}
```

Figure 6.20: Backend API Root

6.9.2 Query Parameters

By introducing query parameters, the complete query logic can be transferred to the backend. The backend is then responsible for making various queries in the cache or the database and answering the respective queries correctly. A more complex example of this is displaying all VRFs with a specific export tag, which is passed as a query parameter. This query is needed to make sure that the two VRFs have a connection. The backend has to query different models in the cache and match the different export tags with the VRFs to answer this query. Because this logic is completely implemented in the backend, the frontend has to make one request only.

```
GET /api/vrf/?export_tags=1:190
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 8,
        "belongs_to_node": 11,
        "name": "VPN3",
        "rd": "12.12.12.12:3",
        "ext_community_list": [
            "1:190"
        ]
    },
    {
        "id": 6,
        "belongs_to_node": 10,
        "name": "VPN3",
        "rd": "11.11.11.11:3",
        "ext_community_list": [
            "1:190"
        ]
    }
]
```

Listing 6.13: API Query Parameter Example

6.9.3 Actions

To make the deployment, revert or acceptance of the better path as easy as possible for the user, various actions have been introduced. With the help of these actions, the user can make regular `GET` requests on a specific policy, and the backend can then react accordingly to this action. Table 6.5 lists the different available actions in more detail.

Request	Description
`GET /api/policy/<pk>/deploy`	Deploy the specific policy.
`GET /api/policy/<pk>/revert`	Revert the specific policy.
`GET /api/policy/<pk>/expand`	Get the non-clustered graph of the specific policy.
`GET /api/policy/<pk>/showbetterclustered`	Get the clustered graph of the better path of the specific policy.
`GET /api/policy/<pk>/showbetterfull`	Get the non-clustered graph of the better path of the specific policy.
`GET /api/policy/<pk>/acceptbetter`	Accept the better path of the specific policy.

Table 6.5: API Action Description

6.10 Application Deployment

The goal of the entire development was to make the application as scalable and highly available as possible. In order to achieve these goals, a deployment option had to be found that would also make this possible. Due to the cloud-native approach, the choice fell on Kubernetes. By deploying on a Kubernetes cluster, the individual services within the application can be easily scaled.

The complete deployment diagram can be seen in figure 6.21. As we can see there, different services are scaled - these are the frontend, the backend, and the workers. For a detailed description of how the whole communication between the different services works, please refer to section System Overview.

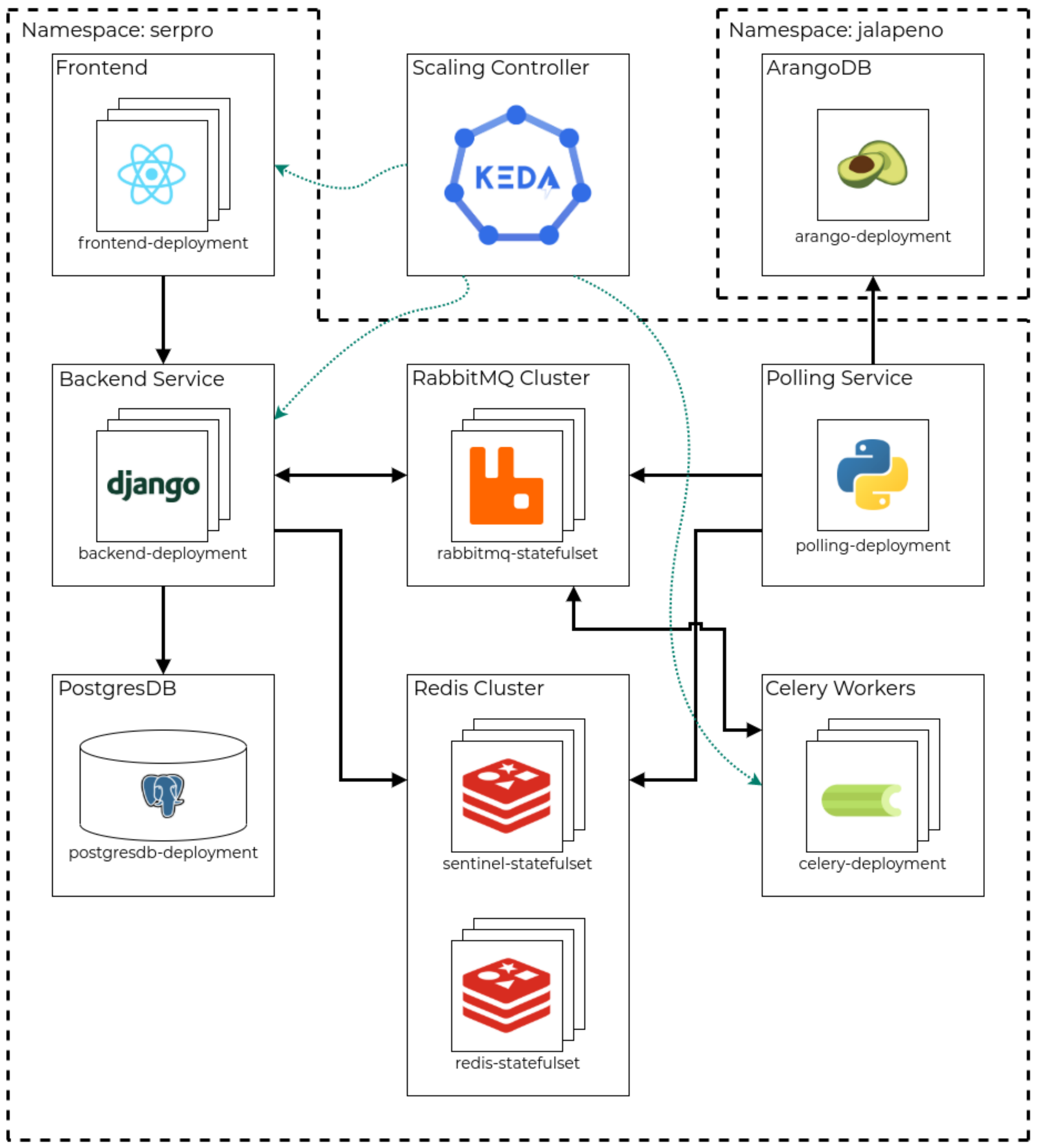


Figure 6.21: Kubernetes Deployment

The Kubernetes Package Manager Helm was used for the entire deployment. More information about the deployment with Helm can be found in section Helm Chart.

6.10.1 Helm Chart

To deploy the application on different clusters as efficiently as possible and to have the possibility to manage the application with different so-called revisions, a Helm Chart with various sub-charts was written. Using a Helm Chart, the individual configurations of the various components can be made central in a so-called `values` file. That allows the customer to deploy and manage the application on any cluster. The use of Helm Charts also offers the possibility to deploy new so-called releases of the application and jump back to earlier releases.

A Helm Chart consists of standard Kubernetes object definitions, which are defined as templates. Helm then renders this template with the predefined values from the `values` file. That provides a generic application template for the deployment on a Kubernetes cluster. In listing 6.14 an extract from the `serpro` Helm chart can be seen.

```
{{- if (default .Values.backend.autoScaling true) }}
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: {{ .Values.backend.name }}-scaledobject
spec:
  scaleTargetRef:
    name: {{ .Values.backend.name }}
  minReplicaCount: {{ .Values.backend.replicas }}
  triggers:
  - type: cpu
    metadata:
      type: AverageValue
      value: "50"
{{- end }}
```

Listing 6.14: Helm chart template example

In total, a main chart was written which deployed the entire SerPro application. This chart has several so-called sub-charts which deploy the different components which the SerPro application depends on. A detailed description of the different charts can be found in table 6.6.

`serpro`	Main chart, which will deploy all SerPro related Kubernetes objects for the frontend, backend and workers.
`serpro/rabbitmq-cluster`	Sub-chart of `serpro` chart, which will deploy a RabbitMQ cluster which is needed for the SerPro application.
`serpro/redis-cluster`	Sub-chart of `serpro` chart, which will deploy a Redis cluster with Redis slaves and Redis sentinels.

Table 6.6: Helm chart description

```
$ helm install serpro serpro --namespace test
NAME: serpro
LAST DEPLOYED: Sat Jun  5 09:26:12 2021
NAMESPACE: test
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

Listing 6.15: Helm chart deployment output

6.10.2 Autoscaling

Keda is used so that the application does not have to be scaled manually in the event of a high load volume. With the help of Keda, so-called scaled objects can be created in Kubernetes.

The scaled objects then define which metrics should be used to scale the pods.

The frontend and backend pods are scaled based on the load, ensuring that there is no latency if many different requests come to these pods. The worker pods are scaled based on the number of jobs. That has the significant advantage that, if many jobs are available for the workers, they are scaled up automatically, and the jobs can be distributed and processed faster. The scaled object connects to the RabbitMQ queue celery (see section Service Communication) and can then detect how many messages are waiting to be processed in this queue to detect the number of jobs. Keda also detects automatically when the load volume is low or when there are no waiting jobs in the queue. If this is the case, Keda can scale down automatically the pods, which is especially useful for a public cloud provider because then costs can be reduced.

6.11 Persistence

The basic persistency logic developed in the preliminary work SerChio was not changed very much in principle. However, it was decided that most of the data should be stored in the cache; the exact reasons for this can be found in section Caching. By dividing the different data between the cache and a relational database, a solution for the linking of the data had to be found. The different IDs mainly do this linking. In sections Cache Persistence and Database Persistence, the different persistency diagrams on the cache and the database are described more in detail.

In order to keep the two persistency diagrams in the same style, the different points are declared as listed which should make the different attributes graphically distinguishable.

- **Primary keys** are bold and marked with an underline.
- Foreign keys are underlined.
- *Uniqueness* is marked with an italic font.
- Not Null constraints are displayed with a purple color.

6.11.1 Cache Persistence

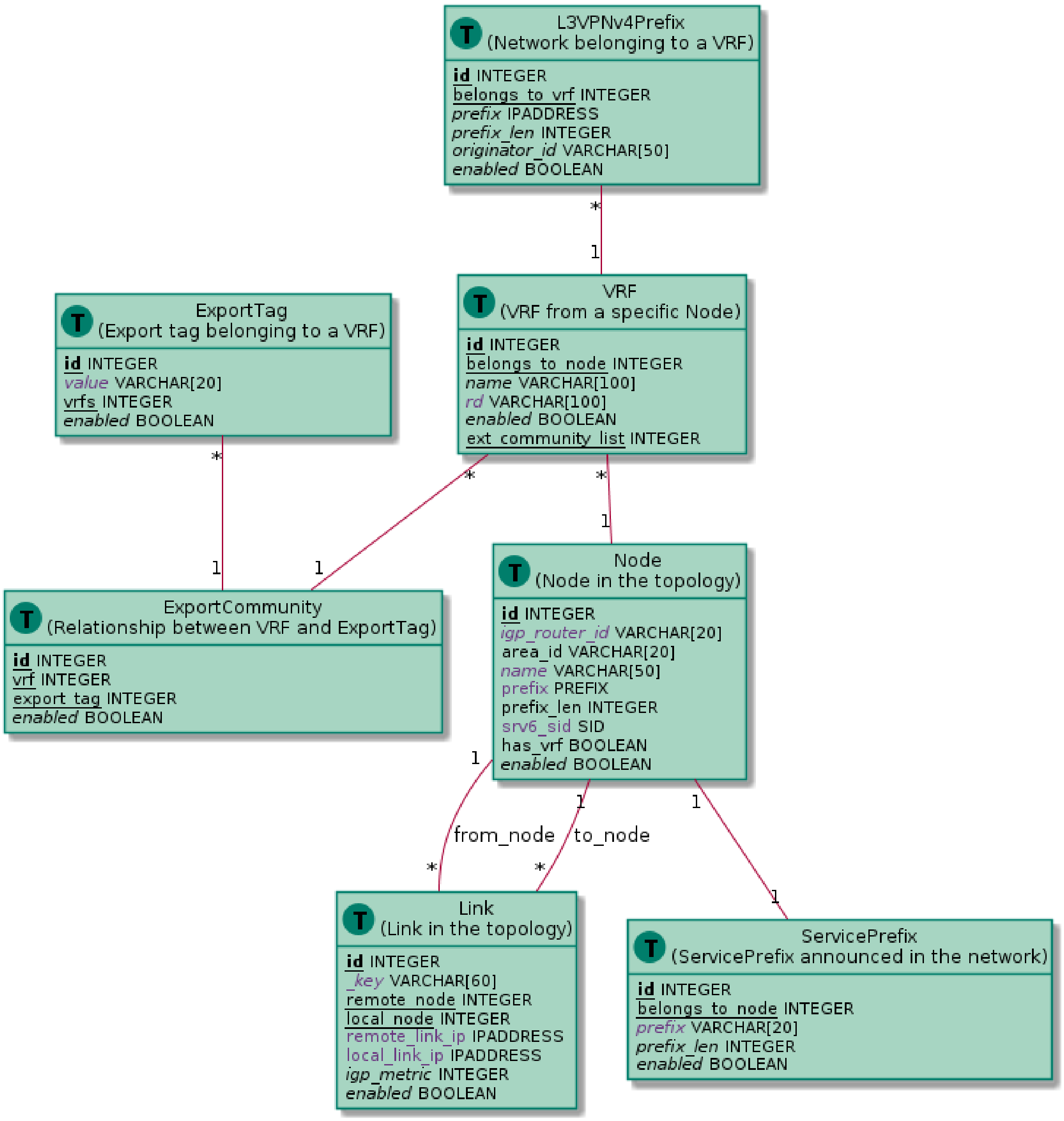


Figure 6.22: Cache Persistence Diagram

Node The Node table contains all nodes from the topology. Behind the node, a service can be installed, therefore, this table has a connection to the ServicePrefix table. A VRF is always configured on a node and a node has links connected to it.

Link is the table which contains all links connected to the nodes. A link has always a `remote_node` and `local_node` connection in order to have a relation in which direction the traffic goes.

VRF is always connected to a node. A VRF has also a L3VPNv4Prefix connection and a connection to the ExportCommunity in order to have a relation which VRF can communicate with another VRF.

ServicePrefix is the table which holds all announced service prefixes. The ServicePrefix connection is connected to the Node table because a service prefix is always configured behind a node.

ExportTag The ExportTag table contains all ExportTags that are created by the application.

ExportComunity The ExportCommunity table is used to create a relationship between the VRF table and the ExportTag table.

L3VPNv4Prefix are different networks that a VRF has. A L3VPNv4Prefix is always associated with a VRF.

