Bachelor Thesis

SD-WAN Topology Viewer

Eastern Switzerland University of Applied Sciences

Department of Computer Science

Period: 22.02.2021 - 18.06.2021

Authors Dominic Gabriel
Lars Barmettler

Supervisor Prof. Laurent Metzger
Expert Marcel Witmer
Counterreader Prof. Mirko Stocker
Industry Partners KSAT Satellite Services
Insoft Services

Task description

Task Description: SD-WAN Topology Viewer

The goal of that thesis is to create an application that displays the IPSEC connections of a Cisco SD-WAN network.

The application sits on top of the controller of the SD-WAN network, the Vmanage.

The following uses cases are some example of areas of investigation during the bachelor thesis:

- A customer portal offering a partial view of the connections
- The control of access to the information of the network
- Filtering options to display specific connections
- The "live" information about the metrics of the tunnel
- The historical information about the metrics of the tunnel
- The export of the metrics to a PDF for reporting purposes
- The filtering and the sorting of the metrics
- A color-tag of the connections based on the metric
- The alarming based on a metric threshold.

The security of the information, the reliability, the usability and the efficiency of the application and a special emphasis on scalability are the essential focus of the application.

Location, Date: Rapperswil, 14.06.2021

Prof. Laurent Metzger

Abstract

Software defined WAN (SD-WAN) is a fast emerging and trending new technology to interconnect worldwide distributed company branches or customers. Many of the market leading network equipment providers, like Cisco, have developed their own SD-WAN solution. For IT professionals using SD-WAN solutions in global companies, the complexity of their network quickly becomes overwhelming. As a result, it is even harder to keep track of the network topology.

Cisco's SD-WAN solution vManage is a web application shipped alongside their SD-WAN technology and is primarily designed for configuration purposes. Although it provides a simple graphical overview of the distribution of the individual routers on a world map, it lacks to apply a proper user centered approach. It is not designed for active monitoring of the infrastructure and does not display the IPSec tunnels. With the rise of SD-WAN, its products and non-existent monitoring solutions, the foundation for a solution to this problem has already been laid.

In contrast to Ciscos vManage web application, the SD-WAN Topology Viewer (SDWANTV) puts the emphasis on a clean visual representation of the topology and a historical view of the metrics. As the foundation of the bachelor thesis, we have taken the result of a previous semester thesis. We extended the existing software with much needed new functionalities and dramatically improved the software architecture. With a historical metric dashboard and a customer limited view, the SDWANTV extended the target audience to customer and network administrators alike.

Our system fetches all information from Ciscos vManage API with a scheduled task runner. The tasks then propagate the collected data to a timescale database and via active WebSocket connections directly to state-of-the-art single page application written in React. Due to the strict separation between frontend and backend, a user can enjoy real time state updates of the topology and observe the latest as well as historical tunnel metrics via a REST Application Programming Interface (API) in one single cloud ready application, shipped with a helm chart for easy installation.

Management Summary

Baseline

Globally distributed companies are often faced with an increasing complexity of their wide network infrastructure. In the last years, new ideas have emerged to cope with this issue. Software-Defined Wide Area Networks (SD-WAN) is one of those. SD-WAN makes it possible to connect remote locations via several different transport protocols such as the internet, mobile network or even MPLS. These connections are automatically established based on an intelligent policy and reduce the operational effort. Furthermore, it is possible to collect metrics about the connections and perform intelligent routing based on this and use the path with the best quality of service. With the increase in internet quality in recent years, SD-WAN now makes it possible to use the internet for business critical applications. These days moving resources to the cloud has become the strategy for many companies, it is even better that SD-WAN also supports the integration of cloud networking.

SDWANTV has the goal to improve the monitoring experience of the network infrastructure based on Ciscos vManage SD-WAN solution. Although there are already some tools which achieve this, our solution takes a different approach and puts emphasis on a visual representation of the network infrastructure itself. Professionals and non-professionals alike, can watch the changes made to the network directly on a monitoring screen at the office and are able to detect failures and anomalies nearly real-time.

The focus of our project is the construction of a monitoring software solution on top of the widely known Ciscos SD-WAN solution. We fetch the infrastructure data and connection metrics from Ciscos vManage and display the topology and connection metrics in a web application. The topology displayed is updated automatically if it changes and displays anomalies.

Approach, Technologies

In the first phase we worked together with a domain expert to pinpoint the most important features. With the result of this phase, we created a functional and non-functional requirement catalog which was the baseline for our software.

Due to the fact that SDWANTV heavily relies on Ciscos vManage API, the second step was to analyze the dependent system. To determine which data needs to be retrieved and how to efficiently query those, we put a lot of effort into studying the vManage API documentation.

Based on the information gathered we designed our software solution. We split the software into two tiers. A backend implemented in Django Rest Framework and a frontend running in a web browser with React. For performance reasons, we chose to have an internal representation of the network topology. This topology is stored in a Timescale database and is updated by scheduled tasks which fetch the latest data from the vManage API. In addition to the topology, we also periodically fetch the metrics of all connections and store them in time-series tables in the Timescale database. This makes it possible to perform efficient metrics queries and display metrics data historically.

Results

The software solution we created is able to monitor the topology of a network infrastructure based on Ciscos vManage API. The main focus on simplicity, robustness and efficiency can be noticed during the user experience and the quality of the code behind it.

The user can limit the complexity of the topology by using filters to only see a subset of it. By clicking on a node or a tunnel, the user is able to examine the requested resource in more

details. The backend takes less than 30 seconds to fetch new topology data and informs the frontend immediately if changes occur. If something goes wrong, the backend will report it and the frontend displays the error message in a status bar.

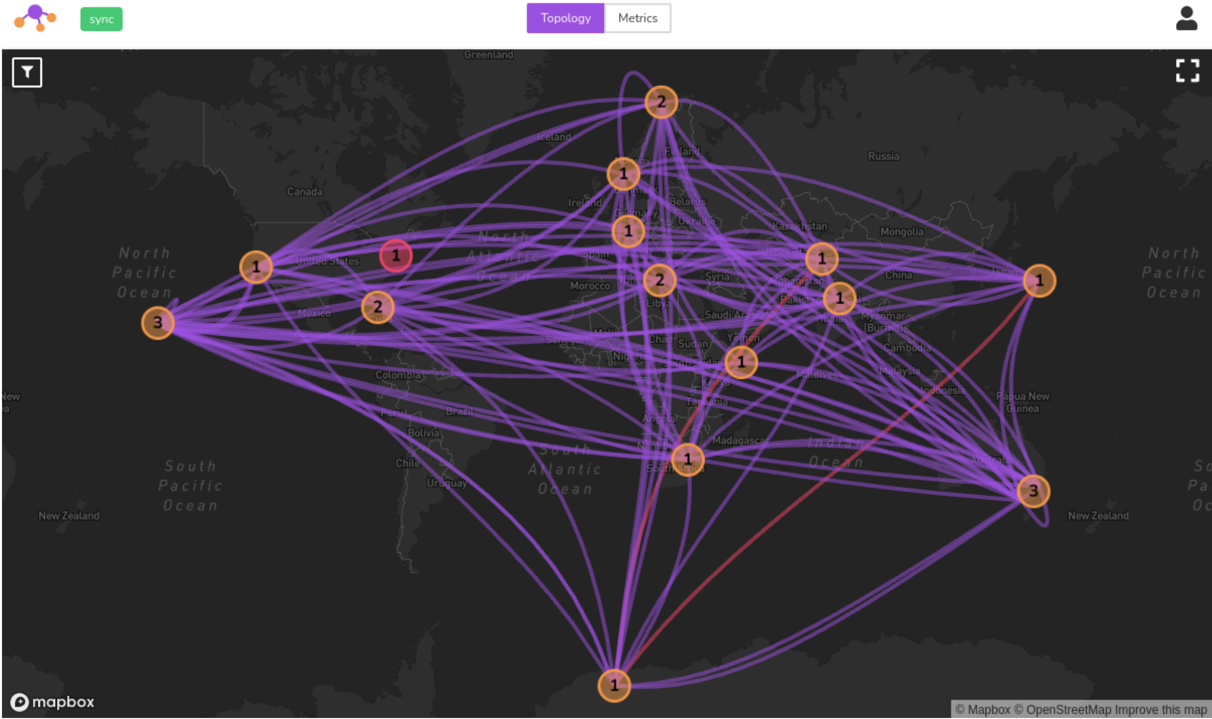

Figure 1: SD-WAN Topology Viewer Topology

SDWANTV is not only able to display the network topology but also to provide connection quality insights. A second view displays all available tunnels and their representing metrics. The user can observe the historical course of the metrics by selecting a specific connection. On top of that, the time range can be defined. To reduce the amount of connections displayed, the user has the possibility to filter the metrics based on a variety of criteria. To organize the listing it is possible to sort the entries.

sync

Topology Metrics

Last hour

State	From	To	Local tloc / Remote tloc	Jittering (ms)	Package loss (%)	Latency (ms)
●	Hawaii	cedge-Sydney	biz-internet, biz-internet	0	0.00	281
●	Orlando	cedge-Sydney	biz-internet, biz-internet	0	0.00	281
●	RS-Bern-a	cedge-Sydney	biz-internet, biz-internet	0	0.00	280
●	RS-Miami-a	cedge-Sydney	biz-internet, biz-internet	0	0.00	278
●	RS-Bern-b	cedge-Sydney	biz-internet, biz-internet	0	0.00	278
●	RS-Tromsoe-b	cedge-Sydney	biz-internet, biz-internet	0	0.00	278
●	RS-Miami-b	cedge-Sydney	biz-internet, biz-internet	0	0.00	278

Figure 2: SD-WAN Topology Viewer Metrics

The user management feature enables the administrator to create users and grant customers access to SDWANTV and observe their own topology and metrics.

To better embed our tool for the daily usage, we created features to enable the software to run on a monitor positioned in the office. A toggle button removes all irrelevant elements of the topology viewer and puts the page in a full screen mode. The "keep me logged in" option enhances the usability and extends the session time up to two weeks.

Everything is built to run on container platforms such as Kubernetes and is packed together as a simple to install helm chart.

Forecast

SDWANTV is constructed with extendibility in mind. This makes it easy to add new features to the existing code base.

For example, it would be helpful to have a notification system to inform users about tunnels with abnormal metrics behaviour. Another useful feature would be to color tunnels on the topology map according to their latest metrics data.

If one would think even further, this tool could completely replace Ciscos vManage and provide a more user-friendly solution for not only the monitoring of the SD-WAN infrastructure but also controlling and configuration of the SD-WAN infrastructure.

Acknowledgements

We thank the following people for their support during our term thesis:

- Supervisor: Prof. Laurent Metzger
- Co-Supervisor: Jessica Hilti
- Possibility for this thesis: KSAT, Thomas Torsteinsen
- Knowledge support: Insoft Services
- Technical support: INS & IFS Team Members
- Architecture documentation review: Prof. Olaf Zimmermann
- Orthographic proofreading: AnneMarie O'Neill

Contents

Glossary and List of Abbreviations xii

List of Figures xiv

List of Tables xviii

I Technical Report 1

1 Technical Report 2
1.1 Introduction and Overview . . . 2
1.1.1 Problem . . . 2
1.1.2 Previous work . . . 3
1.1.3 Goals . . . 3
1.1.4 Limitations . . . 5
1.1.5 Work structure . . . 5
1.2 Evaluation . . . 5
1.2.1 Information acquisition . . . 5
1.3 Concept . . . 6
1.4 Solution . . . 7
1.4.1 Implementation . . . 8
1.5 Conclusion . . . 9
1.6 Forecast . . . 10
1.6.1 Non-technical improvements . . . 10
1.6.2 Technical improvements . . . 10
1.6.3 Technical debt . . . 10

II Project Documentation 12

2 Requirements Specification 13
2.1 Thesis Requirements . . . 13
2.2 Actors . . . 13
2.2.1 Administrator . . . 13
2.2.2 Customer . . . 13
2.2.3 System . . . 13
2.3 Use Cases . . . 14
2.3.1 UC1: Customer portal . . . 15
2.3.2 UC2: User management . . . 15

2.3.3 UC2.1: Update customer . . . 15
2.3.4 UC2.2: Password change enforcement . . . 16
2.3.5 UC3: Add filter between two sites . . . 16
2.3.6 UC4: Acknowledge down tunnels . . . 16
2.3.7 UC5: Display tunnel metrics . . . 17
2.3.8 UC5.1: Specify historical metrics time range . . . 17
2.3.9 UC5.2: Export metrics to pdf . . . 17
2.3.10 UC5.3 Filtering metrics . . . 18
2.3.11 UC5.4 Sorting metrics . . . 18
2.3.12 UC6: Observe tunnel metric state by coloring . . . 18
2.3.13 UC7: Manage Metric Alarms . . . 19
2.3.14 UC7.1: Send alarm to the external syslog server . . . 19
2.3.15 UC8: Global Filtering . . . 19
2.4 Non functional requirements . . . 20
2.4.1 Functionality . . . 20
2.4.2 Reliability . . . 20
2.4.3 Usability . . . 20
2.4.4 Efficiency . . . 20
2.4.5 Supportability . . . 21
2.4.6 Portability . . . 21
2.4.7 Scalability . . . 21

3 Analysis **22**
3.1 Domain model . . . 22
3.2 Data model . . . 23
3.3 Cisco vManage API analysis . . . 25
3.3.1 Authentication . . . 25
3.3.2 Obtain XSRF token . . . 25
3.3.3 List devices . . . 26
3.3.4 VPN ID . . . 27
3.3.5 IPsec connections . . . 28
3.3.6 Tunnel list . . . 29
3.4 Cisco vManage API metrics analysis . . . 30
3.4.1 Fetch metric history . . . 30
3.4.2 Fetch live metrics . . . 31
3.4.3 Metrics analysis conclusion . . . 32
3.5 Database analysis . . . 33
3.6 Prototype . . . 33

4 Architecture & Design Specification **34**
4.1 Scope . . . 34
4.2 Software Architecture . . . 34
4.2.1 Context Diagram . . . 35
4.2.2 Container Diagram . . . 36
4.2.3 Component Diagram . . . 38
4.3 Deployment . . . 44
4.3.1 Helm Deployment . . . 44
4.3.2 Development with Docker-compose . . . 45
4.3.3 Goals . . . 45
4.4 Design . . . 46
4.4.1 Twelve Factors . . . 46
4.5 Architectural Decisions . . . 49

4.5.1 AD1: Kubernetes Deployment . . . 49
4.5.2 AD2: WebSocket . . . 50
4.5.3 AD3: TimescaleDB . . . 50
4.5.4 AD4: WebGL . . . 51
4.5.5 AD5: Tunnel Aggregation . . . 51
4.5.6 AD6: Events fetching . . . 52
4.5.7 Client/Server Cut (CSC) . . . 53
4.6 Package Diagram . . . 54
4.6.1 Frontend . . . 54
4.6.2 Communication . . . 56
4.6.3 Backend . . . 56
4.7 Sequence Diagrams . . . 58
4.7.1 Fetch topology . . . 58
4.7.2 Partial topology update . . . 59
4.7.3 Fetching metrics . . . 60
4.8 Tools & Frameworks . . . 62
4.8.1 Frontend . . . 62
4.8.2 Backend . . . 62
4.8.3 CI/CD . . . 62
4.9 UI-Design . . . 65
4.9.1 Tools . . . 65
4.9.2 Design process . . . 65

5 Implementation & Testing **67**
5.1 Implementation . . . 67
5.1.1 Python Django Backend . . . 67
5.1.2 Frontend . . . 80
5.2 Automated Testing . . . 85
5.2.1 Unit Tests . . . 85
5.2.2 Integration Tests . . . 86
5.2.3 Test Coverage . . . 86
5.3 Manual Testing . . . 87
5.3.1 System Tests . . . 87
5.3.2 Non functional Requirements Tests . . . 87

6 Project Management **88**
6.1 Project organization . . . 88
6.2 Project Meetings . . . 89
6.3 Process Model . . . 89
6.4 Software Development Process . . . 90
6.5 Releases . . . 91
6.6 Milestones . . . 92
6.7 Project Plan . . . 93
6.8 Risk Analysis . . . 94
6.8.1 R3: Not testable . . . 95
6.8.2 R5: Overstrain of technical complexity . . . 95
6.8.3 R6: Lifecycle of dependencies . . . 95
6.8.4 R9: vManage outage . . . 95
6.8.5 R10: vManage is not scalable . . . 95
6.8.6 R11: Faulty topology state . . . 95
6.9 Logging . . . 96
6.10 Time Report . . . 96

6.11 Quality Control . . . 97
6.11.1 Linting . . . 97
6.11.2 Definition of Done . . . 98
6.11.3 Coding Guidelines . . . 98
6.12 MVP . . . 98

7 Project monitoring **99**
7.1 Project reporting . . . 99
7.1.1 Working times . . . 99
7.1.2 Project phases . . . 100
7.1.3 Issues per task types . . . 102
7.1.4 Milestones . . . 103
7.2 Code statistics . . . 104

III Appendix **105**

A User Manual **106**
A.1 Requirements . . . 106
A.2 Deployment . . . 106
A.3 Deployment configuration - values.yaml . . . 107
A.4 Operational tasks . . . 108
A.5 Update . . . 110
A.6 Termination . . . 110
A.7 vManage adjustments . . . 110

B Systemtest protocol **111**
B.1 UC1: Customer Portal . . . 113
B.2 UC2: Customer management: Test 1 . . . 114
B.3 UC2: Customer management: Test 2 . . . 115
B.4 UC2: Customer management: Test 3 . . . 117
B.5 UC2.1: Update customer . . . 119
B.6 UC3: Add filter between two sites . . . 121
B.7 UC5: Display tunnel metrics: Test 1 . . . 123
B.8 UC5: Display tunnel metrics: Test 2 . . . 124
B.9 UC5.1: Specify historical metrics time range . . . 125
B.10 UC5.3: Filtering metrics: Test 1 . . . 126
B.11 UC5.3: Filtering metrics: Test 2 . . . 127
B.12 UC5.4: Sorting metrics . . . 129
B.13 AD5: Tunnel aggregation . . . 130
B.14 S1: Monitoring topology: Test 1 . . . 132
B.15 S1: Monitoring topology: Test 2 . . . 133
B.16 S1: Monitoring topology: Test 3 . . . 134
B.17 S1: Monitoring topology: Test 4 . . . 135
B.18 S1: Monitoring topology: Reported bugs . . . 136
B.19 S2: View node information: Test 1 . . . 137
B.20 S2: View node information: Test 2 . . . 138
B.21 S2: View node information: Test 3 . . . 139
B.22 S3: Display connection metrics . . . 140
B.23 S4: Apply customer filter . . . 141
B.24 S5: Apply site filter . . . 142

C Non functional requirement test protocol **143**
C.1 Security . . . 144
C.2 Fault tolerance, user data . . . 146
C.3 Fault tolerance, vManage data . . . 148
C.4 Maturity . . . 148
C.5 Understandability . . . 149
C.6 Failure management . . . 150
C.7 Time behaviour . . . 151
C.8 Response time . . . 152
C.9 Supportability . . . 153
C.10 Portability . . . 154
C.11 Scalability . . . 155

D Wireframe & Prototypes **156**
D.1 Wireframes . . . 156
D.1.1 UC2: User management . . . 156
D.1.2 UC2.1: Update customer . . . 157
D.1.3 UC3: Add filter between two sides . . . 158
D.1.4 UC4: Acknowledge tunnel down changes . . . 158
D.1.5 UC5: Display the metrics of each tunnel . . . 159
D.1.6 UC8: Global Filtering . . . 159
D.2 Prototypes . . . 160
D.2.1 UC2: User management . . . 160
D.2.2 UC2.1: Update user . . . 161
D.2.3 UC5: Display the metrics of each tunnel . . . 161
D.2.4 UC5.1: Specify historical metrics time range . . . 162

E Risk analysis **163**
E.1 Risk Analysis Table . . . 163

F vManage API Request & Responses **165**
F.1 Devices list response . . . 165
F.2 Tunnel list response . . . 166
F.3 IPsec inbound response . . . 167

G vManage API Metrics Request & Responses **168**
G.1 Metric history request aggregation query . . . 168
G.2 Metric history response . . . 169
G.3 Live Metric response . . . 170

Glossary and List of Abbreviations

API Application Programming Interface. ii, xii, 10, 20, 25–27, 30, 35–37, 47, 53, 56, 57, 62, 70, 71, 73, 74, 78, 86, 94, 132, *see* Application Programming Interface

Application Programming Interface Programming interface of a software. Can be used from other system components or extended.. ii

CI Continous Integration is the practice of merging all developers' working copies to a shared mainline several times a day.. 48, 62, 86, 87, 90, 97, 98

Cisco Cisco is the worldwide leader in IT, networking, and cybersecurity solutions.. ii, iii, v, 2, 5, 10, 25, 34, 35, 37, 47, 48, 67, 68, 95

GitLab Git repository server to host our code on. 46–48, 63, 90, 97, 104

Helm The package manager for Kubernetes.. 98

INS Abbreviation for Institute for Networked Solutions Rapperswil. vi, 34, 143

Json Web Token Is a technology to secure claims between two parties. 3, 57

JSX Template language of React. 85

Kubernetes Kubernetes, also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications.. 98

MVP Abbreviation for Minimum viable Product, defines a minimal set of features an application has to provide.. 98

OST Ostschweizerische Fachhochschule.. 90

REST API This is a common used set of rules, on which application interface are designed.. 2, 3, 8, 37

Scrum Agile project management method which breaks goals into small work packages which are completed in sprints. 89

SD-WAN Technology to manage wide area networks (WAN) via software.. ii, iii, v, 2, 5, 35, 37, 52, 85, 111

SD-WAN Topology Viewer This is the name of the application which is being developed in this thesis. ii

SDWANTV SD-WAN Topology Viewer. ii–v, xiii, 3, 5–7, 10, 16, 20, 25, 32–46, 48, 49, 51–53, 67, 72, 96, 98, 106, 107, 110, 111, 113–115, 117, 119, 121, 123–127, 129, 130, 137–142, 148, 150, *see* SD-WAN Topology Viewer

SIGTERM A signal that is sent by the process handler to the running program to safely shut down the process.. 48

Single Page Application SPA are a way to implement web application. SPA is coded mainly in Javascript. 36, 37, 39, 53, 58, 59

SonarCube Tool to measure Code metrics. 91

Stdout Standard output, which in a docker container is the container log. 20, 48, 96, 144

TimescaleDB Open-source relational database for time-series data. xiv, 9, 33, 37, 42, 44, 45, 50, 61, 79, 106

Unified Process Agile project management method breaks a project into 4 phases (Inception, Elaboration, Construction, Transition). 89

VCS Version Control System. xiii, 46, *see* Version Control System

Version Control System Version Control Systems like GIT or Subversion which makes it possible to keep track of the code with history entries. 46

vManage The API of Ciscos SD-WAN solution vManage. The API can be queried to gather information about the SD-WAN infrastructure.. ii, iii, v, 2, 5, 10, 11, 20, 26, 31, 34, 35, 50, 52, 69, 70, 78, 95, 133, 135, 148, 151, 155

vManage API The RESTful API of the vManage.. ii, iii, 2, 4–7, 9–11, 20, 23, 25, 27, 30, 31, 33, 37, 38, 47, 48, 50, 52, 56, 58, 60, 62, 67–71, 75–78, 86, 92, 94, 95, 132, 148, 150

WebSocket Communication protocol, providing full-duplex communication channels over a single TCP connection. ii, 4, 6–9, 20, 36, 37, 50, 56–60, 62, 73–77, 92, 95, 98, 108

List of Figures

1 SD-WAN Topology Viewer Topology . . . iv
2 SD-WAN Topology Viewer Metrics . . . v

1.1 Deploymentdiagram of the application . . . 6
1.2 Screenshot of the SDWAN Topology Viewer . . . 7
1.3 Topology states of SDWANTV . . . 9

2.1 UseCase Diagram . . . 14

3.1 Domain model . . . 22
3.2 Data model . . . 23
3.3 Live metric speed test . . . 32

4.1 SDWANTV Context Diagram . . . 35
4.2 SDWANTV Container Diagram . . . 36
4.3 Component Diagram Backend . . . 38
4.4 Component Diagram Frontend . . . 39
4.5 Component Diagram Celery . . . 40
4.6 Component Diagram Beat . . . 41
4.7 Component Diagram TimescaleDB . . . 42
4.8 Component Diagram Redis . . . 43
4.9 Runtime behaviour based on runtime . . . 52
4.10 Runtime behaviour based on the size of the topology . . . 52
4.11 Package diagram . . . 54
4.12 Fetch topology sequence diagram . . . 58
4.13 Partial topology update sequence diagram . . . 59
4.14 Fetching metrics sequence diagram . . . 60
4.15 GitLab CI/CD pipeline . . . 62
4.16 GitLab CI/CD pipeline . . . 64
4.17 UC5 wireframe . . . 65
4.18 UC5 display metrics . . . 66
4.19 UC5 running application screenshot . . . 66

5.1 Tunnels between Bern-A and Main-A . . . 70
5.2 Metrics between Bern-A and Main-A . . . 71
5.3 OpenAPI Documentation Overview . . . 72
5.4 OpenAPI Documentation Request . . . 72
5.5 OpenAPI Documentation Response . . . 72
5.6 States and topology flow . . . 77
5.7 Prototype app for deck.gl . . . 80

5.8 Brezier with 5 intermediate points . . . 81
5.9 Brezier with 10 intermediate points . . . 81
5.10 Brezier with 20 intermediate points . . . 81
5.11 Prototype app for deck.gl . . . 82
5.12 Aggregated tunnel popup . . . 84
5.13 Topology admin . . . 86
5.14 Topology service provider . . . 86
5.15 Topology customer 2 . . . 86
5.16 Topology customer 3 . . . 86

6.1 YouTrack Sprint Planning . . . 90
6.2 YouTrack Issue Overview . . . 91
6.3 Project Plan . . . 93
6.4 Risk analysis matrix . . . 94

7.1 Hours spent per component per team member . . . 99
7.2 Cake diagram time per phase . . . 100
7.3 Cake diagram issues per phase . . . 101
7.4 Cake diagram issue count per type . . . 102
7.5 Cake diagram issues per milestone . . . 103

A.1 Mapbox dashboard . . . 108
A.2 Failed sync status . . . 109
A.3 HTML friendly backend healthcheck . . . 109
A.4 vManage device groups . . . 110

B.1 Topology admin . . . 113
B.2 Topology customer . . . 113
B.3 Metrics admin . . . 113
B.4 Topology customer . . . 113
B.5 User list . . . 114
B.6 User list . . . 115
B.7 Add user empty form . . . 115
B.8 Add user filled in form . . . 115
B.9 User create confirmation . . . 116
B.10 User list after creation . . . 116
B.11 User list . . . 117
B.12 User delete popup . . . 117
B.13 User delete confirmation . . . 118
B.14 User list after deletion . . . 118
B.15 User list . . . 119
B.16 User form before update . . . 119
B.17 User form after update . . . 119
B.18 Udpate confirmation . . . 119
B.19 User list after update . . . 120
B.20 Unfiltered topology . . . 121
B.21 One site selected . . . 121
B.22 Two sites selected . . . 121
B.23 Topology filtered by two sites . . . 122
B.24 Initial topology . . . 123
B.25 Metrics overview . . . 123
B.26 Metrics overview . . . 124

B.27 Historical metrics 124
B.28 Specific datapoint selected 124
B.29 Metrics overview 125
B.30 Time range filter opened 125
B.31 Time range filter selected 125
B.32 Metrics overview 126
B.33 Company filter selected 126
B.34 Company filter and value selected 126
B.35 Metrics overview filtered 127
B.36 Metrics overview 127
B.37 Package loss filter selected 128
B.38 Package loss filter and threshold defined 128
B.39 Metrics overview filtered 128
B.40 Metrics overview 129
B.41 To sorting descending 129
B.42 To sorting ascending 129
B.43 Topology overview 130
B.44 Biz-internet 130
B.45 MPLS 130
B.46 Card 1 131
B.47 Card 2 131
B.48 Topology admin 132
B.49 Topology customer 132
B.50 Topology admin 132
B.51 Topology customer 132
B.52 Topology admin 133
B.53 Topology customer 133
B.54 Topology admin, Miami-a down 133
B.55 Topology customer, Miami-a down 133
B.56 Topology admin, Miami-a down 134
B.57 Topology customer, Miami-a down 134
B.58 Topology admin, Miami-a up 134
B.59 Topology customer, Miami-a up 134
B.60 Topology admin, full mesh 135
B.61 Topology customer, full mesh 135
B.62 Topology admin, policy active 135
B.63 Topology customer, policy active 135
B.64 Topology overview 137
B.65 Topology node selected 137
B.66 Node popup 137
B.67 Topology overview 138
B.68 Topology node group selected 138
B.69 Node group popup 138
B.70 Topology overview 139
B.71 Zoomed topology 139
B.72 Topology overview 140
B.73 Tunnel selected 140
B.74 Topology overivew, no filter 141
B.75 Company filter applied 141
B.76 Topology overivew, no filter 142
B.77 Site filter applied 142

C.1 Password stored in DB 144
C.2 Sample JWT payload 144
C.3 JWT token logging 144
C.4 Logs of the stdout stream 145
C.5 Backend validates password 146
C.6 Login failure message 147
C.7 Statistics of all task runns 148
C.8 Sync state if system runs successful 150
C.9 Sync is in error state 150
C.10 Sync information could not be fetched 150
C.11 Full match without policy applied 151
C.12 Policy applied 30s later 151
C.13 Render performance 152
C.14 Sonarqube evaluation frontend 153
C.15 Sonarqube evaluation backend 154
C.16 Full match fetching speed 155
C.17 Applied policy fetching speed 155

D.1 UC2 wireframe 156
D.2 UC2.1 wireframe 157
D.3 UC2.1 wireframe 2 157
D.4 UC3 wireframe 158
D.5 UC4 wireframe 158
D.6 UC5 wireframe 159
D.7 UC8 wireframe 159
D.8 UC8 wireframe 2 160
D.9 UC2 User overview 160
D.10 UC2.1 Update user 161
D.11 UC5 display metrics 161
D.12 UC5 display metrics 162

List of Tables

1.1 Use Case states 8
1.2 Architectural Tasks 8

2.1 Specification Use Case 1 15
2.2 Specification Use Case 2 15
2.3 Specification Use Case 2.1 15
2.4 Specification Use Case 2.2 16
2.5 Specification Use Case 3 16
2.6 Specification Use Case 4 16
2.7 Specification Use Case 5 17
2.8 Specification Use Case 5.1 17
2.9 Specification Use Case 5.2 17
2.10 Specification Use Case 5.3 18
2.11 Specification Use Case 5.4 18
2.12 Specification Use Case 6 18
2.13 Specification Use Case 7 19
2.14 Specification Use Case 7.1 19
2.15 Specification Use Case 8 19

3.1 vManage list devices response 26
3.2 vManage OMP services response 27
3.3 vManage IPsec connections response 28
3.4 vManage IPsec connections response 29
3.5 vManage list metrics aggregation response 30
3.6 vManage fetch live metric response 31

4.1 12 Factors 46

6.1 Team Members and Responsibilities 88
6.2 SDWANTV Releases 91
6.3 Risk analysis 94
6.4 Quality control measures 97

7.1 Working times per team member 99
7.2 Time spent per project phase 100
7.3 Issues per phase 101
7.4 Issues per task type 102
7.5 Time spent per milestone 103
7.6 Issues per milestone 104
7.7 Code statistics 104

B.1 Systemtest protocol UseCases 111
B.2 Systemtest protocol further scenarios 112

C.1 NFR test protocol 143

Part I

Technical Report

Chapter 1

Technical Report

1.1 Introduction and Overview

Towards the trend to software defined networks (SDN), more and more globally operating companies are choosing a SD-WAN solution to manage their wide area network infrastructure. SD-WAN is the latest technology to interconnect globally distributed company branches or customer locations over multiple transport layers like MPLS or also the internet. Companies are no longer constrained to lease expensive direct links from service providers but also can use the internet to route business critical traffic. SD-WAN not only simplifies the network management by building IPsec tunnels according to specific policies and rules, but also follows the "move to the cloud" trend and ability to connect to cloud networks.

Another big advantage of this approach are the monitoring capabilities. Since all routers are connected to the central controller over the control plane, they can also send state information and metrics to this central location. This functionality is especially interesting for network specialists who need to monitor the infrastructure in real time and react to anomalies as fast as possible. Having metric insights about the IPsec tunnels makes it possible to perform intelligent routing and always choosing the best path to the destination.

At the moment the biggest vendor of SD-WAN solutions is Cisco with their vManage. This tool collects all data of the infrastructure and provides it to third parties via a REST API. So far there is no software which can monitor the whole topology with routers and IPsec tunnels.

A software which is able to connect to the vManage API and display the topology state, would be able to satisfy this need from the industry and embrace the software defined networking on a global scale even more.

1.1.1 Problem

Cisco's SD-WAN solution vManage is a well working networking product. It provides a web application which is primarily designed for configuration and also has a simple graphical overview of the distribution of the individual routers on a world map. Unfortunately it is not designed for active monitoring of the infrastructure. Especially the lack of a visual representation of the IPsec tunnels between the routers and the historical metrics is a missed opportunity. Furthermore, Cisco's SD-WAN only targets the network operators and misses out to provide a monitoring tool for network customers.

A semester thesis has already built the foundation to solve this problem, but it lacks several important features such as the metric analysis and their historical evaluation or the customer portal which would enable a limited topology view to customers. In addition to that, the

infrastructure and the technology used are not adequate for a scaleable cloud ready software system.

1.1.2 Previous work

Our bachelor thesis is a follow up work based on our semester thesis. The semester thesis already contained some key elements of the final SDWANTV product. We were strongly dependent on the results from the semester thesis and therefore roughly state some features of it. For more detailed differentiation between those two artifacts, we used a git versioning system and strategically placed release tags.

Topology view From the semester thesis we already had a working topology view on the map. As the render engine in the frontend we used leaflet. To update the topology, we pulled the whole topology each 15 seconds from a REST API endpoint. The used authentication mechanism was Json Web Token.

Fullscreen Feature The full screen feature to toggle the full screen view was already a part of the semester thesis.

Show node and tunnel information With a click on a node and a tunnel the user already could see detailed information about the resources. This only worked for the leaflet map engine but not for the later deck.gl [15] and had less features.

Customer and site filter The topology could be filtered by the customer and the node site id. It was not possible to filter more than one site and combined the filters in a smart way.

Status display We already implemented the little tag on the top right corner of our frontend application, which displays the state of SDWANTV.

Authentication The authentication with Json Web Token for the REST API was already implemented in the frontend and the backend.

Authorization We had only one admin user which was able to see the whole topology. There was no user management or customer portal.

Development environment We used the same development environment as in the semester thesis, this includes the docker-compose as well as the linting and the testing configurations.

Deployment The deployment of SDWANTV was done with docker-compose. Docker-compose files were shipped which included all application parts as well as a Traefik proxy for TLS termination and routing.

1.1.3 Goals

In the context of the bachelor thesis, we want to extend the existing SDWANTV solution with new features and a better architecture. Furthermore, we want to target a new group of users, the customers, who are only allowed to see a limited view of the topology.

Features Our application fetches in a regular interval the required information from the vManage API and processes it. In a next step the changes of the data are stored in a database and directly forwarded to a progressive web application. The user will only see those topology changes which are relevant for the company he belongs to. On the web application itself, the user can monitor not only the status of the topology, but also the metrics of the tunnels. The metrics will be further separated into jittering, latency and package loss and are accessible over a dynamical adjustable historical chart.

It is difficult to describe each feature in detail and thus we provide the following list of features we want to implement:

- Limited topology and metrics view for customers
- A simple user management
- Filtering between two sites
- Historical metric analysis of the tunnels.
- Filtering and sorting to simplify the metric analysis

If possible, we want to implement other features such as:

- Tunnel colouring by metric statistics
- Global filtering
- Enforce password change for the users
- Export metrics to a PDF
- Metric alarms

For a more detailed description of the requirements, we refer to the Use Case Section 2.3.

Architecture Regarding the architecture, some key technologies should be introduced to improve various aspects of the software system.

One of them is the introduction of the WebSocket technology which is used by the backend server to only push topology changes to the web application, if they occur instead of pulling the topology constantly. Another one is the change of the map engine in the frontend, which will calculate the graphics on the graphic card instead of the CPU. There are in total six of them and their changes affect nearly every aspect of our system. The following is a list of all architecture changes we want to achieve in the bachelor thesis:

- Kubernetes and Helm Deployment
- WebSocket for communication
- WebGL powered map engine
- TimescaleDB
- Tunnel aggregation
- Event fetching

For a more detailed description of the architectural decisions, we refer to the Architectural Decision Section 4.5.

1.1.4 Limitations

Due to the bachelor thesis the personal expenses are limited. We have approximately 720h evenly distributed between two people at our disposal.

SDWANTV is heavily dependent on the vManage API. Consequently, we can only fetch those resources which are accessible via its endpoints.

We also have a lot of technical limitations regarding the scaleability of the SD-WAN monitoring. The more routers and IPsec tunnels are part of a SD-WAN, the more computational resources are required. This is especially a problem if we want to render the topology via a web browser on a normal computer.