6.11.2 Database Persistence

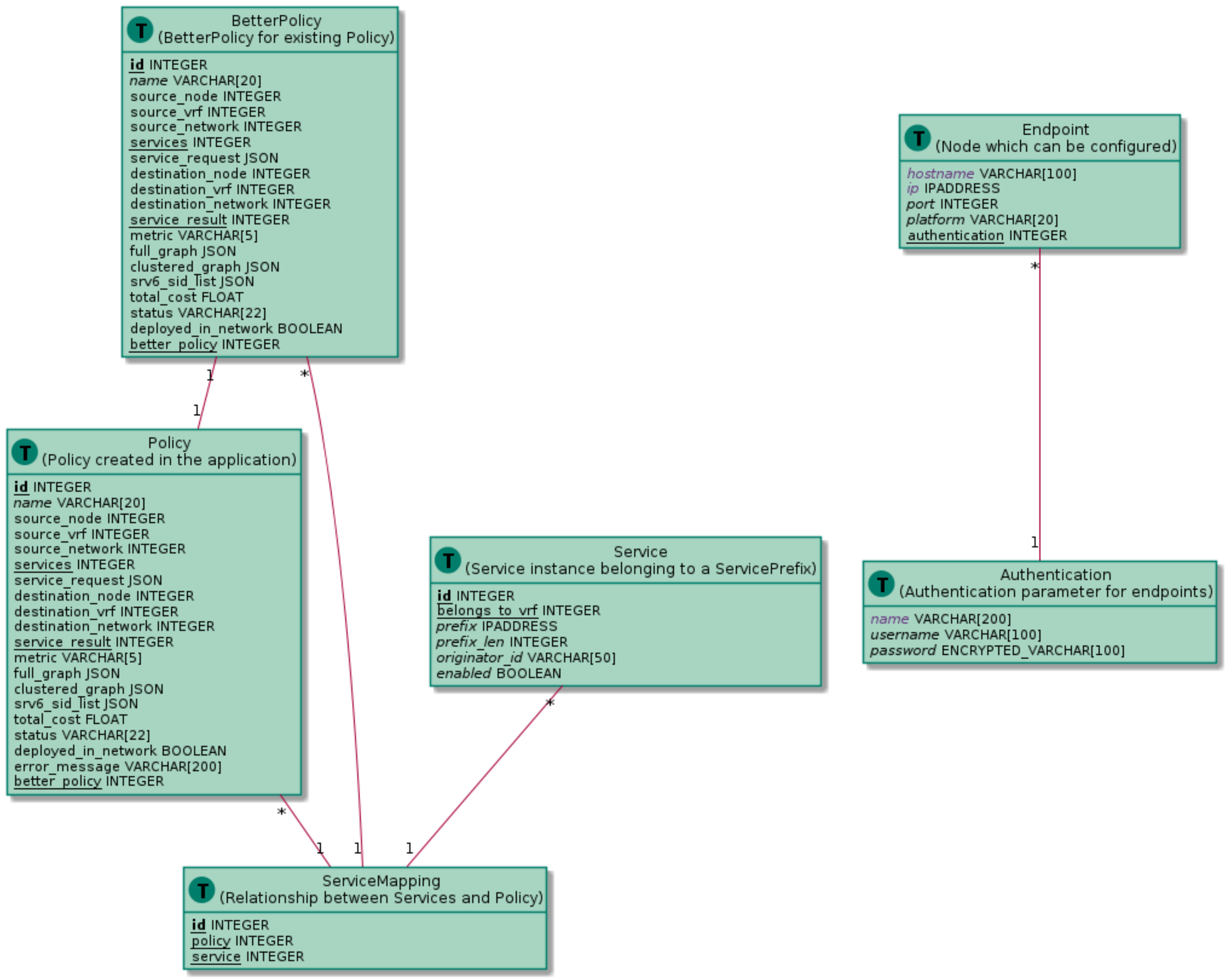


Figure 6.23: Database Persistence Diagram

Policy The Policy table is the one that contains all the information and results from the application. A policy has a connection to the BetterPolicy table if one exists. Additionally, it has a connection to the ServiceMapping, which the Django ORM implicitly creates.

BetterPolicy The BetterPolicy is like a copy of the Policy table but only contains data if a better policy was found.

Service The Service table holds a configured service instance, for example an IDS or a FW.

Authentication is the table which holds the username and the password for the endpoint connection.

Endpoint is the table which holds the management ip address of the different endpoints of the topology.

6.12 Packages and Classes

The following section will discuss the different classes and packages of the polling and backend service in more detail. During the entire development, it was essential to program against the interface and work with dependency injection, where it does make sense. That simplified the testing of the complete application time and again.

6.12.1 Backend

The backend was divided into different packages. Because the framework was used (see Programming Language and Framework), the packages could be implemented as single Django apps. Figure 6.24 shows that the entire backend application consists of a total of eight different packages, which will be discussed in more detail below.

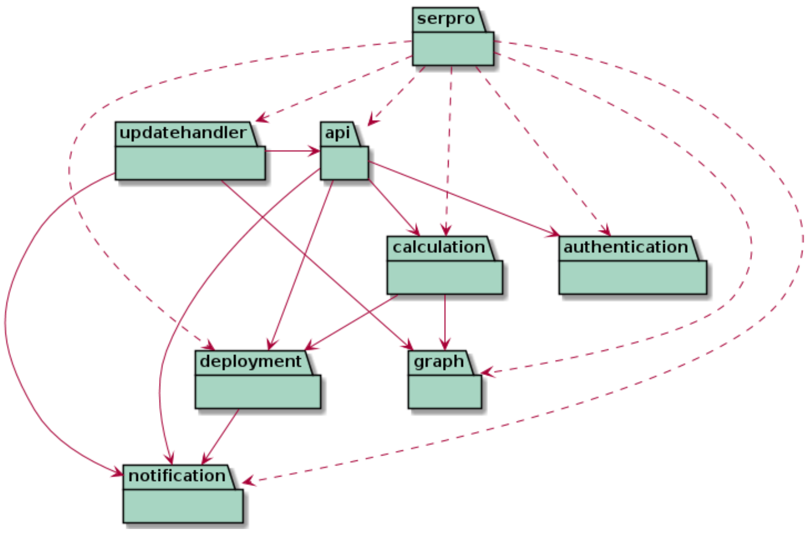


Figure 6.24: Package Diagram Backend

Package serpro

The serpro package is the main package of the whole application. This package handles the complete configurations and the different URL routes for the different Django apps. It is mainly responsible for the management of the different Django apps and has no explicit classes.

Package api

The api package is responsible for the complete REST API. It contains all data models for the cache and the relational database, and the associated data providers allow access to the respective databases. In addition, the api package contains all views, serializers, and validators needed for the REST API. Because of the size of the whole package, the complete class diagram is in the appendix under section Class Diagrams (see figure A.1). In figure 6.25, a minor diagram can be seen.

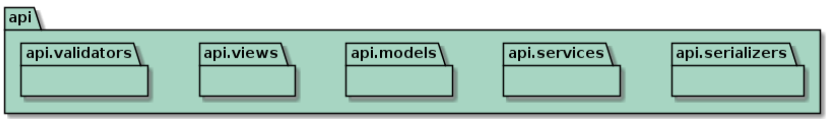


Figure 6.25: Class Diagram api

Package authentication

Since it was decided to rely on JWTs for the complete authentication (see Role Based Access Control), a separate authentication package was written, which handles the complete WebSocket authentication via its middleware. In addition, the package also takes care of the special permissions that do not come directly from the Django framework and the creation of special JWTs tailored to the frontend needs.

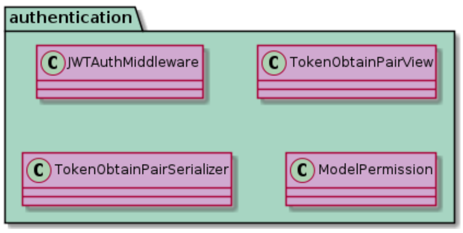


Figure 6.26: Class Diagram authentication

Package calculation

The calculation package takes care of the complex calculations of the different paths needed for the policies. The package also handles all the preprocessing and recalculations. For more information: section Calculation and Preprocessing can be consulted.

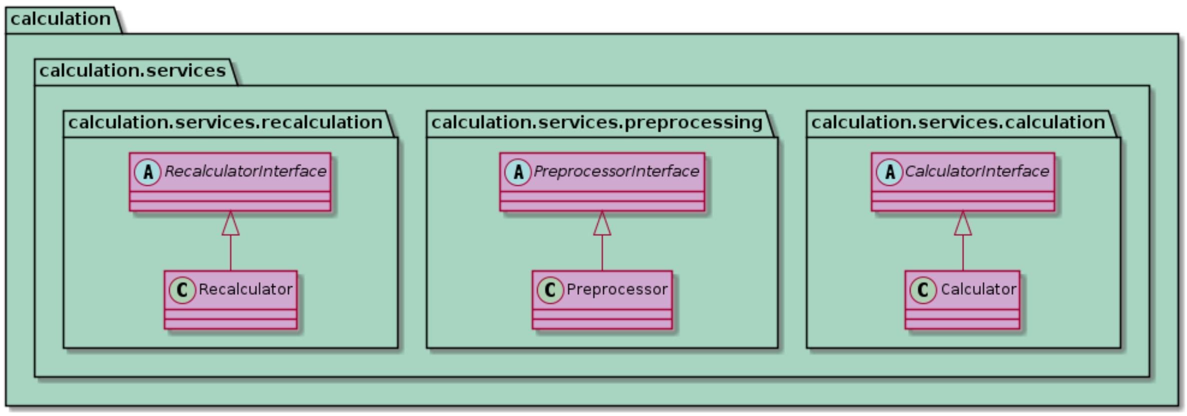


Figure 6.27: Class Diagram calculation

Package deployment

To be able to deploy all the policies, a complete deployment package was written. The deployment package handles the different tasks for Celery, the admin-specific settings to generate the inventory (see Policy Deployment), and the job results' handling.

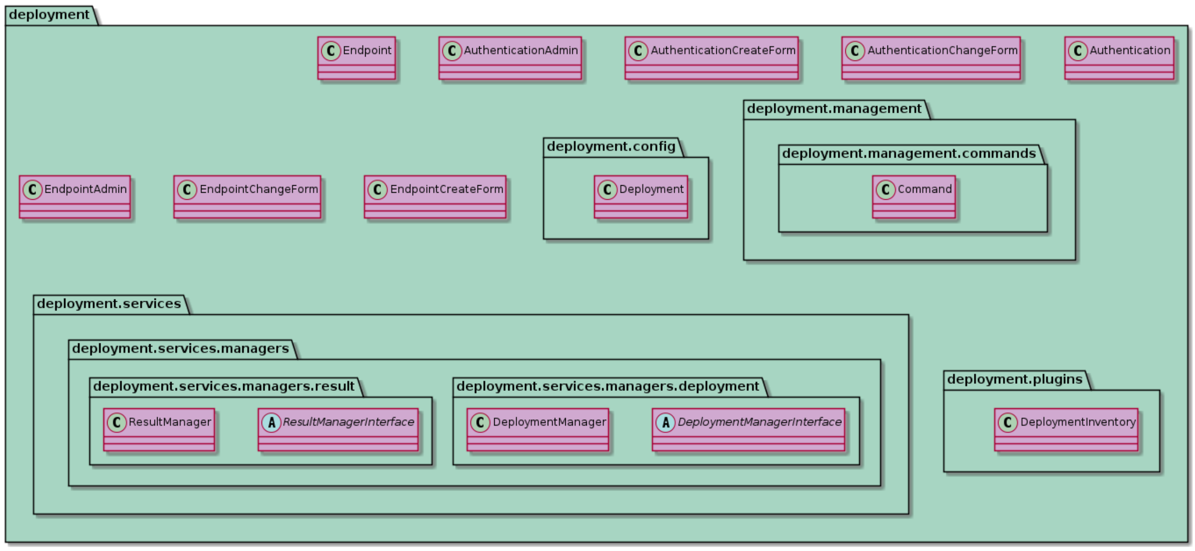


Figure 6.28: Class Diagram deployment

Package graph

The graph package is responsible for the complete graph, which is needed for the different path calculations. In order to maintain only one graph at a time in a backend, it was decided to implement the graph class as a singleton. That ensures that the respective backend instances only maintain one graph instance and perform calculations on it. Additionally, that minimizes error sources and ensures that no inconsistencies can occur.

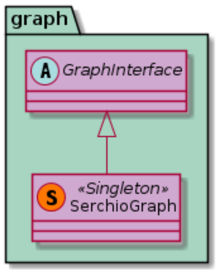


Figure 6.29: Class Diagram graph

Package notification

The complete notification package was developed to provide a unified solution for the other packages to send WebSocket messages to the clients. If another package wants to inform the frontend about any changes, it can simply use this package and send the messages through it.

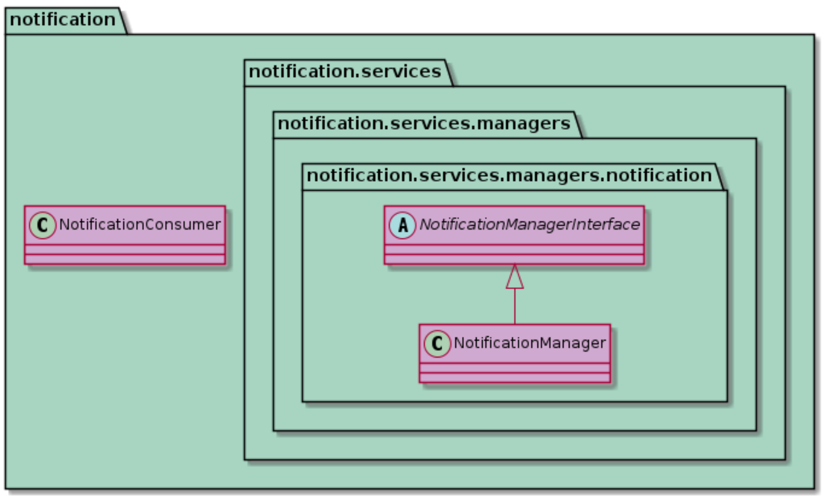

notification
notification.services
notification.services.managers
notification.services.managers.notification
NotificationConsumer
NotificationManagerInterface
NotificationManager

Figure 6.30: Class Diagram notification

Package updatehandler

The updatehandler package was written to process the update messages generated by the polling service and keep the complete data in the backend up-to-date. This is responsible for maintaining the various data in the cache and keeping the graph up-to-date. Due to the generic structure of the updatehandler package, listeners can be added without much effort, which allows the package to perform updates on different levels (currently on the cache and on the graph).

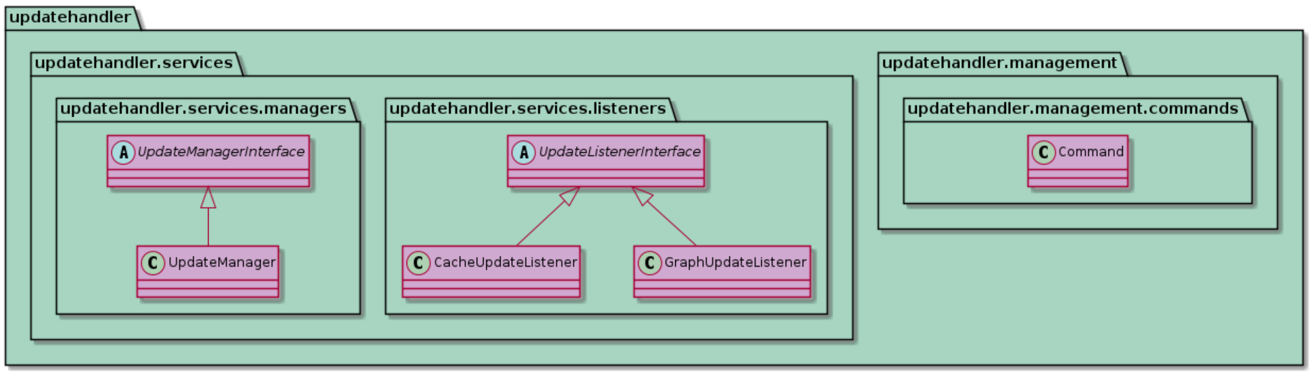


Figure 6.31: Class Diagram updatehandler

6.12.2 Polling

The complete polling service consists of one package only. The polling service was designed, so that the individual components can be replaced as efficiently as possible should something change in the external software system Jalapeño. Therefore, it was paid attention to work everywhere with dependency injection to keep a change of a class as simple as possible. By strictly adhering to the instruction to program against the interface as much as possible, many external components, that are not so easy to test, could be replaced by different mocks during testing.

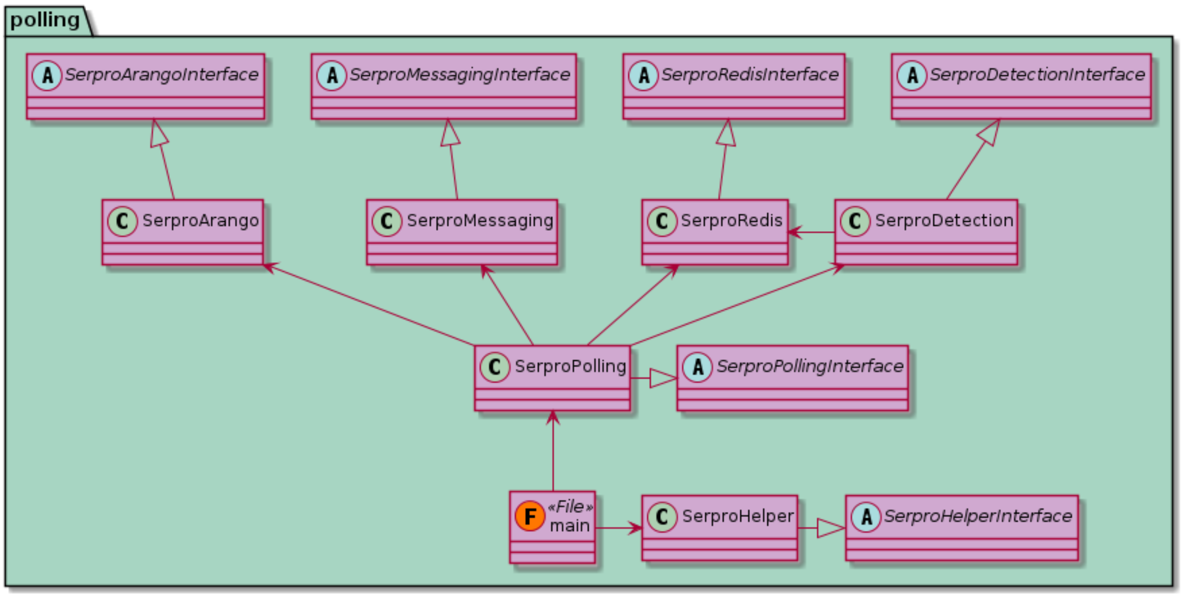


Figure 6.32: Class Diagram Polling

Chapter 7

Project Management

This chapter contains the entire project management-specific topics. It includes an overview of the project plan, including milestones, sprints, and time management. Besides, it includes developer concepts and the project-related risk analysis. During the project, this chapter is changed and adjusted based on the course of the project. In table 7.1. the version history can be found stating the mentioned changes.

Date	Version	Changes	Author
24.02.2021	1.0	Initial project plan was created.	jklaiber, sdellsperger
03.03.2021	1.1	Update Iteration Planning (E1)	sdellsperger
10.03.2021	1.2	Risk Management Update (R3 eliminated - prototype)	sdellsperger
17.03.2021	1.3	Update Iteration Planning (S1)	jklaiber
31.03.2021	1.4	Update Iteration Planning (S2)	sdellsperger
14.04.2021	1.5	Update Iteration Planning (S3)	sdellsperger
23.04.2021	1.6	Risk Management Update (R1, R2 eliminated - Alpha Release Software)	jklaiber
28.04.2021	1.7	Update Iteration Planning (S4)	jklaiber
12.05.2021	1.8	Update Iteration Planning (S5)	sdellsperger
20.05.2021	1.9	Risk Management Update (R4, R5, R7 eliminated - Beta Release Software)	sdellsperger
26.05.2021	1.10	Update Iteration Planning (S6)	jklaiber
29.05.2021	1.11	Update Responsibilities, Repositories, Infrastructure, Development Concept and Environment, Continuous Integration, Code Metrics	jklaiber, sdellsperger
30.05.2021	1.12	Update Exception Handling, Logging, Testing	jklaiber
09.06.2021	1.13	Update Iteration Planning (T1) and remove last meeting info	jklaiber

Table 7.1: Version History Project Mangement

7.1 Project Management

The introduced project management method is called Scrum Plus, suggested by the Eastern Switzerland University of Applied Science. The procedure is a mixture of the well-known methods Rational Unified Process and Scrum. Therefore, it combines certain aspects of an agile and iterative approach (Scrum) with a classical strategy (RUP). The result is visualized in the project timeline in picture 7.1: The project contains the four RUP phases Inception, Elaboration, Construction, and Transition. Furthermore, various milestones to take project management measures are included. Also, the thesis is organized with sprints to develop the results incrementally.

7.2 Scheduling

The bachelor thesis at the Eastern Switzerland University of Applied Science is accredited with 12 ECTS credits. Because one credit is considered 30 hours of work, the work per person is 360 hours. Following, the two-member team has a total of 720 hours for the completion of the thesis at disposal.

During the thesis, the team will use the whole available time. Because the project is organized with an agile approach, the most considerable possible scope is developed in the available time. The scope will be adjusted and prioritized with exact planning to fulfill the needs and result in the best possible end product.

Next, the project timeline is shown. It contains information about events, terms, and fixed dates. Besides, it shows the project-specific topics like milestone and sprints in a visualized manner. Because this bachelor thesis is a follow-up project of a project thesis, the planning was optimized to fulfill the needs and give the authors the most suitable approach to deliver the best result in the end.

The sprints in the Construction phase are fixed and defined as a length of two weeks. In this part of the project, the application is realized, and therefore, it is quite essential to have orderliness in it. The sprints *"E1"* and *"T1"* are adjusted for the optimal length to start respectively finish the project. More information on Iteration Planning can be obtained in table 7.2.

The project contains four milestones to check if the project is on track. This number is the perfect cap between project controlling and minimizing the management lead. This decision is strongly related to the fact that this thesis is a follow-up project. More detailed information can be found in chapter Milestones

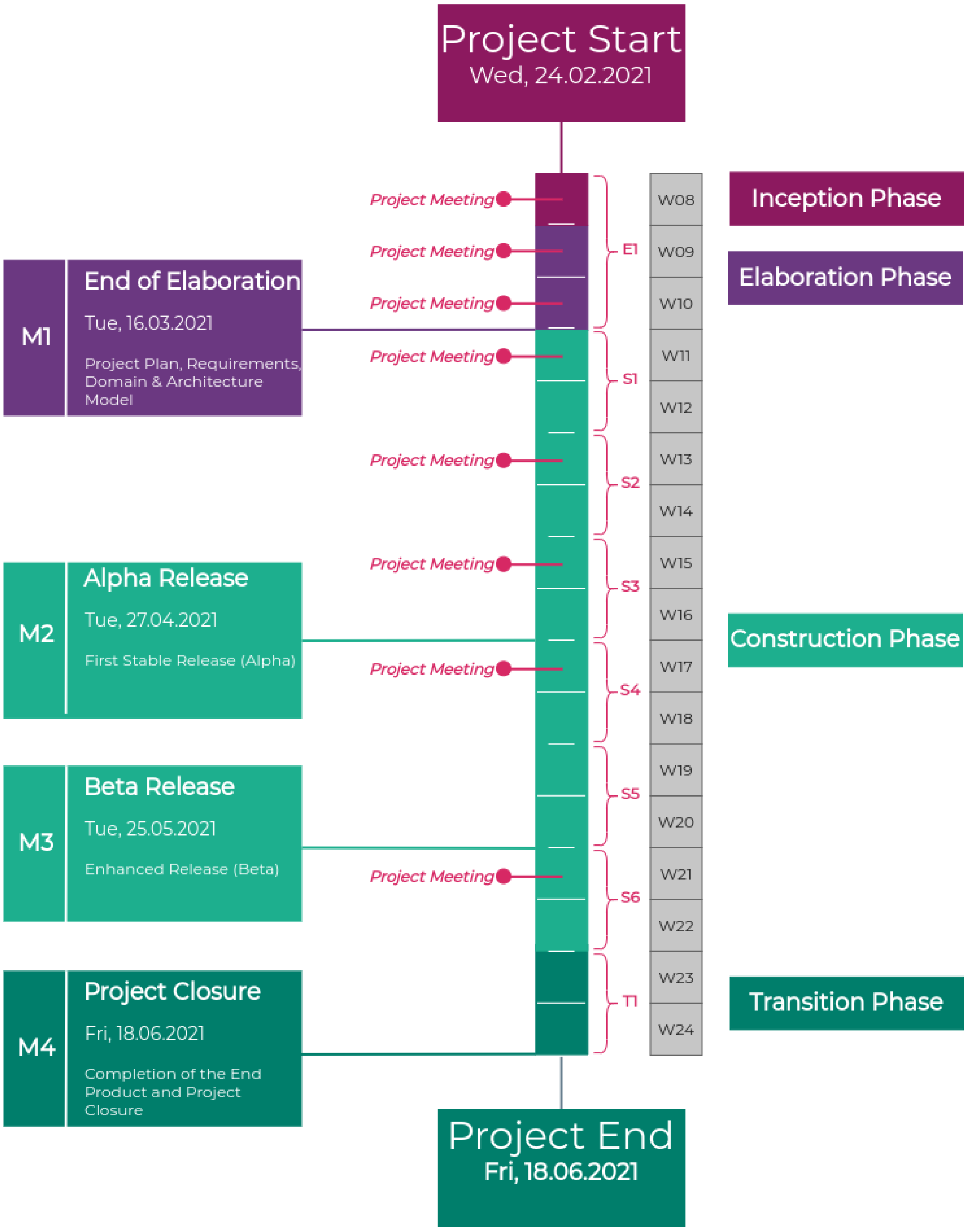


Figure 7.1: Project Timeline

7.2.1 Iteration Planning

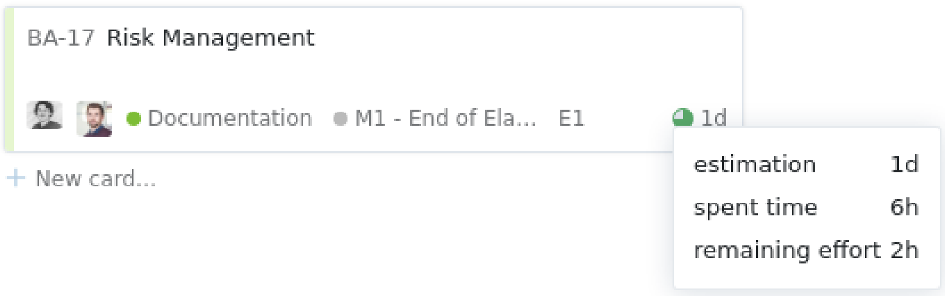
During planning, it was decided that there would be regular sprints in the Construction phase. Due to the very short Inception and Elaboration phase, which was given by the fact that it is a follow-up project, a so-called Elaboration sprint was introduced which lasts longer than usual. This ensured that the Construction sprints could be carried out properly. Due to the agile approach, the respective work packages are always defined at the beginning of the sprint. Therefore, table 7.2 is continuously being updated during the project.

	Inception	
Elaboration 1	24.02. - 02.03.	Kickoff meeting with industrial partner
		Creation of the project plan
	Elaboration	
Elaboration 1	03.03. - 16.03.	Use Cases and Non-Functional Requirements
		Development Environment
		Architecture Prototyp
	Construction	
Sprint 1	17.03. - 30.03.	Adjust Polling Application to SRv6
		Data Caching
		Communication between Polling and Backend
Sprint 2	31.03. - 13.04	Backend Graph Implementation
		API Route
		Calculation and Preprocessing
Sprint 3	14.04. - 27.04.	Improve Calculation and Preprocessing
		Worker Implementation
		Architecture Prototyp
Sprint 4	28.04. - 11.05.	Improve Calculation and Preprocessing
		Authentication and Login
		Handle Topology Changes
		Enabling/Disabling of Components
Sprint 5	12.05. - 25.05.	Exception Handling
		Unit Testing
Sprint 6	26.05. - 08.06.	System Testing
		Documentation
	Transition	
Transition 1	09.06. - 18.06.	Documentation Finalization

Table 7.2: Iteration Planning

7.2.2 Estimates

During the project, each task is created, estimated and the spend time is assigned. These tasks are performed in the YouTrack tool, making it possible to track the time and export different time reports.


BA-17 Risk Management

Documentation · M1 - End of Ela... E1 · 1d

New card...

estimation	1d
spent time	6h
remaining effort	2h

Figure 7.2: YouTrack Estimates

7.2.3 Time Evaluation

The complete time evaluation is made with the information available in the YouTrack application. Thus, the authors can not only review but also improve their time-relevant actions, such as estimation.

7.3 Milestones

A total of four milestones were defined to track the project progress. Because this bachelor thesis is a follow-up project of a project thesis, this number of milestones seems to be the ideal choice. Because there is no need for an intensive Inception phase, it was decided to set the first milestone to the end of the Elaboration phase. There are two significant milestones in the construction phase: The Alpha and the Beta Release of the application under development.
Three out of the four milestones are finished on a Tuesday. The milestone review is made at the next project meeting on the subsequent Thursday. The project's status is presented to the supervisor and the industrial partner, and essential topics are reflected at this meeting. The last milestone, the Project Closure, is presented at the Bachelor presentation, which point in time has to be defined after the actual project.
During the project, the work is done as agile as possible, and therefore, the result is developed incrementally.

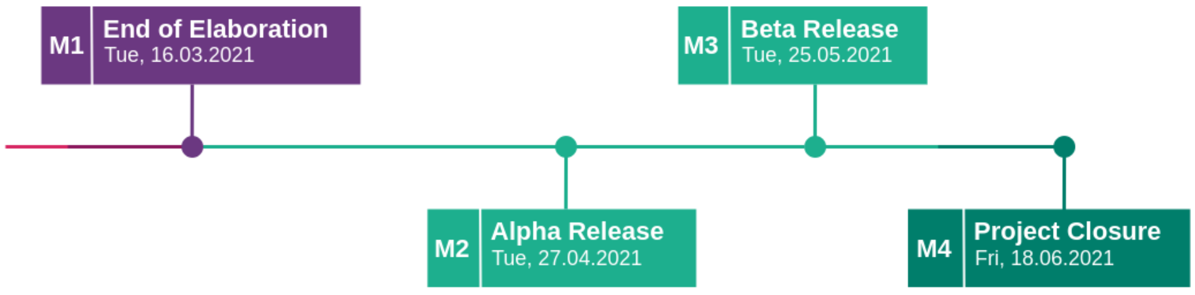


Figure 7.3: Milestone Overview

Milestone	Due Date	Goals
M1 - End of Elaboration	16.03.2021	The goal is to complete the whole project plan, the requirement specification, including the use cases and the non-functional requirements, and the domain analysis. All the risks should be minimized to an acceptable level or eliminated with the help of a prototype.
M2 - Alpha Release	27.04.2021	The goal is to have a first running version of the application under development.
M3 - Beta Release	25.05.2021	The goal is to have a more mature and improved release; also, there is a feature freeze.
M4 - Project Closure	18.06.2021	The goal is to close the project with a working application and complete the essential parts.