Furthermore, most of the technologies are already set from the beginning because we used the semester thesis project as the baseline.

1.1.5 Work structure

Our project is split into two parts. The first part of the project is to gather all required information together, analysing the new vManage API endpoints, getting familiar with the new technologies and test it in an early prototype. In the second part of the project we apply architectural changes and implement new features.

The more detailed description of the project can be found in the chapter project management 6.

1.2 Evaluation

1.2.1 Information acquisition

In order to understand the problem domain of SD-WAN, an online research needed to be made. Especially Cisco had good online video tutorials [13], which were very helpful in the beginning.

The SDWANTV is heavily dependent on the vManage API. It is not only the baseline of the whole functionality of our application, but also a big performance bottleneck. With the help of the vManage API documentation, we can locate the important endpoints. The requests should be designed in a very efficient way and only query the required information.

During the development we will rely on the domain experts. They should show us the possibilities and the important requirements. Especially the supervisor, who has set up the development vManage environment should support us with tips for the testing.

1.3 Concept

After some evaluation of the vManage API for the new features, we enhanced the infrastructure design of our application. We knew we wanted to add some bigger architectural changes by introducing the WebSocket protocol between frontend and backend and storing historical metric statistics in the database.

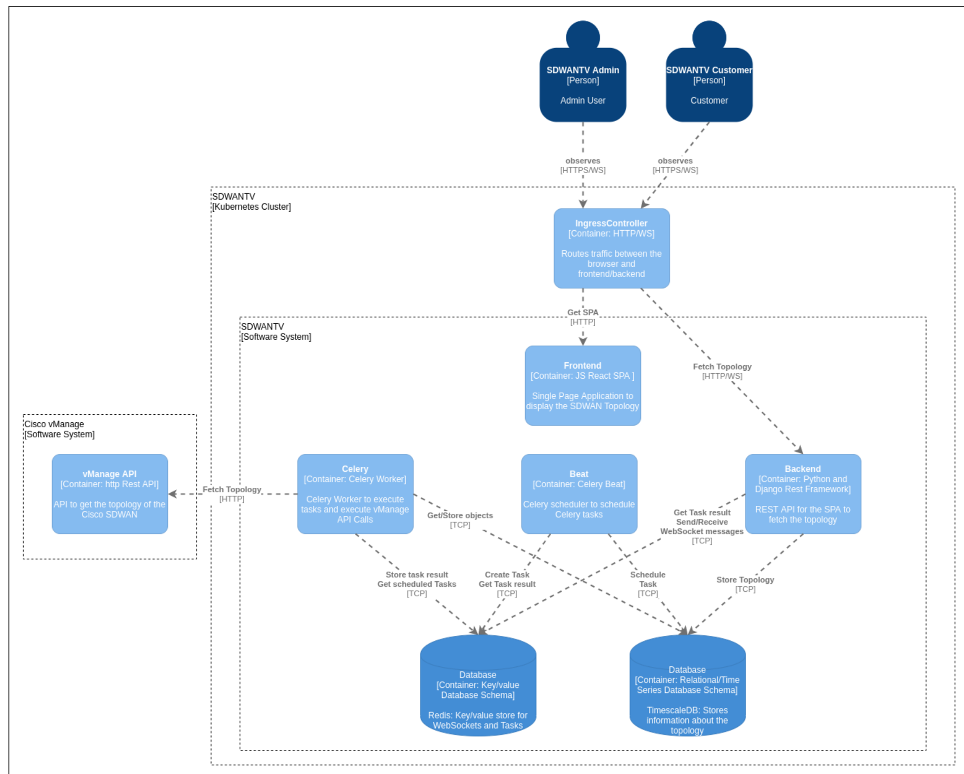

Figure 1.1: Deploymentdiagram of the application

The most parts of the architecture are the same as in the semester thesis. We fetch the information from the vManage API and store it in our internal Timescale database. The task logic should be decoupled in its own error boundaries from the rest of the application. This has been achieved by using a separate container for the tasks runner. Next to the WebSocket connection we still use the REST API of the Django backend to access information such as the status of the task execution or the metrics.

As depicted in the graph we have now an additional user in our system, the customer. Compared to the admin user, the customer has only limited access to the functionalities and can only see a subset of the topology graph.

It should be mentioned that the IngressController container in the architecture drawing is not part of SDWANTV but is a Kubernetes cluster component that is used to route the traffic to the correct container.

1.4 Solution

During the bachelor thesis we improved and extended the already existing SDWANTV application. We reduced the communication overhead between frontend and backend with WebSockets, increased the speed of the frontend rendering with WebGL and packed the system with Helm into a Kubernetes and cloud ready environment. Furthermore, we added some exciting new features and refined the old ones. Because of those improvements, the application is now in a stable and professional state. It is ready to be shipped and used on a daily basis.

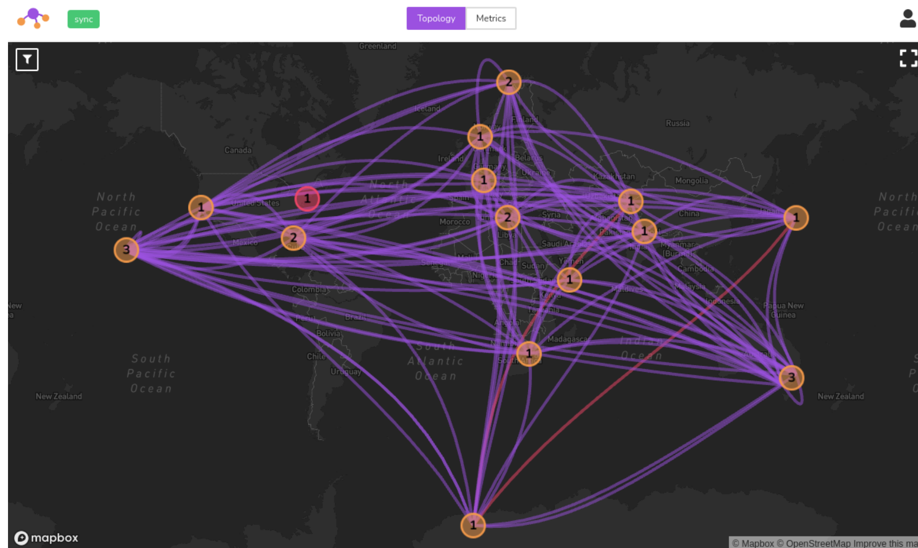

Figure 1.2: Screenshot of the SDWAN Topology Viewer

During the implementation we put a lot of emphasis on the performance. We finally ended up fetching a topology with 25 routers and 700 IPsec tunnels from the vManage API to our backend in an average of 12 seconds. For the metric statistic task, we are able to make a metric probe with six pings for all the 700 connections in an average of 8 seconds.

In addition to the performance, we wanted to keep the code quality in a good state in order to align with our non-functional requirements. This fact is represented by our test coverage which is more than 69% in the backend and more than 75% in the frontend.

The following table shows the state of the functional requirements. Further information about the use cases can be found in the requirements specification section 2.3.

Use Case	State
UC1: Customer Portal	Done
UC2: User Management	Done
UC2.1: Update customer	Done
UC2.2: Password change enforcement	Open
UC3: Add filter between two sites	Done
UC4: Acknowledge down tunnels	Omitted
UC5: Display tunnel metrics	Done
UC5.1: Specify historical metric time range	Done
UC5.2: Export metrics to pdf	Open
UC5.3: Filtering metrics	Done
UC5.4: Sorting Metrics	Done
UC6: Observe tunnel metric state by coloring	Open
UC7: Manage metric alarms	Open
UC7.1: Send alarm to external syslog server	Open
UC8: Global filtering	Open

Table 1.1: Use Case states

The following table shows the state of the architectural tasks. More information about architectural decision is in the project documentation section 4.5:

Use Case	State
AD1: Kubernetes deployment	Done
AD2: Websocket	Done
AD3: TimescaleDB	Done
AD4: WebGL	Done
AD5: Tunnel Aggregation	Done
AD6: Events fetching	Omitted

Table 1.2: Architectural Tasks

1.4.1 Implementation

WebSocket

For the WebSocket problem, we have discussed how we fetch the data and propagate them as messages in a clean way to the customer. Since we wanted to use the WebSocket and the REST API together it was necessary to separate those two endpoints under the same hostname. The frontend now needs to know the location of the REST endpoints and the one of the WebSocket.

First of all, we created WebSocket rooms for each company stored in our system. The task runner, which checks differences in the topology, can then decide based on the content of the changed resource into which room the update notification should go. Each active WebSocket connection that receives the update from the task runner can then check if the filter set by

the customer applies. If this is the case, the message is forwarded to the customer via the WebSocket protocol.

Later we noticed that the active WebSocket connection cannot detect the correct update only on the changed resource itself. Especially, if policies are applied and some nodes should disappear. This problem will be described in more detail in a later section. However, this problem forced us to overthink our software architecture and led to a new concept.

We needed to introduce a new topology representation in each active WebSocket connection. If a user logs in and accesses the topology the backend will open a new WebSocket connection and loads a sub topology into the in-memory state of the WebSocket connection. The sub topology is defined by the company to which the user belongs. On this in-memory sub topology we are now able to apply more complex topology operations like detecting the correct relationship between the nodes and tunnels. Eventually, this change enabled us to get rid of all concurrency problems and simplified the overall system.

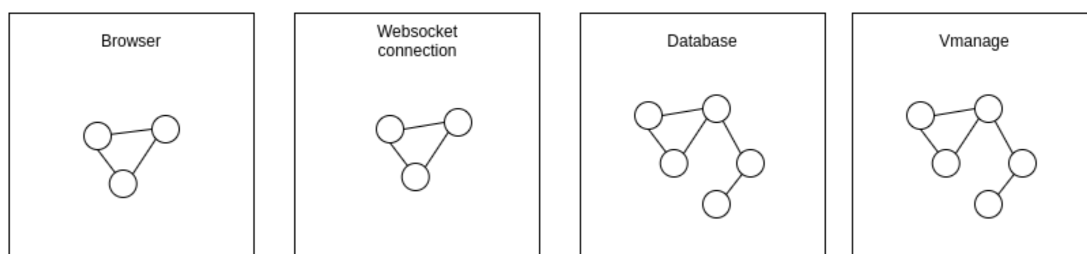

Figure 1.3: Topology states of SDWANTV

Historical metrics

In order to prepare for the historic metric related features, we changed some aspects of our architecture. The historical storage of the metrics quickly became a big data problem, since we had approx. 700 connections each receives two new metric data points every 2 minutes. Therefore, we will collect one million data points per day. Facing such a huge amount of data we decided to switch the database from PostgreSQL to TimescaleDB. Timescale is built on top of PostgreSQL and provides extra functionality to efficiently store time series data. If we want to query the metrics between a time range, we can do that by passing the two dates and the delta of the datapoints to timescale and we will receive so called data buckets with the correctly aggregated metrics.

1.5 Conclusion

Since we already worked on the product in the semester thesis, the baseline and the context were clear from the beginning. We were able to quickly start with the implementation without a lot of analysis beforehand. This gave us a kickstart at the beginning of the project.

As we applied our architectural changes, we realized that we underestimated the impact of them. Especially the change to WebSocket had such an impact, that a big part of the work done in the backend was to solve the problems in connection with it. Furthermore, we faced the same difficulties as we already had in the semester thesis. The vManage API is badly documented and the vManage API endpoints are not tailored to our needs. Although, we tried to mitigate those problems as much as possible but we still face a relatively bad scaleability and are not completely sure if we use the endpoints the way they are intended.

Disregarding the limitation of the vManage API we consider the bachelor thesis and its result as a huge success. We really improved the existing application and added important new

features. In comparison with its competition, the SDWANTV is now quite well placed and produces a real alternative. Although, the feature variety is still small, the ones which are implemented perform extremely well.

1.6 Forecast

1.6.1 Non-technical improvements

Some non-technical improvements to SDWANTV could be made. First of all, the not yet implemented Use Cases could be applied. Refer to the solutions section 1.4 to see which of the Use Cases have not been implemented yet.

If we are fancy enough, we can go beyond the Use Cases and think about the functionalities which we did not documented yet.

There is a big potential to improve the topology map furthermore. We could add more user experience support if we will highlight all connected tunnels of a node if we select a specific node. We also could make a deeper analysis of the data and visually show the paths the packages take in the network.

We already mentioned in the optional Use Cases that we would like to colour the tunnels. This is quite complicated task. The major problem is that a connection between the South Pole and the North Pole has per default a higher latency than a connection between Zurich and Bern. To solve this problem the thresholds for the colouring must be set per tunnel. Otherwise, it would be necessary to use data analysis to analyse what is the normal metrics value per tunnel and detect if the value deviates from the normal ones.

We already thought about a statistic page for the customer during the bachelor thesis but had eventually no time to implement it. The static page would contain some benchmark figures about the topology as a whole, for instants the number of nodes or tunnels or also the average time the backend takes to query data from the vManage API.

1.6.2 Technical improvements

Prometheus exporter Because we are using Kubernetes to orchestrate the running application we have an easy interface to export the application metrics to Prometheus. Prometheus processes the application metrics which afterwards can be queried with PromQL or even more the data could be used together with Grafana and display statistics about the application. When using the Prometheus exporter for Django application statistics are automatically exported and there is a community driven dashboard that for example shows the number of requests to the backend that were answered with a status code of 200.

Contact Cisco As we previously stated, the vManage API was not very well documented and difficult to understand. If we would have a contact to someone who is well informed about the API from Cisco, we could optimize our requests to the vManage API endpoints.

Checksum for verification To further verify if the topology shown on the user's screen is a correct representation, we could check in a regular interval with a checksum the state in the frontend and in the backend. This would improve the confidence of correctness of the system.

1.6.3 Technical debt

Correctly handle 503 vManage erros That vManage is the source of the most problems we are facing right now shows either that we did our job very good or that we did not understand

the vManage endpoints correctly. Nevertheless, vManage sometimes returns a 503 error as a response instead of the requested resources. We somehow need to properly handle this error and process the topology state accordingly. Sometimes it also happens that the requests to the vManage API runs into a read timeout failure. We already have increased the read timeout but it does not seem to be enough.

Refactor the tasks At the moment the code which is responsible for fetching the data from vManage is plugged into one giant file. On top of that we have not invested a lot of time for cleaning up the fetching and processing logic. Before we can implement more features in the future the code base needs to undergo some refactoring.

Improve testing Due the lack of time, we were not able to test the backend code as much as we wished. At the beginning of the bachelor thesis, we set a global threshold of 75% but at the moment we have 76% in the frontend and approximately 70% in the backend.

Part II

Project Documentation

Chapter 2

Requirements Specification

2.1 Thesis Requirements

2.2 Actors

2.2.1 Administrator

The administrator is the first user on the system. The administrator can see all information from all companies and can apply filters on those. The administrator also can invite new users to the system.

2.2.2 Customer

The customer is getting invited by the administrator. The customer can only see the company topology. The filter can only be applied on the information concerning his company.

2.2.3 System

The system is the software system as an independent actor.

2.3 Use Cases

The use case diagram shows an overview over all our use cases. The core requirements are marked in red and the optional requirements in blue. The grey use case got obsolete during the thesis.

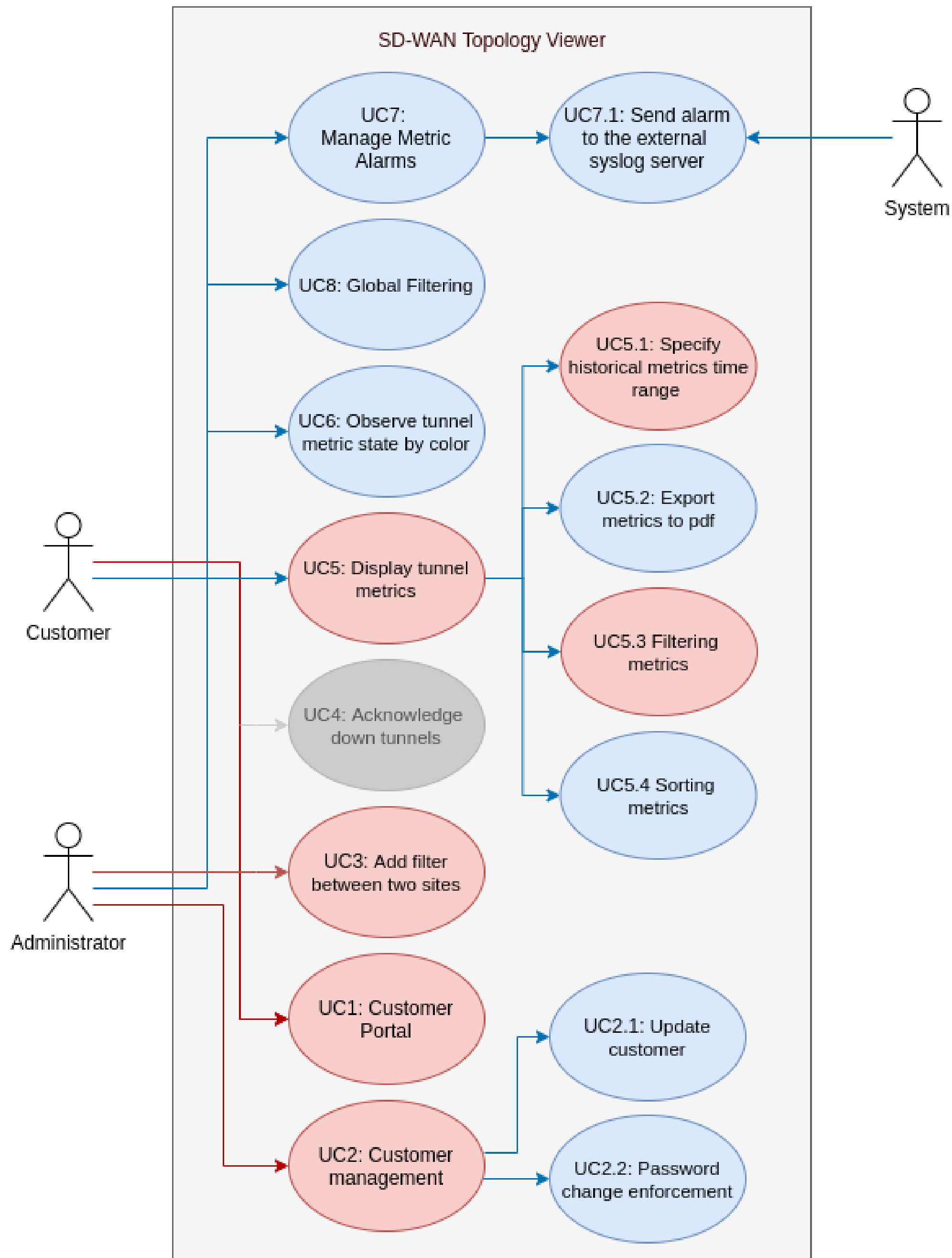

Figure 2.1: UseCase Diagram

2.3.1 UC1: Customer portal

Use Case Section	Description
Main Actor	Customer
Main Success Scenario	The customer can access the topology and the metrics overview. He will only see the information concerning his company.
Success Guarantee	The system evaluates the access rights and only delivers a subset of the data to the customer.
Alternative Success Scenario	The user has an admin role. This grants him unlimited access to the information of all companies.

Table 2.1: Specification Use Case 1

2.3.2 UC2: User management

Use Case Section	Description
Main Actor	Administrator
Main Success Scenario	The administrator can perform create, read and delete operation on the customer data. If the administrator creates a new user, he needs to attach a company and a role to it.
Success Guarantee	The system saves the performed operations successfully and adjusts the depiction for the administrator accordingly.
Alternative Success Guarantee	After a user was created the administrator sends the login credentials to the user. Eventually, the user is able to login.
Alternative Success Guarantee	After a user was deleted the user is not able to login anymore.

Table 2.2: Specification Use Case 2

2.3.3 UC2.1: Update customer

Use Case Section	Description
Main Actor	Administrator
Main Success Scenario	The administrator can update existing users. When editing a user the users values are filled in the update form and can be edited and updated.
Success Guarantee	The user details are successfully updated and the updated details are shown in the user management panel.

Table 2.3: Specification Use Case 2.1

2.3.4 UC2.2: Password change enforcement

Use Case Section	Description
Main Actor	Administrator, Customer
Precondition	The user received the initial login credential from the administrator.
Main Success Scenario	The customer is forced to change the password at the first login.
Success Guarantee	After system updated the password, the customer can access the system with the updated password.
Alternative Scenario	The customer tries to access the system before he updated his password. The access to the system will be denied and he will be prompted to enter a new password.

Table 2.4: Specification Use Case 2.2

2.3.5 UC3: Add filter between two sites

Use Case Section	Description
Main Actor	Customer & Administrator
Main Success Scenario	In order to focus only on the connection between two sites, the customer can select both of them via a filter.
Success Guarantee	The topology displays only the two sites and the IPSec tunnels between them.

Table 2.5: Specification Use Case 3

2.3.6 UC4: Acknowledge down tunnels

Use Case Section	Description
Main Actor	Customer & Administrator
Main Success Scenario	The user uses the SDWANTV to monitor the state of the network. As soon as a tunnel goes down, it is marked red. The user can now investigate why this happened. If the down state of the tunnel is expected, the user can acknowledge the state.
Success Guarantee	After the user acknowledged the issue, the corresponding tunnel will disappear from the topology map.
Alternative Success Scenario 1	If the tunnel went up again during the investigation, the red marking and the option to acknowledge the tunnel will be removed automatically by the system.
Alternative Success Scenario 2	If multiple tunnels go down at once, each tunnel needs to be acknowledged separately.

Table 2.6: Specification Use Case 4

2.3.7 UC5: Display tunnel metrics

Use Case Section	Description
Main Actor	Customer & Administrator
Main Success Scenario	The customer can see a list of all tunnels of his company network. The customer can then select a tunnel and display a historical summary of the jittering, package loss and delay. The customer can reside on the selected tunnel if he wants to watch in real time how new data points are added to the statistics.
Success Guarantee	The metric statistics will successfully be rendered and updated in a regular interval.
Alternative Success Scenario 1	An administrator wants to access the statistics. He either sees a list of all tunnels or limit the listing to only one company via a filter.

Table 2.7: Specification Use Case 5

2.3.8 UC5.1: Specify historical metrics time range

Use Case Section	Description
Main Actor	Customer & Administrator
Main Success Scenario	The customer can select a time range to limit the displayed statistic to a certain time frame.
Success Guarantee	Only statistics that belong to the defined time frame are loaded.

Table 2.8: Specification Use Case 5.1

2.3.9 UC5.2: Export metrics to pdf

Use Case Section	Description
Main Actor	Customer & Administrator
Main Success Scenario	The customer selects a time window and exports the statistics to a PDF. This exported PDF can be downloaded immediately after its creation.
Success Guarantee	A PDF is generated and ready to download for the customer.

Table 2.9: Specification Use Case 5.2

2.3.10 UC5.3 Filtering metrics

Use Case Section	Description
Main Actor	Administrator
Main Success Scenario	The administrator can apply filters to the metrics. It is possible to apply following filters: company, site, tloc color, node from, node to, package loss, latency and jitter. For package loss, latency and jitter a number defines the upper or lower bound of the threshold.
Success Guarantee	Only metrics that belong to the filtered property are shown to the user.

Table 2.10: Specification Use Case 5.3

2.3.11 UC5.4 Sorting metrics

Use Case Section	Description
Main Actor	Customer & Administrator
Main Success Scenario	The user sorts the metrics based on name, delay, jitter or package loss. He can decide for each sort value if it should be sorted ascending or descending.
Success Guarantee	The metrics are listed according to the sorting rules.

Table 2.11: Specification Use Case 5.4

2.3.12 UC6: Observe tunnel metric state by coloring

Use Case Section	Description
Main Actor	Customer & Administrator
Main Success Scenario	The user sees the health of the metrics per IPsec tunnel via a coloring on the tunnel on the topology map.
Success Guarantee	Each connection in the topology is colored according to its corresponding metrics.

Table 2.12: Specification Use Case 6

2.3.13 UC7: Manage Metric Alarms

Use Case Section	Description
Main Actor	Customer & Administrator
Main Success Scenario	The user can perform CRUD operations on alarms. He can add, delete and update an alarm based on the bandwidth utilization of a tunnel percentage of package loss, jitter or delay. He needs to provide a percentage threshold of the metric.
Success Guarantee	The user can see on the topology the connection that is actually used.
Alternative Scenario	The user input is incorrect. The system notifies the user about this.

Table 2.13: Specification Use Case 7

2.3.14 UC7.1: Send alarm to the external syslog server

Use Case Section	Description
Main Actor	The System
Preconditions	A metric threshold is being violated.
Main Success Scenario	The system sends a syslog message to the configured syslog server.
Success Guarantee	The external syslog server receives the message.
Alternative Scenario	The delivery of the message fails and the system receives an error. The system displays to the user that an alarm was triggered and that it was not able to send the message to the external syslog server.

Table 2.14: Specification Use Case 7.1

2.3.15 UC8: Global Filtering

Use Case Section	Description
Main Actor	Customer & Administrator
Main Success Scenario	The user adds a filter that is applied on the topology and on the metrics statistics.
Success Guarantee	The filter is applied on the topology and the metrics page of the application.

Table 2.15: Specification Use Case 8

2.4 Non functional requirements

This section defines the non functional requirements for SDWANTV. The implementation of the components should align with these requirements. The requirements are tested and the result can be found in the non functional requirements test protocol in the appendix C.

2.4.1 Functionality

Security The authorization should be 0Auth [1] conform. The passwords are never stored in plain text and are hashed and encrypted. We use the provided functionality from the django framework and do not implement the security features on our own. The database that stores the user information, should only be accessible from our system.

We should manage the session with a simple JWT [40]. The user should be able to authenticate to the backend with this token. The token should have a limited lifetime.

The authentication for the WebSocket connection should be implemented via the query parameter of the request url. The query parameter are on the application layer therefore is encrypted according to the TLS standard [45]. For the implementation we should prevent the logging of the connection url.

To understand security problems, the system needs to log all relevant information to the Stdout stream.

2.4.2 Reliability

Fault tolerance, user data The system should not fail because of an action done in the frontend. This means the backend checks the data and only accepts valid ones from the API endpoint. The frontend should also support the user in a way that he is able to insert the correct values. If the frontend validation fails the user should be notified about that on the fly.

Fault tolerance, vManage data The system should not fail if the vManage data could not be fetched correctly from the API. It should rather notify the user about the error and display the other resources which could be fetched successfully.

Maturity The system should be able to fetch the API data under normal circumstances successfully at least 90% of the times.

2.4.3 Usability

Understandability The user should be able to interact with our system without introduction or tutorials. Therefore, we want to keep our design as simple as possible. The interaction flow should not be interrupted with unnecessary design elements.

Failure management If an error occurs while synchronizing the data with the frontend, the user should be notified about that.

2.4.4 Efficiency

Time behaviour The system should be able to propagate all relevant changes in the vManage infrastructure to our frontend at least every 2 minutes.

Efficency compliance The system should be able to serve at least 100 user at the same time with each of them having a topology of at least 10 connections. Thus our system should be able to keep at least 1000 connection synchronized with the vManage API.

Response time If a customer visits the page and initializes the first page rendering, the web application should not load longer than 5 seconds under normal conditions.

2.4.5 Supportability

A developer that is familiar with the technology and has some experience with the project, should be able to track down a minor bug and fix it in at least 48h.

2.4.6 Portability

The system should be deployable in any system that supports Kubernetes as a container orchestration. It should also be cloud ready and all 12 factors for a cloud ready application should be applied.

2.4.7 Scalability

It should be possible to assign more resources to the system. The system should scale in O(n). That means if we double the resources, we should able to process twice as much data at the same time.

It should be possible to scale up the parts of the software system by deploying more instances of the same container.

Chapter 3

Analysis

3.1 Domain model

The domain model provides an overview about the problem domain. The detailed description of the entities can be found in the data model section 3.2 below.

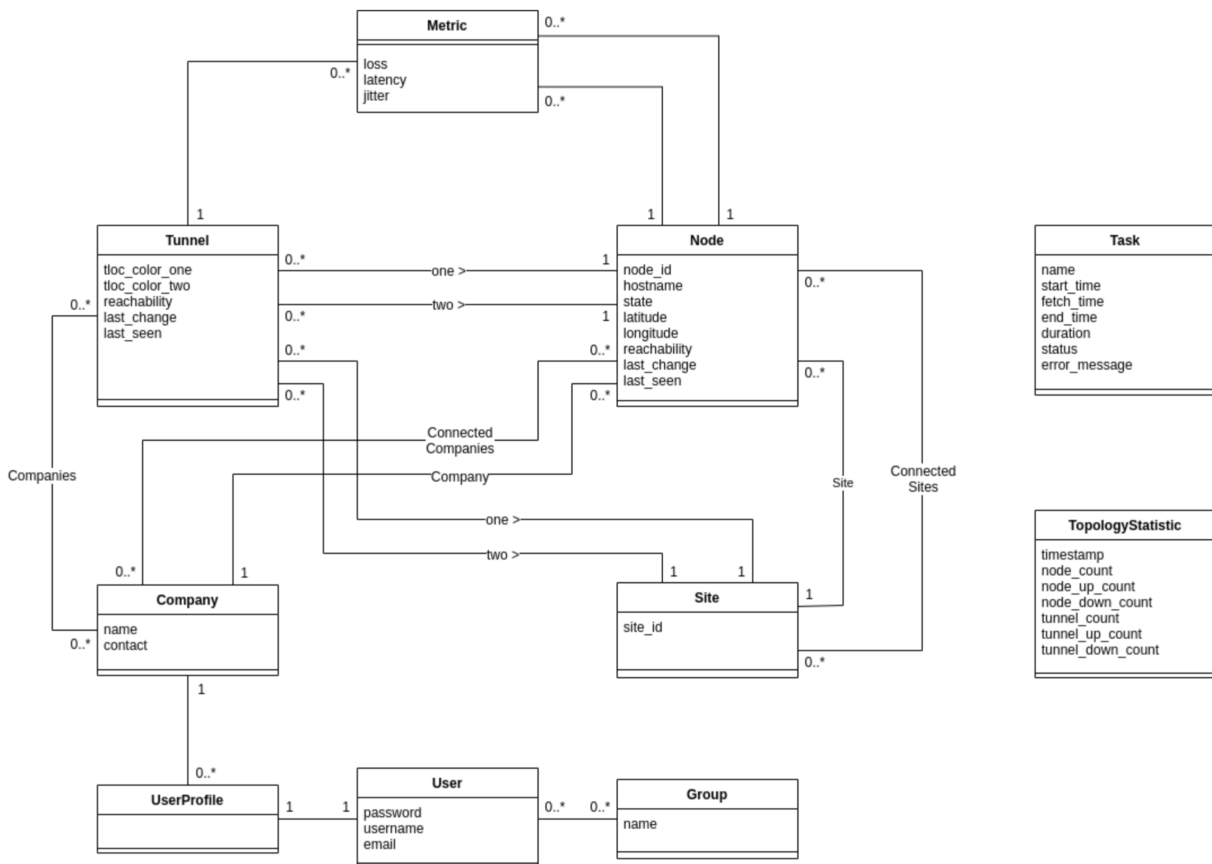

Figure 3.1: Domain model

3.2 Data model

The data model shows how the topology is represented in the Timescale database.

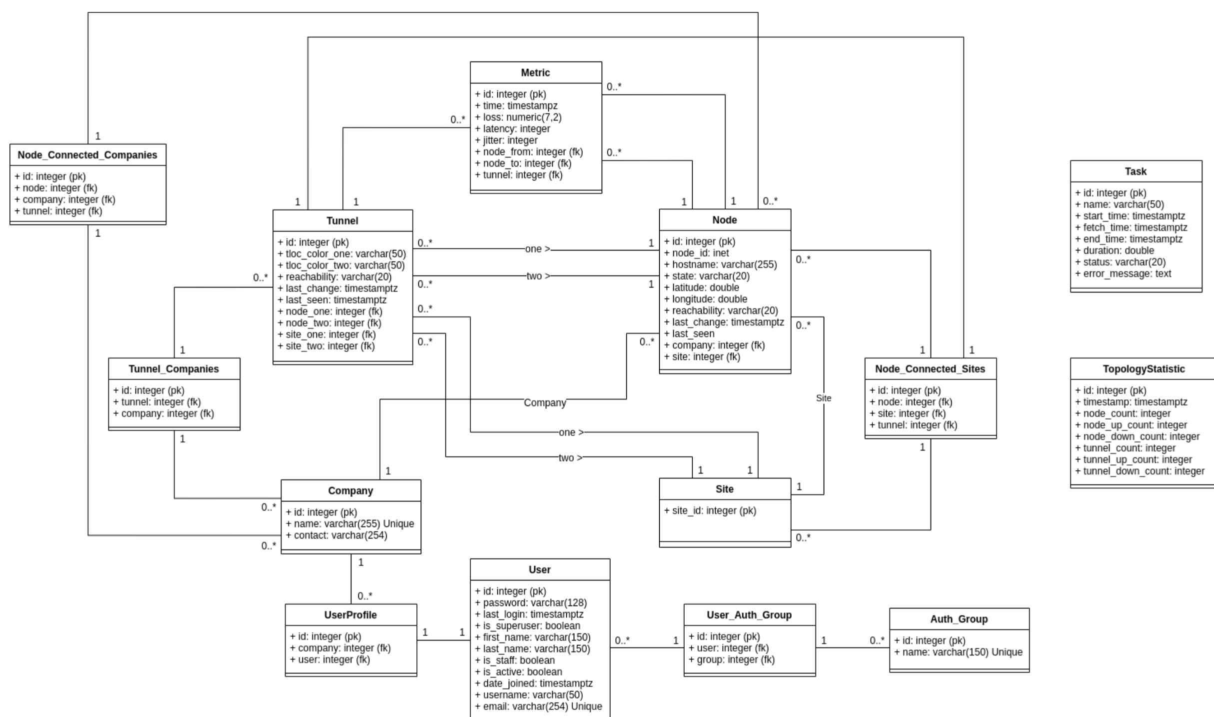

Figure 3.2: Data model

Auth_Group

As an entity it will be named Auth_Group and it is a default entity provided by the auth extension of the Django framework. We always create two groups; the customer and admin group. The groups are used to determine what kind of actions and endpoints a user is allowed to access.

User

The user is an entity that is created by us but inherited from the Django auth user class.

UserProfile

The UserProfile class has a one-to-one relationship with the User class and is an extension to it, to define more fields. It primarily is used to connect a user with a company.

Company

The company represents a customer who owns some nodes and tunnels in the topology. This field is fetched from the device-groups attribute of the nodes when fetched from the vManage API. A User always needs to be connected to a company. A company is used to identify which resources a user is allowed to see.

Node

A node is unique and only exists once in the topology. Every node belongs to one company and one site. Every node also contains a list of all companies and sites of every node that is connected through a tunnel.

Tunnel

The tunnel represents the bidirectional connection between two nodes on a transport layer. The tunnel contains the sites and companies of both nodes that it is connected to. This makes it easier to perform queries and requires less joins.

Site

A site represents a location with one or multiple nodes.

Metric

A metric is unidirectional and belongs to an tunnel. A tunnel typically has two metrics, one for each direction. A metric contains the package loss, latency, and jitter of a tunnel for a specific time. The referenced node objects on a metric object define into which direction the metric is.

Task

The tasks entity is used to manage the topology_sync and metrics_sync tasks running on celery workers. It is used to know when a task was exectued, how long it took to execute and in which state a task is at every given time. Furthermore, we want to display the state of the system in the frontend to reassure the user that the fetching works properly. Because tasks run independently from the rest of the system, we have no relationship to it.

ToplogyStatistic

The topologystaticstic represents the state how many nodes and tunnels are in the system at a specific point in time. Because it is only a statistics table it has no relationship to other tables.

3.3 Cisco vManage API analysis

Due to the fact that SDWANTV heavily depends on Ciscos vManage API, this section describes the API analysis of it. The following sections describe the REST endpoints and the attributes we require for our application to fetch the whole topology. Based on the API endpoints and our restriction of the non functional requirements, we need to achieve to call the **Authentication**, **Obtain XSRF token**, **List devices**, **OMP Servies** and **Tunnel list** endpoints.

3.3.1 Authentication

The first request against the vManage API is to authenticate with a username and password.

Request

```
POST /j_security_check
```

Request data

```
{
    'j_username': 'username',
    'j_password': 'secure-password'
}
```

Listing 3.1: Authentication POST request data

Response If the authentication with the provided user credentials was successful the vManage API returns a cookie which then needs to be saved in a session. Afterwards this session, containing the authentication cookie, can be used to perform further requests to the vManage API. In conjunction with the `set-cookie` response header the vManage API also sets the cookie attributes `path`, `secure` and `HttpOnly`.

```
set-cookie:
    JSESSIONID=MqOryjm2ZgSM9_U_ucGTdtq-sfd4dSZMJsLC7skG.212fffa6-6b9d-40bb-a9d2-03c834a2ad77;
    path=/; secure; HttpOnly
```

Listing 3.2: Authentication POST response header

3.3.2 Obtain XSRF token

After the authentication request was successful and the session is created, another request is required before it is possible to retrieve data from the vManage API. This request is to obtain an XSRF-token that always needs to be sent together with the sessionid cookie to perform requests.