Table 7.3: Milestone Description

7.4 Meetings

Meetings are supposed to discuss open relevant project circumstances and speak about the stage of the project. At these meetings, the thesis's authors, the advisor, the co-advisor, and the industrial partner are invited. Francois Clad and Ahmed Abdelsalam represent the project partner.

Due to the unique situation (COVID-19), the project meetings will take place online in a Microsoft Teams meeting.

Because communication is quite important to set the direction and goals initially, there will be a weekly meeting on Thursdays in the first four weeks. Afterward, the Thursday meetings will be repeated in a two-week stroke.
After the Alpha release, the mid-term presentation is held, and also the current Alpha Release is presented. Afterward, in week 19, the meeting is cancelled due to a public holiday. The rest of the time, the meetings are executed in the defined rhythm. There is no meeting planned in the transition phase because the team will concentrate on the finish of the thesis in this period.

An overview about the planned meetings can be found in illustration 7.1

7.5 Responsibilities

According to the application structure, the application to develop is divided into two primary responsibilities: the front- and the backend.
In the beginning, it was defined that the authors do not maintain the frontend application. The supervisor decided that this task is allocated at the Institute of Networked Solutions.

The second responsibility - the backend and polling application - is a crucial part of this bachelor thesis. Therefore, the responsibilities and tasks are shared between the two thesis authors. The authors decided to distribute all the tasks and responsibilities according to their interests, their main working topics, abilities and strengths.

After the application design was finished, the responsibilities were defined and distributed. Severin Dellsperger focused more on the network-related parts, whereas Julian Klaiber concentrated on the cloud-native parts like the Kubernetes deployment and CI/CD. So, the team amends their skills to produce the most beneficial output during this thesis.

7.6 Repositories

In order to have the possibility to develop the different services independently from each other and through the usage of a dedicated message queue, which allows a standardized way of communication between the different services, two complete separated repositories are created for the services. One repository holds all the code for the backend service, the other is only responsible for the polling service code. Because the complete deployment contains a lot of different files for all the different components, the team members decided to create an own repository for the deployment base.

7.6.1 SerPro Backend Repository

The serpro-backend repository holds the code base for the whole backend service. The structure of the repository can be overviewed in table 7.4

SerPro Backend	
README.md	Project description
images	Image directory for Readme
.gitlab-ci.yml	Pipeline definition
requirements.txt	Pyhon requirements
requirements-dev.txt	Pyhon development requirements
Makefile	Development commands
set-env.sh	Development environment file
manage.py	Django management service
build	Build related files
Dockerfile.prod	Multistage Dockerfile for production
entrypoint.prod.sh	Docker entrypoint script for production
supervisord.conf	Supervisord configuration file
api	API module
authentication	Authentication module
calculation	Calculation module
common	Common module for development and testing
deployment	Deployment module
exceptions	Exception module
graph	Graph module
notification	Notification module
serpro	General application module
updatehandler	Updatehandler module

Table 7.4: SerPro Backend Repository

7.6.2 SerPro Polling Repository

The serpro-polling repository holds the code base for the whole polling service. The structure of the repository can be overviewed in table 7.5

SerPro Polling	
README.md	Project description
images	Image directory for Readme
.gitlab-ci.yml	Pipeline definition
requirements.txt	Pyhon requirements
requirements-dev.txt	Pyhon development requirements
Makefile	Development commands
set-env.sh	Development environment file
Dockerfile.prod	Multistage Dockerfile for production
Dockerfile	Dockerfile for development
main.py	Application entrypoint
polling	Polling module
serpro*.py	Polling related classes
interfaces	Interface module
iserpro*.py	Polling related interfaces
tests	Test module
test_serpro*.py	Polling related classes
mocks	Mock module
exceptions	Exception module

Table 7.5: SerPro Polling Repository

7.6.3 SerPro Kubernetes Repository

The serpro-k8s repository holds the code base for the whole deployment. The structure of the repository can be overviewed in table 7.6

SerPro K8s

```
├── README.md ............................................ Project description
├── images ......................................... Image directory for Readme
├── dev-env ...................................... Values files for development
└── serpro ................................................... SerPro Helm Chart
    ├── charts ................................. Subcharts for SerPro application
    │   ├── rabbitmq-cluster ......................... RabbitMQ cluster Helm Chart
    │   ├── redis-cluster ............................... Redis cluster Helm Chart
    │   └── postgresql-10.3.14.tgz .................... Fixed PostgreSQL Helm Chart
    ├── templates ......................................... Template file directory
    │   ├── backend ............................... Backend related deployment files
    │   ├── frontend ............................. Frontend related deployment files
    │   ├── polling ............................... Polling related deployment files
    │   ├── worker ................................. Worker related deployment files
    │   └── _helpers.tpl ................................................ Helm Chart helpers
    ├── Chart.lock ............................... Chart lock for PostgreSQL Chart
    ├── Chart.yaml ..................................................... Chart description
    └── values.yaml ................................................ Default values file
```

Table 7.6: SerPro Kubernetes Repository

7.7 Infrastructure

For the external system Jalapeño, which is needed for the topology data and developer services like Sonarqube, a dedicated virtual machine with a kubeadm Kubernetes cluster was installed in the network of the Institute network for Networked Solutions.

The application is developed directly in dedicated containers on a Kubernetes cluster at the Institute for Networked Solutions. A detailed description of the development environment can be found in section 7.9.

The Institute for Networked Solutions provided two Kubernetes clusters to test the application and to run it in production.

As version control, the official GitLab instance was used with a dedicated runner, which also runs on the virtual machine, to build the containers.

7.8 Development Concept

7.8.1 Definition of Done

In order to be always on the right track and to close issues only when they are completely finished, a definition of done was defined.

- Code Style Guidelines are met.
- The Continuous Integration run without errors.
- All unit tests run successfully.
- The system tests were successful.

7.8.2 Code Style Guidelines

It was decided to follow the PEP8 style guidelines for both team members to write the cleanest code possible and speak from the same guidelines. Additional typing was used to make the code even more readable and understandable for external programmers. More about this can be found in section 6.4.2.

In addition to the style guidelines, it was decided to use an auto formatter to ensure that the code has the same style everywhere.

7.8.3 Development Workflow

A development workflow, which can be seen in Figure 7.4 was already created in the preliminary work[3]. This workflow proved itself very well, which is why it was decided to adopt it and continue it in the bachelor thesis.

For the sake of completeness, the workflow has been included here again.

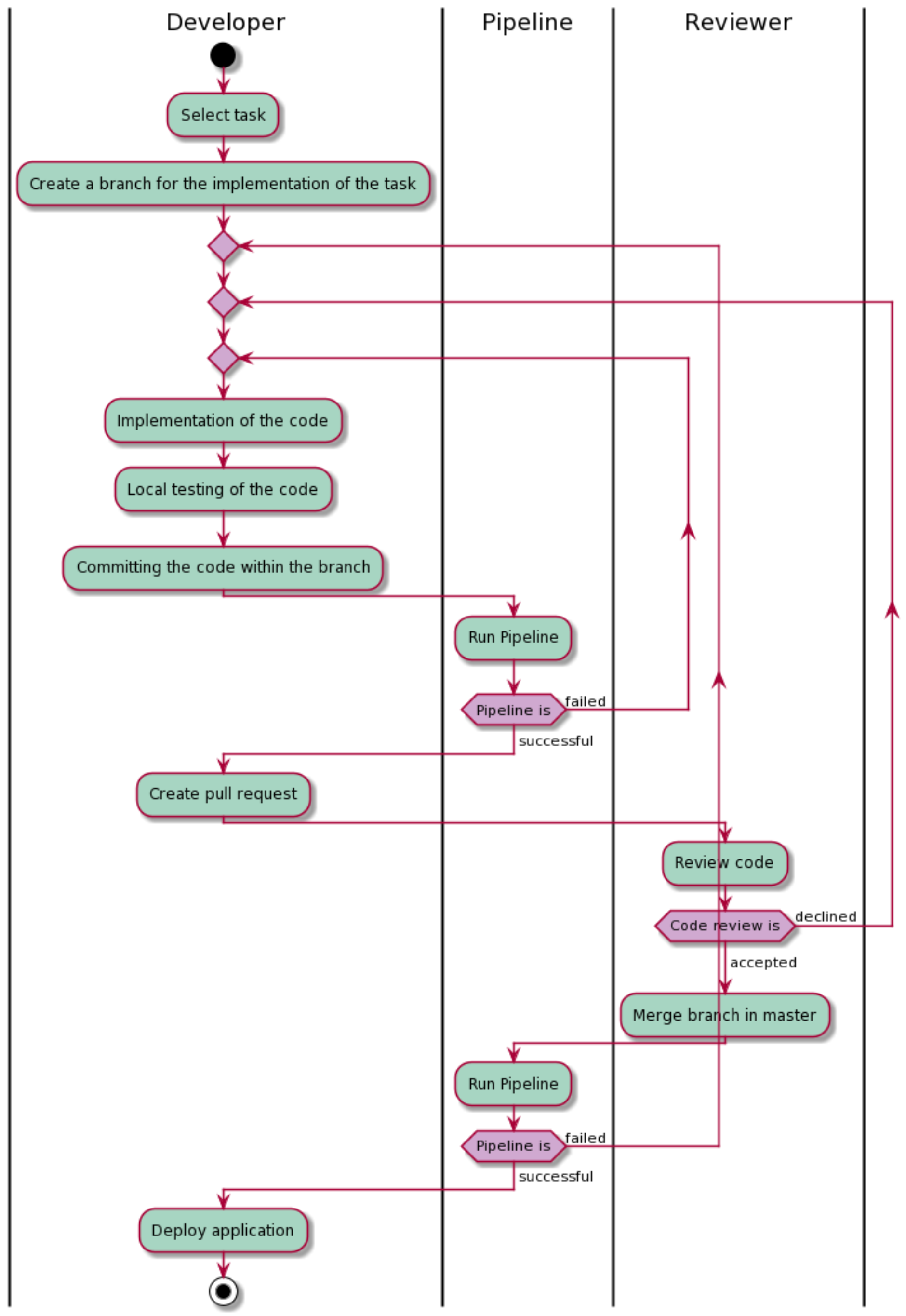


Figure 7.4: Development Workflow[3]

7.9 Development Environment

A uniform development environment was used for the entire development of the application. Since the entire application is ultimately to run on a Kubernetes cluster, it was decided that development should also take place directly in the cluster. The decision was made to use Visual Studio Code as the development environment since both team members had already worked with it, and as it offers good Python support. For the development directly in the cluster, the tool Okteto was chosen, which, together with the Remote Development Extension in Visual Studio Code, allows to set up an own Python container directly in the cluster. The codebase is synchronized from the local system directly in the dedicated container in the cluster and vice versa. The entire development environment can be viewed graphically in Figure 7.5.

This approach offers several advantages, which are described below:

- Use the components like in the production environment.
- Possibility to use the Kubernetes services.
- No local dependencies needed to install.
- Complete cloud-native development.

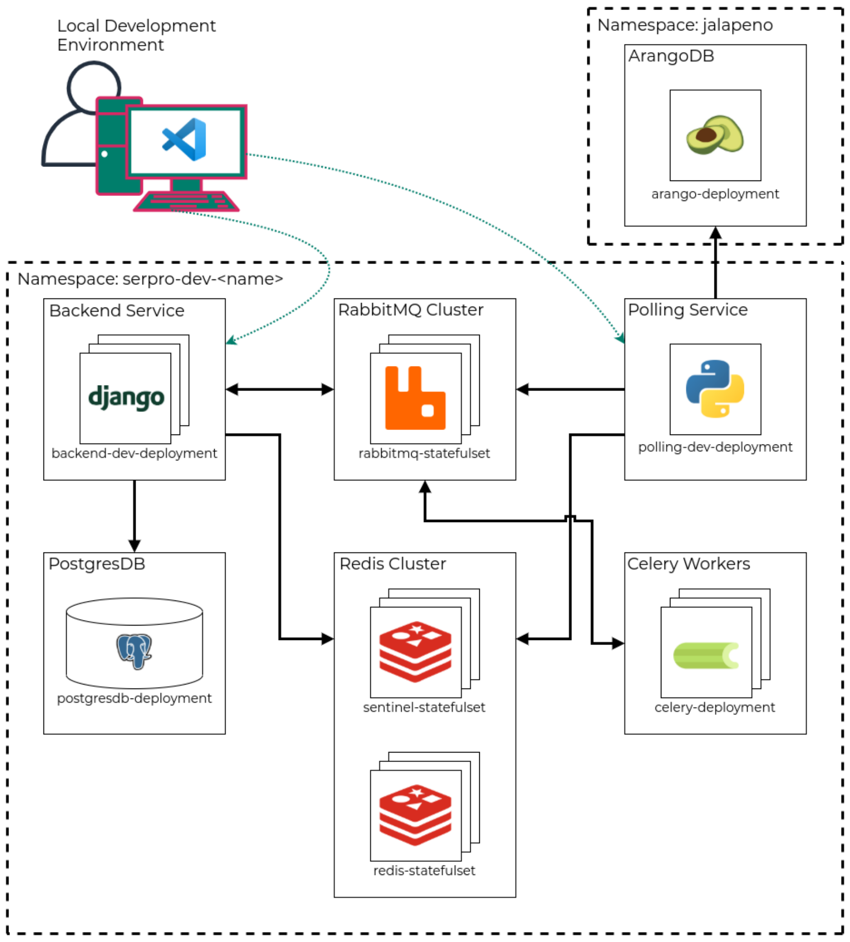
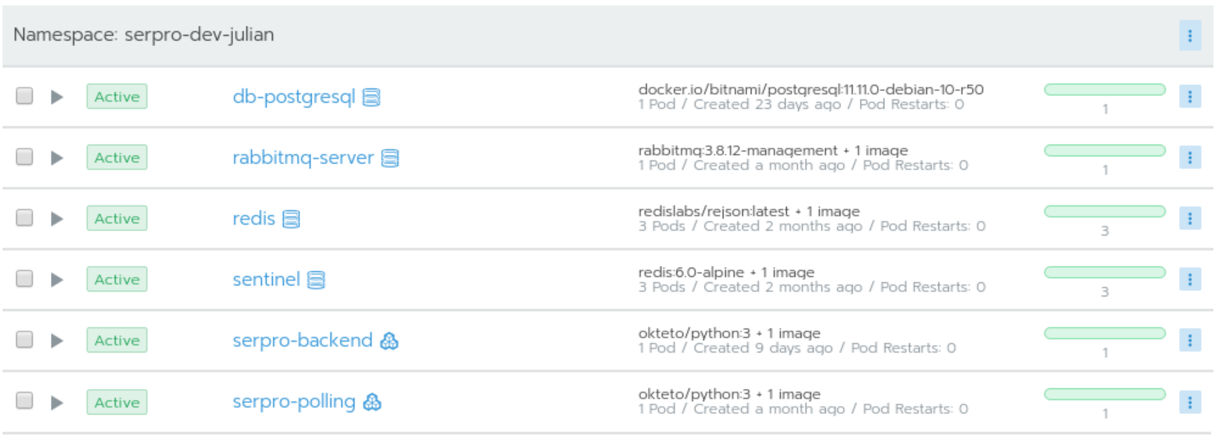


Figure 7.5: Development Environment



Namespace: serpro-dev-julian

Active	db-postgresql	docker.io/bitnami/postgresql:11.11.0-debian-10-r50 1 Pod / Created 23 days ago / Pod Restarts: 0	1
Active	rabbitmq-server	rabbitmq:3.8.12-management + 1 image 1 Pod / Created a month ago / Pod Restarts: 0	1
Active	redis	redislabs/rejson:latest + 1 image 3 Pods / Created 2 months ago / Pod Restarts: 0	3
Active	sentinel	redis:6.0-alpine + 1 image 3 Pods / Created 2 months ago / Pod Restarts: 0	3
Active	serpro-backend	okteto/python:3 + 1 image 1 Pod / Created 9 days ago / Pod Restarts: 0	1
Active	serpro-polling	okteto/python:3 + 1 image 1 Pod / Created a month ago / Pod Restarts: 0	1

Figure 7.6: Development Environment Namespaces in Rancher

7.10 Continuous Integration

In order to have continuous integration, a pipeline description was created in the backend and polling repository. Both pipelines have the same stages and execute the same commands. For the sake of simplicity, all the commands are defined in a Makefile. The pipeline definition can therefore be held short and clean. The stages are described in summary in table 7.7.

Stage	Description	Execution
1. `execute-tests`	Tests of the backend with the django built-in testing tools.	On every commit
2. `dev-build`	Builds Docker image with dev tagging	On every commit exclude the master branch
3. `sonar`	Generates the Sonarqube report.	When merging in the master branch
4. `prod-build`	Builds Docker image with latest tagging	When merging in the master branch

Table 7.7: Continuous Integration Stages

Figure 7.7 shows the pipeline workflow graphically. The deployment and redeployment are described in more detail in the section Application Deployment.

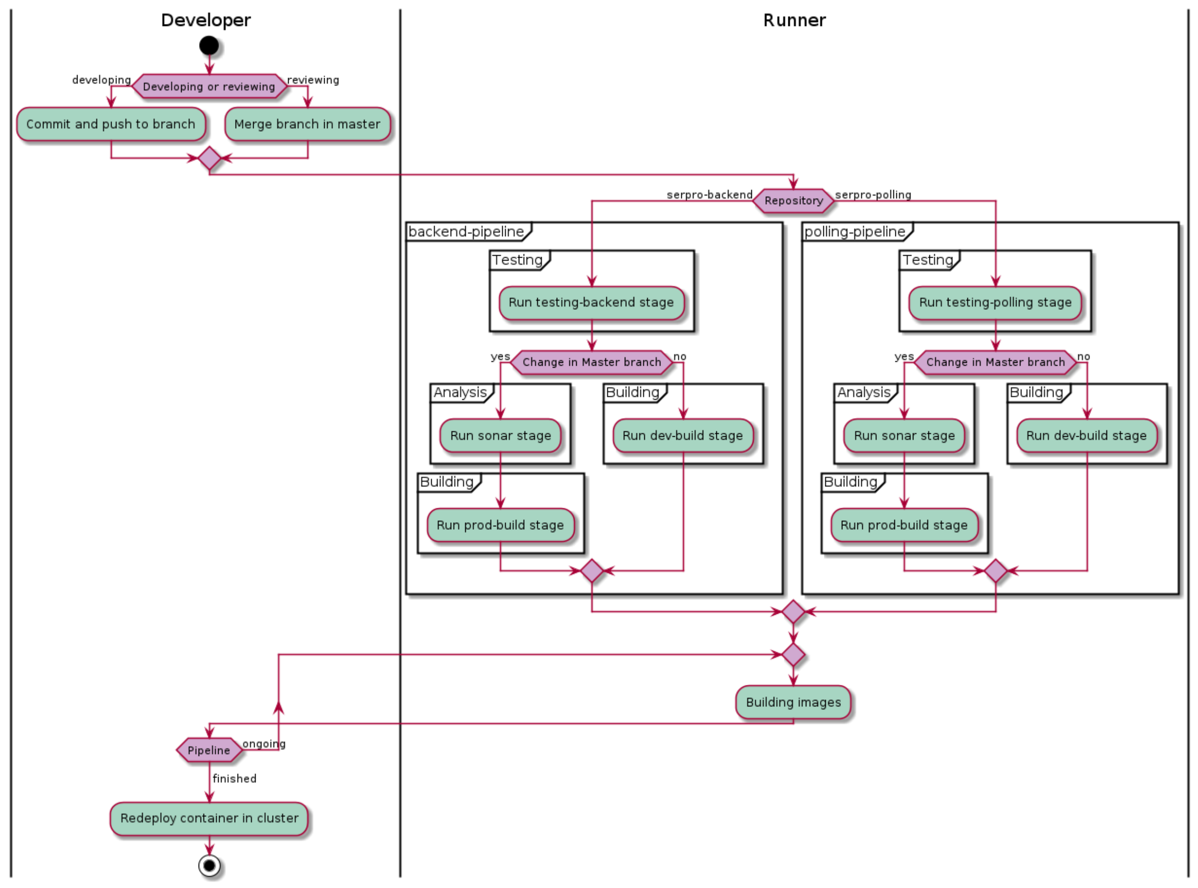


Figure 7.7: Pipeline Workflow

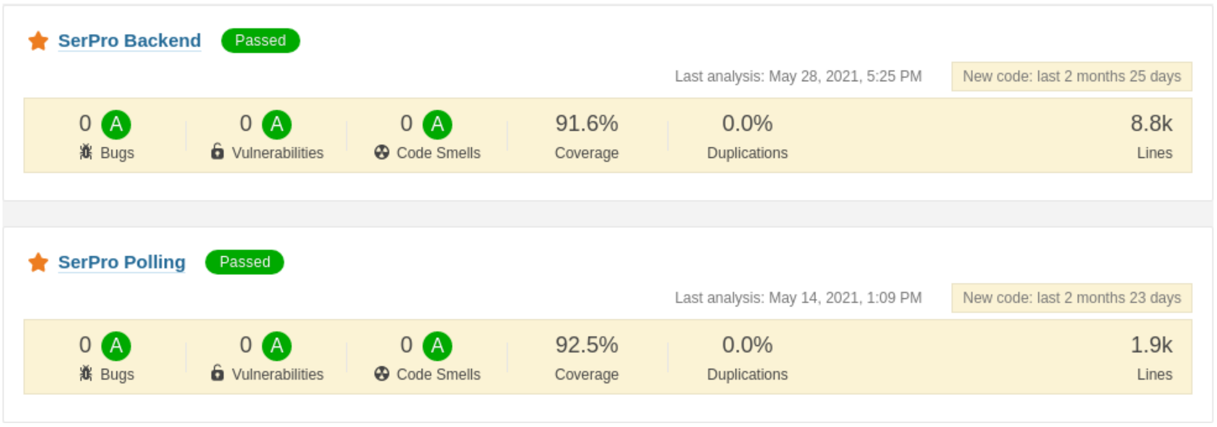
7.11 Code Metrics

For the complete code analysis, the tool Sonarqube was used, which is located on a dedicated server in the network of the Institute for Networked Solutions. The Sonar report is generated and analyzed each time a merge into the master branch takes place. Thus, there is always a current code analysis for the master branch.

The goal was to pass certain quality attributes described in the table 7.8. The quality attributes have not changed compared to the previous thesis[3] and have been listed again here for the sake of completeness.

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 describe what percentage of the code is duplicated.	max. 1%

Table 7.8: Quality Attributes[3]


SerPro Backend Passed

Last analysis: May 28, 2021, 5:25 PM New code: last 2 months 25 days

0 A Bugs	0 A Vulnerabilities	0 A Code Smells	91.6% Coverage	0.0% Duplications	8.8k Lines

SerPro Polling Passed

Last analysis: May 14, 2021, 1:09 PM New code: last 2 months 23 days

0 A Bugs	0 A Vulnerabilities	0 A Code Smells	92.5% Coverage	0.0% Duplications	1.9k Lines

Figure 7.8: Sonarqube Overview

7.12 Risk Management

The following chapter serves to show the project risks graphically (figure 7.9) as well as textually (table 7.9). Subsequently, various preventions and measures were defined to minimize the individual risks.

The most significant risk was identified in the implementation of the desired policies on the device. This risk should be reduced as much as possible at the beginning of the project.

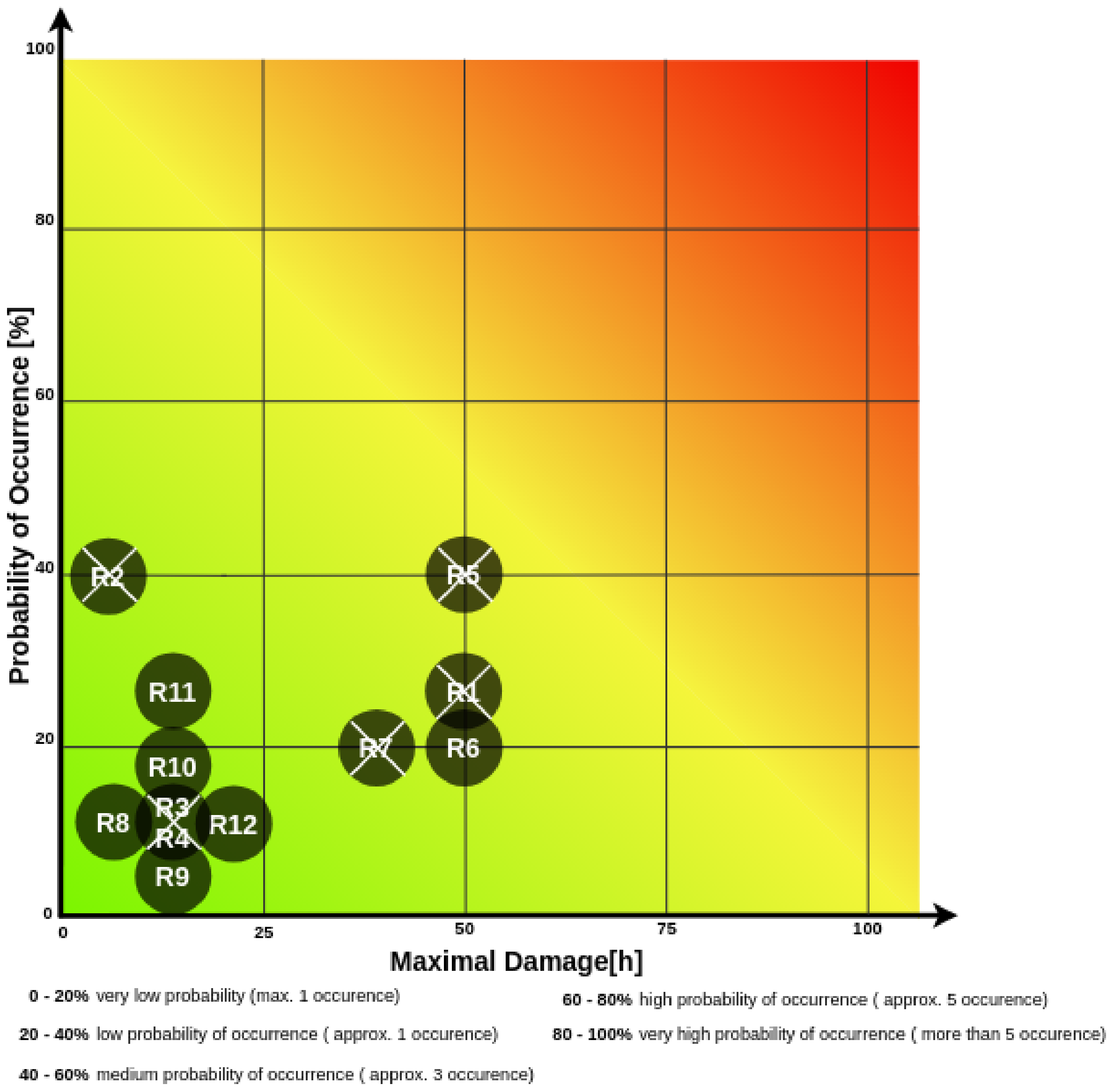


Figure 7.9: Risk Overview

#	Risk Description	Damage [h]	Probability	Weighted Damage
R1	Adjustment of the existing code base to the new data plane (SRv6) leads to delays.	50	~~25~~ 0	~~12.5~~ 0
R2	The Jalapeño application does not provide all the information which is needed.	8	~~40~~ 0	~~3.2~~ 0
R3	Unable to connect/login to the network devices.	16	~~10~~ 0	~~1.6~~ 0
R4	Unable to write policies to the network devices.	16	~~10~~ 0	~~1.6~~ 0
R5	Unable to create policy configuration on the devices which influences the traffic like desired.	50	~~40~~ 0	~~20~~ 0
R6	A team member has to suspend working due to illness or an accident.	50	20	10
R7	Difficulties to extract SRv6 Traffic Engineering information to realize policies in the application.	40	~~20~~ 0	~~8~~ 0
R8	Services are not working properly due to misconfigurations.	10	10	1
R9	Constant changes from the frontend-developers lead to constant adjustments in the backend.	16	5	0.8
R10	The targeted caching solution leads to problems.	16	15	2.4
R11	The targeted message queue solution leads to problems.	16	25	4
R12	The used tools, frameworks or components have unknown bugs or faults.	20	10	2

Table 7.9: Risk List

7.12.1 Dealing with Risks

For each risk, which was collected and analyzed in table 7.9, preventions and measures were defined. These elements can be found in the table 7.10.

#	Prevention	Measures
R1	The team verifies the new data stored in the graph database of the Jalapeno application and checks if the data can be used to realize the use cases as soon as possible.	If the adjustment takes longer than planned, the team will concentrate on implementing a solid base to adapt the application to the newly introduced data structure.
R2	The team members contact the development team of the Jalapeño application before the construction phase in order to check if all needed data is available.	If information is missing in the Jalapeño application, the team members will add manually the missing data to the application, and inform the Jalapeño development team.
R3	The team members consider common network connection plugins.	The team members will use a plain SSH connection to connect to the network devices.
R4	The team members consider common network configuration frameworks.	The team members will write the configuration with the CLI over a plain SSH connection.
R5	The team members inform about Segment Routing Traffic Engineering topics to build knowledge on how to configure SR-TE policies.	If there is no solution found on how to configure the policies, the supervisor and/or Cisco experts are asked for help.
R6	The team members try to adhere to hygiene measures in order not to get a disease.	If one team member falls ill, the situation is discussed with the supervisor. Depending on the duration of the illness, the scope of the thesis will be reduced.
R7	The necessary data is saved in a form, which enables comfortable extraction of the information.	If the necessary information can not be extracted, the developer team will work intensively to solve it.
R8	The team members have contact with the developer of these services.	If there is no solution found on how to configure the services correctly, the development team of the service is contacted directly.
R9	The team members have the lead which feature will be implement at which point.	Features in the frontend will be freezed until the team members are ready in the backend.
R10	A prototype in the elaboration phase will be developed to test if the caching system solution can be used for the application.	If the message queue solution cannot be realized, the architecture will be changed according to the planned needs.
R11	A prototype in the elaboration phase will be developed to test if the message queue solution can be used for the application.	If there is a problem with the utilized software components, the Institute of Networked Solution employees will be asked for help. They have considerable knowledge in this area.
R12	The same tools and framework as in the project thesis will be used.	When there are problems with the utilized frameworks, issues will be opened on the corresponding projects.