Request

```
GET /dataservice/client/token
```

Response The vManage API returns an XSRF-token if the provided session-id cookie was valid. This token needs be stored in the `X-XSRF-TOKEN` sessions header together with the session-id cookie and used for future requests to retrieve data from the vManage API.

```
BDAD385CAB1CF2F19DE83B027E85CA2E371AA972BDB17CB2DE210354D48CAAC21EC25D0E3FB736C0B2BAE55B5EBB
```

Listing 3.3: XSRF token response

3.3.3 List devices

The `device` API endpoint lists all existing devices present in vManage and can be retrieved for the whole topology with one request.

Request

```
GET /dataservice/device
```

Response body

Response attribute	Description
device-type	We use this attribute to only filter for routers of the type vedges.
deviceId	Is actually the System-IP address of this device. We will use it as an unique identifier for the devices.
site-id	The site-id helps us to identify the location of a device. All devices with the same site-id belong to the same location.
host-name	The host-name will be used to display the edge in the GUI.
reachability	Informs if the device is reachable and therefore operating normally or if it is unavailable.
device-groups	Device groups helps us to decide which company the node belongs to.
latitude	Will be used to place the node on the right horizontal spot on the map.
longitude	Will be used to place the node on the right vertical spot on the map.

Table 3.1: vManage list devices response

Sample Response The most important fields of the response are shown in the sample response below. The full response can be found in the appendix F.1.


```
[
    {
        "deviceId": "10.255.255.133",
        "host-name": "Customer-king-Hawaii",
        "reachability": "reachable",
        "device-type": "vedge",
        "device-groups": [
            "\"king\""
        ],
        "site-id": "32",
        "latitude": "19.5429",
        "longitude": "-155.6659",
    },
    ...
]
```


Listing 3.4: Devices sample response

3.3.4 VPN ID

The OMP Service API contains the vpn-id, which we need to group the devices to one company. Sadly we have to request the vpn-id in a separate request for each device. If we had 100 companies with at least 10 nodes each, we might do up to **1000** Requests to the vManage API. It is possible to omit these requests if devices are not grouped together by the vpn-id and just use the `device-group` field of the node.

Request

```
GET /dataservice/device/omp/services?deviceId={{deviceId}}
```

Response body

Response attribute	Description
originator	We use this attribute to match the deviceId of the device.
vpn-id	All devices with the same VPN-id belong to the same company. A device can have multiple VPNs.

Table 3.2: vManage OMP services response

Sample Response The most important fields of the response are shown in the sample response below.

```
[
    {
        "originator": "10.255.255.163",
        "vpn-id": "20"
    },
    ...
]
```

Listing 3.5: OMP services sample response

3.3.5 IPsec connections

The IPsec inbound endpoint returns the connections information between two devices. A huge problem is that this needs to be executed for each device in the topology. This fact makes our application not very scalable. For 100 companies with 10 connections each we need **1000** requests and there is no way to fetch the connection information in another way.

Request

```
GET /dataservice/device/ipsec/inbound?deviceId={{deviceId}}
```

Response body

Response attribute	Description
local-tloc-address	Shows which device the connection originates from.
local-tloc-color	This attribute is the transport protocol used by the originator. It is used for displaying purposes.
remote-tloc-address	Shows with which other device the current device has an inbound connection.
remote-tloc-color	This attribute is the transport protocol used to reach the destination device. It is used for displaying purposes.

Table 3.3: vManage IPsec connections response

Sample Response The most important fields of the response are shown in the sample response below. The full response can be found in the appendix F.3.

```
[
    {
        "local-tloc-address": "10.255.255.162",
        "local-tloc-color": "mpls"
        "remote-tloc-address": "10.255.255.111",
        "remote-tloc-color": "mpls",
    },
    ...
]
```

Listing 3.6: IPsec inbound sample response

3.3.6 Tunnel list

After all devices were retrieved it is required to fetch all the existing connections between the devices. This is what the bfd state endpoint can be used for. This request needs to be performed for every single device in the topology and returns the list of connections/tunnels it has to other devices. Seeing as there is no general API endpoint that returns all the existing connections between the devices, this request needs to be done one time for every device in the topology which is not very scalable if the topology contains a lot of devices. Unfortunately there is no other method to fetch all existing connections between devices so this is how tunnels are fetched.

Request

```
GET /dataservice/device/bfd/state/device?deviceId={{deviceId}}
```

Response body

Response attribute	Description
vdevice-name	The device where the connection originates from.
local-color	The transport protocol used by the originator.
vsystem-ip	The device where the connection ends in.
color	The transport protocol used to reach the destination device.
state	The state of the tunnel. Either up or down.

Table 3.4: vManage IPsec connections response

Sample Response The most important fields of the response are shown in the sample response below. The full response can be found in the appendix F.2.

```
[
    {
        "vdevice-name": "10.255.255.162",
        "local-color": "mpls"
        "vsystem-ip": "10.255.255.111",
        "color": "mpls",
        "state": "up"
    },
    ...
]
```

Listing 3.7: BFD state sample response

3.4 Cisco vManage API metrics analysis

The following tables displaying the REST endpoints and the attributes we require for fetching the metrics. Based on the API endpoints and our restriction of the non functional requirements, we need to achieve to call the **Metrics history** and **Live metrics** Endpoint. This led us to the conclusion that there are 3 different ways how to process these metrics.

1. Storing both live metrics and historical metrics in our database.
2. Storing live metrics in our database and pass-through historical metrics requests.
3. Not storing any metrics at all and pass-through all requests to the vManage API.

This analysis helped us to decide which of the above 3 possible solutions suits the needs of the customer best.

3.4.1 Fetch metric history

The `device` API endpoint lists the historical metrics for one tunnel. The result of the request is used to display the metric statistics. For this request the HTTP POST method is used along with an aggregation query parameter. The aggregation query parameter can be found in the attachment section G.1.

Request

```
POST /dataservice/statistics/approute/fec/aggregation
```

Response body We actually receive one array with all the timestamps and one array with all the datapoints. The values received in the datapoint array are discussed in more detail.

Response attribute	Description
entry_time	This is the timestamp of the data point.
count	As the query specifies the granularity of the result, we receive a count that shows how many data points have been merged.
jitter	This is the average jitter of the merged data points.
loss_percentage	This is the average percentage of package loss merged over the given intervall.
latency	This is the average latency of the merged data points.

Table 3.5: vManage list metrics aggregation response

Sample Response The most important fields of the response are shown in the sample response below. The full response can be found in the appendix G.2.

```
[{
        "entry_time": 1615262400000,
        "count": 6,
        "jitter": 0,
        "loss_percentage": 0,
        "latency": 2
    }, ...
]
```

Listing 3.8: Approute statistic sample response

Speed It was very important for our design decision how fast the query can be performed from the vManage API. For that we check two variants. vManage API could response in the first variant, which fetched a time range of 24h with a 144 data points, in only **280ms**. The second one delivered us in **600ms** a time range of one week with 336 data points.

Load testing In order to make sure that we can use the vManage API for multiple concurrent users we executed a performance test for this endpoint. The performance test consisted of 100 request with 168 data points distributed over one week. The response time averaged over all queries has been **128ms**.

3.4.2 Fetch live metrics

As we are building a monitoring application, we want to update the metrics statistics live. If we fetched the whole statistic each time we wanted to fetch the newest data point, we would face performance issues.

Request

```
GET /dataservice/device/app-route/statistics?deviceId={deviceId}
```

The request query can be made more specific by using `remote-system-ip={remoteIP}`, `local-color={localColor}` or `remote-color={remoteColor}`.

Response body The request will take on average eight seconds to respond. This long response time is due the fact that vManage sends a probe to the respective device. The probe consists of six measurements with a gap of one second between. From the six measurements we will calculate the average values and store them in our database.

Response attribute	Description
average-jitter	The devices will send their information in an regular interval to the vManage. Therefore we will always have summary over a time window and never a real live metrics. This property is the average jitter value summarized over the time frame.
average-latency	The average latency in milliseconds summarized of the time frame.
total-packets	The total packages transmitted in the summarized time frame.
index	We will receive six measurements with a gap of one seconds. The index differentiates the six measurements from each other.
loss	We use this value in combination with the total-packets to calculate the relative packet loss in percent.

Table 3.6: vManage fetch live metric response

Sample Response The most important fields of the response are shown in the sample response below. The full response is in the attachment G.3.

```
[
   {
          "total-packets": "664",
          "loss": "0",
          "average-latency": "2",
          "average-jitter": "0",
   },
   ...
]
```

Listing 3.9: Live metrics sample response

Performance testing The live metric will take at least six or seven second to response, because the request does six probes with one second gap in between. As we wanted to use this endpoint for our own service, we decided to make a performance test to check how reliable the response time really is. For this we executed 360 times a metric request to all nodes on the network. As one can see in the figure below, the response time never exceeds 15 seconds. We simultaneously watched the workload on the routers to check if our requests do not strain the CPU too much. However, our performance tests was not even perceivable in the CPU statistics.

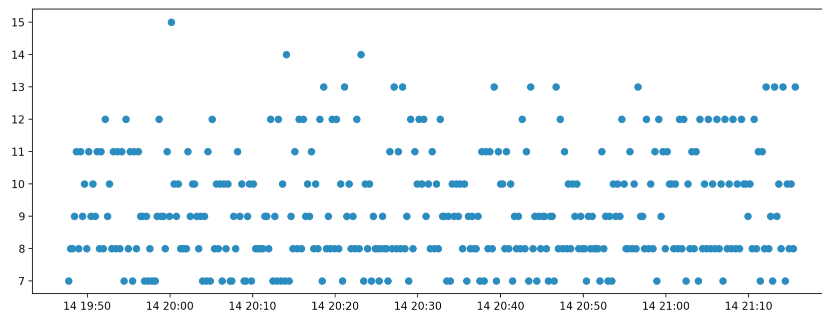

Figure 3.3: Live metric speed test

3.4.3 Metrics analysis conclusion

According to the metrics analysis and the good performance of performing repetitive requests we decided to go with option number one and store both historical and live metrics in our database. Furthermore, storing the metrics in the SDWANTV database would simplify to fulfill the Use Cases UC6 (section 2.3.12), UC7 (section 2.3.13) and UC7.1 (section 2.3.14).

3.5 Database analysis

A central part of the bachelor thesis is the requirement to provide metrics for every IPsec tunnel. This requirement is written down in Use Case UC5 2.3.7 and required to do an analysis of the vManage API metrics endpoints which is described in section 3.4.

Typically metrics data are time-series data which require to be saved in a time-series database. So far we were using a PostgreSQL database which is a relational database and does not provide a way to store in an efficent way time-series data out of the box. That is why it was required to perform a database analysis that fulfills the following requirements:

- Compatible with Django.
- Minimal effort for integration.
- Able to store time-series and relational data.
- Data retention to delete/rotate old data.
- Aggregation of data.

Because of the facts that a key requirement is to store time-series and relational data in one database and that it needs minimal integration effort to use it, solutions like InfluxDB, Elasticsearch, Graphite or CrateDB were not an optimal solution.

After some research and a recommendation we received in the semester thesis, we found that TimescaleDB [43] fits our needs the best and fulfills all our requirements. TimescaleDB is a relational database that supports also time-series data and claims to be even faster than native time-series databases. It is based on PostgreSQL and therefore fully compatible.

The analysis went even further than online research but also included to test Timescale together with Django in a prototype which is described in section 3.6. Another part of the analysis was to make sure that it is compatible with SDWANTV and the existing PostgreSQL database can be exchanged by TimescaleDB.

3.6 Prototype

The feasibility of the architectural decisions, documentend in section 4.5 should be ensured with the prototype. The goal of the prototype was to make sure all the technical components work together and to also get familiar with the new technologies. To make sure everything works also in a container based environment, it was built with docker-compose. The following components are part of the prototype.

- TimescaleDB database
- Django with WebSockets
- React with WebSockets
- React with DeckGL

The prototype was a success and all the components worked together neatly.

Chapter 4

Architecture & Design Specification

4.1 Scope

This chapter describes the architecture and design of the SDWANTV application. The installation and configuration of Cisco vManage is not part of this thesis and will be provided by the industry partner and for the development by the INS.

4.2 Software Architecture

The software architecture documentation is based on the C4 diagram technique [10]. Due to the fact that we deploy SDWANTV on Kubernetes the diagrams were sligthly adjusted but still contain the same design approach as we had in the semester thesis.

- Context Diagram: Cluster overview
- Container Diagram: Pod overview
- Component Diagram: Components around the pods

4.2.1 Context Diagram

The context diagram shows the overview of SDWANTV and the surrounding systems.

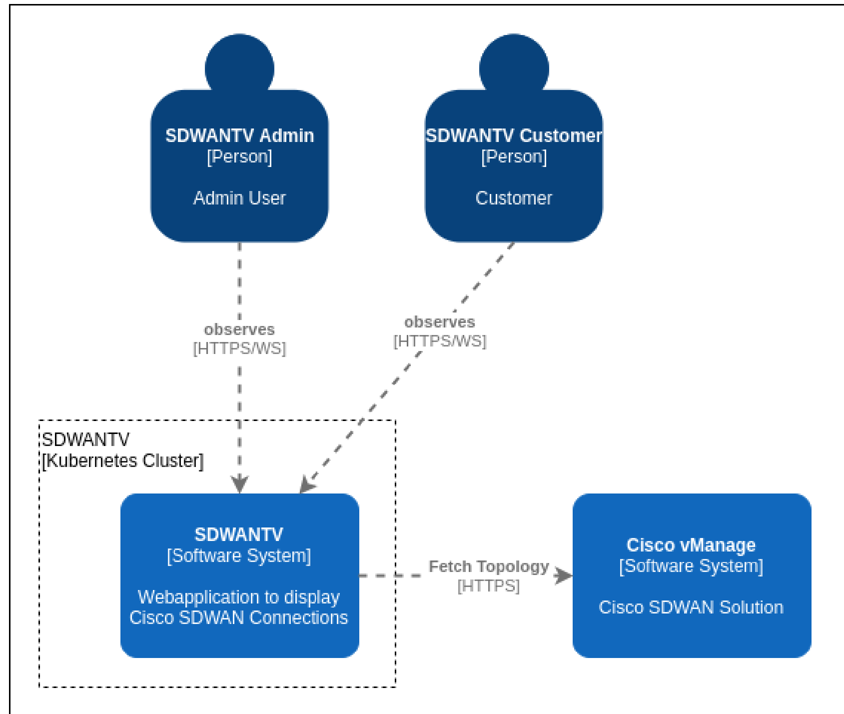

Figure 4.1: SDWANTV Context Diagram

SDWANTV SDWANTV is the software system that is developed in this thesis. It is designed to run inside a Kubernetes cluster next to other applications. The user interacts with the SDWANTV application to observe the SD-WAN topology. Admin users can view the entire topology, while customers are only shown their own topology.

Cisco vManage The Cisco vManage system is the SD-WAN solution from Cisco. It provides an API which we query to get the necessary data to build the topology and displays it in the SDWANTV UI.

4.2.2 Container Diagram

The container diagram provides a brief overview of how each container interacts with the other containers. Due to the fact that SDWANTV is designed to run in a Kubernetes cluster, the containers represent the pod construct of Kubernetes.

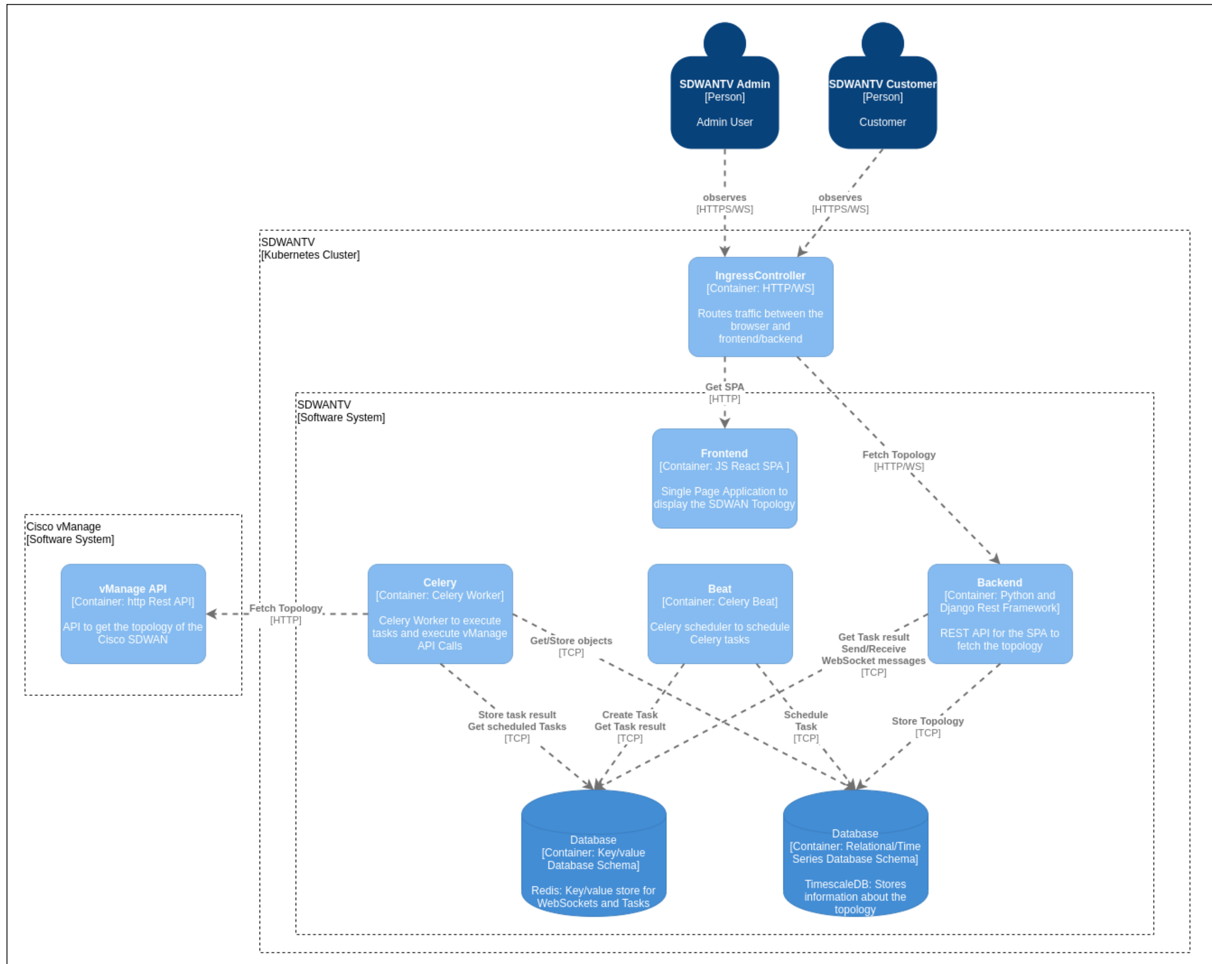

Figure 4.2: SDWANTV Container Diagram

SDWANTV Admin & Customer

The admin and customer are the primary users who use SDWANTV. They both interact with SDWANTV over a certain web domain. The user first accesses the "/"-Endpoint via HTTPS. The ingress controller forwards the request to the Nodejs server. The Nodejs server delivers the required assets for a Single Page Application to run in the browser. From then on the client only accesses the "*/api*"-Endpoint for REST API calls and the "*/ws/*"-Endpoint to connect to the WebSocket.

Ingress controller

The ingress controller is a proxy system that is responsible for the traffic routing from the outside of the Kubernetes cluster to the applications running inside the cluster. It is a component that is provided by the Kubernetes platform. Therefore it is not part of SDWANTV and displayed outside of the SDWANTV software system boundaries in the container diagram. However, the helm chart for the deployment provides a configuration for ingress resources that are processed by the ingress controllers. This approach makes it possible for the Ku-

bernetes cluster maintainer to use its preferred ingress controller implementation like Nginx or Traefik. The ingress controller only needs to be capable of processing HTTP and WebSocket connections. Ingress controller are also responsible for providing the application to the outside of the cluster with a secured TLS certificate and are responsible for the certificate management.

Frontend

As mentioned before, the frontend pod delivers the assets such as HTML, CSS, graphics and JS, to the browser. Once the Single Page Application is loaded into the browser the client can use it to display the SD-WAN topology and metrics. From this point in time no communication between the browser and the frontend server is done anymore. The data is provided by the backend pod.

Backend

The backend pod provides the topology and metrics data over a REST API and WebSockets. Another part of the backend pod is the handling of authentication and authorization. It is the primary pod that communicates with the TimescaleDB to gather all the required data to perform its duties.

Beat

Beat is a part of Celery and responsible to periodically schedule tasks at a defined time. The periodically scheduled tasks are sent into the Redis database which be consumed and processed by Celery pods.

Celery

The Celery pod fetches is a Celery worker and capable of processing the tasks that are sent to the Redis queue by the Beat pod. We will schedule two types of tasks. The first one is the fetching of the whole topology from the vManage API. The second one is the fetching of the network metrics of every IPsec tunnel.

Database: Redis

Redis is the database that is used by Beat to schedule tasks and send them into the task queue. Redis is also required for the WebSocket connections. Messages that are sent from the consumer to the provider or vice versa are sent into another Redis queue in the same container.

Database: TimescaleDB

The database used for SDWANTV is TimescaleDB. TimescaleDB is a relational and time series database the is built upon PostgreSQL. All data that needs to be saved persistently are stored in the TimescaleDB. Hence we will not take advantage of PostgreSQL specific optimization for indexing or other speedups which would require stable data.

vManage API

The vManage API is an external API. It is the gate to the SD-WAN solution [12] of Cisco. The vManage API is a full blown control tool to mutate and monitor the SD-WAN solution. We will only need a subset of all the functionalities.

4.2.3 Component Diagram

The component diagram zooms in on the different pods and describes the additional Kubernetes resources that are required for the pods to operate.

Backend

The diagram shows all required resources for running the backend container of SDWANTV.

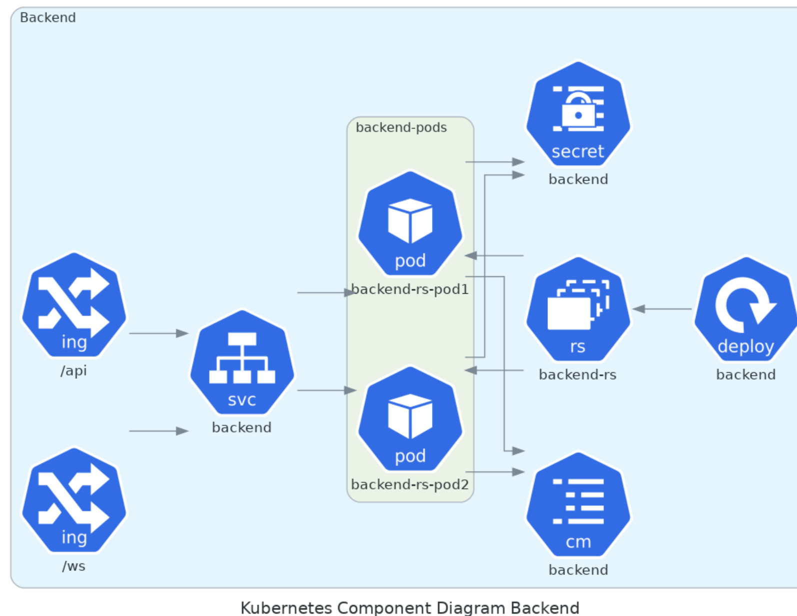

Figure 4.3: Component Diagram Backend

- **Ingress**: The two ingress objects are defining the hosts */api* and */ws* paths that should be forwarded to the backend pod. Both ingress resources point to the backend service.
- **Service**: The backend service object is of type ClusterIP and points to the backend pods on port 8000.
- **Pod**: The backend pods are the actual container that process the requests. There must be at least one pod. But since we aim for a high availability application, we deploy two pods.
- **ConfigMap**: The backend configmap contains all the environment variables that are required for the backend pod to run correctly. It is mounted into every backend pod.
- **Secret**: The backend secret contains the credentials to access the database, redis and the vManage API. It is mounted into every backend pod.
- **ReplicaSet & Deployment**: These are meta objects that define how many instances of the backend pods should be started, define the update strategy and other metadata around the backend pods.

Frontend

The diagram shows all required resources for running the frontend container of SDWANTV.

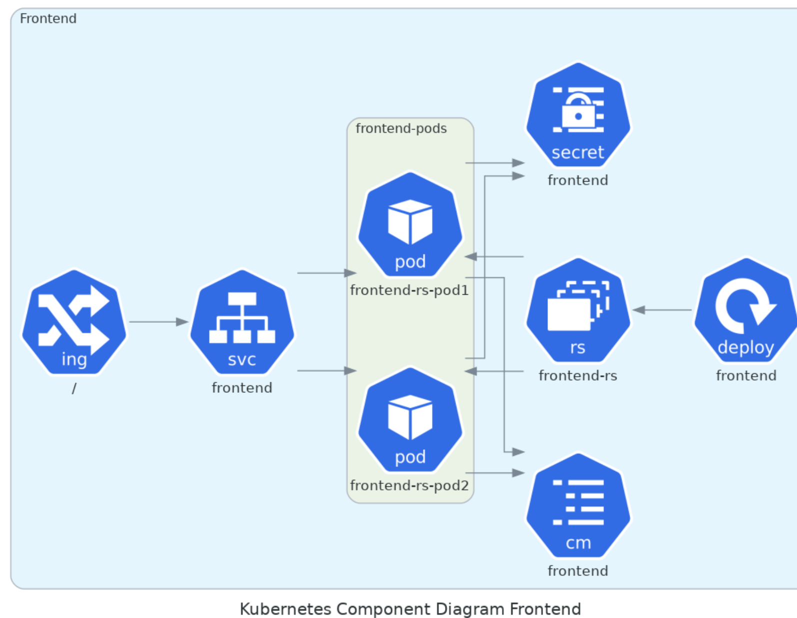

Figure 4.4: Component Diagram Frontend

- **Ingress**: The ingress object defines the hosts, / path that should be forwarded to the frontend pod. The ingress resource points to the frontend service.
- **Service**: The frontend service object is of type ClusterIP and points to the frontend pods on port 3000.
- **Pod**: The frontend pods are the pods that provide the Single Page Application. There must be at least one pod but for high availability concerns we deploy two.
- **ConfigMap**: The frontend configmap contains all the environment variables that are required for the frontend pod to run correctly. It is mounted into every frontend pod.
- **Secret**: The frontend secret contains the mapbox access token that is required for the mapbox Map. It is mounted into every frontend pod.
- **ReplicaSet & Deployment**: These are meta objects that define how many instances of the frontend pods should be started, define the update strategy and other metadata around the frontend pods.

Celery

The diagram shows all required resources for running the Celery worker container of SD-WANTV.

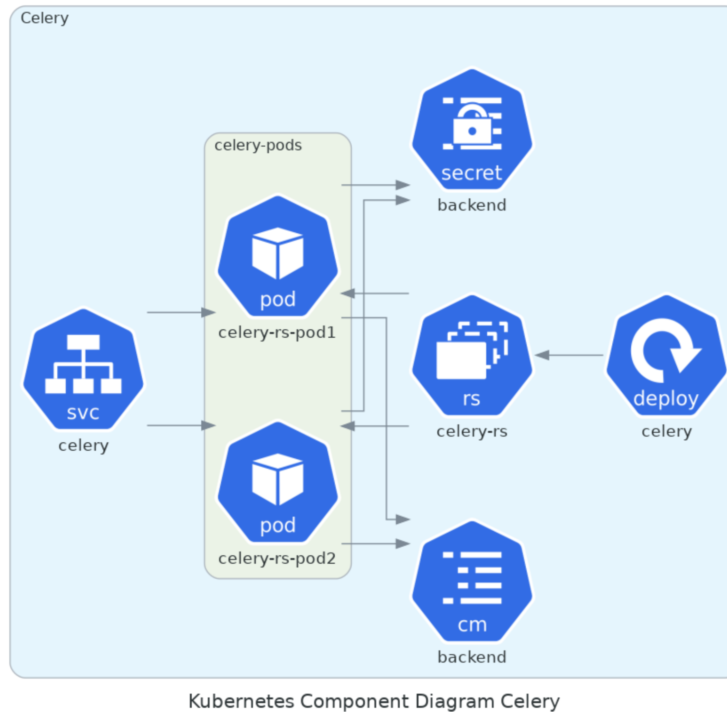

Figure 4.5: Component Diagram Celery

- **Service**: The celery service object is of type ClusterIP and points to the celery pods on port 8000.
- **Pod**: The celery pods are the pods that process tasks. There must be at least one pod but for high availability concerns we deploy two.
- **ConfigMap**: The celery pods use the backend configmap because celery uses the same environment variables like the backend pods. It is mounted into every celery pod.
- **Secret**: The celery pods use the backend secret because celery uses the same environment variables like the backend pods. It is mounted into every celery pod.
- **ReplicaSet & Deployment**: These are meta objects that define how many instances of the celery pods should be started, define the update strategy and other metadata around the celery pods.

Beat

The diagram shows all required resources for running the Celery Beat container of SD-WANTV.

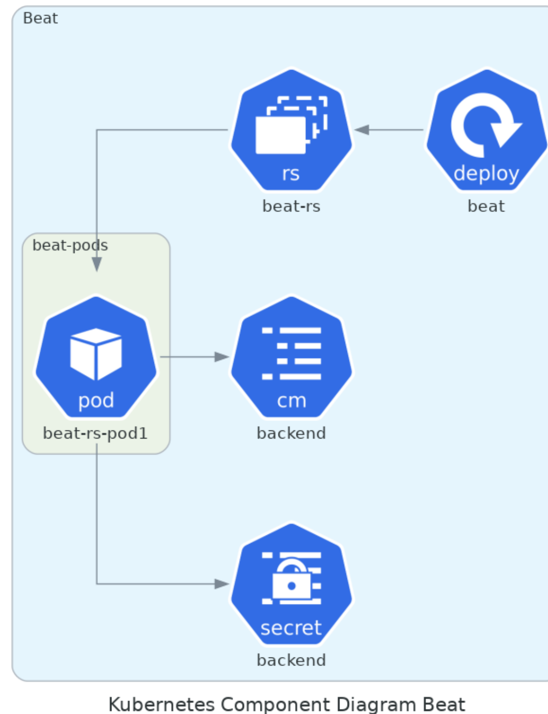

Figure 4.6: Component Diagram Beat

- **Pod**: The beat pod is the pod that schedules tasks into the task queue. Because tasks would be scheduled twice with 2 instances there is only one instance. This means when updating SDWANTV there will be no task scheduling for some time.
- **ConfigMap**: The beat pod uses the backend configmap because beat uses the same environment variables like the backend pods. It is mounted into the beat pod.
- **Secret**: The beat pod uses the backend secret because beat uses the same environment variables like the backend pods. It is mounted into the beat pod.
- **ReplicaSet & Deployment**: These are meta objects that define how many instances of the beat pod should be started, define the update strategy and other metadata around the beat pod.

TimescaleDB

The Timescale database is a requirement of SDWANTV. Seeing as the deployment of a Timescale database cluster is not in the scope of the SDWANTV, only the most important objects are described. However, it is possible to deploy a TimescaleDB cluster from the SDWANTV helm chart.

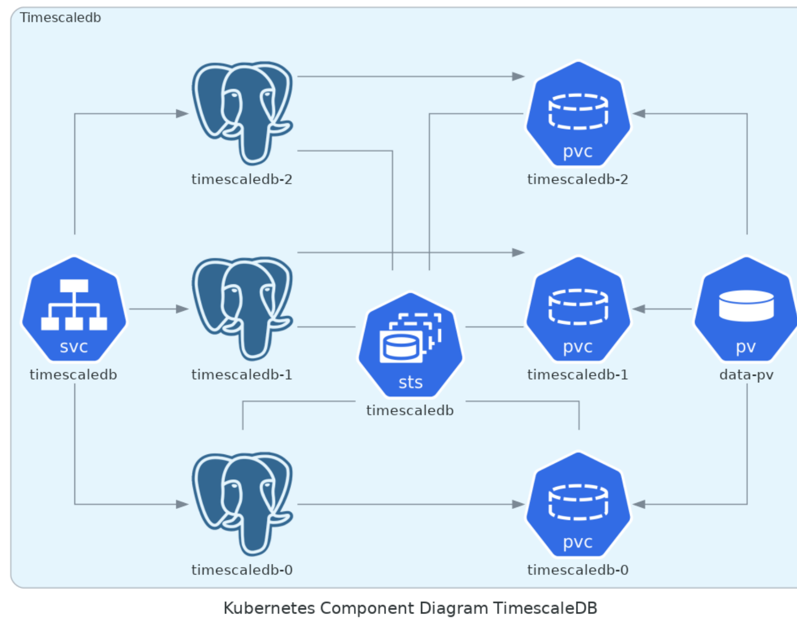

Figure 4.7: Component Diagram TimescaleDB

- **Service**: The TimescaleDB service object is of type ClusterIP and points to the TimescaleDB pods on port 5432. The TimescaleDB service object is the endpoint where the backend, celery and beat can connect to.
- **Pod**: The TimescaleDB pods start a Timescale database cluster. By default there are always an odd number of TimescaleDB pods because a cluster always needs to evaluate a master and talk to each other to have a quorum.
- **StatefulSet**: The statefulset defines how many instances of TimescaleDB pods should be started, define the ordered deployment and update strategy, automatically create a persistent volume claim and define other metadata around the TimescaleDB pods. StatefulSets are like a deployment but for stateful applications like databases.
- **PersistentVolumeClaim**: For every TimescaleDB pod a persistent volume claim is created which defines the required capacity of the storage that should be requested on the defined storage. This automatically triggers the creation of the persistent volume.
- **PersistentVolume**: Because TimescaleDB is a stateful application every pod needs storage in the name of volumes. A volume is always requested by a persisten volume claim and is mounted into the TimescaleDB pods.

Redis

The Redis database is a requirement of SDWANTV. Because the deployment of a Redis database cluster is not in the scope of the SDWANTV only the most important objects are described. However, it is possible to deploy a Redis cluster from the SDWANTV helm chart.

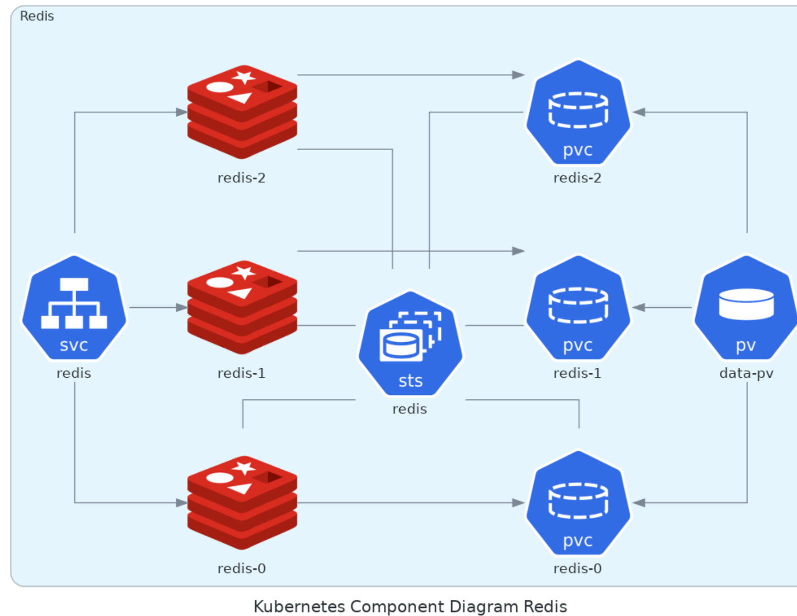

Figure 4.8: Component Diagram Redis

- **Service**: The redis service object is of type ClusterIP and points to the redis pods on port 6379. The redis service object is the endpoint where the backend, celery and beat can connect to.
- **Pod**: The redis pods start a redis database cluster. By default there are always an odd number of redis pods because a cluster always needs to evaluate a master and talk to each other to have a quorum.
- **StatefulSet**: The statefulset defines how many instances of redis pods should be started, define the ordered deployment and update strategy, automatically create a persistent volume claim and define other metadata around the redis pods. StatefulSets are like a deployment but for stateful applications like databases.
- **PersistentVolumeClaim**: For every redis pod a persistent volume claim is created which defines the required capacity of the storage that should be requested on the defined storage. This automatically triggers the creation of the persistent volume.
- **PersistentVolume**: Because redis is a stateful application every pod needs storage in the name of volumes. A volume is always requested by a persisten volume claim and is mounted into the redis pods.

4.3 Deployment

In a normal software system the deployment diagram describes which part of the system runs on which tier. Due to the fact that SDWANTV is deployed on a Kubernetes cluster the distribution on the tiers is handled by Kubernetes itself. Therefore a normal deployment diagram does not provide any value and was left out.

4.3.1 Helm Deployment

The deployment to Kubernetes is done by Helm [27]. Helm is a package manager to deploy Kubernetes applications and includes the GO templating engine to make the whole deployment dynamic and configurable. By using Helm as the deployment method for SDWANTV we achieve a 3 way separation of responsibilities. The software system code is separated from the helm deployment yaml files and further separated from the configuration values for the deployment and the application code. Moreover, all three parts are stored in their own git repository and are independent of each other.