Table 7.10: Dealing with Risks

7.12.2 Implications

If one or more risks occur that cannot be mitigated, the supervisor will analyze whether the scope of work can be adjusted. The scope of work is then reduced, especially for non-MVP requirements.

7.13 Exception Handling

In addition to the standard HTTP messages, two are especially important for the exception handling of the application. One is `HTTP 400 Bad Request`, and the other is `HTTP 403 Forbidden`. Both are described in more detail below.

All requests will be validated in the API module to ensure that the backend can only work with validated data. That ensures that calculations are only performed with correct data. If invalid data is sent to the backend, the backend returns an `HTTP 400 Bad Request` response (see listing 7.1).

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

{
    "source_node": "The source node with id 99 could not be found! Please provide a
    valid source node!"
}
```

Listing 7.1: Backend Validation Example

The backend sends `HTTP 403 Forbidden` responses for unauthorized requests to give the user feedback on whether operations with his permissions are allowed or not. An example `HTTP 403 Forbidden` response can be found in listing 7.2

```
HTTP 403 Forbidden
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "detail": "You do not have permission to perform this action."
}
```

Listing 7.2: Backend Authentication Exception Example

Due to the high complexity and the interaction of different components, a way had to be found to inform the customer about errors in a component. Using a dedicated messaging system can ensure that messages are only acknowledged and removed from the queue if they can be processed entirely and without errors. If an error were to occur, the application would pass an error message (see Listing 7.3) through the messaging system, which would then be forwarded to the client's browser via a web socket. This way, the customer gets feedback on what exactly failed. The unacknowledged message can then be received from another backend, and therefore, processed by them.

```
{
    "type": "deployment",
    "status": "failed",
    "message": "Policy testpolicy deployment failed",
    "data": {
        "id": 1,
        "name": "testpolicy"
    }
}
```

Listing 7.3: Websocket Error Message Example

The application process is automatically stopped in case of a fatal error to keep the containers in an error-free state, which results in the container being shut down and redeployed automatically by Kubernetes. This approach ensures that the containers are always in a valid state.

7.14 Logging

Since the whole application is to be deployed cloud-native, it was decided that all logs should be written to the Standard Output (stdout). This was included for both services, the polling, and the backend. For the other components, the logging was used already integrated by the manufacturer. With this solution, the container log can be easily viewed. At a later time, it is possible to send the logs to an external logging system and monitor them at a central location without much effort.

It is possible to analyze much more information in the log during development, using different logging levels. That also makes it easy to see where exactly, for example, messages have been sent and where they arrive. That makes sense, especially when the application is tested in a scaled state, as this simplifies troubleshooting considerably.

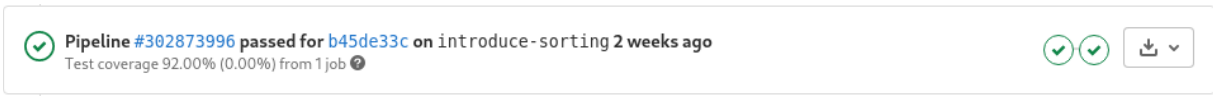
In the listing 7.4 an example log of a backend container can be found. In this example, it is visible that, for example, messages from the messaging system are only displayed in the log-level debug.

```
2021-05-30 10:05:35,761 | INFO | Connect to RabbitMQ and subscribe to queue: changes
2021-05-30 10:05:35,845 | INFO | Connected to queue changes
2021-05-30 10:05:36,028 | INFO | Connect to RabbitMQ and subscribe to queue:
    workerresults
2021-05-30 10:05:36,051 | INFO | Connected to queue workerresults
2021-05-30 10:05:50,825 | INFO | Watching for file changes with StatReloader
2021-05-30 10:09:07,329 | INFO | HTTP POST /authentication/token/ 200 [0.28,
    10.42.5.3:53110]
2021-05-30 10:09:07,584 | INFO | HTTP GET /api/link/ 200 [0.05, 10.42.5.3:53110]
2021-05-30 10:13:55,844 | DEBUG | {"type": "lsnode", "status": "add", "message":
    "incoming changes", ...}
2021-05-30 10:13:56,019 | DEBUG | {"type": "lsnode", "status": "add", "message":
    "incoming changes", ...}
```

Listing 7.4: Backend Container Example Log Output

7.15 Testing

In order to test the application thoroughly, we mainly used unit tests in combination with integration tests which were performed together with the unittest framework. The coverage of the individual files will be created with the tool coverage. All unit/integration tests will be executed on each commit to the repository, and the pipeline runs only if all these tests are passed. The exact pipeline workflow can be seen in section Continuous Integration. Due to the automatic coverage detection of GitLab, the coverage is displayed directly with each merge request. That has the advantage that current coverage can be observed when merging in the master branch. A total of 234 unit/integration tests were written.



Pipeline #302873996 passed for b45de33c on introduce-sorting 2 weeks ago
Test coverage 92.00% (0.00%) from 1 job

Figure 7.10: GitLab Coverage on Merge Request

```
...................................................................................
Name                                                            Stmts   Miss  Cover
-----------------------------------------------------------------------------------
api/serializers/endpointserializer.py                               6      0   100%
api/serializers/l3vpnv4prefixserializer.py                          7      0   100%
api/serializers/linkserializer.py                                   8      0   100%
api/serializers/nodeserializer.py                                   9      0   100%
api/serializers/policyserializer.py                               250     53    79%
api/serializers/serviceprefixserializer.py                          6      0   100%
api/serializers/serviceserializer.py                               12      0   100%
api/serializers/vrfserializer.py                                    9      0   100%
api/services/dataprovider/cache/cachedataprovider.py              309     22    93%
api/services/dataprovider/cache/decorators.py                      52      3    94%
api/services/dataprovider/database/dbdataprovider.py              125      1    99%
api/services/verification/policyverifier.py                        84     16    81%
api/validators/policyvalidator.py                                 145     33    77%
api/views.py                                                      274      0   100%
authentication/permissions.py                                      21      9    57%
authentication/serializers.py                                      10      0   100%
authentication/tokenviews.py                                        4      0   100%
calculation/services/calculation/calculator.py                    234      3    99%
calculation/services/preprocessing/preprocessor.py                 42      1    98%
calculation/services/recalculation/recalculator.py                 38     16    58%
deployment/config/config.py                                        64      0   100%
deployment/management/commands/handle_worker_messages.py           19      2    89%
deployment/plugins/deploymentinventoryplugin.py                    19      0   100%
deployment/services/managers/deployment/deploymentmanager.py       54      0   100%
deployment/services/managers/result/resultmanager.py               20      0   100%
deployment/tasks/task.py                                           16      0   100%
exceptions/edgenotfoundexception.py                                 5      0   100%
exceptions/pathnotfoundexception.py                                 5      2    60%
exceptions/serproexception.py                                       4      0   100%
exceptions/vertexnotfoundexception.py                               7      0   100%
graph/decorators.py                                                 8      0   100%
graph/serprograph.py                                              238      4    98%
notification/consumers.py                                          16      0   100%
notification/services/managers/notification/notificationmanager.py 10      0   100%
updatehandler/management/commands/handle_polling_messages.py       28      2    93%
updatehandler/services/listeners/cacheupdatelistener.py            42      0   100%
```

```
updatehandler/services/listeners/graphupdatelistener.py           40      0   100%
updatehandler/services/managers/update/iupdatemanager.py           9      2    78%
updatehandler/services/managers/update/updatemanager.py           38      1    97%
-------------------------------------------------------------------------------------
TOTAL                                                             100    686    91%
-------------------------------------------------------------------------------------
Ran 158 tests in 9.394s
```

Listing 7.5: Backend Testreport Output from Pipeline

```
Name                                Stmts    Miss   Cover
---------------------------------------------------------
polling/helper.py                     180       1     99%
polling/serproarangodb.py              64      11     83%
polling/serprodetection.py             61       5     92%
polling/serpromessaging.py             33       1     97%
polling/serpropolling.py               79      32     59%
polling/serproredis.py                 58       6     90%
---------------------------------------------------------
TOTAL                                 976      81     92%
---------------------------------------------------------
Ran 76 tests in 0.037s
```

Listing 7.6: Polling Testreport Output from Pipeline

Due to the high scalability and the complex interaction of different components from the network to the frontend, we focused firmly on system tests in addition to unit tests. A total of 80 system tests were written in order to be sure that the whole application works as expected. The system test protocol can be found in the appendix (see Test Protocols).

Because the frontend was developed by the Institute for Networked Solutions and is not part of this bachelor thesis, no usability tests were performed.

Part III

Appendix

Appendix A

Class Diagrams

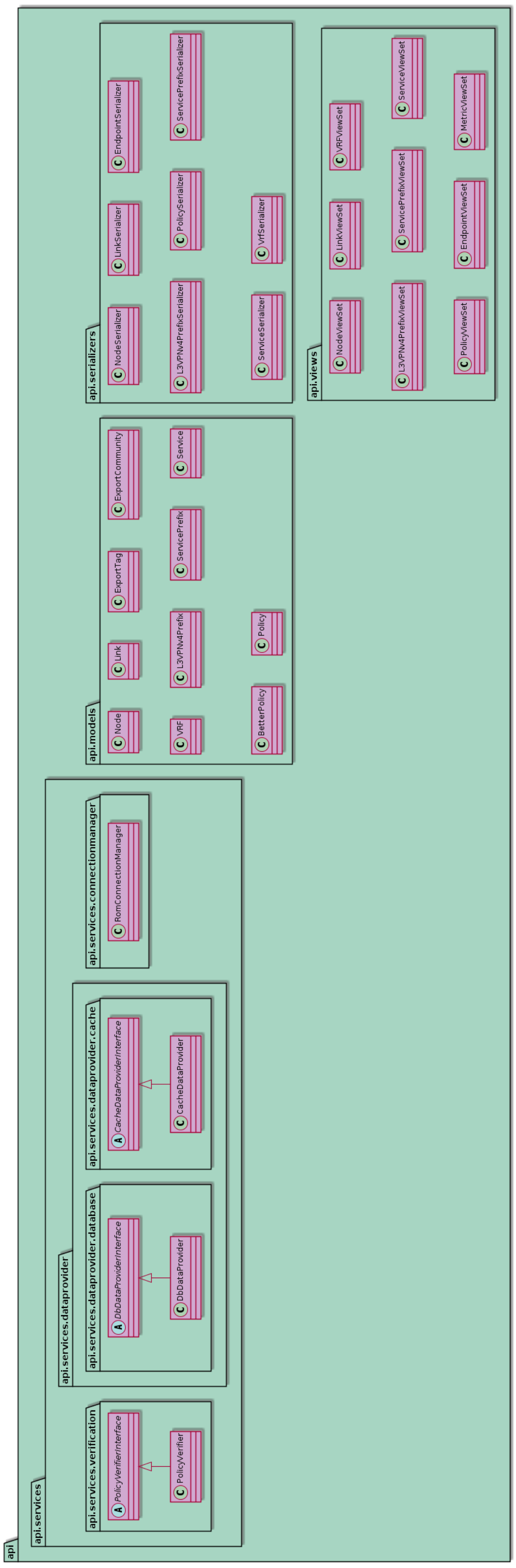


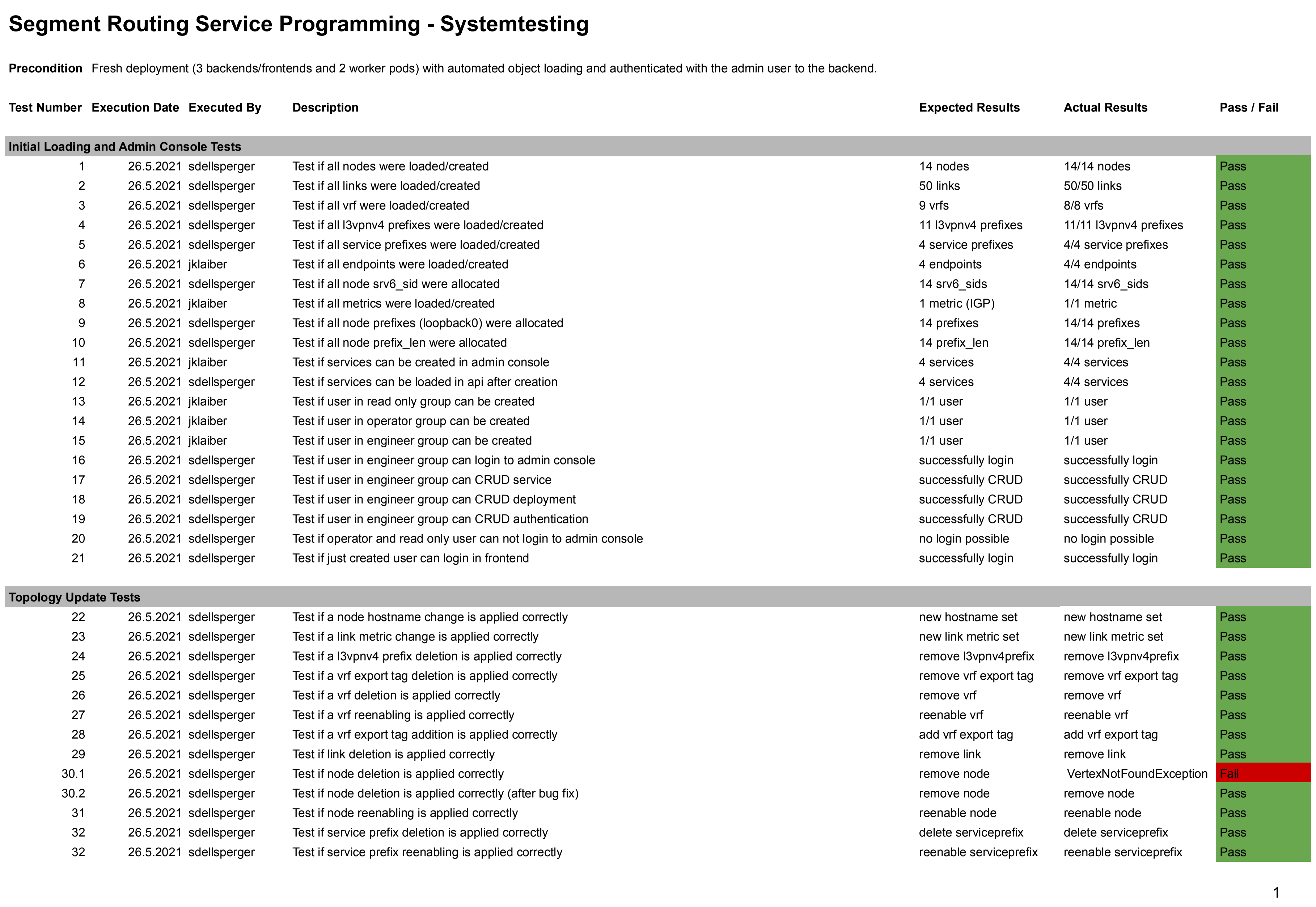
Figure A.1: Class Diagram Backend API Complete

Appendix B

Test Protocols

B.1 System Tests

Due to the fact that the application was to be tested in a productive environment and therefore in a scaled state, manual system tests were performed. All system tests were performed jointly by both authors of this thesis and documented in an Excel sheet. The different system tests can be seen in figure B.1, B.2 and B.3. Extended comments have been omitted to improve presentability. These can be taken from the Excel sheet provided with the bachelor thesis.



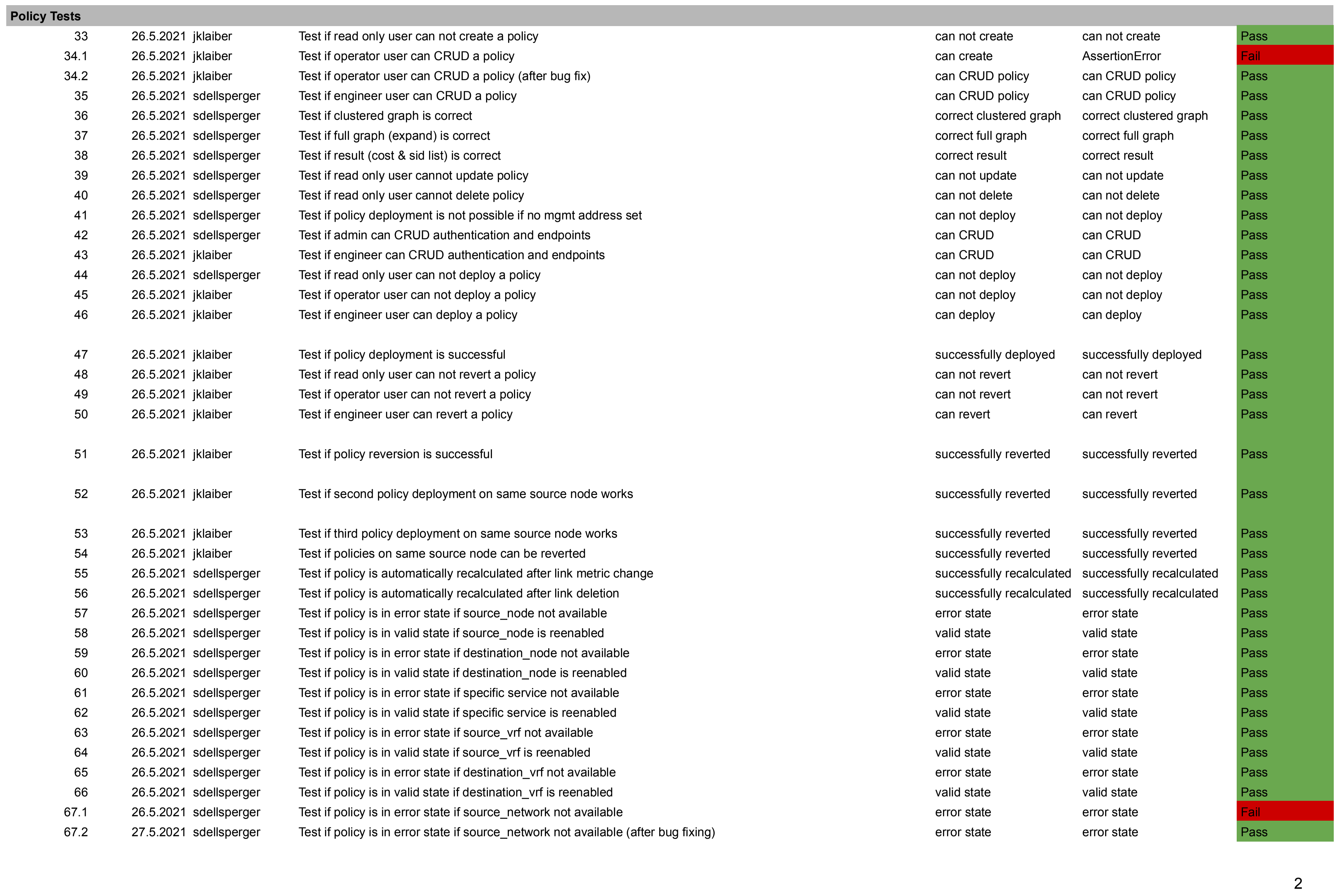
Segment Routing Service Programming - Systemtesting

Precondition Fresh deployment (3 backends/frontends and 2 worker pods) with automated object loading and authenticated with the admin user to the backend.

Test Number	Execution Date	Executed By	Description	Expected Results	Actual Results	Pass / Fail
Initial Loading and Admin Console Tests						
1	26.5.2021	sdellsperger	Test if all nodes were loaded/created	14 nodes	14/14 nodes	Pass
2	26.5.2021	sdellsperger	Test if all links were loaded/created	50 links	50/50 links	Pass
3	26.5.2021	sdellsperger	Test if all vrf were loaded/created	9 vrfs	8/8 vrfs	Pass
4	26.5.2021	sdellsperger	Test if all l3vpnv4 prefixes were loaded/created	11 l3vpnv4 prefixes	11/11 l3vpnv4 prefixes	Pass
5	26.5.2021	sdellsperger	Test if all service prefixes were loaded/created	4 service prefixes	4/4 service prefixes	Pass
6	26.5.2021	jklaiber	Test if all endpoints were loaded/created	4 endpoints	4/4 endpoints	Pass
7	26.5.2021	sdellsperger	Test if all node srv6_sid were allocated	14 srv6_sids	14/14 srv6_sids	Pass
8	26.5.2021	jklaiber	Test if all metrics were loaded/created	1 metric (IGP)	1/1 metric	Pass
9	26.5.2021	sdellsperger	Test if all node prefixes (loopback0) were allocated	14 prefixes	14/14 prefixes	Pass
10	26.5.2021	sdellsperger	Test if all node prefix_len were allocated	14 prefix_len	14/14 prefix_len	Pass
11	26.5.2021	jklaiber	Test if services can be created in admin console	4 services	4/4 services	Pass
12	26.5.2021	sdellsperger	Test if services can be loaded in api after creation	4 services	4/4 services	Pass
13	26.5.2021	jklaiber	Test if user in read only group can be created	1/1 user	1/1 user	Pass
14	26.5.2021	jklaiber	Test if user in operator group can be created	1/1 user	1/1 user	Pass
15	26.5.2021	jklaiber	Test if user in engineer group can be created	1/1 user	1/1 user	Pass
16	26.5.2021	sdellsperger	Test if user in engineer group can login to admin console	successfully login	successfully login	Pass
17	26.5.2021	sdellsperger	Test if user in engineer group can CRUD service	successfully CRUD	successfully CRUD	Pass
18	26.5.2021	sdellsperger	Test if user in engineer group can CRUD deployment	successfully CRUD	successfully CRUD	Pass
19	26.5.2021	sdellsperger	Test if user in engineer group can CRUD authentication	successfully CRUD	successfully CRUD	Pass
20	26.5.2021	sdellsperger	Test if operator and read only user can not login to admin console	no login possible	no login possible	Pass
21	26.5.2021	sdellsperger	Test if just created user can login in frontend	successfully login	successfully login	Pass
Topology Update Tests						
22	26.5.2021	sdellsperger	Test if a node hostname change is applied correctly	new hostname set	new hostname set	Pass
23	26.5.2021	sdellsperger	Test if a link metric change is applied correctly	new link metric set	new link metric set	Pass
24	26.5.2021	sdellsperger	Test if a l3vpnv4 prefix deletion is applied correctly	remove l3vpnv4prefix	remove l3vpnv4prefix	Pass
25	26.5.2021	sdellsperger	Test if a vrf export tag deletion is applied correctly	remove vrf export tag	remove vrf export tag	Pass
26	26.5.2021	sdellsperger	Test if a vrf deletion is applied correctly	remove vrf	remove vrf	Pass
27	26.5.2021	sdellsperger	Test if a vrf reenabling is applied correctly	reenable vrf	reenable vrf	Pass
28	26.5.2021	sdellsperger	Test if a vrf export tag addition is applied correctly	add vrf export tag	add vrf export tag	Pass
29	26.5.2021	sdellsperger	Test if link deletion is applied correctly	remove link	remove link	Pass
30.1	26.5.2021	sdellsperger	Test if node deletion is applied correctly	remove node	VertexNotFoundException	Fail
30.2	26.5.2021	sdellsperger	Test if node deletion is applied correctly (after bug fix)	remove node	remove node	Pass
31	26.5.2021	sdellsperger	Test if node reenabling is applied correctly	reenable node	reenable node	Pass
32	26.5.2021	sdellsperger	Test if service prefix deletion is applied correctly	delete serviceprefix	delete serviceprefix	Pass
32	26.5.2021	sdellsperger	Test if service prefix reenabling is applied correctly	reenable serviceprefix	reenable serviceprefix	Pass

1

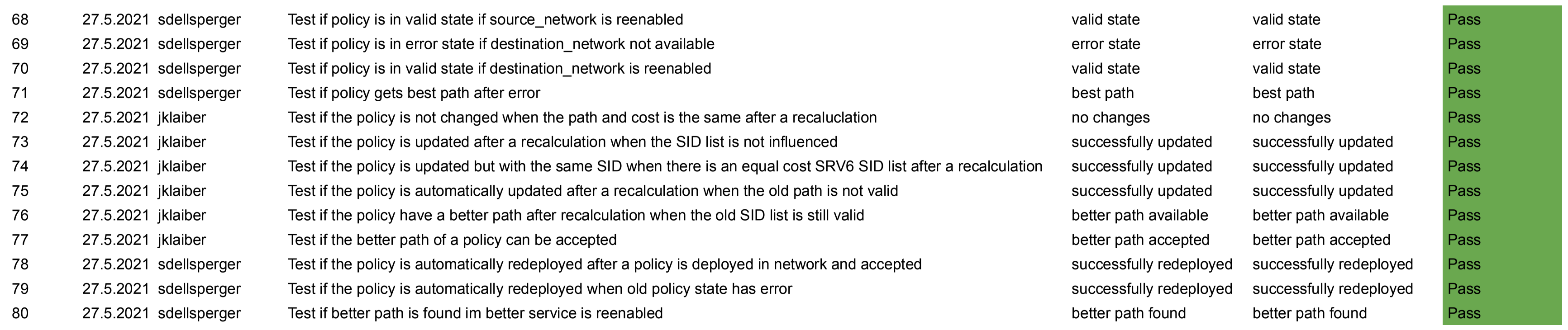
Figure B.1: System Testing Page 1



Policy Tests						
33	26.5.2021	jklaiber	Test if read only user can not create a policy	can not create	can not create	Pass
34.1	26.5.2021	jklaiber	Test if operator user can CRUD a policy	can create	AssertionError	Fail
34.2	26.5.2021	jklaiber	Test if operator user can CRUD a policy (after bug fix)	can CRUD policy	can CRUD policy	Pass
35	26.5.2021	sdellsperger	Test if engineer user can CRUD a policy	can CRUD policy	can CRUD policy	Pass
36	26.5.2021	sdellsperger	Test if clustered graph is correct	correct clustered graph	correct clustered graph	Pass
37	26.5.2021	sdellsperger	Test if full graph (expand) is correct	correct full graph	correct full graph	Pass
38	26.5.2021	sdellsperger	Test if result (cost & sid list) is correct	correct result	correct result	Pass
39	26.5.2021	sdellsperger	Test if read only user cannot update policy	can not update	can not update	Pass
40	26.5.2021	sdellsperger	Test if read only user cannot delete policy	can not delete	can not delete	Pass
41	26.5.2021	sdellsperger	Test if policy deployment is not possible if no mgmt address set	can not deploy	can not deploy	Pass
42	26.5.2021	sdellsperger	Test if admin can CRUD authentication and endpoints	can CRUD	can CRUD	Pass
43	26.5.2021	jklaiber	Test if engineer can CRUD authentication and endpoints	can CRUD	can CRUD	Pass