A helm chart consists of templated yaml files for Kubernetes resources and a values yaml file that provides the values that are filled in the templated resource yaml files. The structure of a Helm chart looks like the following:

```
sdwantv/
    charts/
        redis-12.8.1.tgz
        timescaledb-single-0.8.2.tgz
    templates/
        - configmap-backend.yaml
        - configmap-frontend.yaml
        - deployment-backend.yaml
        - deployment-beat.yaml
        - deployment-celery.yaml
        - deployment-frontend.yaml
        - ingress.yaml
        - ingress-ws.yaml
        - rbac-depends-on.yaml
        - secret-backend.yaml
        - secret-frontend.yaml
        - service-backend.yaml
        - service-celery.yaml
        - service-frontend.yaml
        - serviceaccount.yaml
    - Chart.yaml
    - values.yaml
    - README.md
```

charts/ The *charts* directory contains dependency helm charts that can be installed as a dependency of the SDWANTV helm chart. Because SDWANTV requires a Redis and a TimescaleDB installation the Bitnami Redis Helm chart [8] and the TimescaleDB Helm chart [44] are added as dependency charts. This means that it is possible to not only install SDWANTV but also all dependencies with the provided helm chart. By default, both TimescaleDB and Redis are installed as a dependency. However it is also possible to disable the installation of Redis or TimescaleDB if one already has a TimescaleDB or Redis cluster and wants to use these or deploy these separately.

templates/ The *templates* directory contains all the yaml template files that will be renderered with the values from the *values.yaml* file and installed, in an automatically determined order, on the cluster.

Chart.yaml The *Chart.yaml* file contains the helm chart version, the app version, defined dependencies (Redis and TimescaleDB) and some other meta information about the chart.

values.yaml The *values.yaml* file contains all the variables/values that can be set to configure the installation. For example passwords, connection credentials but also configuration options for the kubernetes deployment can be set here.

README.md The *README.md* file provides the installation steps for this chart, any other required information and a table of all possible configurable values and their default value.

4.3.2 Development with Docker-compose

For the development of all containers we use the tool docker-compose [21]. Docker-compose is a tool to run multi-container docker applications. Applications, which are represented as services inside docker-compose, are described in a YAML file together with persistent storage and networks.

Backend Development The docker-compose file for the development of the SDWANTV backend starts the TimescaleDB database, adminer to inspect the database, Redis, Celery, Celery Beat and the Python backend container with the Python development webserver inside. All python containers are configured to run in debug mode. Additionally, the backend container is configured for hot reloading which means that changes on the application will be replicated inside the container and the webserver restarted. This makes it perfect for development.

Frontend development For the frontend development the backend development docker-compose file is started. The development of the frontend occurs outside of a docker container and connects to the backend python container. This is due to a faster development experience, where the docker file sync can be omitted.

4.3.3 Goals

SDWANTV is split into SDWANTV frontend and backend to enable flexibility and extensibility. The backend offers business services, holds the core logic and implements the persistence layer. The frontend is only responsible for presenting the data gathered from the backend in an accurate manner. This makes the core functionality independent from the frontend and enables a high level of automation.

4.4 Design

4.4.1 Twelve Factors

Due to the fact that we decided to containerize our application, it is important to meet the Twelve Factors [2] to design a good and clean cloud ready application. Our goal is to meet the requirements of all of the Twelve Factors.

Factor	Description	How?
I. Codebase	Code hosted in a Version Control System (VCS)	4.4.1
II. Dependencies	Explicitly declare and isolate dependencies	4.4.1
III. Config	Store config in the environment	4.4.1
IV. Backing Services	Treat backing services as attached resources	4.4.1
V. Build, release, run	Strictly separate build and run stages	4.4.1
VI. Processes	Execute the app as one or more stateless processes	4.4.1
VII. Port binding	Export services via port binding	4.4.1
VIII. Concurrency	Scale out via the process model	4.4.1
IX. Disposability	Maximize robustness with fast startup and graceful shutdown	4.4.1
X. Dev/prod parity	Keep development, staging and production as similar as possible	4.4.1
XI. Logs	Treat logs as event streams	4.4.1
XII. Admin processes	Run admin/management tasks as one-off processes	4.4.1

Table 4.1: 12 Factors

I. Codebase

This factor describes how the code of a cloud-native application should be hosted on a VCS. There always needs to be a one-to-one correlation between the code repository and the app. The frontend and the backend are both hosted in their own code repository on GitLab. Therefore, we consider this factor as fulfilled.

II. Dependencies

A twelve-factor app never relies on implicit existence of system-wide packages. All dependencies need to be declared completely in a dependency declaration file. Furthermore, it uses a dependency isolation tool to ensure that no implicit dependencies from the surrounding system affect the application. All components of SDWANTV are packed into their own container and therefore are completely isolated from the surrounding system outside the container. To manage compile time dependencies, the package managers pip for python and npm for react are used. Therefore, this factor is considered to be fulfilled.

III. Config

A strict separation of the config from the code is required to meet this factor. Apps never store config constants in code. We strictly separate configuration from the code. All configuration parameters are passed to the application by docker environment variables. This makes it

possible to change the behaviour of the application from outside the container. Therefore, this factor is considered to be fulfilled.

IV. Backing services

Backing services are services that the app consumes over the network as part of the normal operation. For example, databases, messaging/queueing systems, caching systems or also third party services like the Twitter API or Amazon S3 storage. A 12 factor app should make no distinction between local and third party services. The access URL to these services needs to be configurable and exchangeable. For example, it should be possible to change the database from a local instance to a cloud hosted one without any changes to the code. We developed the backend in the so named manner. The URLs of the PostgreSQL database, Redis cache and Cisco vManage API can both be configured via environment variables. Therefore, this factor is considered to be fulfilled.

V. Build, release, run

The code needs to be transformed into a production deploy through the stages build, release and run. All 3 stages need to be separated strictly. The build stage takes the code and bundles it into an executable object. The release stage takes the executable object and enriches it with the configuration. The run stage takes the output of the release stage and runs it on the execution environment. The GitLab pipeline we configured for both repositories takes the *Dockerfile* and build the container image. In the deploy stage of the pipeline, the helm chart and the configuration values are checked out and finally deployed on the Kubernetes cluster. With the automation of the deployment this factor is considered fulfilled.

VI. Processes

Twelve-factor processes are stateless and share nothing. Any data that needs to persist must be stored in a stateful backing service, typically a database. The app never assumes that anything cached or on disk will be available on a future request. All data is strictly stored in the stateful PostgreSQL backing service. The disk for the database is handled by a persistent volume on Kubernetes. If the app is started and no persistent volume is available, a new one will be created. The application does not share a state with each other. Therefore, this factor is considered to be fulfilled.

VII. Port binding

The twelve-factor app is completely self-contained and does not rely on an external webserver to run. The app exports HTTP as a service by binding to a port and listening to requests coming in on that port. Both the frontend and the backend container already have a production-ready webserver included and listen to HTTP requests on a defined port. Therefore, this factor is considered to be fulfilled.

VIII. Concurrency

In a twelve-factor app, processes are meant to run concurrently. There may be a various amount of process types of which a various amount of processes should be able to run concurrently and do their tasks. When the traffic increases, the resources can also be increased to scale up the application capacities. Gunicorn, the production webserver we use in the backend, uses a master/worker model where it is possible to define the number of workers [26] that respond to HTTP requests concurrently. The same is used in the frontend but with

nodejs. Furthermore it is possible to depoy multiple replicas of a container in Kubernetes which makes it very scalable. Therefore, this factor is considered to be fulfilled.

IX. Disposability

Processes are disposable, they need to be started or stopped at any moment. Processes should minimize startup time and shut down gracefully if a SIGTERM signal is received. As soon as a pod receives the shutdown command, Kubernetes will send a SIGTERM signal to the containers root process (PID 1) and tell it to end its tasks and shutdown. If the graceful period is exceeded and the pod did not shutdown Kubernetes will send the pod a SIGKILL signal and force shutdown it. Because we use Kubernetes as application management and the described behaviour is part of it, this factor is considered to be fulfilled. Restarting pods is possible at any times. Error handling is implemented in the core functionalities. Occurred errors are captured and handled in an appropriate way. The frontend is always aware of the sync status of the backend. Therefore, this factor is considered to be fulfilled.

X. Dev/prod parity

Twelve-factor compliant apps should be designed for continuous deployment to have a minimal gap between development and production. The time between deployment should be minimal. Additionally, the code authors should be the same people as the ones that deploy the app. We use GitLab and SA pipelines to automatically build and deploy our application every time a push into a branch happens. Testing in the development environment happens with an sqlite database and requests to the external Cisco vManage API are mocked. Therefore, this factor is considered to be fulfilled.

XI. Logs

A twelve-factor app should never need to manage the routing or storage of its output stream. It should not write logs to files or manage logfiles but write log events to the Stdout stream. All components of SDWANTV are configured to write log messages directly to the Stdout stream and not into logfiles. Therefore, this factor is considered to be fulfilled. More about logging can be found in the section logging 6.9.

XII. Admin processes

One-off admin tasks, such as database migrations, should be run in an identical environment as the regular processes of the app. Admin code should be shipped together with the application code. Django ships the admin code in the same repository with the normal code. Admin tasks can be executed automatically at the startup or from a shell inside the container, or with a scheduled timer. Admin tasks can only be executed by users with admin privileges. Therefore, this factor is considered to be fulfilled.

4.5 Architectural Decisions

Based on the semester thesis we will continue to improve the software quality. The technical improvements are not directly linked to a specific Use Case. However, they have a massive influence on the non-functional requirements.

To back-trace the design decision made at the early state of the project, we will create with the help of the Y-template [47] a transparent decision reasoning.

The individual tasks can be very time-consuming and are therefore prioritised.

4.5.1 AD1: Kubernetes Deployment

SDWANTV will run in a cloud environment (public or private cloud). Hence, for maximal portability and reliability an application management standard to describe SDWANTV with all its containers. Kubernetes is the most widespread orchestration software how to perform automated deployment, scaling and managing container based applications.

Y-template section	Description
In the context of	a production and cloud ready application
Facing	a rapid changing cloud environment with many different cloud provider. And a small project team, with little time to learn new complex technologies for the deployment.
We decided	to use Kubernetes
And neglected	other technologies like docker-compose, or a cloud provider specific software management solution
To achieve	independence of specific cloud provider, requiring less application management overhead and a minimum of learning effort.
Accepting the downside	of having to invest more time in transforming the application to run on Kubernetes.

4.5.2 AD2: WebSocket

We will place the WebSocket technlogy as an additional communication protocol between the backend and the frontend. The backend will fetch the data for the topology from the vManage API and pushes the processed data to the frontend. This technology will solve two problems for us, scalability because it will only update the frontend when a topology change happened and speed by reducing the time when a topology change happens in the vManage and until is visible to the user.

Y-template section	Description
In the context of	the communication between the backend and the frontend.
Facing	a scalability and a response speed bottleneck.
We decided	to use WebSocket
And neglected	a classic pull mechanism on the fronend for fetching the data or a newer technology like Server-Send-Events.
To achieve	a faster system which is flexible enough to adjust itself according to the amount of processing time the backend needs. It also makes the resource usage more economical because it will only require a communication to the frontend, if actual change happened in the topology.
Accepting the downside	having a more complex solution.

4.5.3 AD3: TimescaleDB

The new use cases in the bachelor thesis require us to adjust our database technology. The usecase 2.3.7 will introduce time series data, that needs to be fetched in a very efficient way. TimescaleDB is an database engine based on PostgreSQL and therefore will retain the old functionalities of PostgreSQL.

Y-template section	Description
In the context of	a database engine to manage relational and time series data.
Facing	potential speed and usability problems.
We decided	to use TimescaleDB
And neglected	a manual configured PostgreSQL database or other technologies for time series data like InfluxDB.
To achieve	a convenient system that provides the speed and an easy usage during development.
Accepting the downside	of introducing an unknown technology that is not as well supported by Django as the old PostgreSQL solution.

4.5.4 AD4: WebGL

The frontend which runs on a browser engine is one major bottleneck in SDWANTV. We already optimized our code to work as efficiently as possible under these circumstances, but if we used the WebGL technology, it would be possible to take advantage of the GPU and render more nodes and edges in a shorter time.

Y-template section	Description
In the context of	performance problems in the frontend running in the browser.
Facing	a massive bottleneck in SDWANTV.
We decided	to use webgl
And neglected	to spend more time in optimize the actual rendering.
To achieve	a fast and scalable frontend on devices owning a graphic card.
Accepting the downside	introducing a new technology which adds a layer of complexity to the system and not having an performance gain for devices which do not own a graphic card.

4.5.5 AD5: Tunnel Aggregation

Although this is a technical task, it influences more the usability than the actual functionality of the system. We want to establish a solution, which reduces the number of connections displayed on the map by merging the tunnels into a bigger aggregate.

Y-template section	Description
In the context of	the usability on the topology map.
Facing	an overload of tunnels between the nodes.
We decided	to reduce the amount of tunnels by aggregating them based on rules.
And neglected	the possibility to keep it as it is.
To achieve	an more user friendly interface.
Accepting the downside	introducing a rules set for aggregation and abstract certain information away.

4.5.6 AD6: Events fetching

The current implementation is designed to fetch the whole topology that is present in vManage in a defined interval. However, as soon as the number of routers and IPsec tunnels increases, the time to fetch the topology will rise. The vManage API also provides an endpoint to fetch occurring events. To lower the time to fetch the topology, we could fetch the whole topology only every hour and in between fetch the occurring events every 5 seconds. However, this still requires a lot of analysis and whether it really speeds up the fetching time is unclear.

We assume the **events fetching** provides a counterweight to the current **whole topology fetch** implementation. The events fetch will perform well if the topology is large and the amount of events produced by the system is small. The whole topology fetch on the other side, will perform well if the topology is small and the amount of events is big.

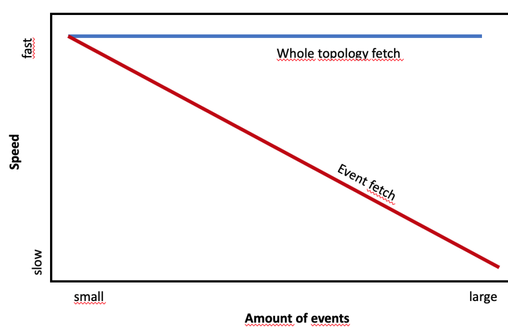

Figure 4.9: Runtime behaviour based on runtime

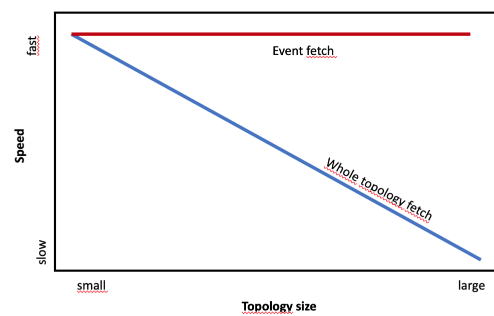

Figure 4.10: Runtime behaviour based on the size of the topology

The biggest challenge in this technical task will be to enable SDWANTV to automatically decide which approach is faster. This can be implemented by a simple if statement or a machine learning algorithm.

There are also situations where both solution will perform badly. In cases where the topology and the amount of events produced is large in size we can neither profit from the fast topology nor the fast event fetching.

Y-template section	Description
In the context of	the fetching of the SD-WAN topology by celery tasks.
Facing	a scalability problem if the SD-WAN network increases in size.
We decided	to update our internal topology representation with event fetching and whole topology fetching.
And neglected	the possibility to only use whole topology fetching or only event fetching
To achieve	a system that performs better on large topologies with a small amount of events.
Accepting the downside	introducing additional complexity to the system.

4.5.7 Client/Server Cut (CSC)

For the distribution design of SDWANTV we decided to use the two well known patterns **remote user interface** and **remote database** from the distribution pattern catalog. These two patterns are also the most common patterns used in today's applications. Additionally, as described below the technologies we decided to use already preset the need to use these patterns. Django by default defines the database to be a remote database and React recommend that not too much business logic should be at the client. Therefore the backend should store the business logic and React should only be responsible to query the backend and display the data in a meaningful way to the user.

Due to the fact that SDWANTV is designed with a cloud native approach the frontend, backend and the database should be separated strictly for scaleability.

Django Backend

For the implementation of the backend API of SDWANTV, we decided to use Python. The decision to use Python as programming language was pretty clear, as we were already familiar with it and the supervisor had recommended it. In addition, the employees of the institute can help us if we encounter problems, as they are also very familiar with Python and its frameworks.

We finally decided to use the Django Rest framework. However, this decision was not easily made. The other option to create the Rest API was to use Flask Restplus [24], which we were already familiar with. We arranged a meeting with a domain expert to discuss this topic, as he knows both of the frameworks. By using flask it would be easier to create an API. In contrast, it is much easier to create the database model with Django. Because Django also provides a built-in user management and easy way to setup API authentication, we decided to use the Django Rest framework.

React Frontend

The decision to use a single page application written in React as the frontend was quite easy. Like Python React was proposed by the supervisor. In addition to that, we already worked with React and as a result would be faster during the construction phase.

But React is also the right framework to use when it comes to dynamic updates. React itself watches the state of the objects on in the Single Page Application and if an object receives a new value React will detect this and update the object. This is exactly what we require to always have an up to date representation of the data without reloading the whole website.

Furthermore, React includes a good amount of security mechanisms. React automatically escapes the input of form fields. This prevents Cross-Site-Scripting and Injection attacks.

Alternatives like Angular or VueJS are either too elaborated or are difficult to test.

4.6 Package Diagram

We separate our application in a frontend and a backend. The frontend is located in the user's browser and the backend on our server. This setup is called a remote user interface and is a common approach in modern web applications.

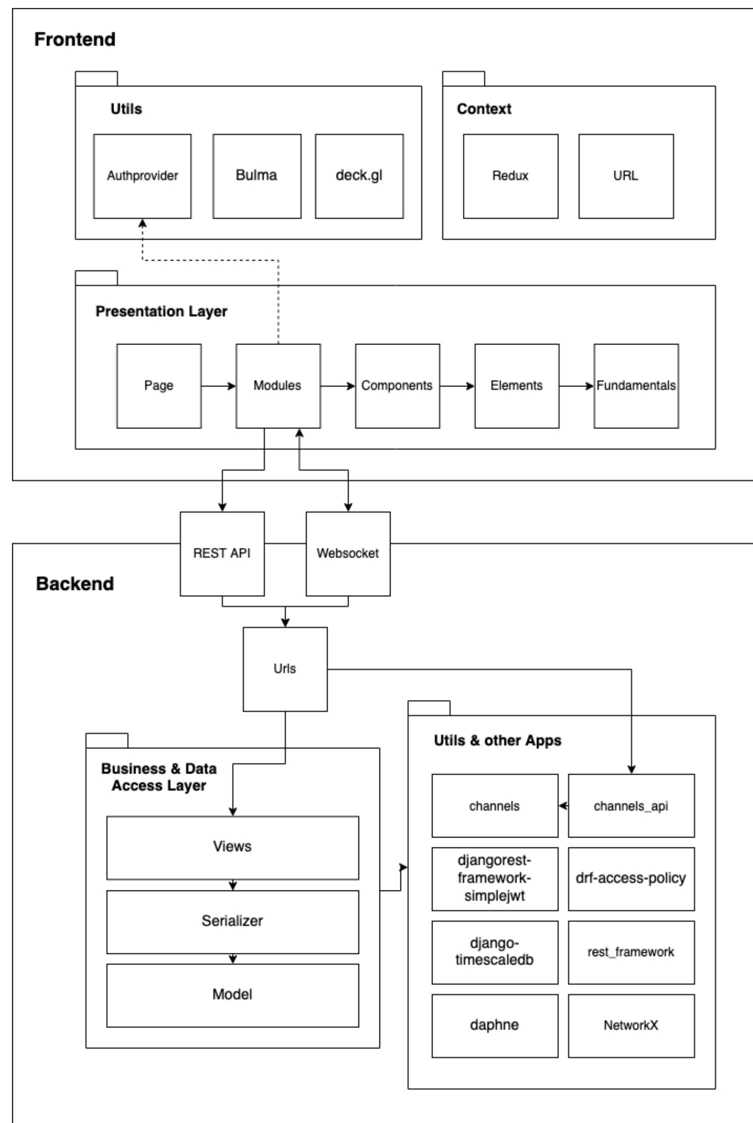

Figure 4.11: Package diagram

4.6.1 Frontend

Context

Context are features which contain an application wide state. It sets the context of the the running application.

Redux We use Redux to manage the application wide state. Redux is stored as a React context and accessible via React Hooks. At the moment the global state contains the global notifications and the global user data.

URL We use the URL for the filter and the sort state on the metrics overview page. Since we can consider the URL as a kind of global state and therefore as a context to the frontend application.

Presentation Layer

React recommends structuring the frontend in a hierarchical form. But how exactly we should do this is not defined. Therefore, we decided to use a common approach to solve this problem.

Pages The physical appearance on the screen. It is the container which contains all other elements, such as modules, components, etc. Paging are mainly relevant for routing, because they are accessible over the URL. Eg. HomePage, MonitoringPage.

Modules Independent code-abstraction with its own state. Apart from the session management, we only communicate via the modules with the backend. One can think of the module as the implementation of a feature in the frontend like DisplayMonitor, ListUsers.

Example file structure of a module:

```
Modules/
    EventList/
        - index.ts
        - EventList.tsx
        - EventListWithData.tsx
        - EventList.test.tsx
        - someStuffonlyThisComponentUses.ts
```

Components Reusable parts which can be used by multiple modules. For instance, a form can be used in multiple modules or a header, which is used on every page.

Elements Those are small parts like buttons or form inputs. They are reused all over the place, in modules, component and pages. You could also call them molecules, because they consist of multiple fundamentals.

Fundamentals Is the smallest possible part of the abstraction. As a result, it is also called Atom. It is something that cannot be separated any further for example Colors, Typography or Spacing.

Utils

The utils are used by the presentation layer either to fetch data, to simplify the design or to render the topology map.

Authprovider We created the Authprovider in order to abstract the fetching logic away in a separate service. The Authprovider is responsible for multiple tasks. JWT management, Error handling and data converting.

Bulma It would be too much effort to create all the visual appealing UI Elements by ourselves, hence we decided to use bulma [9] to get a beautifully aligned design out of the box.

Deck.gl Deck.gl is the replacement for leaflet [31]. Deck.gl is based on WebGL and has a much better performance. It is responsible for the rendering of the map and the drawing on top of the map.

4.6.2 Communication

REST API

Primarily we are using a rest API communication between the frontend and the backend. REST is a structured way to define API endpoints and since we are using the Django rest framework, this is the easiest way. For the user management, possible filter values, metrics and other non-monitoring related tasks, we will use simple API calls.

WebSocket WebSocket connections are used to exchange the topology information between the frontend and backend. Either because the frontend requests the whole topology or because the backend has detected topology changes, after the topology was fetched from the vManage API, that need to be sent to the frontend.

Redis

Redis is used as pub/sub system to store the celery tasks and results. With the introduction of WebSockets, Reddis is also used to create WebSocket groups where messages are sent into and afterwards forwarded to all consumers that are connected to the groups.

4.6.3 Backend

For the backend we will use the Django rest API Framework. Frameworks tend to structure already a big chunk of the architecture on the server side. As a result, for more detailed information we will refer to the Django rest framework documentation [19].

Urls

We need to register all endpoints from all apps in this file. It will decide which call will be handled by which view.

Business & Data Access Layer

Those are the traditionally layers for the django webserver. An http request enters the view goes through the serializer and modifies the database over the model.

Views Views contain the functionality to correctly render the serialized JSON result. The view is also the place where API authorization happens. It contains the business logic which user is allowed to see which resources.

Serializer The serializer translates the JSON into a model and back. It will extensively be used by the view logic.

Model The model works as an OR Mapper. It is the in-code representation of the objects stored in the database. The database is only accessed using the models class.

Utils & other Apps

The Django framework consists of several apps. Some of those apps can run independently from each other and some of them are only working embedded in another app. We will not go too much into detail of each of those apps, because those are given by the framework and are better described in their own documentations [17] [19].

Channels_api and channels As we implemented WebSocket to push the topology updates to the frontend we needed the backend to support the WebSocket protocol. Channel is the most popular and well documented python library for WebSocket.

Rest-framework Is a helper app for our own logic. It enables us to easily create a REST API with JSON.

djangorestframework-simplejwt This plugin is responsible for Json Web Token. It manages the creation and the renewal of tokens. Furthermore it provides a fast and easy interface to verify tokens for each request.

drf-access-policy The drf-access-policy plugin provides us with a easy to use role based access management for the API endpoint.

django-timescaledb With the help of django-timescaledb we are able to create time-series tables and simplify the queries to the Timescale database.

daphne Is our python production webserver which hosts the backend code. It supports HTTP, HTTP2 and WebSocket, and automatically handles the protocol negotiation.

NetworkX NetworkX is a in-memory graph library that allows us to store a topology per customer in an active WebSocket connection.

4.7 Sequence Diagrams

4.7.1 Fetch topology

The sequence diagram below visualizes the process flow of fetching the topology.

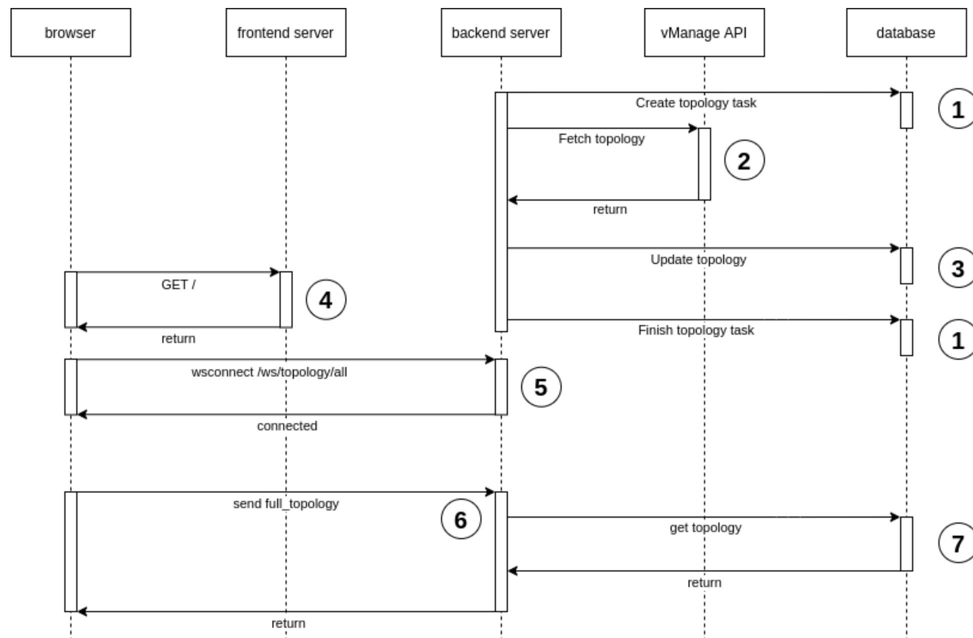

Figure 4.12: Fetch topology sequence diagram

1: Managing topology tasks

A topology task should only run if the previous task execution has either failed or succeeded. To implement this logic we created a database task object that keeps track of the task's state. As a result we access the database before and after the execution of the topology fetch.

2: Fetching the topology

In an effort to keep the topology as close to realtime as possible and to reduce the number of requests to the vManage API, we cache the topology in our own database. To achieve that, we will run a task in a separate container in a regular interval, which fetches the whole topology.

3: Update cached topology

After we fetched the resources, we will format the data in either tunnels or nodes and store them into our database.

4: Initial render

Because we implement a Single Page Application, the initial render will hit the frontend server. The frontend nodejs server will serve all assets like JS, HTML, CSS and images and passes environment variables to the frontend.

5: WebSocket connect

After the frontend was loaded into the browser the Single Page Application connects to the backend container and opens a persistent WebSocket connection. This is a bidirectional con-

nection and will stay open until either the browser or the backend container closes it.

6: Get topology

The browser requests the whole topology by sending a `full_topology` message through the open connection to the backend container. The message optionally contains a filter that is used to filter the topology.

7: Query database

The last step is to query the database for all nodes and tunnels that do match the filter. If no filter was provided the whole topology is returned. After the database query succeeds the backend container returns the topology back to the browser.

4.7.2 Partial topology update

The sequence diagram below visualizes the process flow of fetching the topology and sending partial topology updates to the user.

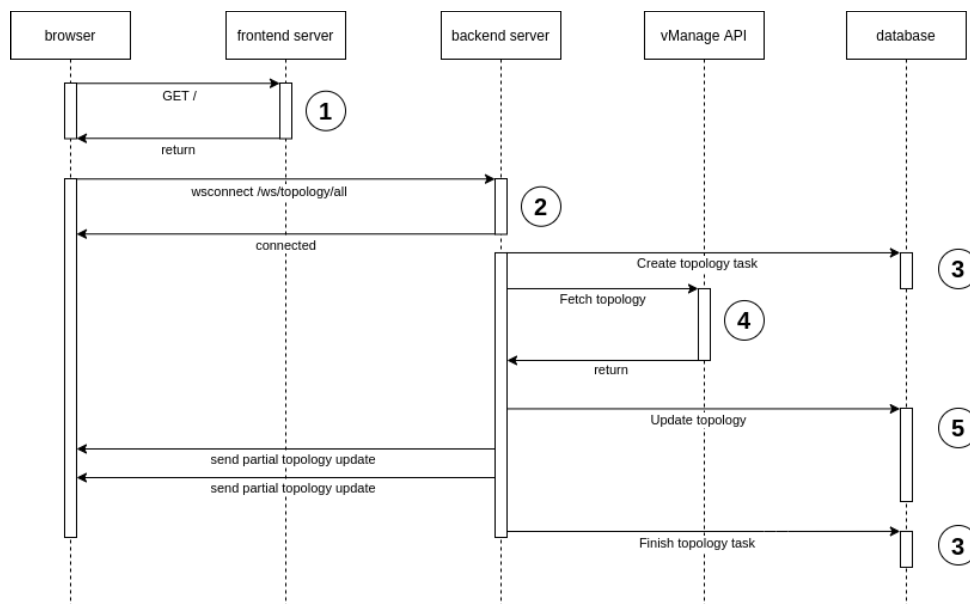

Figure 4.13: Partial topology update sequence diagram

1: Initial render

Because we implement a Single Page Application, the initial render will hit the frontend server. The frontend nodejs server will serve all assets like JS, HTML, CSS and images and passes environment variables to the frontend.

2: WebSocket connect

After the frontend was loaded into the browser the Single Page Application connects to the backend container and opens a persistent WebSocket connection. This is a bidirectional connection and will stay open until either the browser or the backend container closes it.

3: Managing topology tasks

A topology task should only run if the previous task has either failed to execute or succeeded. To implement this logic we created a database task object that keeps track of the task's state. As a result we access the database before and after the execution of the topology fetch.

4: Fetching the topology

We want to keep our topology as close to real time as possible. This is only possible if we cache the topology in our own database. To achieve that, we will run a task in a separate container in a regular interval, which fetches the whole topology.

5: Update cached topology

After we fetched the resources in a task, we will store the tunnels and nodes in our database. If node or tunnel changes were detected it immediately sends a WebSocket message to all connected users and informs them about an add, update or deletion of a node or tunnel. For each changed node and each tunnel a separate message is sent.

4.7.3 Fetching metrics

The sequence diagram below visualizes the process of fetching the IPsec tunnel metrics.

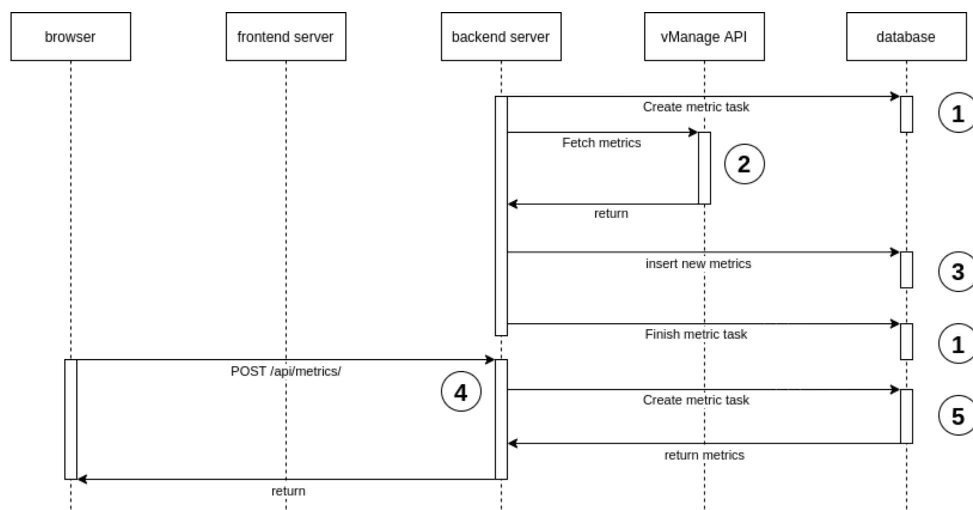

Figure 4.14: Fetching metrics sequence diagram

1: Managing metrics tasks

A metric task should only run if the previous task execution has either failed or succeeded. To implement this logic we created a database task object that keeps track of the task's state. As a result we access the database before and after the execution of the topology fetch.

2: Fetching the metrics

To store metrics for historical analysis and aggregation we decided to store the metrics in our own Timescale database. To achieve this, we query the vManage API and get six metrics measurements per tunnel.

3: Insert new metrics

After we fetched the metrics, we will summarize the measurements and insert them into the Timescale database. To store metrics we created a TimescaleDB time series table that supports storing time series data.

4: Django REST API metrics endpoint

Eventually the frontend accesses the metrics endpoint via HTTP POST method. The backend processes the request, performs authorization and the queries the TimescaleDB to retrieve the metrics. It is to mention that the POST method is used to pass complicated filters and sorting.

5: Get metrics

The backend server will access the required information from the Timescale database and satisfies the request. After the database query succeeds the backend container returns the topology back to the browser.

4.8 Tools & Frameworks

4.8.1 Frontend

In the frontend we decided to keep the tool chain as easy as possible and still have all the functionality we need. For a detailed listing of all technologies used we refer to the `package.json` file in the frontend repository.

Nodejs We use the nodejs server to distribute our frontend assets. It also enables us to fetch the environment variables and pass them down to the frontend. In the future we could implement a so called Server Side rendering to further improve the initial load time of the frontend.

React React is a commonly used frontend UI component library. The rest of our frontend is chosen in a way that they work good together with React. Reacts virtual DOM enables us to keep the re-rendering of the frontend as small as possible.

Typescript For maintainability and static type safety we have chosen typescript.

Redux We use Redux to manage the global frontend state. The major improvement we get with Redux is not the functionality itself, its the debugging view and the good API description.

4.8.2 Backend

Django rest framework Our backend needs to fulfil several tasks. It should be able to communicate with the vManage API. It should respond to the request from the frontend. In addition to that, setting up a user management should be as easy as possible. Django offers far more functionality than that, but we will only need a subset of those.

Channels We use the channels library to establish a WebSocket connection. For each connection we will create a active WebSocket scope. A active WebSocket scope is similar to the active actor in the actor model. Althrough, WebSocket runs in the same container as the django application it does not share any data with it. If the frontend wants to communicate with WebSocket it needs to do it via the WebSocket protocol, like any other request.

4.8.3 CI/CD

For both the frontend and the backend repository CI pipelines were created. The pipline consists of the followind stages:

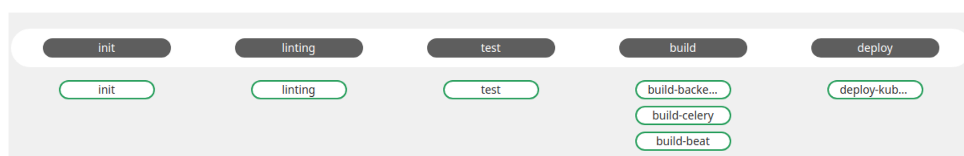

Figure 4.15: GitLab CI/CD pipeline

- **init** Initializes the software which installs the dependencies needed for the build.
- **linting** Perform code linting to ensure the code is formatted according to the selected guidelines.
- **test** Automated tests are executed. Mainly unit tests.

- **build** Builds the docker containers using the dockerfiles.
- **deploy** Deploy the application on the Kubernetes cluster.

Because the build and the deploy job are the most interesting ones they are covered and described further.

Build Jobs The build job uses the dockerfile and the code in the repository to build to docker images. After the docker images are built they are tagged according to the branch name or an explicit git tag. After the tagging the image is pushed to the GitLab container registry.

Deploy Job The deploy job is the last job that runs and only runs if a new tag in the format `vX.Y.Z`. Furthermore it is also possible to completely disable the deployment job by configuring a variable on the repository. The kubeconfig file that is required to authenticate against the Kubernetes cluster can also be defined via a pipeline variable.

1. The Kubeconfig file is also needed in the first step of the deploy job to check the cluster connection.
2. The second step is to clone the Kubernetes infra repository which contains not only the values.yaml file for the helm chart but also the dependencies that are required for the helm deployment.
3. The next step is to clone the helm chart itself from the helm repository.
4. After all dependencies are now present in the pipeline the image tag in the values.yaml file is updated to the git tag which was also used to build and push the images. The updated values.yaml file is then being pushed to the Kubernetes infra repository.
5. The second last step is to to deploy all the dependencies (namespace file, certificate and pullimagesecrets) that were cloned from the Kubernetes infra repository.
6. The last step is the deployment itself which will use the helm chart and the updated values.yaml file and execute the installation on the Kubernetes Cluster.