44	26.5.2021	sdellsperger	Test if read only user can not deploy a policy	can not deploy	can not deploy	Pass
45	26.5.2021	jklaiber	Test if operator user can not deploy a policy	can not deploy	can not deploy	Pass
46	26.5.2021	jklaiber	Test if engineer user can deploy a policy	can deploy	can deploy	Pass
47	26.5.2021	jklaiber	Test if policy deployment is successful	successfully deployed	successfully deployed	Pass
48	26.5.2021	jklaiber	Test if read only user can not revert a policy	can not revert	can not revert	Pass
49	26.5.2021	jklaiber	Test if operator user can not revert a policy	can not revert	can not revert	Pass
50	26.5.2021	jklaiber	Test if engineer user can revert a policy	can revert	can revert	Pass
51	26.5.2021	jklaiber	Test if policy reversion is successful	successfully reverted	successfully reverted	Pass
52	26.5.2021	jklaiber	Test if second policy deployment on same source node works	successfully reverted	successfully reverted	Pass
53	26.5.2021	jklaiber	Test if third policy deployment on same source node works	successfully reverted	successfully reverted	Pass
54	26.5.2021	jklaiber	Test if policies on same source node can be reverted	successfully reverted	successfully reverted	Pass
55	26.5.2021	sdellsperger	Test if policy is automatically recalculated after link metric change	successfully recalculated	successfully recalculated	Pass
56	26.5.2021	sdellsperger	Test if policy is automatically recalculated after link deletion	successfully recalculated	successfully recalculated	Pass
57	26.5.2021	sdellsperger	Test if policy is in error state if source_node not available	error state	error state	Pass
58	26.5.2021	sdellsperger	Test if policy is in valid state if source_node is reenabled	valid state	valid state	Pass
59	26.5.2021	sdellsperger	Test if policy is in error state if destination_node not available	error state	error state	Pass
60	26.5.2021	sdellsperger	Test if policy is in valid state if destination_node is reenabled	valid state	valid state	Pass
61	26.5.2021	sdellsperger	Test if policy is in error state if specific service not available	error state	error state	Pass
62	26.5.2021	sdellsperger	Test if policy is in valid state if specific service is reenabled	valid state	valid state	Pass
63	26.5.2021	sdellsperger	Test if policy is in error state if source_vrf not available	error state	error state	Pass
64	26.5.2021	sdellsperger	Test if policy is in valid state if source_vrf is reenabled	valid state	valid state	Pass
65	26.5.2021	sdellsperger	Test if policy is in error state if destination_vrf not available	error state	error state	Pass
66	26.5.2021	sdellsperger	Test if policy is in valid state if destination_vrf is reenabled	valid state	valid state	Pass
67.1	26.5.2021	sdellsperger	Test if policy is in error state if source_network not available	error state	error state	Fail
67.2	27.5.2021	sdellsperger	Test if policy is in error state if source_network not available (after bug fixing)	error state	error state	Pass

2

Figure B.2: System Testing Page 2



68	27.5.2021	sdellsperger	Test if policy is in valid state if source_network is reenabled	valid state	valid state	Pass
69	27.5.2021	sdellsperger	Test if policy is in error state if destination_network not available	error state	error state	Pass
70	27.5.2021	sdellsperger	Test if policy is in valid state if destination_network is reenabled	valid state	valid state	Pass
71	27.5.2021	sdellsperger	Test if policy gets best path after error	best path	best path	Pass
72	27.5.2021	jklaiber	Test if the policy is not changed when the path and cost is the same after a recaluclation	no changes	no changes	Pass
73	27.5.2021	jklaiber	Test if the policy is updated after a recalculation when the SID list is not influenced	successfully updated	successfully updated	Pass
74	27.5.2021	jklaiber	Test if the policy is updated but with the same SID when there is an equal cost SRV6 SID list after a recalculation	successfully updated	successfully updated	Pass
75	27.5.2021	jklaiber	Test if the policy is automatically updated after a recalculation when the old path is not valid	successfully updated	successfully updated	Pass
76	27.5.2021	jklaiber	Test if the policy have a better path after recalculation when the old SID list is still valid	better path available	better path available	Pass
77	27.5.2021	jklaiber	Test if the better path of a policy can be accepted	better path accepted	better path accepted	Pass
78	27.5.2021	sdellsperger	Test if the policy is automatically redeployed after a policy is deployed in network and accepted	successfully redeployed	successfully redeployed	Pass
79	27.5.2021	sdellsperger	Test if the policy is automatically redeployed when old policy state has error	successfully redeployed	successfully redeployed	Pass
80	27.5.2021	sdellsperger	Test if better path is found im better service is reenabled	better path found	better path found	Pass

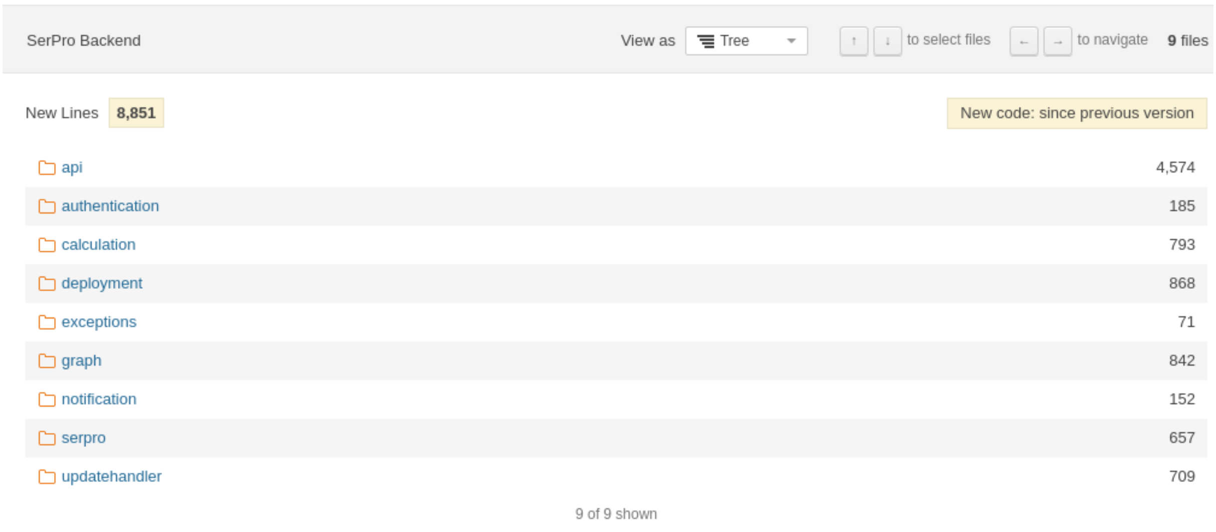
3

Figure B.3: System Testing Page 3

Appendix C

Metrics

In this bachelor thesis, a total of **10'781** lines of Python code were written. These lines of code are divided between the two services backend and polling. In the backend service, **8'851** lines of Python code were written (see figure C.1) with a coverage of **92,2%**. In the polling service, there were **1'930** lines of Python code (see figure C.2) with a coverage of **92,5%**. In Figures C.3 and C.4, the different excerpts from Sonarqube can be viewed. It should also be noted that these lines of code are only related to the implementation in Python; besides that, many more lines were written for deployment, pipeline, and the various Docker definitions.

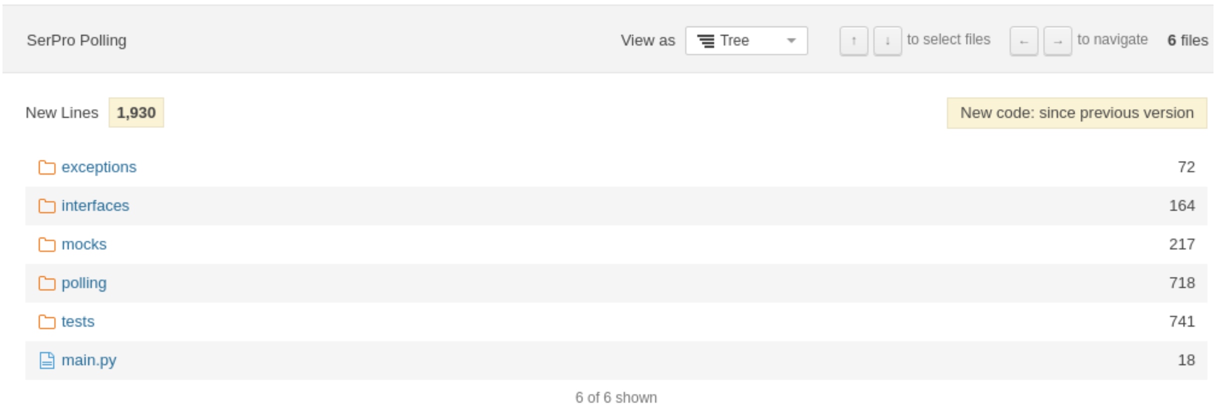


New Lines 8,851

api	4,574
authentication	185
calculation	793
deployment	868
exceptions	71
graph	842
notification	152
serpro	657
updatehandler	709

9 of 9 shown

Figure C.1: Sonar Lines of Code Backend Service



SerPro Polling

View as Tree ↑ ↓ to select files ← → to navigate 6 files

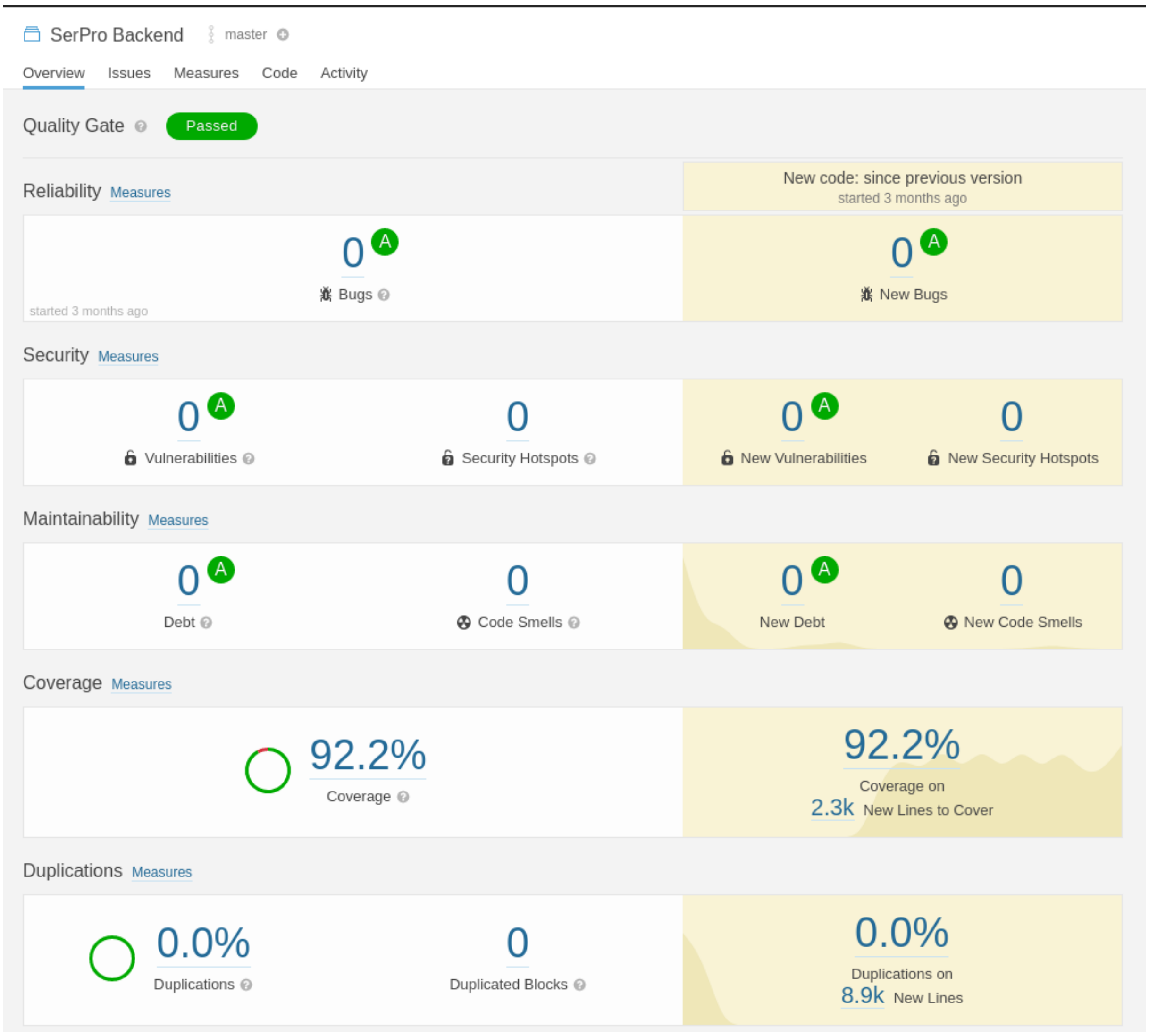
New Lines 1,930

New code: since previous version

exceptions	72
interfaces	164
mocks	217
polling	718
tests	741
main.py	18

6 of 6 shown

Figure C.2: Sonar Lines of Code Polling Service



SerPro Backend master

Overview Issues Measures Code Activity

Quality Gate Passed

New code: since previous version
started 3 months ago

Reliability Measures

started 3 months ago

0 A Bugs

0 A New Bugs

Security Measures

0 A Vulnerabilities

0 Security Hotspots

0 A New Vulnerabilities

0 New Security Hotspots

Maintainability Measures

0 A Debt

0 Code Smells

0 A New Debt

0 New Code Smells

Coverage Measures

92.2% Coverage

92.2% Coverage on 2.3k New Lines to Cover

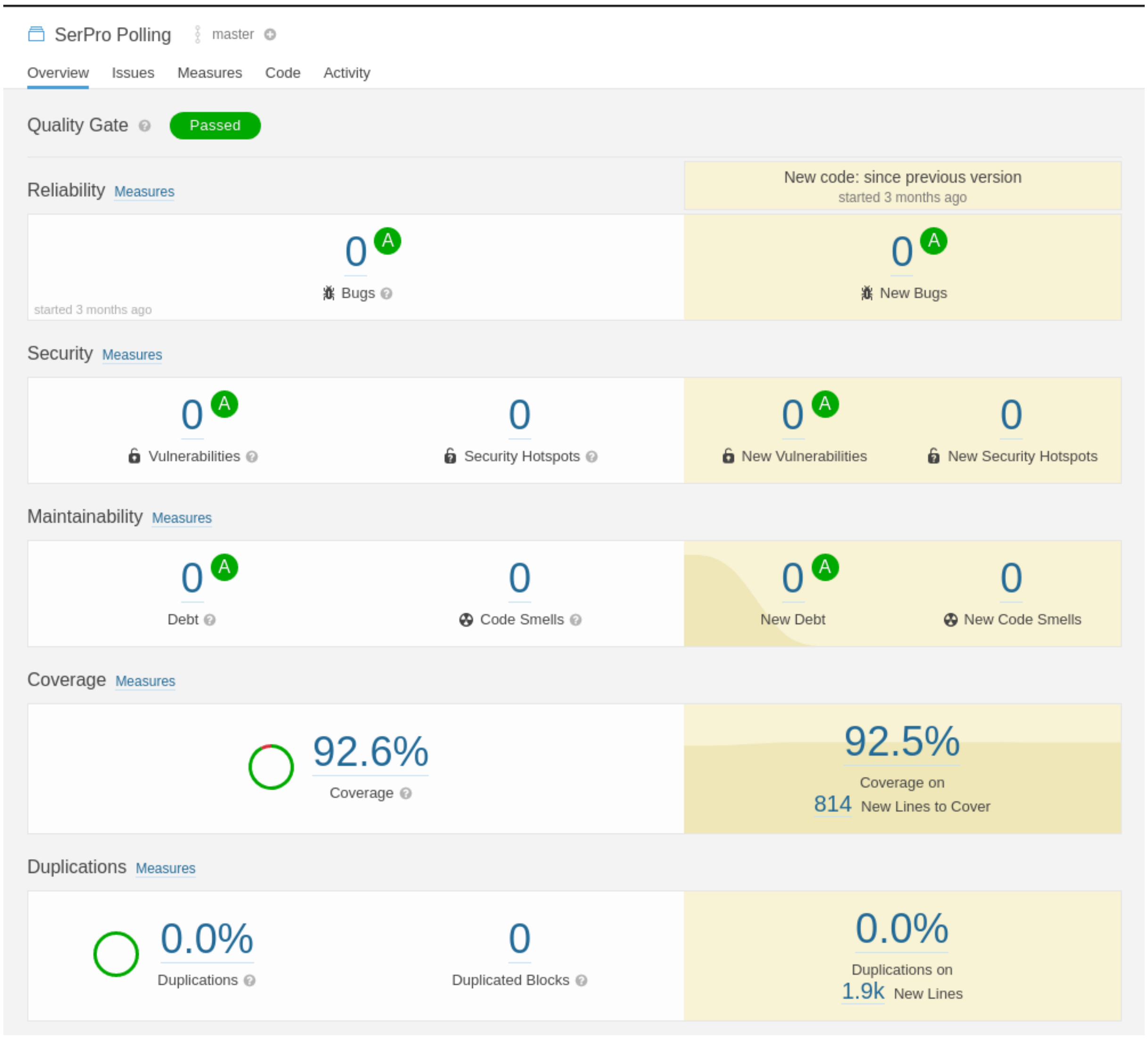
Duplications Measures

0.0% Duplications

0 Duplicated Blocks

0.0% Duplications on 8.9k New Lines

Figure C.3: Sonar Code Analysis Backend Service



Quality Gate Passed

New code: since previous version
started 3 months ago

Reliability Measures

0 A Bugs

started 3 months ago

0 A New Bugs

Security Measures

0 A Vulnerabilities

0 Security Hotspots

0 A New Vulnerabilities

0 New Security Hotspots

Maintainability Measures

0 A Debt

0 Code Smells

0 A New Debt

0 New Code Smells

Coverage Measures

92.6% Coverage

92.5% Coverage on 814 New Lines to Cover

Duplications Measures

0.0% Duplications

0 Duplicated Blocks

0.0% Duplications on 1.9k New Lines

Figure C.4: Sonar Code Analysis Polling Service