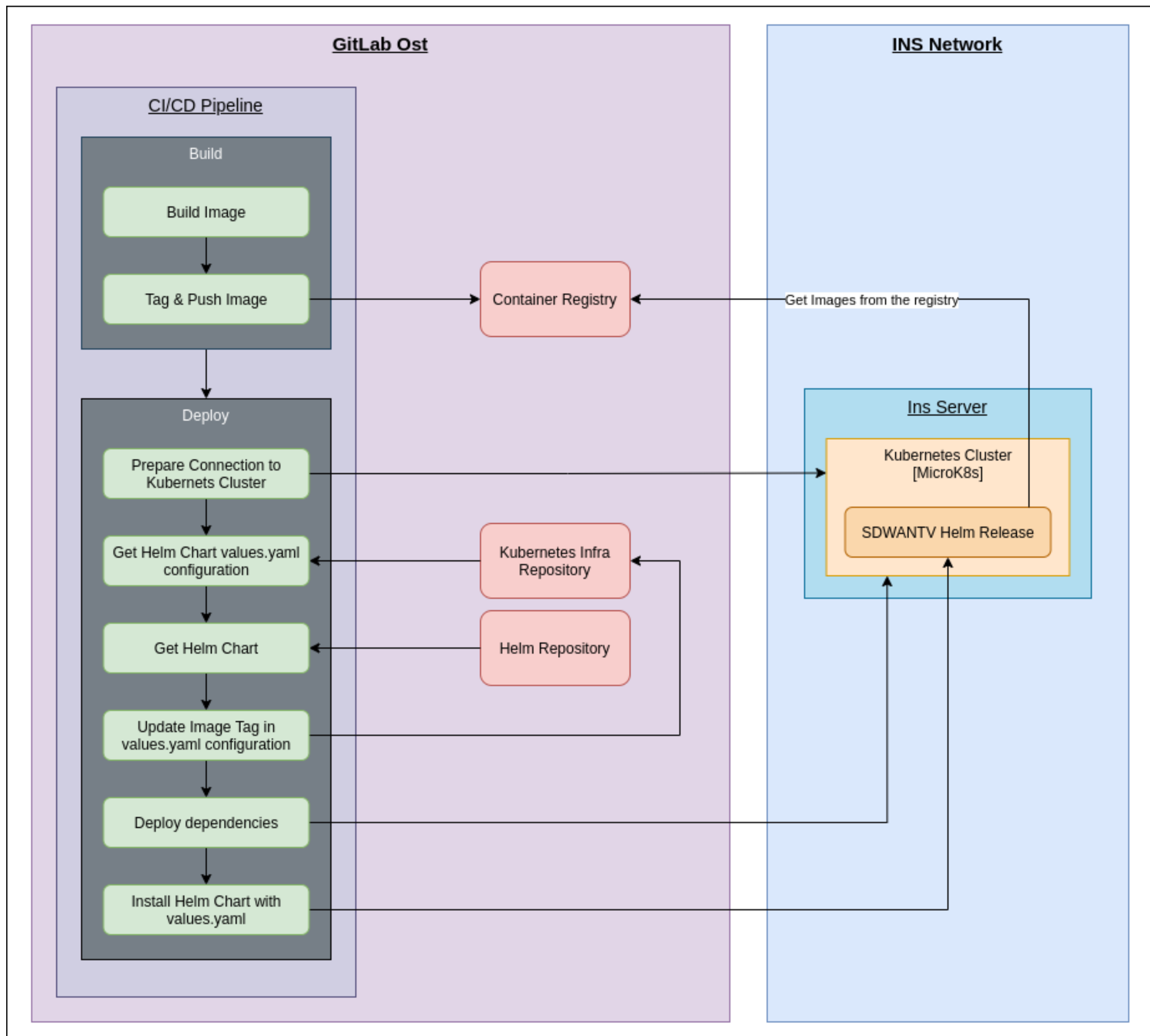

Figure 4.16: GitLab CI/CD pipeline

4.9 UI-Design

4.9.1 Tools

Balsamiq Wirefreame Balsamiq wireframe[6] is a simple tool to create early sketches of an user interface.

Figma For the the user interface design we used figma[23]. Figma is a good easy to use tool to create prototypes for user interfaces. It also provides a proper color and font management. This is important in order to make sure that all fonts and colors used in an application are harmonic.

4.9.2 Design process

At the beginning of every design process a basis to start a discussion is needed. Normally those are rough sketches (wireframes) drawn by hand or created with a tool. After a common consent is made, further refinement can be applied. For this step it is common to use a prototype design tool. A design prototype should already look and feel like the real application but without the complimentary logic.

We decided to create all the mockup for all usecases at the beginning and later design the prototypes in an incremental fashion. This provides us a solid design discussion without doing too much work in advance.

A few screenshots from the various phases are shown below.

Wireframe Already at an early phase of the project we had the wireframe for the metric dashboard. Based on it we discussed which information should the user see in the box header or how should the chart be displayed. The other wireframes are located in the attachment. D.1

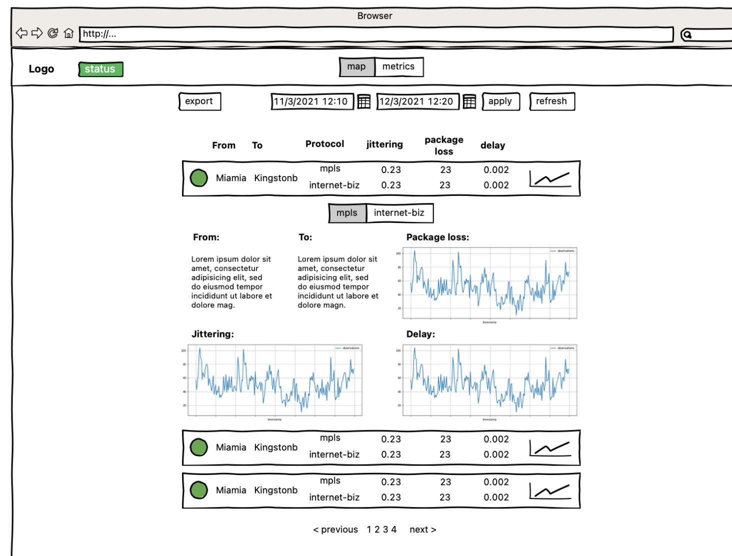

Figure 4.17: UC5 wireframe

Prototype The prototype for the metric dashboard was build on top of the wireframe. It was design-wise congruent with the rest of the application. The other prototype screenshots are located in the appendix D.1.

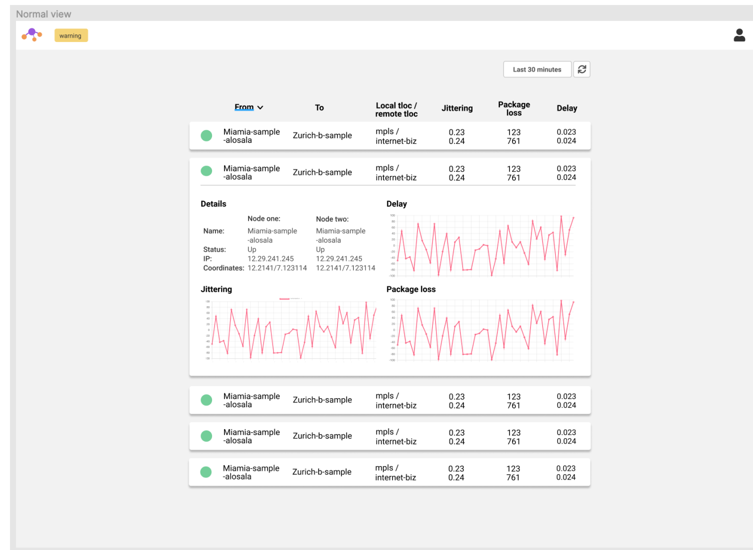

Figure 4.18: UC5 display metrics

Final implementation The frontend engineer now takes the prototype from the designer and implement with code. From the prototype it needs to be clear how the interactions and animations should look like.

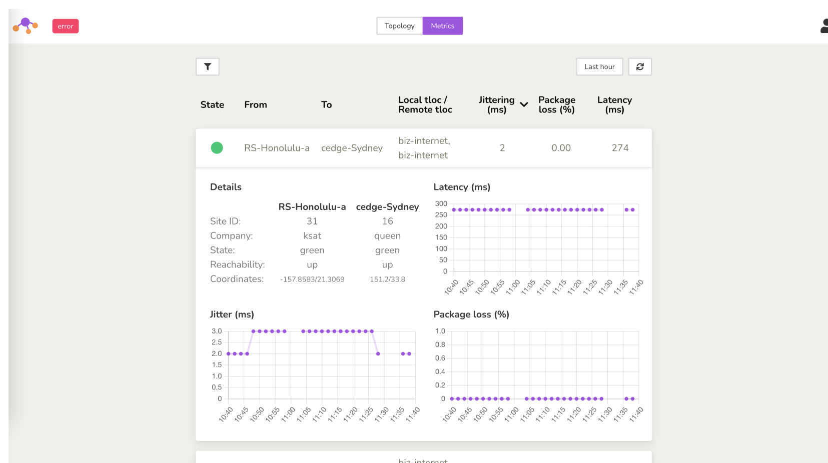

Figure 4.19: UC5 running application screenshot

Chapter 5

Implementation & Testing

5.1 Implementation

As can be seen in the c4 diagrams located in the Architecture chapter 4, we separate the SDWANTV application into several independent components. Because of our professional background and knowledge gained in the cloud solutions module in the spring semester 2020, we decided to use docker containers to separate the components.

This section covers the most noteable parts of the implementation and testing of the SD-WANTV application.

5.1.1 Python Django Backend

Fetching Nodes & Tunnels

The main purpose of the backend is to periodically query the Ciscos vManage API and get all devices (represented as nodes) and IPsec tunnels (represented as tunnels). All fetched nodes and tunnels should than be processed and stored in the database.

To achieve the purpose mentioned above we decided to use Celery and Celery Beat. Celery is a task queue engine which can be used to execute code on workers. Celery Beat is the Celery component which is responsible to periodically schedule tasks. It either uses an interval defined in seconds or a crontab schedule. Because we want to fetch all nodes and tunnels periodically, Celery Beat is the perfect component for us.

We defined a Celery Beat schedule in the `settings.py` file which executes the main task to fetch the nodes and tunnels. The schedule interval can be configured from the environment variable `VMANAGE_TOPOLOGY_SYNC_INTERVALL_SECONDS`.

```
topology_sync_intervall_seconds =
    float(os.environ.get('VMANAGE_TOPOLOGY_SYNC_INTERVALL_SECONDS', 30))

CELERY_BEAT_SCHEDULE = {
    'sync_topology': {
        'task': 'api.tasks.sync_topology',
        "schedule": full_sync_intervall_seconds,
    },
}
```

Listing 5.1: Celery Beat schedule in settings.py

The task `sync_topology` determines if another task is already running. If this is the case the task will be skipped. The `@shared_task` annotation marks this function as a task, so that

Celery is able to detect and execute it on a Celery worker. For reading purposes, logging lines were removed.

```
@shared_task
def sync_topology():
    task_name = 'topology_sync'
    existing_task = existing_tasks(task_name)
    if existing_task:
        no_task_running = not task_running(task_name)
        last_task_older_than_task_intervall = task_older_than_sync_intervall(task_name,
            topology_sync_intervall_seconds)
    task = Task.objects.create(name=task_name)
    task.start()
    if not existing_task:
        import_topology(sync)
        return
    if last_task_older_than_task_intervall:
        if no_task_running:
            import_topology(sync)
        else:
            task.skip('skipped because another sync is running')
    else:
        task.skip('skipped because sync_intervall is not reached yet')
```

Listing 5.2: Sync topology Celery task

If it should fetch nodes and tunnels it will execute the function `import_topology`.

```
def import_topology(task):
    try:
        session = create_session(task)
        token_session = fetch_token(session, task)
        fetched_devices, fetched_nodes = fetch_devices(token_session, task)
        loop = asyncio.new_event_loop()
        fetched_tunnels, errors = loop.run_until_complete(fetch_device_tunnels(token_session,
            fetched_devices, task))
        loop.close()
        if True in errors:
            task = Task.objects.get(id=task.id)
            task.warn()
        process_connected_companies_sites(fetched_tunnels)
        if not task.status == TaskStatus.WARNING:
            cleanup_topology(fetched_tunnels, fetched_nodes)
        create_topology_statistic(task)
        task = Task.objects.get(id=task.id)
        task.end()
    except Exception:
        task = Task.objects.get(id=task.id)
        task.fail(f'An unknown error occured, please investigate log files')
```

Listing 5.3: Fetch topology function

This function performs the following actions:

1. Authenticating against Cisco vManage API using a username and password.
2. Obtaining a token for further communication with the vManage API.
3. Fetching all nodes and store them in the database.
4. Concurrently getting all tunnels of every node and storing them in the database.
5. Add companies and sites to nodes and tunnels.
6. Add a topology statistic entry to the database.

7. Cleanup old nodes and tunnels if needed.

All of the above steps are implemented synchronously except fetching the tunnels for every node. To speed up performance and make it somewhat scalable we used asyncio [5]. Asyncio is a library to write concurrent code using the async/await syntax. Furthermore, we used the AIOHTTP [3] module which provides an asynchronous http client based on asyncio.

The function `fetch_device_tunnels` is executed asynchronously and executes the `get_device_tunnels` function one time for every node in the devices list. It than awaits the response of every concurrent execution and gathers the result together into the `result` variable.

```
async def fetch_device_tunnels(token_session, devices, task):
    result = await asyncio.gather(*[get_device_tunnels(session, device, task) for device in
        devices])
    fetched_tunnels = []
    errors = []
    for device in result:
        fetched_tunnels.append(device['fetched_tunnels'])
        errors.append(device['error'])
    return fetched_tunnels, errors
```

Listing 5.4: Asynchronously fetch tunnels for a device

The function `get_device_tunnels` creates a new aiohttp `ClientSession` and asynchronously gets the tunnels for the device. After the data has been received, it is validated against a JSON schema. If the data is valid the fetched tunnels are processed and saved in the database. For reading purposes some lines were omitted and only the core functionality is documented.

```
async def get_device_tunnels(session, device, task):
    fetched_tunnels = []
    error = False
    device_url = vmanage_url + '/dataservice/device/bfd/state/device?deviceId=' +
        device['system-ip']
    try:
        timeout = aiohttp.ClientTimeout(connect=3, sock_read=10)
        async with aiohttp.ClientSession() as client_session:
            async with client_session.get(device_url, timeout=timeout, headers=session.headers,
                cookies=session.cookies, ssl=False, raise_for_status=False) as response:
                data = await response.json()
                if 'data' in data:
                    validate(instance=data['data'], schema=device_tunnel_schema)
        await client_session.close()
    except aiohttp.ClientConnectionError:
        error = True
    else:
        loop = asyncio.get_event_loop()
        fetched_tunnels = await loop.run_in_executor(None, process_fetched_device_tunnels, data,
            device)
    finally:
        result = {'fetched_tunnels': fetched_tunnels, 'error': error}
    return result
```

Listing 5.5: Asynchronously get tunnels for a device

After the tunnels were successfully fetched from the vManage API they need to be inserted into the database which is done via the `process_fetched_device_tunnels` function. This function contains some business logic because we don't exactly inherit the same tunnel representation as vManage. IPsec tunnels are bidirectional and if there is a tunnel between two nodes (here Bern-A and Main-A) there is exactly one tunnel per transport layer (mpls, biz-internet combination). Having two transport layers results in four tunnels, mpls-mpls,

mpls-biz-internet, biz-internet-mpls and biz-internet-biz-internet. These are the four tunnels that are shown on the picture between Bern-A and Main-A.

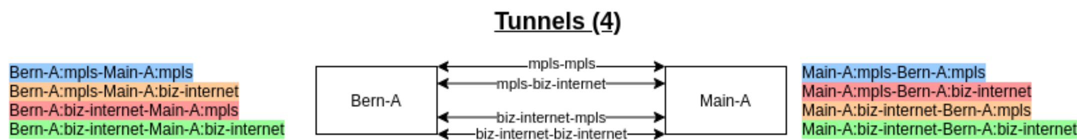

Figure 5.1: Tunnels between Bern-A and Main-A

Because the vManage API does not a have one API endpoint that returns all existing IPsec tunnels, we need to fetch the tunnels per device. This has the result that every tunnels is returned two times. In the example above the four tunnels are returned one time when the tunnels of node Bern-A and one time when the tunnels of node Main-A are queried. If we would ingest the output right away like we received it from vManage we would store every tunnel two times in the database. This is why we need to make sure to only insert a tunnel in one of the two possible combinations and thus only create a new one if none of the two combinations already exist in the database.

```
def process_fetched_device_tunnels(tunnels):
    for tunnel in tunnels['data']:
        try:
            # Get tunnel in normal order
            tunnel = get_and_update_tunnel(node_one=node_one,
                                           node_two=node_two,
                                           tloc_color_one=tloc_color_one,
                                           tloc_color_two=tloc_color_two,
                                           defaults=defaults)
        except Tunnel.DoesNotExist:
            try:
                # Get the tunnel in reversed order
                tunnel = get_and_update_tunnel(node_one=node_two,
                                               node_two=node_one,
                                               tloc_color_one=tloc_color_two,
                                               tloc_color_two=tloc_color_one,
                                               defaults=defaults_reversed)
            except Tunnel.DoesNotExist:
                tunnel = Tunnel(**defaults)
                tunnel.save()
    return fetched_tunnels
```

Listing 5.6: Process returned tunnels

Fetching metrics

Fetching tunnel metrics is designed exactly the same way as fetching nodes and tunnels. First fetch all nodes and after that fetching all metrics for every tunnel of every node. There is just one minor difference. Metrics are not bidirectional like tunnels are but are unidirectional. This means there are always two metrics per tunnel per transport layer, so twice as many as tunnels. The picture shows the metrics between Bern-A and Main-A.

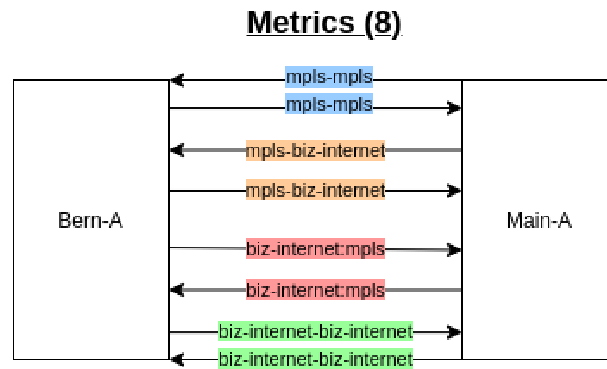

Figure 5.2: Metrics between Bern-A and Main-A

Task objects

To keep track of topology_sync and the metrics_sync tasks we created a task model. The model consists of the following properties:

- name
- start time
- fetch time
- end time
- the duration
- the status of the task
- an optional error message field

With the help of those properties it is possible to manage multiple tasks and make sure only one task with the same name runs at a time.

Each time topology_sync task is started, a task object is created in the database. At the beginning the task object receives the start time and a status of running.

If during the fetching of the data from the vManage API errors occur, the task status will be set to failed and a corresponding error message will be written into the task object.

If no errors occur and the task finishes without any problems, the end time of the task will be updated and the status is set to successful.

If a task starts and another task with the status running is still present in the database the new task will get the status skipped and the task is ended.

Those task objects can also be queried by the frontend under the `api/tasks/` endpoint.

API Documentation

The Django backend is a Rest API and therefore offers several endpoints for interaction. The API documentation provides an overview over the available routes in the backend. OpenAPI was used to document all API endpoints. For our convenience there already is a Django module which brings everything out of the box.

In a first step it is possible to see all available API endpoints.

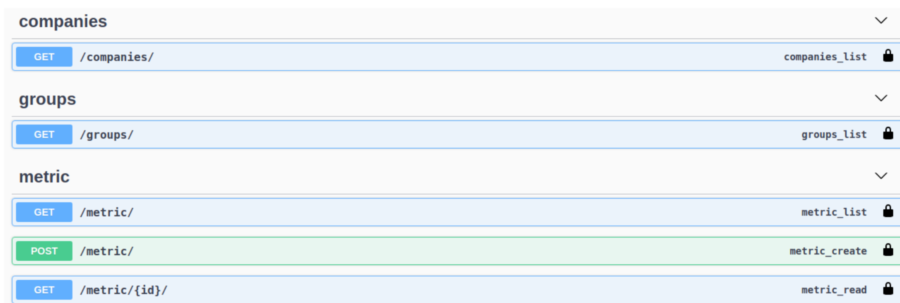

Figure 5.3: OpenAPI Documentation Overview

By selecting a specific endpoint it is possible to also see the required request parameters to interact with this endpoint and the response format the endpoint will produce.

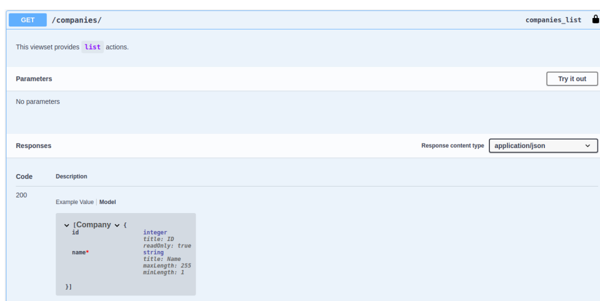

Figure 5.4: OpenAPI Documentation Request

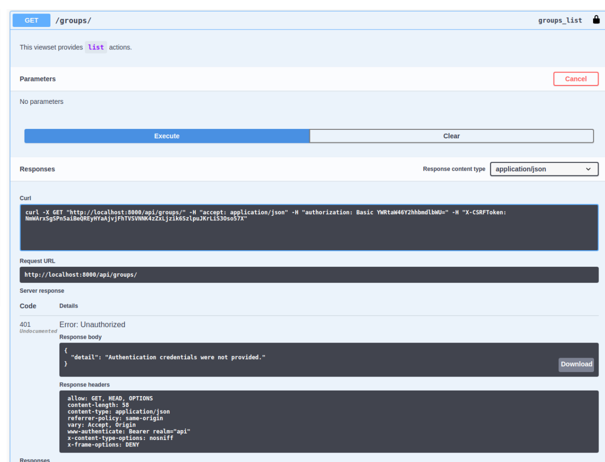

Figure 5.5: OpenAPI Documentation Response

Authentication

Due to the fact that SDWANTV should not be accessible for everybody, we decided to use an authentication mechanism. In a first step we used the TokenAuthentication module of the Django Rest Framework. We quickly realised that this module could not meet our requirements and therefore in a second step switched to Simple JWT [40], the de-facto standard for Django authentication. It uses the JSON Web Token technology commonly used today and the internet standard for creating data.

Simple JWT not only fully satisfies our needs for authentication, but it is simple to use and configure. It requires to install the pip module *djangorestframework-simplejwt* and include it in the Django Rest Framework settings.

```
REST_FRAMEWORK = {
    ...
    'DEFAULT_AUTHENTICATION_CLASSES': (
        ...
        'rest_framework_simplejwt.authentication.JWTAuthentication',
    )
    ...
}
```

Listing 5.7: DRF configuration in settings.py

It furthermore requires to expose a token authentication and a token-refresh endpoint.

```
urlpatterns = [
    path('api/token/', TokenObtainPairView.as_view(), name='token_obtain_pair'),
    path('api/token/refresh/', TokenRefreshView.as_view(), name='token_refresh'),
]
```

Listing 5.8: Include the path for obtaining Tokens

Simple JWT also provides some possibilities to configure the access and refresh token. This can easily be done in the `settings.py` file. We have configured the access token lifetime to one day and the refresh token lifetime to seven days. Furthermore, we have configured the algorithm for signing/verification of the tokens to be `HS512`, the strongest symmetric HMAC algorithm available.

Access tokens can be obtained by a user if he provides the correct email address and password to the `api/token/` endpoint. A request and response using curl could look like the following.

```
curl \
  -X POST \
  -H "Content-Type: application/json" \
  -d '{"email": "admin@sdwantv.com", "password": "changeme"}' \
  http://localhost:8000/api/token/

...
{
  "access":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX3BrIjoxLCJ0b2tlbl90e
           XBlIjoiYWNjZXNzIiwiY29sZF9zdHVmZiI6IuKYgyIsImV4cCI6MTIzNDU2LCJqdGk
           iOiJmZDJmOWQ1ZTFhN2M0MmU4OTQ5MzV1MzYyYmNhOGJjYSJ9.NHlztMGER7UADHZJ
           lxNGOWSi22a2KaYSfd1S-AuT7lU",
  "refresh":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX3BrIjoxLCJ0b2tlbl90
            eXBlIjoicmVmcmVzaCIsImNvbGRfc3R1ZmYiOiLimIMiLCJleHAiOjIzNDU2Nywia
            nRpIjoiZGUxMmY0ZTY3MDY4NDI3ODg5ZjE1YWMyNzcwZGEwNTEifQ.aEoAYkSJjoW
            H1boshQAaTkf8G3yn0kapko6HFRt7Rh4"
}
```

Listing 5.9: Obtain an access token

An access token is returned which afterwards can be used by the user to authenticate to all other endpoints of the API or to the WebSocket connection initialization without providing the email address and password every time.

Authentication for WebSockets using JWT is not part of the module and needed to implemented by us. Luckily there is already a GitHub gist [46] that showed how it can be achieved which just needed some adaption by us to authenticate users also using simplejwt. Unfortunately the Channels WebSocket module does not accept http headers and therefore we needed to pass the JWT as query parameter to the WebSocket endpoint. A WebSocket connection can be opened with a WebSocket client and a valid JWT to the path below.

```
ws://127.0.0.1:8000/ws/topology/all/?token=<very-secure-jwt-token>
```

Listing 5.10: Connect to WebSocket

Authorization

After the user is authenticated to interact with the backend, he still needs to be authorized. The authorization for the backend API happens in two steps.

1. **Endpoint authorization**: This controls if the user is even allowed to access the API endpoint and to perform the provided action against it.

2. **Object level authorization**: If the user is allowed to access the endpoint, which objects is the user allowed to see.

To achieve this fine-granular authorization we made use of the DRF Access Policy module [22]. Using this module requires to write a Class that inherits from the AccessPolicy class and defines which authorizations should be granted for which principal. This class then can be used in a normal Django ViewSet class as an additional entry in the ViewSet's `permission_classes` list. This way it is possible to limit the actions that can be performed on the API endpoint and who can perform them. This way the first of the two authorization steps is fulfilled.

```
class NodeAccessPolicy(AccessPolicy):
    statements = [
        {
            "action": ["retrieve", "list"],
            "principal": ["group:admin", "group:customer"],
            "effect": "allow"
        }
    ]

    @classmethod
    def scope_queryset(cls, request, queryset):
        # return all objects if the user is in the admin group
        if request.user.groups.filter(name='admin').exists():
            return queryset

        # return only the tunnels from the company if the user is in the customer group
        if request.user.groups.filter(name='customer').exists():
            company = request.user.profile.company
            if company is None:
                return queryset.none()
            nodes_company = queryset.filter(company=company)
            nodes_connected = queryset.filter(connected_companies=company)
            return (nodes_company | nodes_connected).distinct()

class NodeViewSet(viewsets.ModelViewSet):
    permission_classes = [permissions.IsAuthenticated, NodeAccessPolicy]
    serializer_class = NodeDetailSerializer

    @property
    def access_policy(self):
        return self.permission_classes[1]

    def get_queryset(self):
        return self.access_policy.scope_queryset(
            self.request, Node.objects.all()
        )
```

Listing 5.11: Authorization using AccessPolicy

Access Policy is also used to perform object level authorization. It requires to add a classmethod `scope_queryset` to the AccessPolicy class and link it in the `get_queryset` of the ViewSet class. This will restrict the queryset which the user is allowed to see. If the user belongs to the admin group, he is allowed to see all the objects. But if he belongs to the customer group, he is only allowed to see objects that belong to his company.

WebSocket

WebSockets are used to send the topology from the backend to the frontend, either if the frontend requests it or if the task sends partial node or tunnel updates. The latter will happen if changes were detected between the local topology and the new topology fetched from the

vManage API. For the implementation of WebSockets the Django Channels [16] module was used.

One WebSocket class called `TopologyConsumer` that inherits from the `AsyncWebsocketConsumer` class was created to receive messages from and send messages to the frontend. To achieve the cloud-native approach and send messages from multiple backend Kubernetes pods it was required to use a Channels layer that acts as a first-in-first-out (FIFO) message queue. Due to the fact that we already had a Redis instance from the Celery module, we stuck with Redis as the Channels layer.

After the frontend got a valid JWT and initiated a WebSocket connetion, the request goes through the authentication middleware. If the token is valid the user object of the requesting user is added to the scope field in the `TopologyConsumer` class else the anonymous user is added. In general, the scope field holds all the request information that were passed when the frontend opens a connection to the WebSocket.

TopologyConsumer The `TopologyConsumer` class contains several methods to mention. The first is the `connect` method. It accesses the scope field to determine the room to which the user wants to connect and gets the user object. In a first step the connection is accepted and the connection joins the room. If in a next step the User is the anonymous user, which means that the authentication in the middleware was unsuccessful, the connection is closed again.

```
async def connect(self):
    self.room_name = self.scope['url_route']['kwargs']['room_name']
    self.room_group_name = f'topology_{self.room_name}'

    # Join room group
    await self.channel_layer.group_add(
        self.room_group_name,
        self.channel_name
    )

    # always accept the connection
    await self.accept()

    # immediately close the connect if the user is the anonymous user, which is the case
    # for unauthenticated users - 4001 (unauthorized)
    if (self.scope["user"].is_anonymous):
        await self.close(code=4001)
        raise DenyConnection("Unauthorized")
```

Listing 5.12: TopologyConsumer connect method

Filters/rooms The question of how we implement filters for the topology, came quite early in the implementation phase. There were two possible solutions.

1. One room per filter and filter combination
2. One room per company and filter as in-memory state

The first solution would have been much cleaner and would have required less logic. With that solution we would not have needed to use the in-memory consumer state. However, this solution would have two major disadvantages. First, it would result in a lot of rooms, as there would be one room for every filter combination. Having 4 companies and 16 sites would already result in 4 * 16 different rooms. If we also add the site-to-site filter that would add another times 16 more possibilities, so 4 * 16 * 16. If new filters are added in the future this solution would not scale anymore. This also leads to the second disadvantage which is the number of messages that would need to be sent. A message would need to be sent in every

room that contains the affected resource, which also would lead to an enormous amount of messages being sent for only one simple change.

This is why we decided to go with the second solution and to have more logic in the consumer. We now have one room per company and the filters are stored in-memory in the consumer class.

Receive method Another important method to mention in the `TopologyConsumer` is the `receive` method. After the frontend has successfully connected to the WebSocket, the frontend sends a request to get the whole topology. This request contains a JSON message with the `type`, `full_topology` and `full_topology` properties and optionally a filter property. The type field defines which method should handle this message. In this case the type property contains the `full_topology` message, the WebSocket contacts the database for all nodes and tunnels matching the filter, serializes them into a GeoJSON format and sends them back to the channels group and therefore to the frontend.

```
{
    'message': 'full_topology',
    'filter': {                 # optional filter
        'company': 2
    }
}
```

Listing 5.13: TopologyConsumer full_topology message

Add/Update/Delete messages There are three more methods worth mentioning that do have the same logic but handle different cases, `topology_add`, `topology_update` and `topology_delete`. As the name already imply these methods are called when one node or tunnel is added, updated or deleted after fetching the topology from the vManage API. All of these methods are called from outside of the consumer. Again the type defines if the `topology_add`, `topology_update` or `topology_delete` method should be exectued in the consumer. The `update_type` defines if the update is for a node or a tunnel, the `geojson_objects` is the object in GeoJSON format and the `serialized_object` is the serialized representation of the object and is used for filtering in the consumer.

```
async_to_sync(channel_layer.group_send)(f"topology_{room_group}", {
    "type": f"topology_{operation}",
    "update_type": update_type,
    "message": geojson_objects,
    "serialized_object": serialized_object
})
```

Listing 5.14: WebSocket add, update, delete messages

The last two methods to mention are the `get_nodes` and `get_tunnels` methods. These are used to query the database according to the defined filter and only return the objects that match the filter. The method loops through all filters that the user has set when requesting the topology. If a filter is set it looks it up in the queries dictionary, executes the lambda function to build the query and appends it to the overall `query_node`. After all filters are concatenated to one query, it is executed on the database and the objects are returned to the caller. This way it is easy to add new filters or adjust the existing ones.

```
self.filter = {}

def get_nodes(self):
    queries = {
        'company_node': lambda company: Q(company=company) | Q(connected_companies=company),
        'site_node': lambda site: Q(site=site) | Q(connected_sites__site_id=site),
        'site_to_site_node': lambda sites: Q(site__in=sites, connected_sites__site_id__in=sites),
    }
    query_node = Q()
    for filter in self.filter:
        query_node &= queries[f'{filter}_node'](self.filter[filter])
    return Node.objects.filter(query_node).distinct().order_by('id')
```

Listing 5.15: TopologyConsumer get_nodes

State management

In a first step we only had two states; the database and the frontend state. Due to the problem determining which connected user needs to receive which message when sending partial topology add, update and delete messages we had to add an intermediate state in the WebSocket consumer class. This now results in three different kind of states.

1. **Database state** The database state holds the whole topology that was retrieved from the vManage API.
2. **Consumer state** The consumer state contains all nodes and tunnels that are relevant for the user, it is determined according to the filters that the user applied and is a subset of the whole topology from the database. It is also used to figure out if a message should be forwarded to a user or not.
3. **Browser state** The browser state contains all the nodes and tunnels the user sees on the topology map. It depends on the consumer state and should be identical.

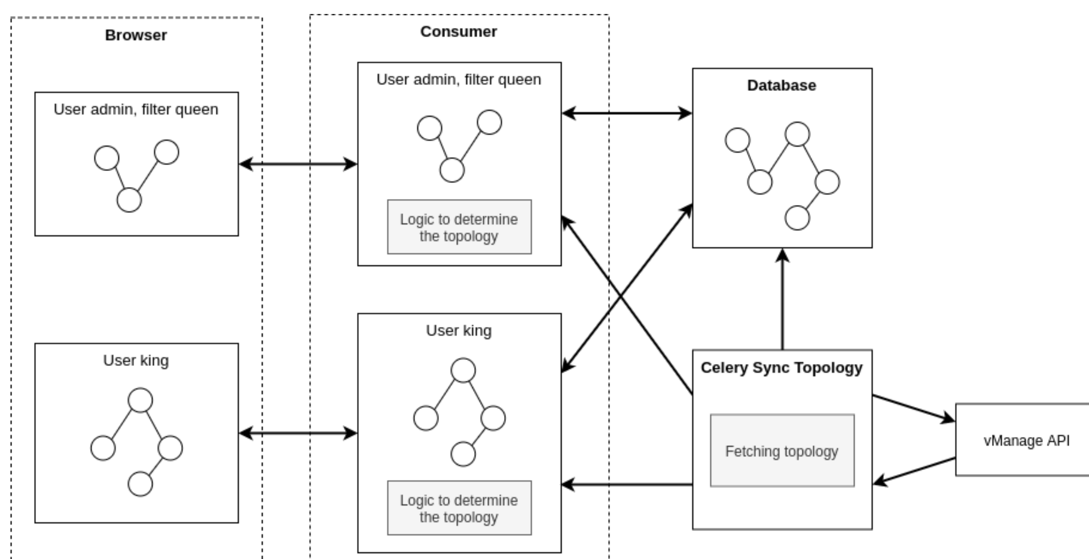

Figure 5.6: States and topology flow

The consumer state is an in-memory state only and is created from the database nodes and tunnels in combination with the filters that were set by the user. It lives as long as the user is connected to the backend and is destroyed as soon as the user disconnects or the backend

closes the connection. For the representation of this state we used a graph module called NetworkX [34]. NetworkX is a module to create an in-memory graph and add nodes and edges to it.

vManage API response validation

While developing the application, the topic of upgrading the vManage to a newer version came up several times. This could mean that the API will change its behaviour or change the response format. To ensure that the response from vManage has the correct format, we decided to use a JSON schema validator. After some research we came up with the Python module `jsonschema`. It provides an easy way to define a json schema and validate the response from the vManage API against it. If the validation of the response fails, a `ValidationError` is raised.

In the code below we defined the JSON schema for the expected response from the `/devices/` endpoint in a separate file and import it. After the `/devices/` endpoint was called and the response has returned, the returned data is validated against the imported JSON schema. If the validation fails, an error will raise and the fetch of the topology aborts. For readability some lines of code were omitted.

```
from jsonschema import validate
from jsonschema.exceptions import ValidationError
from api.json_schemas import device_schema

def get_devices(session, sync):
    device_url = vmanage_url + '/dataservice/device?device-type=vedge'
    try:
        result = session.get(device_url, timeout=(3, 5), verify=False)
        response = result.json()
        if 'data' in response:
            validate(instance=response['data'], schema=device_schema)
    except ValidationError as e:
        raise e
    return response
```

Listing 5.16: vManage response Validation

User Management

The built-in user management [20] solution was one of the major criteria why we have chosen Django in the semester thesis. In the bachelor thesis we have replaced the built-in Django user management with our own user management. It consists of two views; one to register/create new users and one to list and update existing users.

The register and update methods are implemented as POST actions and require the user to always pass all the required parameters as a JSON request payload. When creating a user, the password needs to be passed two times and before the user is created the passwords are validated and accepted if they match.

```
class RegisterSerializer(serializers.ModelSerializer):
    def validate(self, attrs):
        if attrs['password'] != attrs['password2']:
            raise serializers.ValidationError({"password": "Password fields didn't match."})
        return attrs
```

Listing 5.17: User management password validation

TimescaleDB and metrics

Due to the fact that we decided to store a lot of metrics in the database we had the technical requirement to store time series data. This is why we have exchanged the PostgreSQL database with a Timescale database. TimescaleDB makes it easy to efficiently store, query and aggregate time series data.

To create a Timescale time series table instead of a normal relational table it is possible to define a new model class and inherit from the `TimescaleModel`. We used it to create the metrics model.

```
class Metric(TimescaleModel):
    node_from = models.ForeignKey(Node, related_name='node_from', on_delete=models.CASCADE)
    node_to = models.ForeignKey(Node, related_name='node_to', on_delete=models.CASCADE)
    tunnel = models.ForeignKey(Tunnel, related_name='metric', on_delete=models.CASCADE)
    loss = models.DecimalField(max_digits=7, decimal_places=2, default=0)
    latency = models.IntegerField(default=0)
    jitter = models.IntegerField(default=0)
```

Listing 5.18: Metrics model

Because the metrics model is defined as a `TimescaleModel` class it is possible to use Timescale queries to get metrics for a specific time range and tunnel.

```
def query_metrics(tunnel_id, node_from, node_to, ranges, interval, datapoints):
    return Metric.timescale \
        .filter(tunnel_id=tunnel_id, node_from=node_from, node_to=node_to) \
        .filter(time__range=ranges) \
        .time_bucket_gapfill('time', interval, ranges[0], ranges[1], datapoints=datapoints) \
        .annotate(Avg('loss'), Avg('jitter'), Avg('latency'))
```

Listing 5.19: Query TimescaleDB metrics

5.1.2 Frontend

WebGL

In the semester thesis we used Leaflet in combination with OpenStreetMap[36]. At the beginning this worked quite well and was not very complicated to implement. But as we started to render a certain number of resources the leaflet engine came to its limit. The complexity of the topology and over 800 lines and markers were just too much. Especially when a lot of updates arrive from the backend because of some policy changes.

At first, we tried to solve the issue by optimizing the algorithms which are responsible for the rendering. This, however, was not efficient enough. The second idea was to use WebGL to speed up the rendering. We searched online for good solutions that integrate React, Leaflet and WebGL. Although we found some, the existing projects were badly maintained and were not very promising.

Eventually we had the chance to completely replace Leaflet[31] and switch to an alternative, which works with WebGL. After some research the deck.gl[15] technology from Uber was the most suitable one. It was a huge risk to replace the existing topology rendering with a new one, we have never used before. Therefore, we decided to create a prototype to validate the solution and check the advantages.

Figure 5.7: Prototype app for deck.gl

The prototype was a success. With over 20'000 points randomly rendered on the world map each 5 seconds, we had the proof that deck.gl was possible to satisfy our requirements.

The following is a short explanation of what deck.gl exactly is. It is based on luma.gl[32], which comes as well from the Uber kitchen, and is a wrapper around WebGL. WebGL itself implements OpenGL, that is an industry standard for high performance graphics. To use native WebGL, we would have to write in a domain specific WebGL language, which then gets executed directly on the graphic card. Since the graphic card is specialized for executing parallel operation, it is the perfect fit if someone wants to speed up the rendering. We can see the usage of the graphic card in other render intensive environments like gaming or video streaming. With luma.gl we are able to avoid writing the domain specific WebGL part ourselves. We can abstract it away. Eventually, deck.gl combines the power of luma.gl and

connect it with the map of Mapbox[33]. So, what deck.gl is doing at its core, is defining an API to create render layers and synchronize the Mapbox coordinates and the luma.gl coordinates.

The new rendering engine is much faster than the older one. The map zooms smoothly, and we do not have to worry anymore if our software can scale up to more nodes and tunnels. But there are also some downsides. The user now requires a graphic card and a browser that supports WebGL. Luckily, 97.6% of all browser support WebGL.

Rendering the lines

At the first glance line rendering is a simple problem. We just have to draw a line in a certain coordinate system from point A to point B. But what is the result if we want to draw a second line to a third point which happens to be at the exact location as point B? The two lines would overlap, and the viewer would not be able to see the line below.

In our case the user should interact with the tunnels on a map. He should click on the line and a popup with more information should appear. Hence it is not possible to have overlapping lines. The solution to this problem was quite simple, the lines needed to be curved. Unfortunately, we realized a bit too late that deck.gl does not provide us with such a feature. We had do extend the deck.gl library and implement this feature on our own. As the time was limited a quick solution was required, we did not search for the perfect solution, we just needed one that pleases our runtime requirements.

As deck.gl does only provide markers and lines as render objects, we wanted to interpolate a line so that it just looks like a curve. After a quick research the correct formula was found, the quadratic Bézier[7] curve.

$$B(t) = (1-t)^2 * P_0 + 2(1-t) * T * P_1 + t^2 * P_2, t \in [0,1] \tag{5.1}$$

By the characteristics of our algorithm that uses the Bézier curve we have a free variable to set the number of intermediate points. Between those points we perform linear interpolation. Setting the correct value for the variable was a trade of between smoother lines and better performance. We figured out that 20 linear interpolations worked quite well for us.

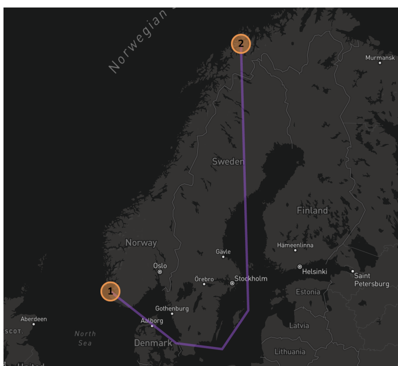

Figure 5.8: Brezier with 5 intermediate points

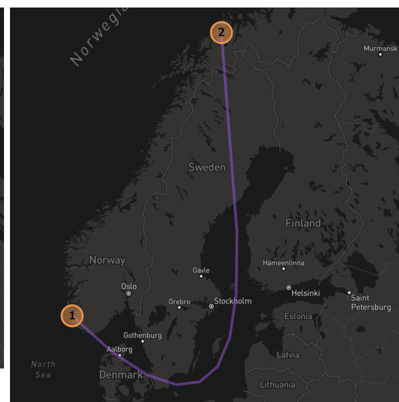

Figure 5.9: Brezier with 10 intermediate points

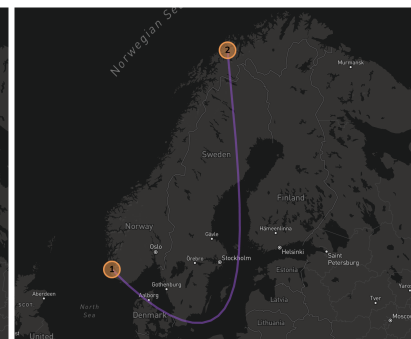

Figure 5.10: Brezier with 20 intermediate points

Right now, we do the brezier curve calculation outside of WebGL and therefore synchronously or at least concurrently in javascript. If we could rewrite a layer of deck.gl manually, we could relocate the calculation onto the graphic card as well to improve the performance further

more. The characteristics of the brezier curve algorithm would allow us to do that, since no global variable are touched during the computation.

Filter over URL

With our filter feature we created something that extends the metric list in a meaningful way. It was not easy to create a filter which supports different kind of filters like tloc color or latency threshold.

In single page applications, like we built with React, one has to rethink the meaning of an URL. The URL is not needed anymore to switch a page because the resources for frontend are loaded only once. The traditional relation between a page and the backend that delivers the page based on the request URL does not exists anymore. With single page applications the URL is just a special type of local state.

What does it mean if we view the URL as a state and not as an address to a page? It means that we can store certain properties of the local application state either in the local storage, as a cookie, in the session or as a query parameter in the URL. The advantage of storing something in the URL shines when the user wants to share his state to another user. He just needs to copy the URL and send it over a communication channel of his choice. We used this characteristic of a single page application to store not only the filter but also the sort of type and direction in the URL. The user can now simply share the metric list to another user and the other one can see the same set of filter and sorting properties, assuming that the other user has the same access level.

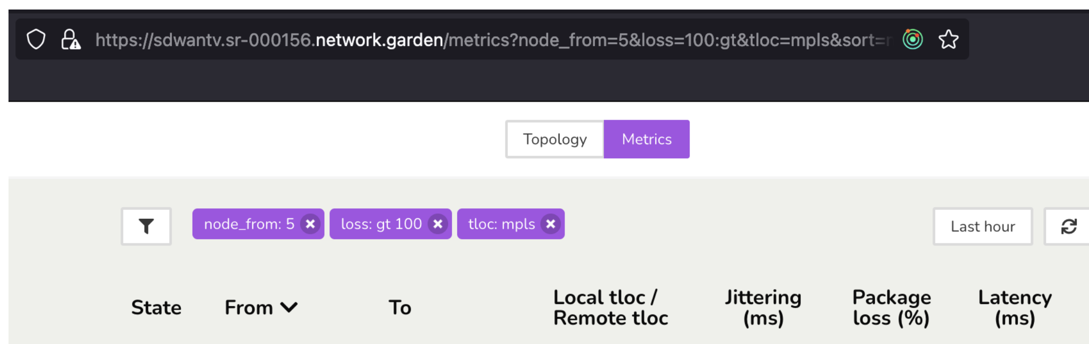

Figure 5.11: Prototype app for deck.gl

Tunnel aggregation

The logic for the tunnel aggregation starts already in the backend. A tunnel which we receive from the backend is actually a bidirectional tunnel which contains the two unidirectional connection with the same tloc color. More about that in the backend implementation section . In the frontend we need to aggregate the tunnels even more. Otherwise too many lines would be rendered and the user would lose the overview.

We use this aggregation function in a reducer. What we essentially do in the function is a mapping from an array into a map datatype. Each tunnel which has the same hash will be added to the value object of the map entry, under `aggregatedTunnels`. To correctly display the aggregated tunnel, we need to keep the coordinates and the reachability state outside of the aggregation core.

```
const aggregateTunnels = (acc: Map<string, RenderTunnelAggregate>, curr: RenderTunnel) => {
  const hash = hashTunnel(curr)
  const existingTunnel = acc.get(hash)

  if (existingTunnel) {
    acc.set(hash, {
      ...existingTunnel, // coordiantes are included here
      properties: {
        reachability:
          existingTunnel.properties.reachability === Reachability.DOWN
            ? Reachability.DOWN
            : curr.properties.reachability,
        aggregatedTunnels:
             existingTunnel.properties.aggregatedTunnels.concat([extractCore(curr)]),
      },
    })
  } else {
    acc.set(hash, {
      ...curr,
      properties: {
        reachability: curr.properties.reachability,
        aggregatedTunnels: [extractCore(curr)],
      },
    })
  }

  return acc
}
```

Listing 5.20: Aggregate function used as callback for reducer

What we actually aggregate becomes clearer if we look at the hash function. We aggregate all tunnels that share the same two coordinates, regardless of the type of the tloc color or the order of the coordinates.

```
export const hashTunnel = (tunnel: RenderTunnel) =>
    tunnel.geometry.coordinates.flat().sort().join()
```

Listing 5.21: Hash function

We eventually get only one connection between two sites despite having multiple nodes in each site. Depending on the property of the topology we can reduce the rendered tunnels up to the factor of four. This design decision greatly improves the user experience and shows our emphasis on a good user centred design. As we can see in the image below, the tunnels are aggregated. The popup contains all the aggregated information and the user can switch between the tunnels in order to see the tunnels specific information.

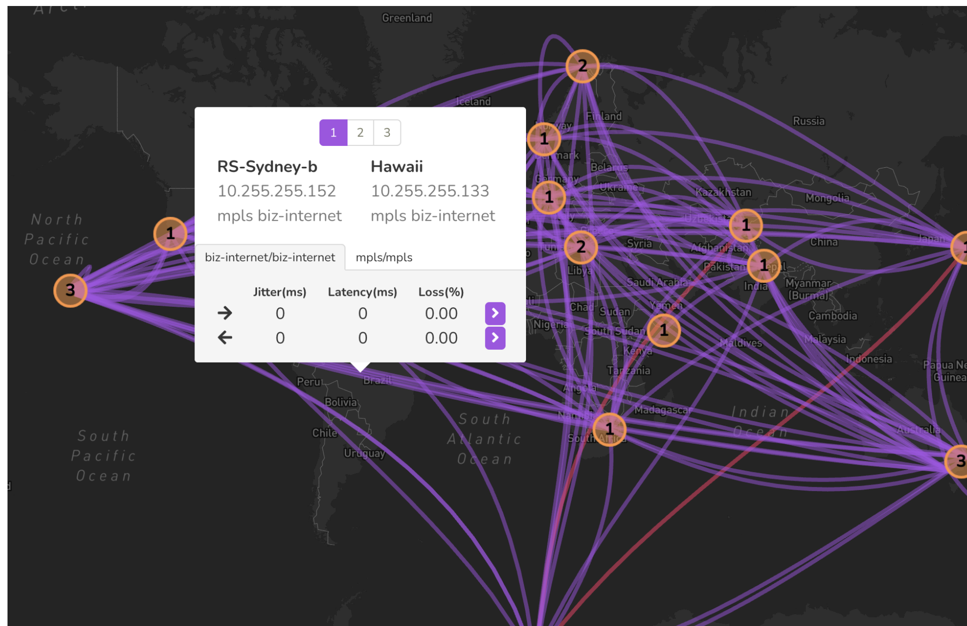

Figure 5.12: Aggregated tunnel popup

Authorization in the frontend

The authorization in a single page application is an interesting discussion, since it works a bit differently as one first might think.

During the initial page request, we will deliver all Javascript, HTML and CSS to the browser at once, regardless of the authentication or authorization of the user. We only differentiate between authorized user or non-authorized user after the user has logged in and stored the respective JWT in the local storage. So, if someone would reverse engineer the single page application, he would quickly figure out the correct local storage properties to simulate a log in. There is no way how we could prevent an adversary to access a certain part of the single page application.

Although we cannot control what happens in the frontend, we can still verify the API request to the backend. For each API request the backend expect a JWT token in the header to authenticate the request. So, the adversary would not get any benefits by just accessing a shallow page without any valuable data. He would need to steal a valid JWT, which is difficult when considering that the token is stored in the local storage of the browser.

Nonetheless, we implemented a simple verification to hide certain information from user without the necessary authorization. This aligns with our non-functional requirements to make the design as simple as possible. A good example of such an authentication barrier in the frontend would be the page access. In the code below we use three different kinds of pages, a normal route, which is accessible to everyone, a private route which is accessible by normal users and administrators and finally the admin route which only the administrators can access.

```
<Route path={router.LOGIN_PAGE} component={LoginPage} />
<PrivateRoute exact path={router.TOPOLOGYVIEW_PAGE} component={TopologyViewPage} />
<AdminRoute exact path={router.USER_OVERVIEW_PAGE} component={UserOverviewPage} /
```

Listing 5.22: Different types of routes in the frontend

If we look at the code of the `AdminRoute` we see that it is essentially a wrapper around a route React component from the react-dom-router library. In this wrapper we check if the user that

accesses the page is logged in, respectively if he has a JWT stored that is still valid, and if he is an administrator, which is just a key value pair in the local storage. When those two preconditions are given the user is allowed to access the page, otherwise he will be redirected to the login page.

```
const AdminRoute = ({ component: Component, ...rest }: AdminRouteProps) => {
  const isLogged = useAuth()
  const admin = useAppSelector(isAdmin)
  return (
    <Route
      {...rest}
      render={(props) =>
        isLogged && admin ? (
          <Component {...props} />
        ) : (
          <Redirect
            to={{
              pathname: routes.TOPOLOGYVIEW_PAGE,
            }}
          />
        )
      }
    />
  )
}
```

Listing 5.23: AdminRoute implementation

5.2 Automated Testing

5.2.1 Unit Tests

Frontend Testing For the frontend testing we are going to use the most common test setup for react applications. The foundation of our test setup builds the *Jest* [28] testing framework. On top of that we use the *react-testing-library* [42], which simplifies the testing with JSX. Similar to the *hot-module-replacement* of *nodejs* [35] we can use the **watch** functionality of jest, which speeds up the development process for the tests. The test will be located next to the code under test.

Backend Testing The backend uses the built-in testing library from Django [18], which is *unittest*. Similar to the frontend we decided to locate the unit tests in the same folder as the code under test. This helps us to find the test suites easily and enables us to clearly differentiate between unit and integration tests.

To test all functionality that depends on existing data in the database a fake topology was created. This fake topology consists of 3 companies, 6 sites, 7 nodes and 15 tunnels over one transport layer. It is a hub-spoke topology which means that there is one service provider (company 1 and the blue nodes) with which every customer (companies 2 and 3 respective green and red nodes) can form tunnels with. This is a typical policy that is used in SD-WAN environments. The pictures demonstrate what topology is visible for an administrator, service provider and for 2 customers.

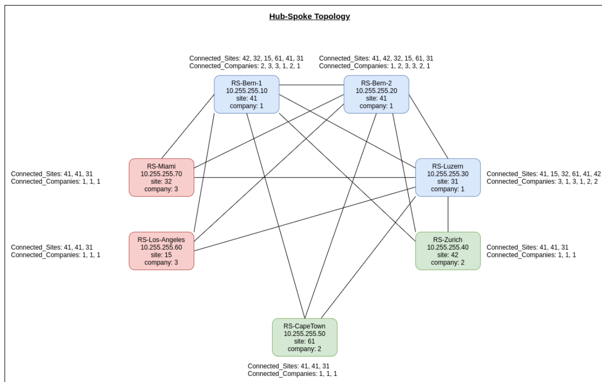

Figure 5.13: Topology admin

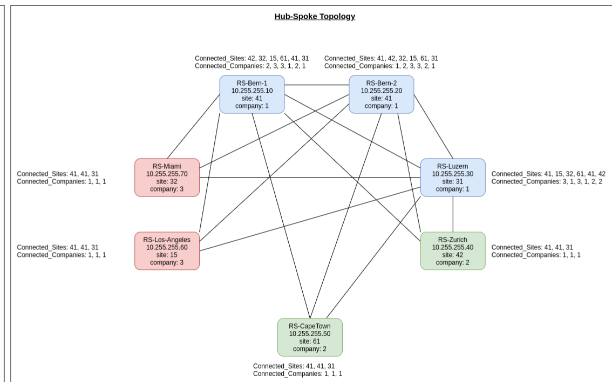

Figure 5.14: Topology service provider

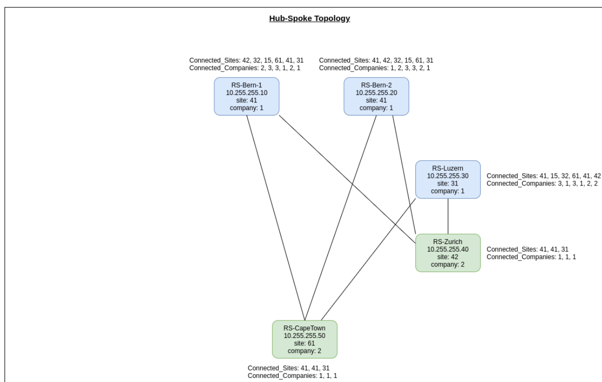

Figure 5.15: Topology customer 2

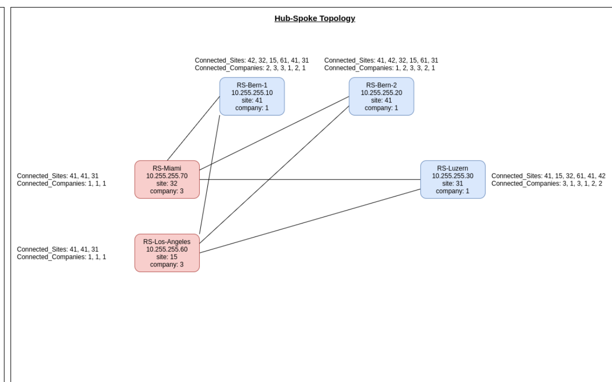

Figure 5.16: Topology customer 3

5.2.2 Integration Tests

Integration tests will be conducted before every merge into the master branch. Those test cases are automated to a reasonable level. The purpose of these tests is to ensure that the interface between the front- and backend is working. For each business service of the API, at least one test case is required. The integration tests are going to be located in a separate folder.

Since the integration test must run on the CI pipeline, we need to create fakes for the database and the vManage API.

5.2.3 Test Coverage

To ensure our code quality, we decided to stick to test coverage thresholds. With the coverage threshold we can let the CI pipeline fail if the committed code does not have the required percentage covered. For the frontend and the backend, we decided to go with following thresholds:

- branches: 75%
- functions 75%
- lines: 75%
- statements: 75%

5.3 Manual Testing

In addition to the unit tests that will be triggered by the CI pipeline every time a push into a branch is done, manual testing is limited to branches with either the prefix `feature-` or `bugfix-`. Whenever a new feature is added or a bug is fixed the affected part of the application will be tested manually on the personal notebook before the branch will be merged into the master branch.

5.3.1 System Tests

System tests will be completed mainly manually, due to the extensive amount of time an automated system test would require. They focus on the scenarios in the Use Cases and requirements. System tests are run before every major release to ensure that the software maturity has reached production readiness.

Non-functional requirements will also mainly be tested on this test level. For each Use Case, at least one test is required (may contain multiple test steps). The testing protocol is produced in the process.

We are going to use a system test specification, which will help us to conduct the tests and write the test protocol down in a structured way. The system test protocol can be found in the appendix B.

5.3.2 Non functional Requirements Tests

At the same time as system tests are being done, we are also doing non functional requirements testing. These tests are carried out manually and should make sure that the non-functional requirements are applied and therefore taken into consideration for the implementation. The non-function requirements test protocol can be found in the appendix C.

Chapter 6

Project Management

6.1 Project organization

The project organisation is made up as follows:

Name	Role	Responsibilities
Prof. Laurent Metzger	Supervisor	Responsible for the thesis and supervision of the team.
Jessica Hilti	Co Supervisor	Assistant of Laurent and also responsible for the thesis.
Prof. Mirko Stocker	Counterreader	Counterreader from the IFS who reviews the thesis.
Marcel Witmer	Expert	External expert from Cisco Systems who will review the thesis.
Thomas Torsteinsen	Industry Partner	Responsible for bringing in his requirements and feedback about the product.
Ali Manzoor	Industry Partner	Responsible for bringing in his requirements and field expertise.
Dominic Gabriel	Developer	Responsible for the architecture as well as the Python backend and supporting with React.
Lars Barmettler	Developer	Responsible for the testing, database and the React frontend and supporting with Python.

Table 6.1: Team Members and Responsibilities

6.2 Project Meetings

The supervisor agreed with us on having a weekly project meeting every Wednesday morning. Whenever possible the industry partner should also participate. If we are in the middle of a project phase and there is no need for a meeting, we would skip the meeting. Due to the COVID-19 pandemic the meetings mostly take place remotely on Microsoft Teams. Meeting minutes are created for every meeting and can be found in the appendix **??**.

Besides the official meeting with the advisor we decided to have a sprint planning and review meeting on Wednesday morning. We also agreed to make a short meeting every Sunday evening to synchronize each other about the status of the project. Of course, these meetings are not necessary every week so we will skip them sometimes as well. Because of the increased communication effort at the beginning of the project we decided to have extra meetings in the inception and elaboration phase whenever needed.

6.3 Process Model

Because we are familiar with Scrum plus from previous lectures, we decided to use this workflow in our thesis as well. Scrum plus is a combination of Scrum and Unified Process. From the Unified Process we are taking the concept of the phases: **Inception**, **Elaboration**, **Construction** and **Transition**. And from Scrum we take the agile development with sprints. The time frame of each phase can be found in the list below. In each project phase we do agile sprints of 2 weeks which allows us to work efficiently and react on unexpected events.

Each sprint is planned in the bi-weekly sprint planning meeting and closed with a sprint review meeting. Sprints always start and finish on Tuesdays.

Phases:

- Inception: 24.02.2021 - 28.02.2021 (4 days)
- Elaboration: 03.03.2021 - 17.03.2021 (2 weeks)
- Construction: 17.03.2021 - 02.06.2021 (11 weeks)
- Transition: 02.06.2021 - 18.06.2021 (2.5 weeks)

Sprints:

- Sprint 1: 22.02.2021 - 10.03.2021
- Sprint 2: 10.03.2021 - 24.03.2021
- Sprint 3: 24.03.2021 - 07.04.2021
- Sprint 4: 07.04.2021 - 21.04.2021
- Sprint 5: 21.04.2021 - 05.05.2021
- Sprint 6: 05.05.2021 - 19.05.2021
- Sprint 7: 19.05.2021 - 02.06.2021
- Sprint 8: 02.06.2021 - 18.06.2021

6.4 Software Development Process

To make the development process as easy as possible, we decided to use the provided GitLab instance by the OST. We created two Git repositories to separate the frontend (React) and backend (Pyhton) code from each other. With this approach we have two completely separated CI pipelines. On every push into the git repositories the CI pipeline automatically runs fully automated tests and will build the docker containers.

For the development of the code we use the GitHub Flow [25] recommendations. This contains the following steps:

- For every issue a dedicated new Git branch is to be created. Issues regarding features have a `feature-` prefix and issues regarding bugfixes will be prefixed with `bugfix-`. The branch name should consist of the number and the title of the corresponding issue.
  - Format: `<type>-<issue number>_<issue title>`
  - Example: `feature-23_implement_login_screen`
- Once all changes for an issue are complete, a merge request will be created. The merge request should be assigned to the other team member.
- The other team member will review the merge request and merge it into the master branch if everything is fine. Otherwise, he will decline the merge request and give his feedback.
- Working directly on the master branch is normally not allowed except for work that is not possible to be completed on other branches efficiently.

We decided to use YouTrack as our issue and time tracking tool. It enables us to manage all epics, tasks and bugs in one place. All resources are tagged for the phase, milestone and component.

YouTrack offers agile scrum boards which we use for the sprint planning and review meetings in order to keep track of the work packages per sprint.

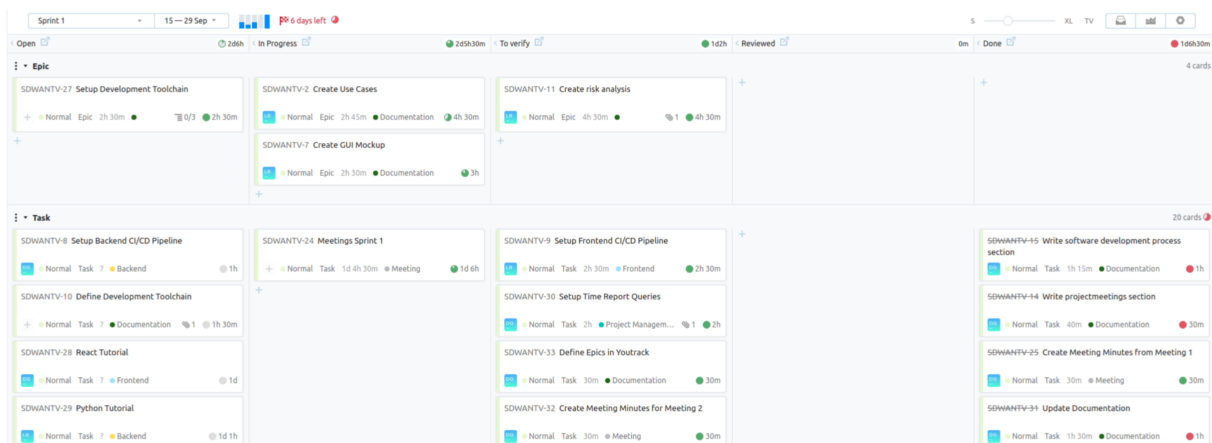

Figure 6.1: YouTrack Sprint Planning

To create our issues we decided to use the 3 issue types *Bug*, *Task* and *Epic*. We use the issue type task to create simple issues and issues that belong to a higher level epic. Furthermore, we use bugs to document problems occuring with the software. To be able to generate time reports we introduce three new custom fields which are required before creating new issues.

The fields *Milestone*, *Phase* and *Component* define some project management topics for every issue.

Figure 6.2: YouTrack Issue Overview

In order to distinguish the activities done we can label our work items with *Development*, *Meeting*, *Documentation*, *Testing* and *Project Management*. This enables us to see at the end of the project how much time we have spent on which activities. The time reports can be found in chapter 7.1.1.

To measure and check the quality of the code we use SonarCube. This makes it possible to see issues in our code and correct them. The code staticstic collected by SonarCube are defined in the project monitoring chapter 7.2.

6.5 Releases

In the process of the Semester Thesis we create four releases.

Nr.	Name	Version	Date	Comment
P1	Prototype	-	17.03.2021	No release for the prototype is planned. It is only used to test the feasibility of the new technologies.
R1	Alpha Release	1.1.0	07.04.2021	
R2	Feature freeze Release	1.4.0	26.05.2021	
R3	Final Release	2.0.0	03.06.2021	

Table 6.2: SDWANTV Releases

Semantic versioning [39] is used for the versioning of the different releases. Semantic versioning uses a 3 number system, *MAJOR.MINOR.PATCH*.

- The major number is increased if changes to the software are done that are incompatible with a previous version.
- The minor number is increased if new features are added that are still compatible with the current release.
- The patch number is increased if changes for bugfixes are introduced.

6.6 Milestones

M1 - End of Inception - 28.02.2021 Project plan is created and risk analysis is created.

M2 - End of Elaboration - 17.03.2021 Requirements are defined, wireframes are designed, Software architecture design defined, C4 and deployment diagrams are generated, domain analysis is completed, kubernetes deployment is set up and the prototype is ready. Furthermore, the vManage API was analysed and documented. After this phase we also have the whole knowledge to start constructing the software.

M3 - Release 1.1.0 - 07.04.2021 WebSockets are introduced, succesfully changed to a new map engine and changed the database engine.

M4 - Feature freeze - 26.05.2021 Feature freeze is done which means that no additional features are going to be added to the software.

M5 - End of Construction - 03.06.2021 All open bugs are fixed and the software is ready for the transition phase.

M6 - Project Closure - 18.06.2021 All documents are finalized and ready to be handed in.

6.7 Project Plan

SD-WAN Topology Viewer	Project Start	CW	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
	22.02.2021	Date	24. Feb.	3. Mar	10. Mar	17. Mar	24. Mar	31. Mar	7. Apr.	14. Apr.	21. Apr.	28. Apr.	5. May	12. May	19. May	26. May	2. Jun	9. Jun
Task		**Date**																
Inception																		
Create Projectplan																		
Risk Analysis																		
M1 - End of Inception		**28.02.2021**																
Elaboration																		
Wireframes																		
Requirements specification																		
vManage API Analysis																		
Domain Model																		
Software Architecture																		
Create Prototype																		
Kubernetes Deployment																		
M2 - End of Elaboration		**17.3.2021**																
Construction																		
Change to WebSocket																		
Change Map Engine																		
M3 - Release 2.0.0		**07.04.2021**																
UseCase Implementation																		
Technical Improvements																		
M4 - Feature freeze		**26.05.2021**																
Bug Fixing																		
Instensive Testing																		
M5 - End of Construction		**03.06.2021**																
Transition																		
Finalize Documentation																		
Presentation																		
M6 - Project Closure		**18.6.2021**																

Figure 6.3: Project Plan

6.8 Risk Analysis

For the risk analysis we created a matrix as an overview. While the events in the white areas are unproblematic, we should reduce the impact of the events in the orange area. In the red area we should avoid the occurrence of an event at all.

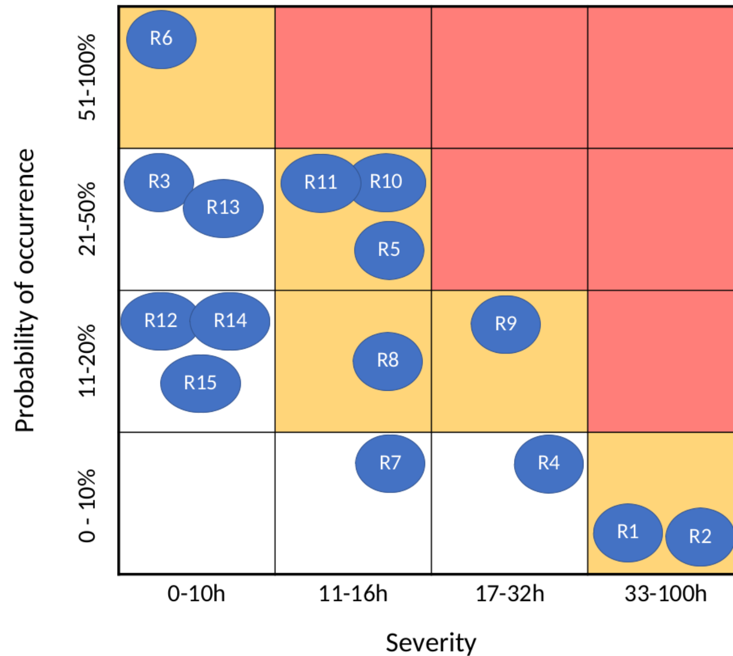

Figure 6.4: Risk analysis matrix

Nr.	Title	Had entered
R1	Interpersonal conflicts	no
R2	Outage of team member	no
R3	Not testable	yes 6.8.1
R4	Customer requirements not met	no
R5	Overstrain of the technical complexity	yes 6.8.2
R6	Lifecycle of dependencies	yes 6.8.3
R7	Lack of security	no
R8	vManage API is not deterministic	no
R9	vManage outage	yes 6.8.4
R10	vManage is not scalable	yes 6.8.5
R11	Faulty topology state	yes 6.8.6
R12	Time format incompatibility	no
R13	No real-time metrics data available	no
R14	Heavy deployment process	no
R15	vManage API version incompatbility	no

Table 6.3: Risk analysis

The full list including the max damage, probability of occurrence, prevention and the behaviour on entry can be found in the attachment E. All risks that had entered are further described below.

6.8.1 R3: Not testable

Some frontend components like the deck.gl are not testable with unit tests. That is why we tested them manually and made sure that it behaves as we expect them to do.

6.8.2 R5: Overstrain of technical complexity

We encountered some difficulties with the technical complexity especially for the WebSocket and deck.gl technology. WebSocket was hard to integrate in our existing system because it made a lot of problems for the inital connection of a new client. We had to ask ourselves some critical questions like how to pass the JWT token and how to handle authentication. With deck.gl it was hard to render the tunnels on the map. It required us to create complex mathematical functions to calculate correct curves on our own. For both of the mentioned problems we needed to invest more time than planned.

6.8.3 R6: Lifecycle of dependencies

This risk was calculated and it was quite obvious that this event would occur. Since the bachelor thesis only lasted one semester we stuck to the same framework and tool versions. Only at the beginning of the bachelor thesis we updated some of the major frameworks to the newest versions and ensured everything still works as expected.

6.8.4 R9: vManage outage

After the the supervisor performed a vManage upgrade on 03.11.2021 vManage was completely down and unavailable. No downgrade or upgrade was possible anymore. A ticket at Cisco was opened by the supervisor. Unfortunately vManage was down for 2 weeks. We mitigated the problem during the two weeks by prioritizing the tasks which did not rely on the availability of vManage. This way we did not lose a lot of time and it only cost us one day. Thankfully the problem could be identified and after two weeks of outage the vManage was up and running again.

6.8.5 R10: vManage is not scalable

Although the vManage API does provide a single endpoint to fetch all the available nodes, there was no endpoint to fetch all the existing IPsec tunnels. One request per node needs to be executed to gather all existing tunnels. With an increasing number of nodes and a strategy to perform requests one after the other, this would not scale. We had to implement Python coroutines and perform these requests in parallel. This was a huge performance improvement that came with some additional complexity tradeoff. However, the switch from sequential to parallel request execution was definitely worth it.

6.8.6 R11: Faulty topology state

Sending incremental add, update and delete messages for every node and tunnel of the topology requires to know which connected user should receive the update messages. Especially when filters are applied, a user can only see a subset of the whole topology. We encountered those problems in the logic which decides who is allowed to see what. This is why we chose to implement an in-memory state using the Python NetworkX library that is created every time a user connects to the WebSocket.

6.9 Logging

SDWANTV consists of mulitple docker containers for which logging need to be configured separately. Overall, the goal is to implement logging based on the best practices from the 12-factor criteria which means that logs should be written to the Stdout stream so they can be collected by docker. For all docker containers the loglevel is adjustable with docker environment parameters and the default log level is *INFO*.

Django Logging Logging in Django is configured in the *settings.py* file. A global log handler is configured which will write the logs to the console (Stdout stream). The loglevel can be set by providing the docker environment parameter *DJANGO_LOG_LEVEL*.

Daphne webserver The Daphne production webserver logs all the http access and error requests. The webserver is configured to also write log messages to the Stdout stream.

Nodejs Webserver Log messages from the frontend webserver are also configured to be sent to the Stdout stream. This webserver does not have a lot of log messages anyway.

React SPA Because the react single page application runs locally in the client's browser log, messages are directly sent to the console of the browser. These logs are not forwarded to a containers Stdout stream.

6.10 Time Report

We manage the time reporting with YouTrack. All epics, tasks and bugs have assigned tags for the phase, milestone and working area (backend, frontend, documentation, meeting). This makes it easy for us to get time reports per milestone, sprint, phase and working area. The time reports can be found in the project monitoring chapter 7.

6.11 Quality Control

Some Quality control measures are briefly listed in the table below. The important measures are described in more detail further down.

Measure	Time Range	Goal
Code reviews	On every merge request	Merge requests into the master branch need to be approved by the other team member. This improves not only the code quality, it also promotes knowledge sharing.
Unit testing	On every push	By building and running the code through the GitLab CI pipeline all tests will be executed and failures will be detected before they get into the master branch.
Integration testing	Before every merge request	Automated tests with a personal computer with both the frontend and backend software components is executed.
Supervisor meetings	On every supervisor meeting	The meetings with the supervisor ensure that the project is on track.
Weekly team-meeting	Weekly on Tuesday	Prevent or correct wrong planning and assign competences and tasks.
Code linting	On every merge request	During every GitLab CI run the code will be linted. This ensures that syntax errors will be detected and are corrected before merging into the master branch.
Definition of done	On every issue	For those issues, whereby the description does not make it implicitly clear as to what needs to be done, we will create a separate definition of done.

Table 6.4: Quality control measures

6.11.1 Linting

We consider the code base to be very important and should be as maintainable as possible. Because of that, we decided to use linting tools with some predefined public rules. The most important ones for the frontend are listed below. The linting will be run during the continuous integration with a `|warnings 0` flag, which prevents the pipeline from running through when we just have one single warning.

- React-app
- Airbnb
- Prettier/@typescript-eslint
- Prettier/recommended
- flake8
- pep8

6.11.2 Definition of Done

The following criteria have to be met before an epic, task or bug is considered finished.

- Successful CI run
- Documentation updated
- Successful run of unit tests

6.11.3 Coding Guidelines

The Code Styleguide for Python is the official PEP8 [37] - Style Guide for Python Code. A tool called `pycodestyle` should be used to observe violations of the style guide and if possible `autopep8` to automatically format code in the PEP8 style.

For the front-end technologies we are using eslint and prettier. The rule set for eslint is based on the AirBnB-Guidlines [4].

6.12 MVP

The MVP will include all the features of the Use Cases UC1 2.3.1, UC2 2.3.2, UC3 defined in the requirements specification chapter 2 and the technical requirements AD1 4.5.1, AD2 4.5.2 and AD4 4.5.4 defined in the architectural decisions chapter 4.5. This will bring the following features:

- SDWANTV can be installed on any Kubernetes distribution by a Helm chart.
- New toptology changes are propagated to the frontend immidiately after they are fetched by using WebSockets.
- Users can be managed through the user management.
- Customers can log on the system and see only their own nodes and tunnels.
- Advanced filtering that limits the displayed number of tunnels to the selected 2 sites.

All other features and functions are not included in the MVP and will only be developed if the time allows it.

Chapter 7

Project monitoring

7.1 Project reporting

In the following sections some project reports for the process of the semester thesis are displayed as charts or metrics.

7.1.1 Working times

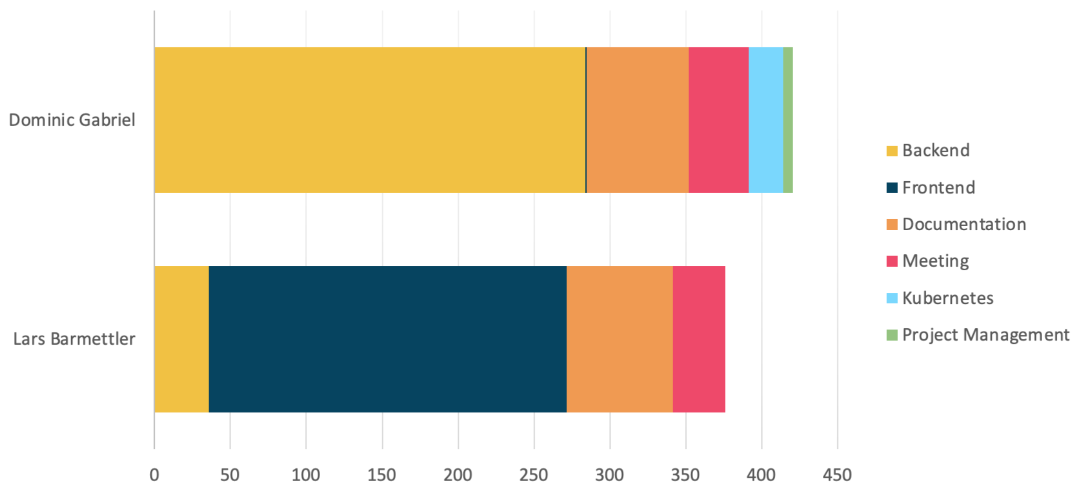

Figure 7.1: Hours spent per component per team member

Member	Time spent
Dominic Gabriel	425h
Lars Barmettler	380h
Total	**805h**

Table 7.1: Working times per team member

7.1.2 Project phases

Time spent per phase

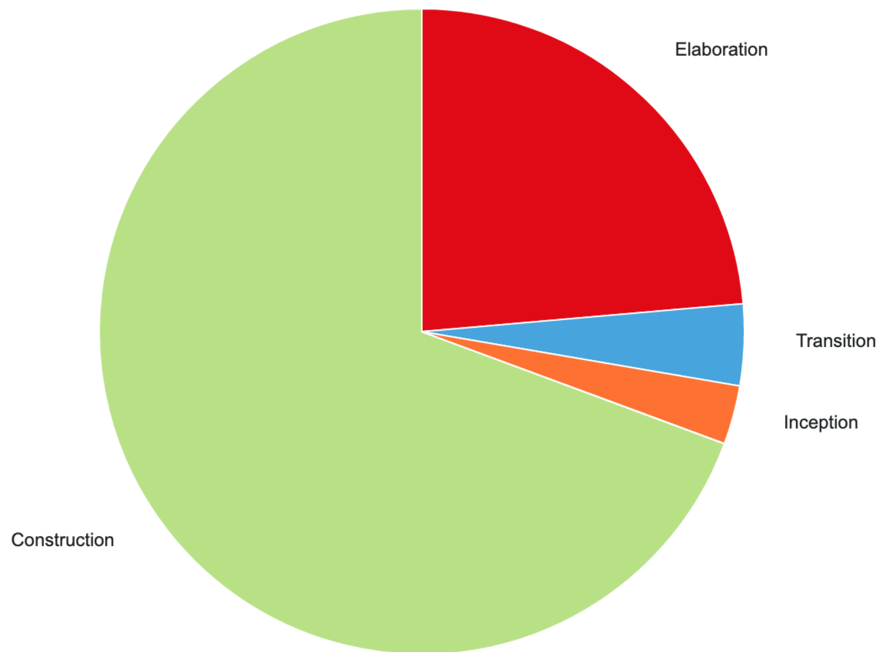

Figure 7.2: Cake diagram time per phase

Phase	Time estimated	Time spent
Inception	35h	33.5h
Elaboration	135h	148h
Construction	530h	577h
Transition	66h	46h

Table 7.2: Time spent per project phase

Issues per phase

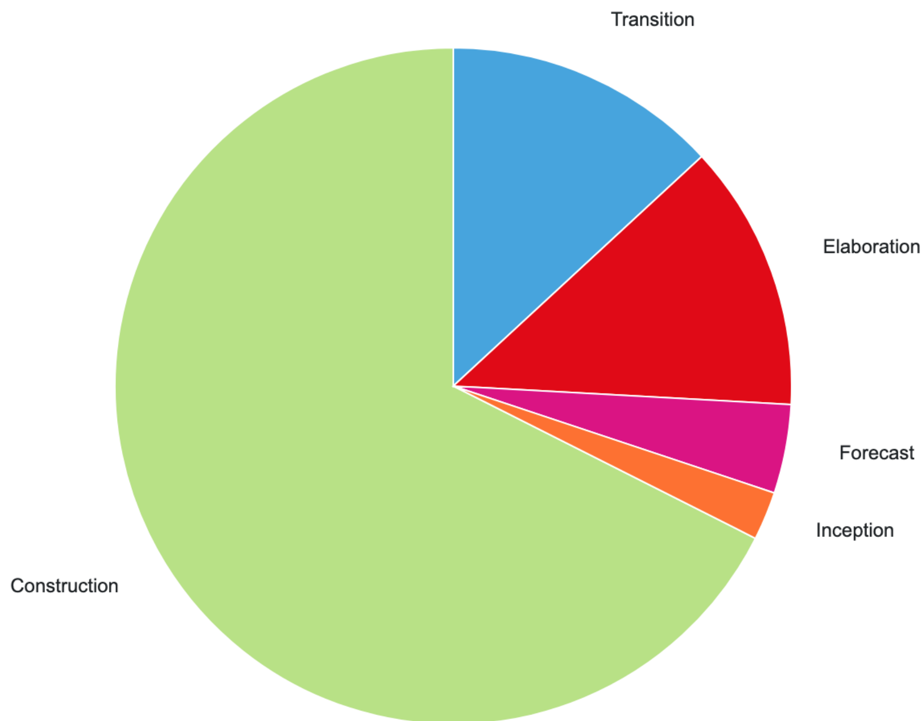

Figure 7.3: Cake diagram issues per phase

Phase	Number of issues
Inception	6
Elaboration	33
Construction	175
Transition	34
Forecast	11

Table 7.3: Issues per phase

7.1.3 Issues per task types

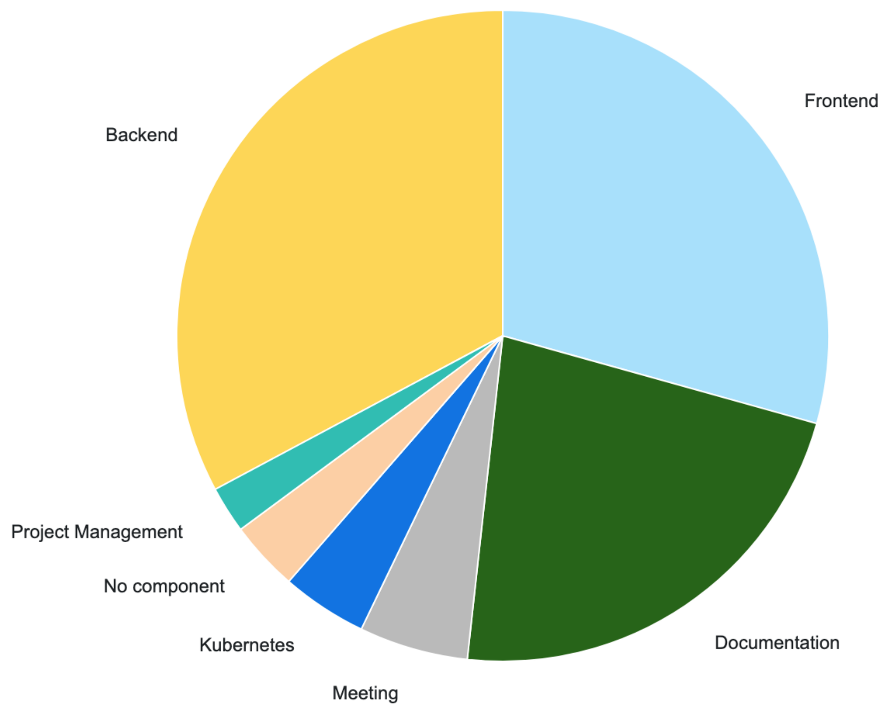

Figure 7.4: Cake diagram issue count per type

Type	Number of issues
Backend	85
Frontend	76
Documentation	58
Meetings	14
Kubernetes	11
Project Management	6
No type	9

Table 7.4: Issues per task type

7.1.4 Milestones

Time spend per milestone

Type	Time estimated	Time spent
M1	59h	56h
M2	113h	126h
M3	110h	144h
M4	329h	353h
M5	85h	76h
M6	71h	49h

Table 7.5: Time spent per milestone

Issues per Milestone

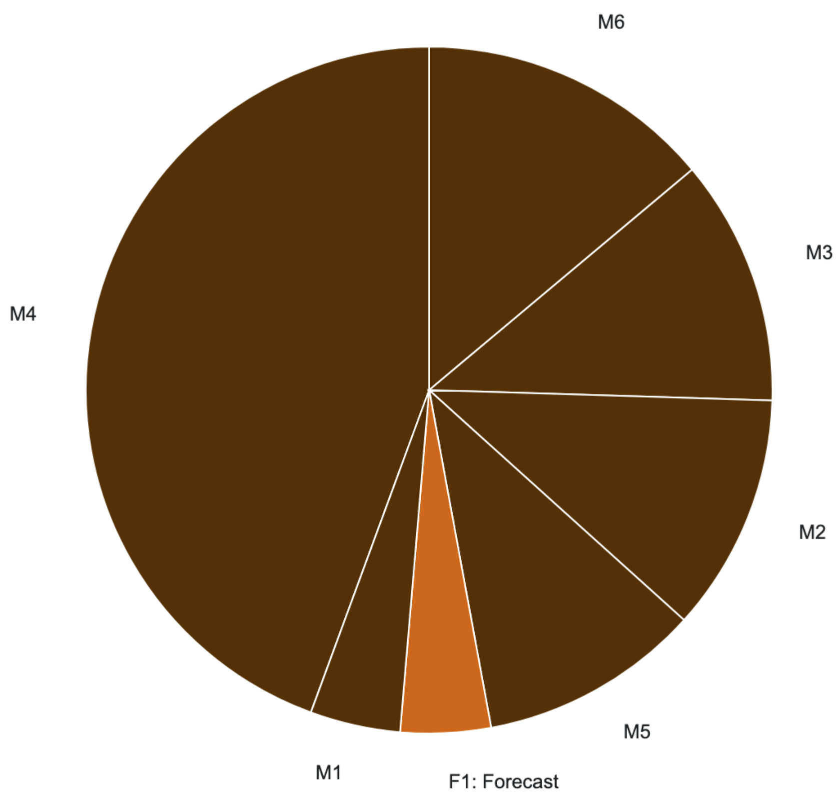

Figure 7.5: Cake diagram issues per milestone

Milestone	Number of issues
M1	11
M2	29
M3	30
M4	115
M5	27
M6	36
F1	11

Table 7.6: Issues per milestone

7.2 Code statistics

The code metrics are obtained with Sonarqube [41] and GitLab repository statistics.

Tier/Metric	Backend	Frontend	Helm & K8s Infra	Total
Lines of Code (with tests)	4700	6587	1729 (configuration)	13016
Statements	3503	1708	-	5211
Comments (%)	10.8%	1.1%	-	5.95% avg.
Commits	183	178	56	417
Testing coverage	70%	81%	-	75.5% avg.
Docker image size	41.18 MiB	43.50 MiB	-	84.68 MiB
Cyclomatic complexity	633	962	-	1595
LoC Docker configuration	179	28	-	207

Table 7.7: Code statistics

Part III

Appendix

Appendix A

User Manual

The user manual provides information about how to install and configure SDWANTV. SDWANTV can be installed on every Kubernetes [30] compatible container distribution. Furthermore it describes some operational tasks that may be necessary.

A.1 Requirements

The requirements to get SDWANTV up and running are the following:

- A Kubernetes cluster
- Kubectl binary
- Helm 3 binary
- vManage API user and password

A.2 Deployment

To deploy SDWANTV a notebook with access to the Kubernetes cluster, kubectl and helm installed locally is required. Afterwards it is needed to have access to the SDWANTV Helm chart repository [38]. The installation process is describe below but also in the `README.md` of the Helm chart repository. All files paths are relative to the directory where the `Chart.yaml` files is located.

First of all create a namespace where to install SDWANTV into.

```
kubectl create namespace sdwantv
```

Clone the SDWANTV Helm chart. Because it is a username and password protected registry the user needs to be authenticated.

```
git clone ssh://git@gitlab.ost.ch:45022/sdwantv/helm.git
```

Execute this step only if a new TimescaleDB instance should be deployed. If you want to use an already existing TimescaleDB instance this step can be skipped. After the namespace was created the dependency files need to be applied. These files create a user called `sdwantv`, create a database called `sdwantv` with the password `sdwanTV2020` after the TimescaleDB instances were installed.

```
kubectl apply -f dependencies/timescaledb-tls-secret.yaml --namespace sdwantv
kubectl apply -f dependencies/timescaledb-credentials-secret.yaml --namespace sdwantv
kubectl apply -f dependencies/configmap-extra-db.yaml --namespace sdwantv
kubectl apply -f dependencies/configmap-extra-db-pw.yaml --namespace sdwantv
```

After the dependencies have been installed it is time to deploy the SDWANTV Helm Chart. Make sure to first adjust the `values.yaml` that should be used for the installation. For more information about the possible configurations, read the next section.

```
helm install sdwantv . --namespace sdwantv
```

A.3 Deployment configuration - values.yaml

The deployment is easily adjustable via the `values.yaml` file that is the source for all installation parameters. All available configuration values are documented in the `README.md` of the Helm chart repository [38]. Some are further described here.

Access private image registry

If the docker images are located in a username and password protected registry, it is required to create a `Secret` resource to use for the `imagePullSecret` option and apply it before the installation of the chart. How to create a `Secret` from a docker config file can be read in this Kubernetes documentation [14].

```
apiVersion: v1
kind: Secret
metadata:
  name: gitlab-auth
stringData:
  .dockerconfigjson: {"auths":{"registry.gitlab.ost.ch:45023":{"username":"gitlab-registry",
                     "password":"very-secure-password","email":"email@ost.ch",
                     "auth":"Z2l0bGFiLXJlZ2lzdHJ5OjFUazYtTXRpXzlzczQ5V2tVd2Fz"}}}
type: kubernetes.io/dockerconfigjson
```

After the secret was applied configure it in the `values.yaml` file for all the components frontend, backend, celery, beat.

```
imagePullSecrets:
  - name: gitlab-auth
```

MapBox access token

In order to get a mapbox accesst token you need to go to the mapbox website and create an account. After the account is created you can see your personal access token directly on the dashboard page. Copy the token and put it in your adjusted `values.yaml` file under the frontend section.

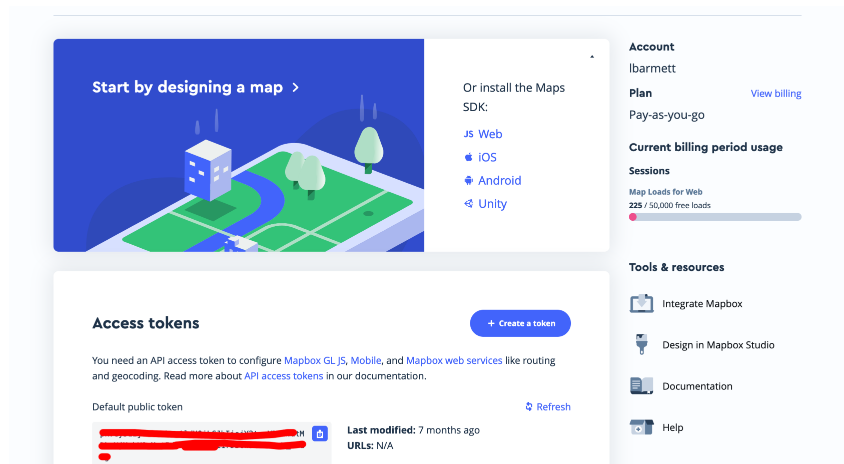

Figure A.1: Mapbox dashboard

If the monthly request limit exceed the 50'000, it is possible to upgrade.

Ingress

If ingress is enabled in the `values.yaml` it will automatically create ingress resources. It is important to note that in order to make WebSockets work it is required to understand how the clusters ingress controller does handle WebSocket connections. For the Traefik proxy no special configuration is required to allow WebSocket connections. For the Nginx ingress controller it is required to pass the following annotations to the WebSocket ingress.

```
ingress:
  annotationsWebsocket:
    nginx.ingress.kubernetes.io/proxy-read-timeout: "3600"
    nginx.ingress.kubernetes.io/proxy-send-timeout: "3600"
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
```

TLS certificates

There are a lot of methods how to create and use TLS with Kubernetes clusters, for example using the companies wildcard certificates or using the Cert-Manager and Let's encrypt generated certificates. If TLS certificates should be used is configurable in the ingress section in the `values.yaml`.

```
ingress:
  tls:
   - secretName: sdwantv-tls-cert
     hosts:
       - sdwantv.mycluster.company.com
```

A.4 Operational tasks

Cleanup failed tasks

If for any reason the status batch in the left upper corner in the frontend shows up some errors for a longer time it might be that there is a task that is still in a running state and preventing other tasks from starting. If this happens it is required to exec into one of the backend pods and exectue the cleanup command that will fix this unwanted state and delete all tasks with a running state.

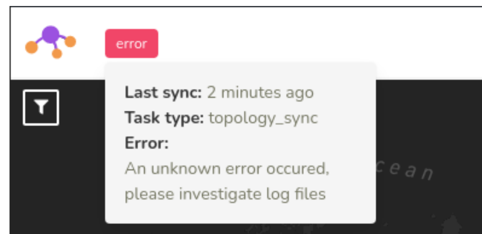

Figure A.2: Failed sync status

```
$ kubectl get pods -n sdwantv | grep backend
sdwantv-backend-569dbc9f8d-5mj5g 1/1    Running  0          5d23h
sdwantv-backend-569dbc9f8d-7hq5q 1/1    Running  0          5d23h

kubectl exec -it sdwantv-backend-569dbc9f8d-5mj5g -n sdwantv -- bash
bash-5.1$
bash-5.1$ python manage.py cleanup_tasks
Info: Using postgresql Database on Host sdwantv_db
Info: Using Celery broker redis://sdwantv_redis:6379/0

Successfully cleaned up running tasks
```

After the tasks were cleaned up, everything should work again and the sync status should turn green again and show up that is recently synced.

Backend healthcheck

The backend container features a healthcheck endpoint that can be accessed to check if the backend components are running as expected.

```
https://sdwantv.mycluster.company.com/api/ht
```

This displays the healthcheck in a html readable presentation and is good for using in browsers.

System status

Service	Status	Time Taken
✔ Cache backend: default	working	0.0101 seconds
✔ DatabaseBackend	working	0.0212 seconds
✔ DefaultFileStorageHealthCheck	working	0.0101 seconds
✔ MigrationsHealthCheck	working	0.0313 seconds
✔ RedisHealthCheck	working	0.0046 seconds

Figure A.3: HTML friendly backend healthcheck

Or to see the status of the backend components in a json format.

```
https://sdwantv.mycluster.company.com/api/ht?format=json
```

This produces the json output below.


```
{
    "Cache backend: default": "working",
    "DatabaseBackend": "working",
    "DefaultFileStorageHealthCheck": "working",
    "MigrationsHealthCheck": "working",
    "RedisHealthCheck": "working"
}
```


Default admin user

By default a user with the username `admin` and the password `changeme` exists. This user has administrative rights and is able to login to the admin panel.

A.5 Update

To update the installation of SDWANTV because somehting has been adjusted in the `values.yaml` file simply run the update command.

```
helm upgrade sdwantv . --namespace sdwantv
```

A.6 Termination

To uninstall SDWANTV simply uninstall the release of the SDWANTV helm chart.

```
helm uninstall sdwantv --namespace sdwantv
```

If the data is not needed anymore it the persistentvolumeclaims also need to be deleted manually using kubectl. Do this for every volume claim in the namespace.

```
kubectl delete pvc -n sdwantv <pvc-name>
```

A.7 vManage adjustments

To tell SDWANTV which node belongs to which company, set the `device groups` field on every node. By default nodes without any `device groups` configured belong to the `no_groups` company. Our system is only able to handle one group per device.

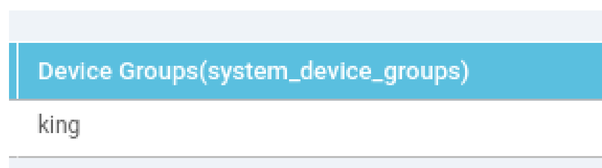

Figure A.4: vManage device groups

Appendix B

Systemtest protocol

The system test protocol covers all tests that are relevant for the Use Cases. They test the whole application and show that the application is doing what it is intended to do. The test results can be found further below and are linked in the result column in the table.

The requirements for performing these tests are to have an instance of the SDWANTV application running. How to setup an instance of the SDWANTV can be found in the installation guide A.

UseCase	**Implemented**	**Result**	**Status**
UC1: Customer Portal	yes	result B.1	passed
UC2: Customer management	yes	result B.2	passed
UC2.1: Update customer	yes	result B.5	passed
UC2.2: Password change enforcement	no	no result	not tested
UC3: Add filter between two sites	yes	result B.6	passed
UC4: Acknowledge down tunnels	no	no result	not tested
UC5: Display tunnel metrics	yes	result B.7	passed
UC5.1: Specify historical metrics time range	yes	result B.9	passed
UC5.2: Export metrics to pdf	no	no result	not tested
UC5.3: Filtering metrics	yes	result B.10	passed
UC5.4: Sorting metrics	yes	result B.12	partly passed
UC6, UC7, UC7.1, UC8	no	no result	not tested
AD1: Tunnel aggregation	yes	result B.13	passed

Table B.1: Systemtest protocol UseCases

Further, the main goal of SDWANTV is to show a live monitoring of the SD-WAN topology. This was tested in separate tests that are not part of any UseCase.

Scenario	Result	Status
S1: Monitoring topology	result B.14	passed
S2: View node information	result B.19	passed
S3: Display connection metrics	result B.22	passed
S4: Apply customer filter	result B.23	passed
S5: Apply site filter	result B.24	passed

Table B.2: Systemtest protocol further scenarios

B.1 UC1: Customer Portal

The customer is able to access the topology and metrics overview.

Initial state

The customer is not logged in and does not have access to the topology and metrics.

Test procedure

The customer receives the login credentials from an SDWANTV administrator and is able to login. After the login SDWANTV by default displays the topology overview with all nodes and tunnels of the customers company. The customer can access the metrics page from the metrics button in the page header and get insights of the metrics of all tunnels that he is allowed to see.

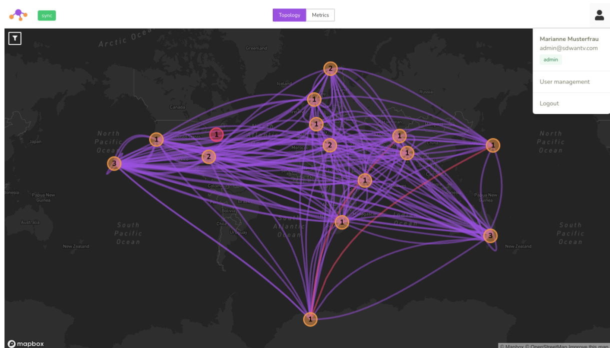

Figure B.1: Topology admin

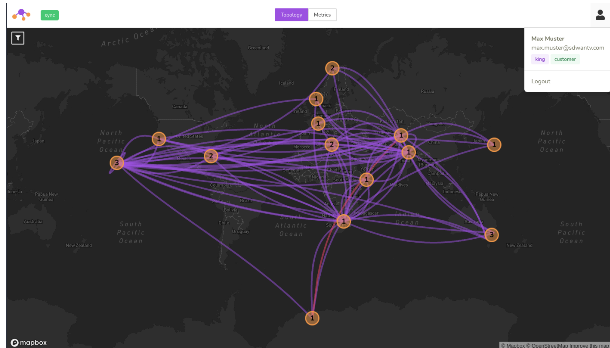

Figure B.2: Topology customer

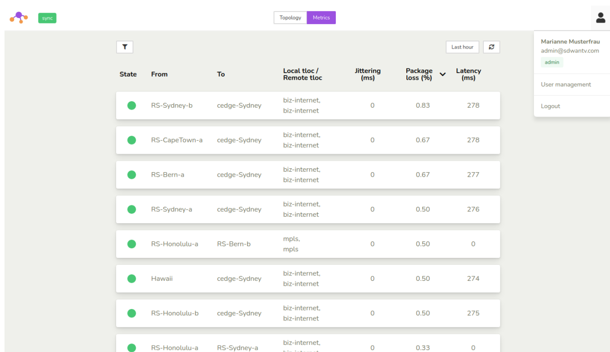

Figure B.3: Metrics admin

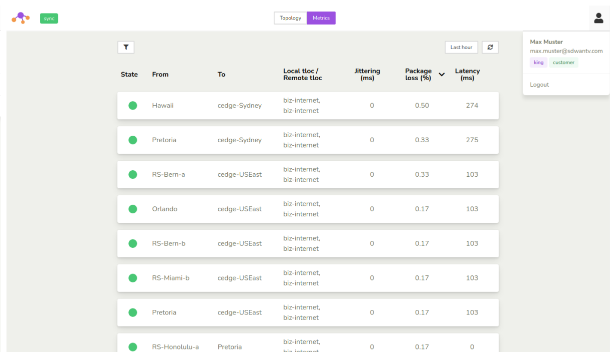

Figure B.4: Topology customer

Test results

The customer is only allowed to see a subset of the whole topology and metrics according to which company the customer belongs.

The test was successful!

B.2 UC2: Customer management: Test 1

Administrator accesses the customer management and lists users.

Initial state

An administrator is logged into SDWANTV and the topology is displayed.

Test procedure

The administrator accesses the user management panel from the droplist that appears by clicking on the user profile icon in the top right corner. Users are grouped by companies. The administrator can extend the company groups to view all users that belong to a certain company.

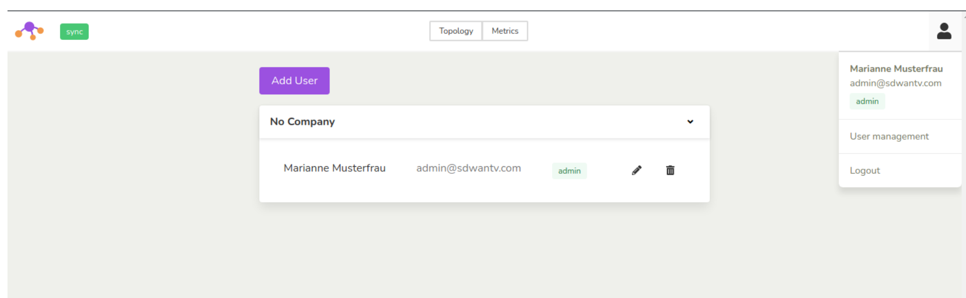

Figure B.5: User list

Test results

The administrator is able to access the user management and list the users.

The test was successful!

B.3 UC2: Customer management: Test 2

Administrator accesses the customer management and creates a user.

Initial state

An administrator is logged into SDWANTV and acesses the user management panel from the droplist that appears by clicking on the user profile icon in the top right corner. One User is present.

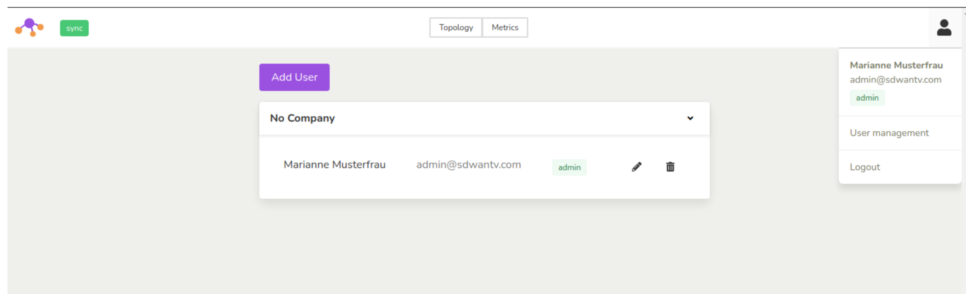

Figure B.6: User list

Test procedure

The administrator clicks on the add user button to add a new user to the system. The form to enter the user details opens and the administrator can fill in the required information.

Figure B.7: Add user empty form

Figure B.8: Add user filled in form

After all information are filled in the administrator clicks the add button to add the user to the system. A new page is displayed with a confirmation that the user was created and the login credentials.

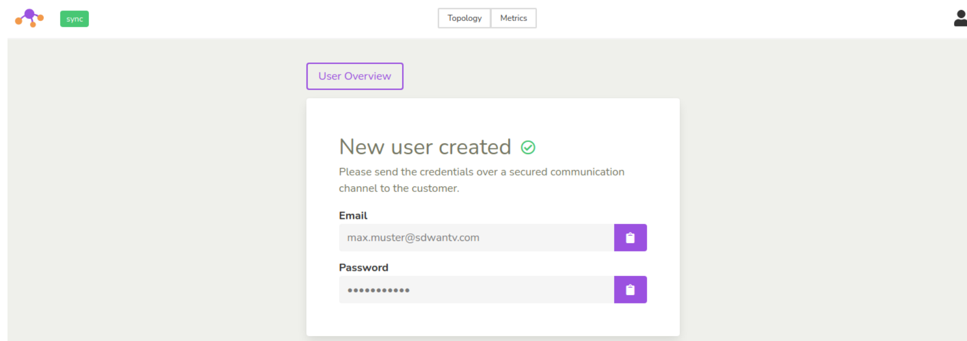

Figure B.9: User create confirmation

The administrator returns to the user overview page by clicking the user overview button.

Test results

The user overview page now also lists the newly created user.

The test was successful!

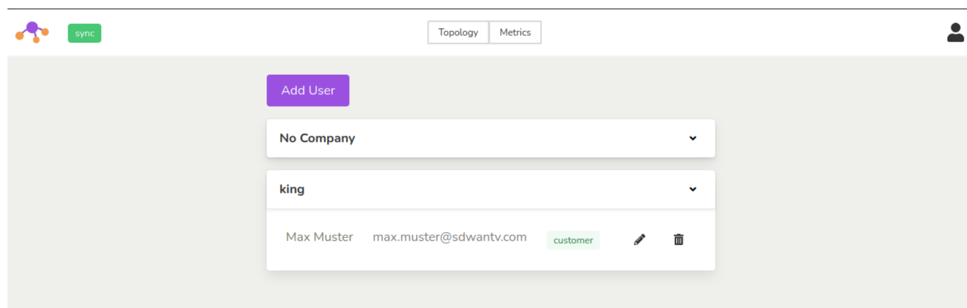

Figure B.10: User list after creation

B.4 UC2: Customer management: Test 3

Administrator accesses the customer management and deletes a user.

Initial state

An administrator is logged into SDWANTV and acesses the user management panel from the droplist that appears by clicking on the user profile icon in the top right corner. 2 users are present in the list of all users.

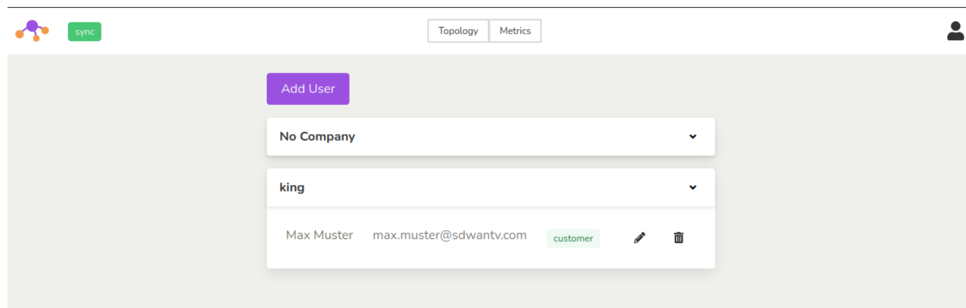

Figure B.11: User list

Test procedure

The administrator deletes the user Max Muster from the system by clicking on the paper bin icon. A user delete popup is display that asks for confirmation to delete the user.

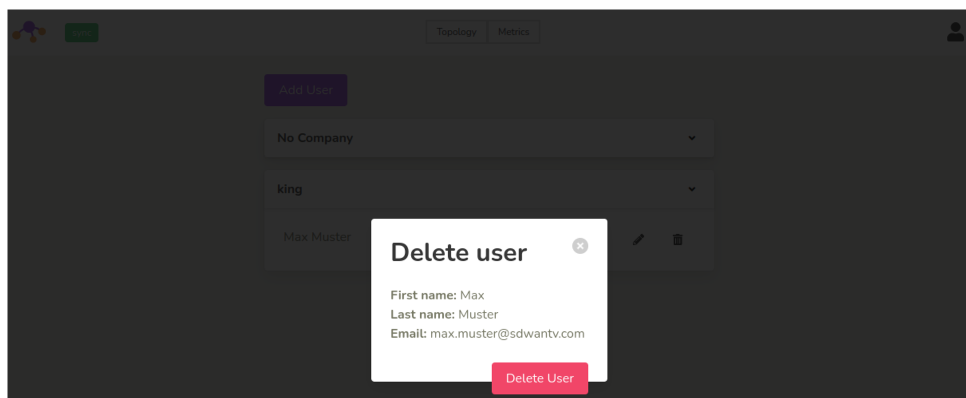

Figure B.12: User delete popup

After clicking the delete button on the popup the user is deleted, the system returns to the user list and a blue delete confirmation message is displayed.

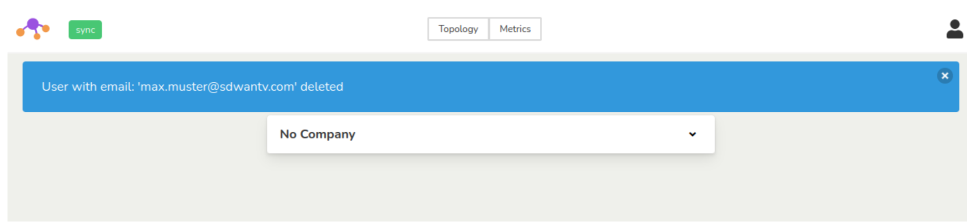

Figure B.13: User delete confirmation

Test results

The user list does not show the deleted user anymore.

The test was successful!

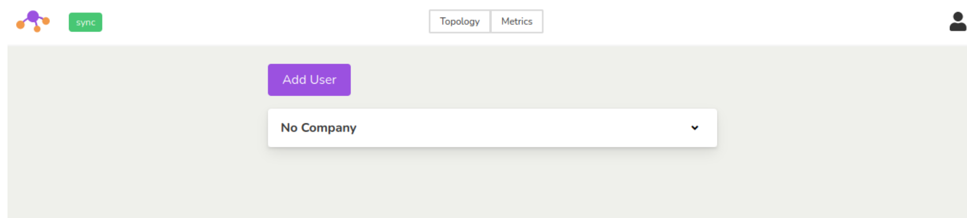

Figure B.14: User list after deletion

B.5 UC2.1: Update customer

The administrator updates profile information of a customer.

Initial state

An administrator is logged into SDWANTV and acesses the user management panel from the droplist that appears by clicking on the user profile icon in the top right corner. 2 users are present in the list of all users.

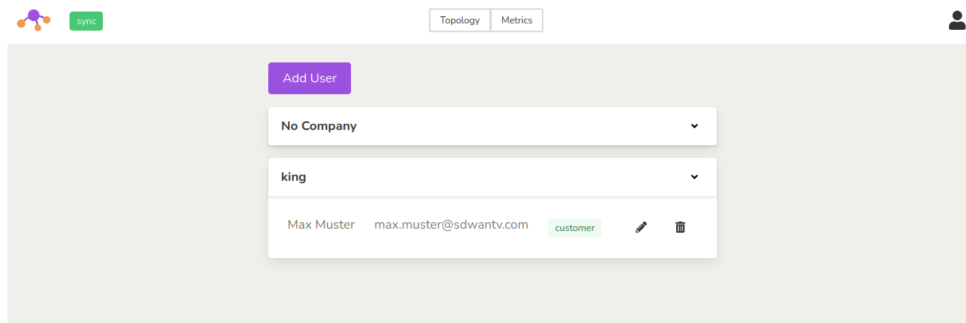

Figure B.15: User list

Test procedure

The administrator clicks on the edit button on the same line as his name and email which will open the user details form.

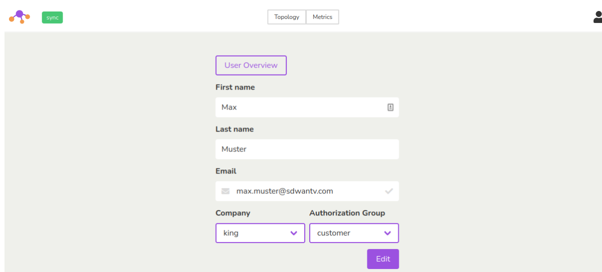

Figure B.16: User form before update

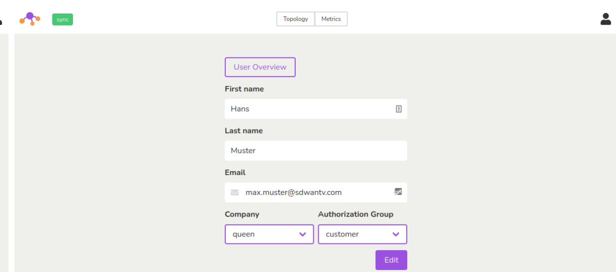

Figure B.17: User form after update

After the user details were modified the administrator clicks the edit button to make the changes final, the system returns to the user list and a blue update confirmation message is displayed.

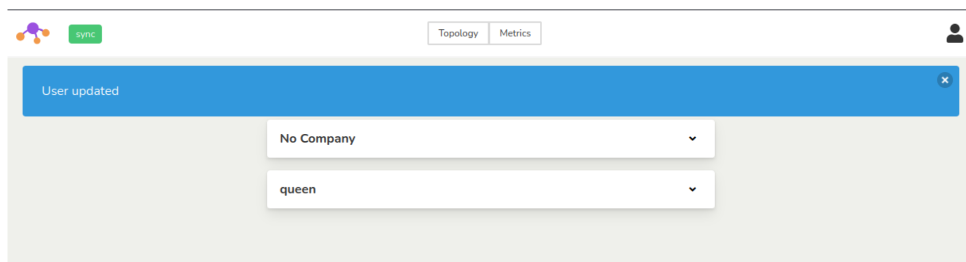

Figure B.18: Udpate confirmation

Test results

The user overview does now show the update user information.

The test was successful!

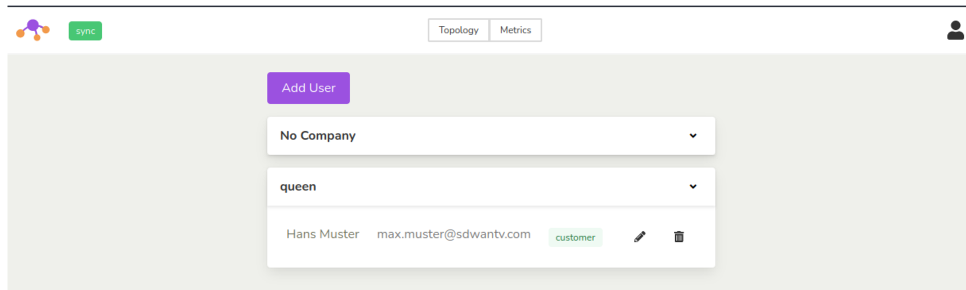

Figure B.19: User list after update

B.6 UC3: Add filter between two sites

Filter between two sites.

Initial state

A user is logged into SDWANTV and the unfiltered topology is displayed.

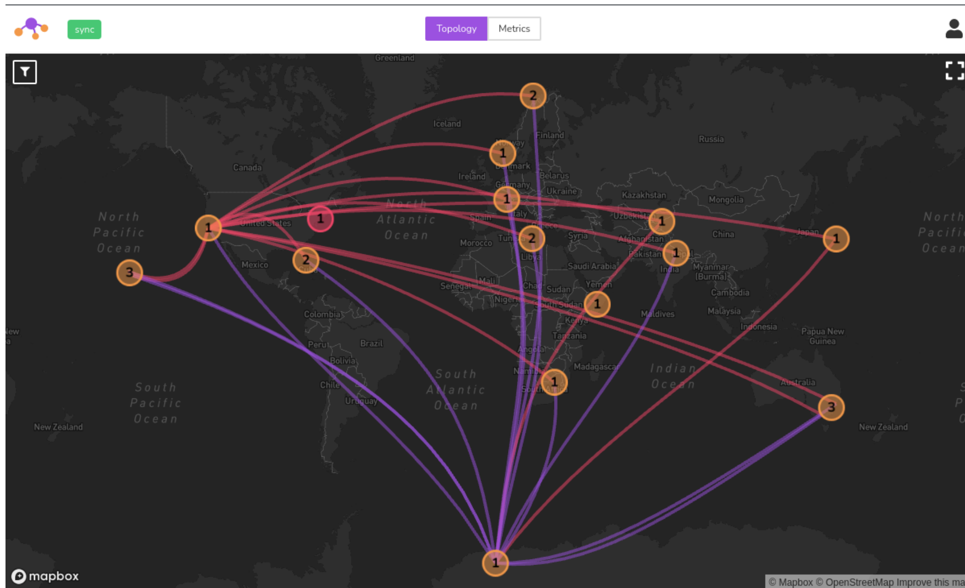

Figure B.20: Unfiltered topology

Test procedure

After the user opens the filter panel, by pressing on the button in the left top corner, a filter can be applied. A site is chosen from the list of available sites. Another site is selected from the site filter panel.

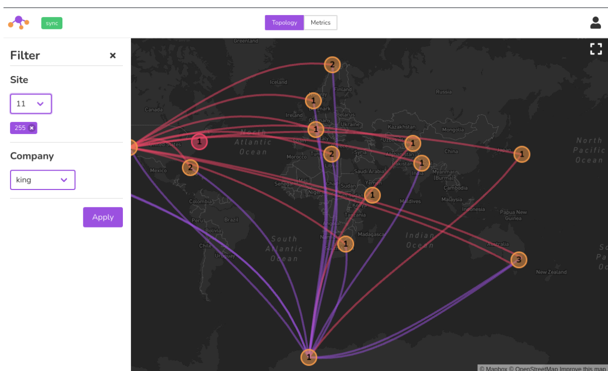

Figure B.21: One site selected

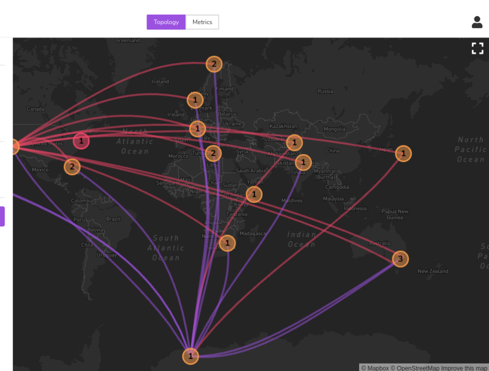

Figure B.22: Two sites selected

After two sites were selected the filter is applied by pressing the apply button.

Test results

The topology is filtered to only show tunnels between the two selected sites.

The test was successful!

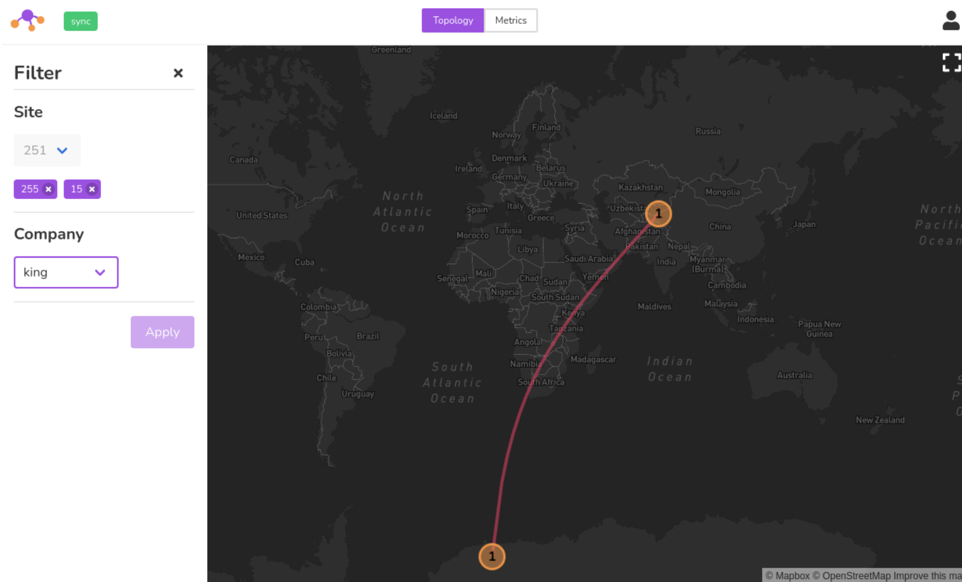

Figure B.23: Topology filtered by two sites

B.7 UC5: Display tunnel metrics: Test 1

The user accesses the metrics overview.

Initial state

A user is logged into SDWANTV and the topology on the world map is displayed.

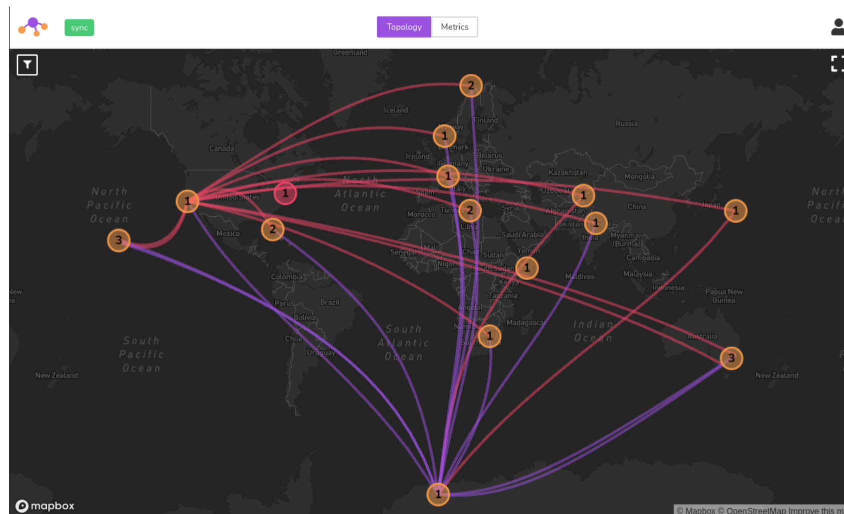

Figure B.24: Initial topology

Test procedure

The user switches to the metrics overview by pressing the metrics button in the center of the page header.

Test results

The metrics page is opened and displays the metrics overview for all available tunnels.

The test was successful!

sync | Topology | Metrics

Last hour

State	From	To	Local tloc / Remote tloc	Jittering (ms)	Package loss (%)	Latency (ms)
●	reserve1	RS-Bern-a	biz-internet, biz-internet	0	100.00	0
●	reserve1	RS-Sydney-a	mpls, mpls	0	100.00	0
●	reserve1	RS-CapeTown-b	biz-internet, biz-internet	0	100.00	0
●	reserve1	RS-Tromsoe-b	mpls, mpls	0	100.00	0
●	reserve1	RS-Miami-a	mpls, mpls	0	100.00	0
●	reserve1	RS-Bern-b	biz-internet, biz-internet	0	100.00	0
●	reserve1	RS-Miami-a	biz-internet, biz-internet	0	100.00	0

Figure B.25: Metrics overview

B.8 UC5: Display tunnel metrics: Test 2

Specific historical tunnel metrics.

Initial state

The user is logged into SDWANTV and heads over to the metrics overview page.

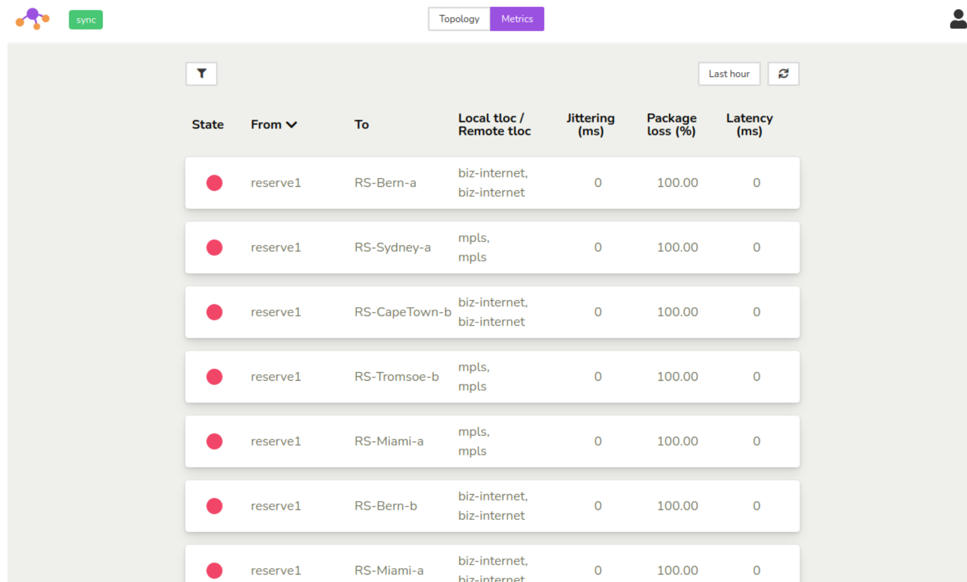

Figure B.26: Metrics overview

Test procedure

The user clicks on the desired tunnel to extend the window and display historical metrics and some information about the node that the connection is in between. By hovering over a datapoint on the chart the value and the exact date is displayed.

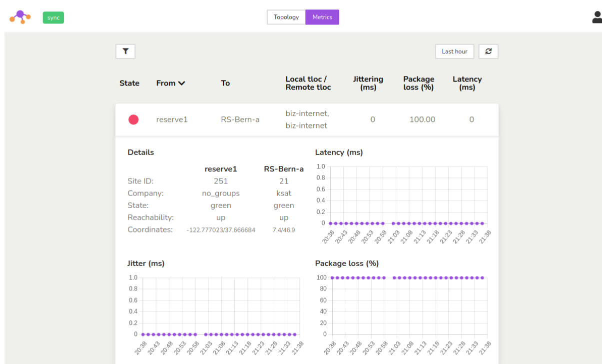

Figure B.27: Historical metrics

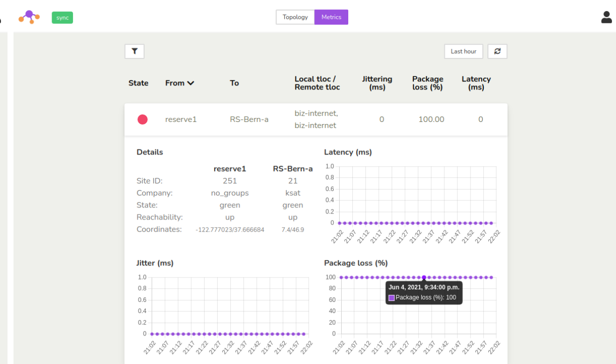

Figure B.28: Specific datapoint selected

Test results

The user is able to see historical metrics about one specific tunnel.

The test was successful!

B.9 UC5.1: Specify historical metrics time range

Specify the time range for historical metrics.

Initial state

The user is logged into SDWANTV and heads over to the metrics overview page. The user clicks on the desired tunnel to extend the window and display historical metrics.

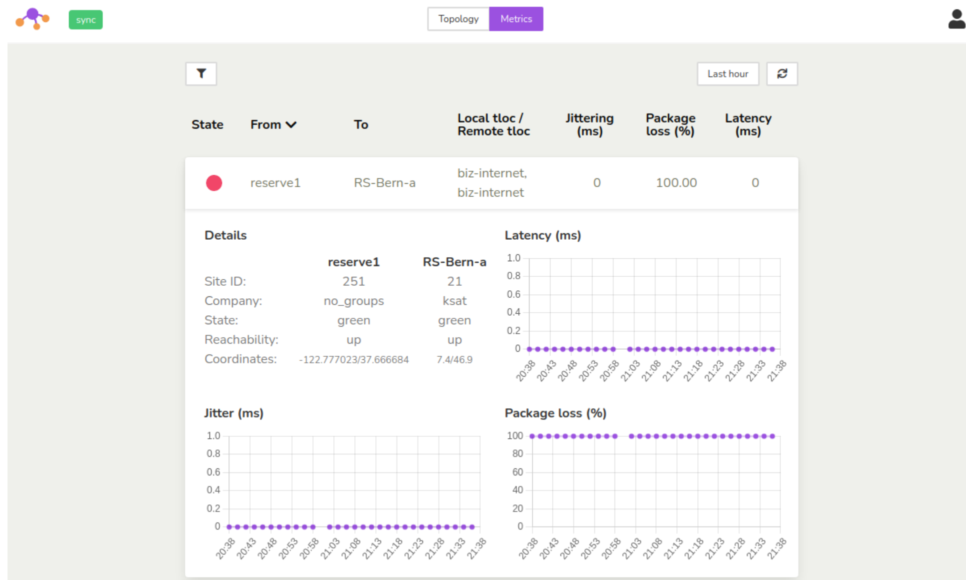

Figure B.29: Metrics overview

Test procedure

The user clicks on button that is called last hour and a time picker popup is opened. The desired time range can be selected and the time range is automatically adjusted and the new datapoints displayed in the chart

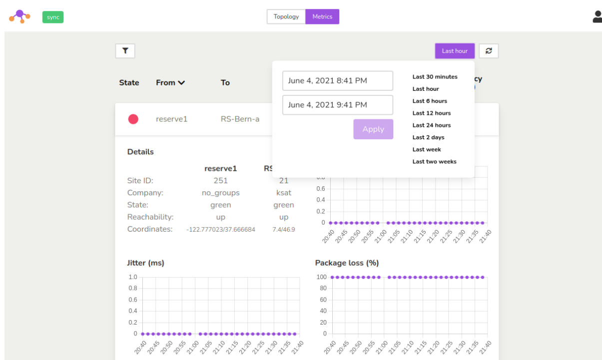

Figure B.30: Time range filter opened

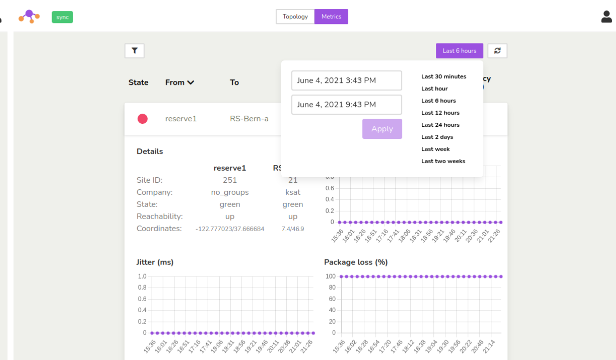

Figure B.31: Time range filter selected

Test results

The time range for the displayed datapoints is adjusted and historical metrics for the desired time range are displayed. The test was successful!

B.10 UC5.3: Filtering metrics: Test 1

Filter metrics by field.

Initial state

The user is logged into SDWANTV and heads over to the metrics overview page.

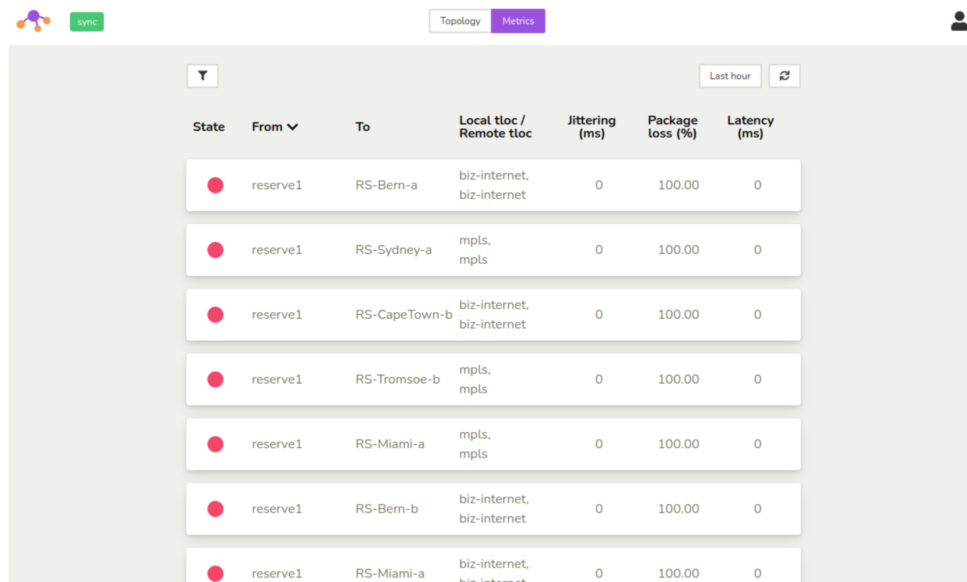

Figure B.32: Metrics overview

Test procedure

The user opens the filter by pressing the filter button on the top left side of the screen. He now chooses the kind of filter that should be applied. The company filter is selected to only display metrics that belong to a certain company. Afterwards the user can select the desired company from in the second dropdown.

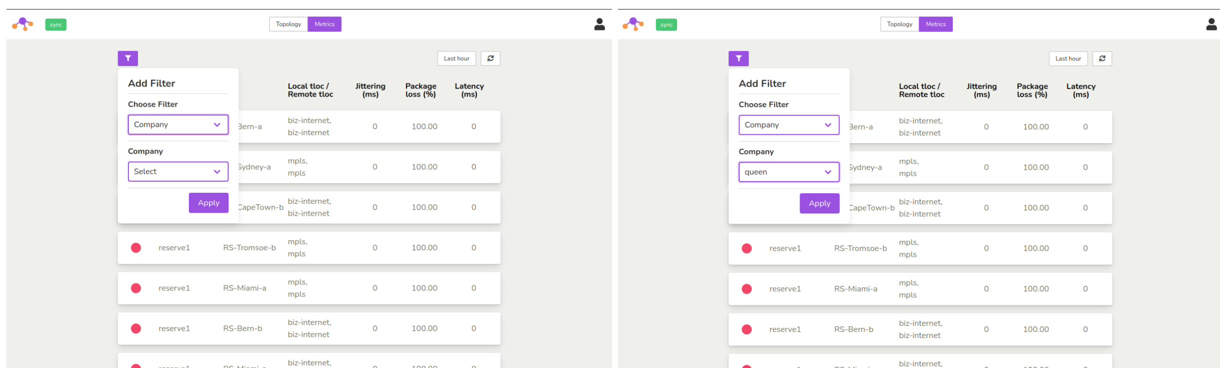

Figure B.33: Company filter selected

Figure B.34: Company filter and value selected

After both the filter and the value were selected to user hits the apply button to filter the metrics overview.

Test results

The metrics overview is filtered by the selected filter.

The test was successful!

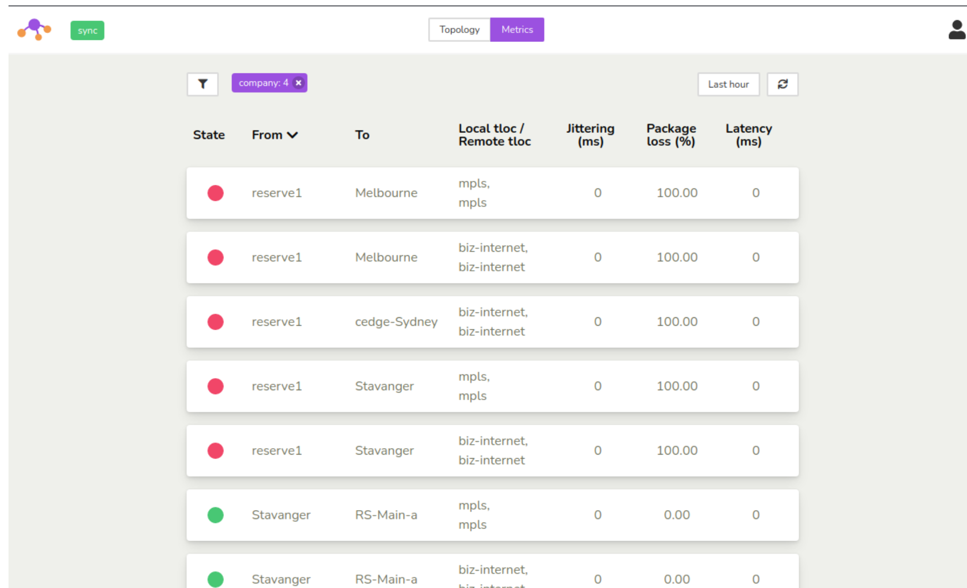

Figure B.35: Metrics overview filtered

B.11 UC5.3: Filtering metrics: Test 2

Filter metrics by metric threshold.

Initial state

The user is logged into SDWANTV and heads over to the metrics overview page.

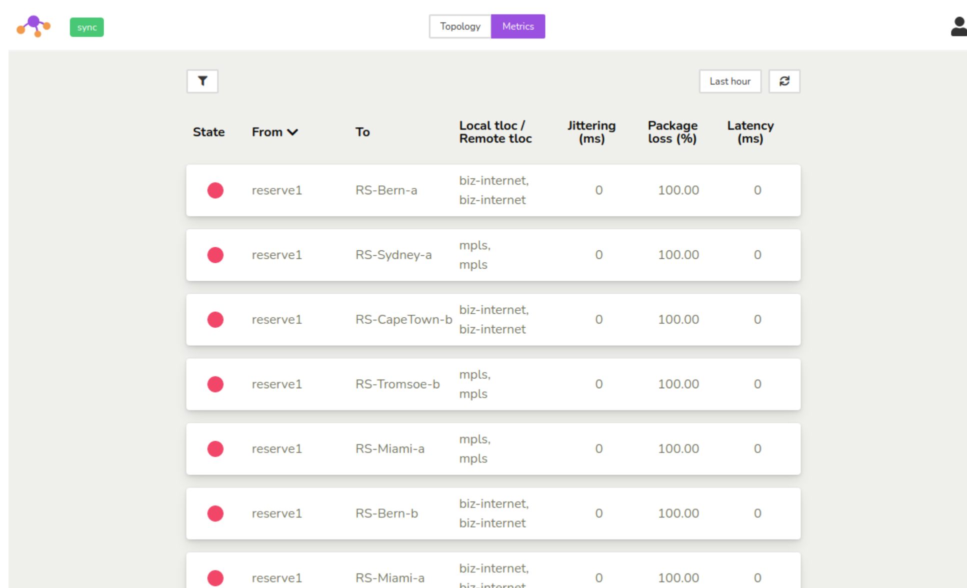

Figure B.36: Metrics overview

Test procedure

The user opens the filter by pressing the filter button on the top left side of the screen. He now chooses the kind of filter that should be applied. The package loss filter is selected to only display metrics that are greater or less than a defined threshold. Afterwards the user types in the desired percentage and selects if the filtered metrics should have a higher or lower value than the defined threshold value..

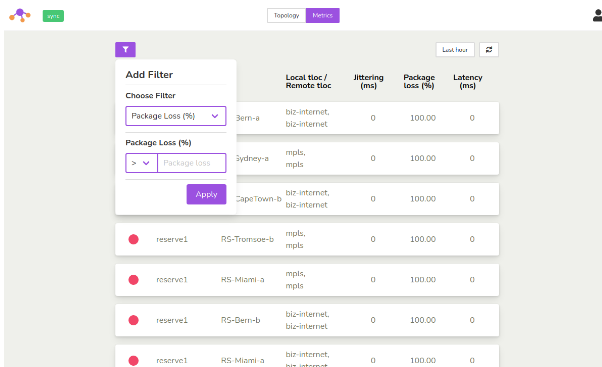

Figure B.37: Package loss filter selected

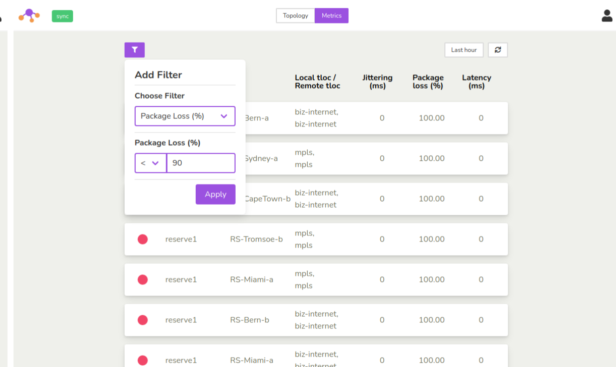

Figure B.38: Package loss filter and threshold defined

After both the filter and the threshold value were defined to user hits the apply button to filter the metrics overview.

Test results

The metrics overview is filtered by the selected filter.

The test was successful!

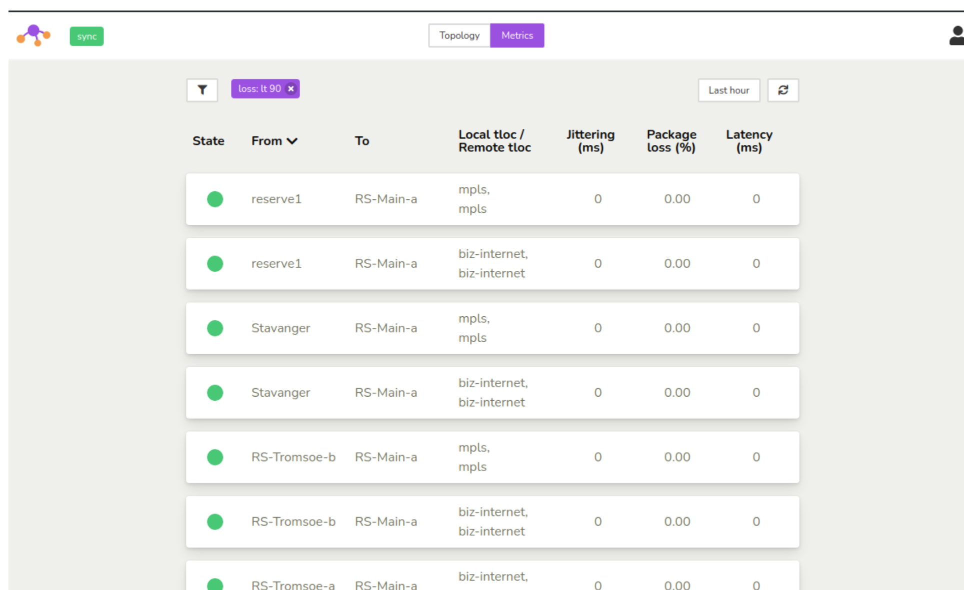

Figure B.39: Metrics overview filtered

B.12 UC5.4: Sorting metrics

Sorting metrics.

Initial state

The user is logged into SDWANTV and heads over to the metrics overview page.

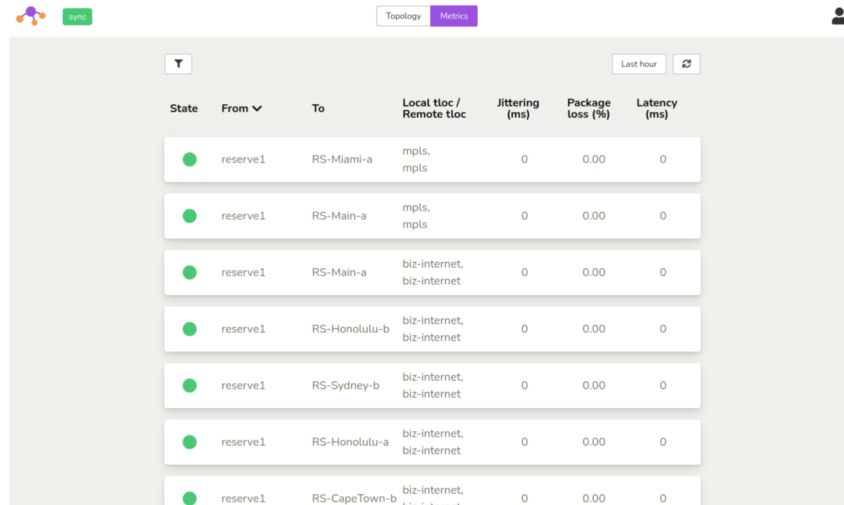

Figure B.40: Metrics overview

Test procedure

The user selects the field on which he would like to sort and clicks on it. Here the to field was selected. When clicking 1 time on the field it will filter descending, when clicking another time on the filter it will filter ascending.

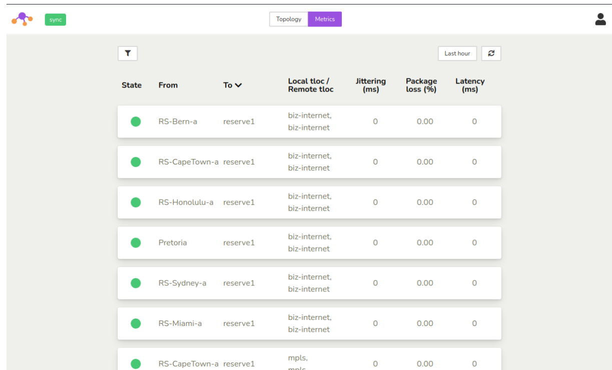

Figure B.41: To sorting descending

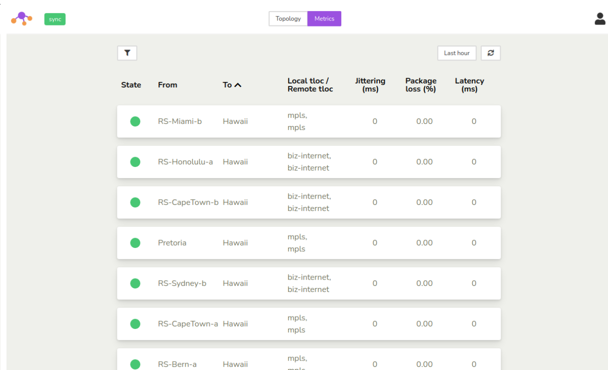

Figure B.42: To sorting ascending

Test results

The metrics are sorted according to the to field. It is to note that sorting by from and to names happens according to the ascii table number representation which is the cause that the sorting does not work properly and a tunnel that starts with ”c” is higher in the list than one that starts with an ”S”. The bug was reported and a respective task was opened to fix it later on. The test was partly successful!

B.13 AD5: Tunnel aggregation

Initial state

The user is logged into SDWANTV and the topology map is displayed. No node or tunnel is selected.

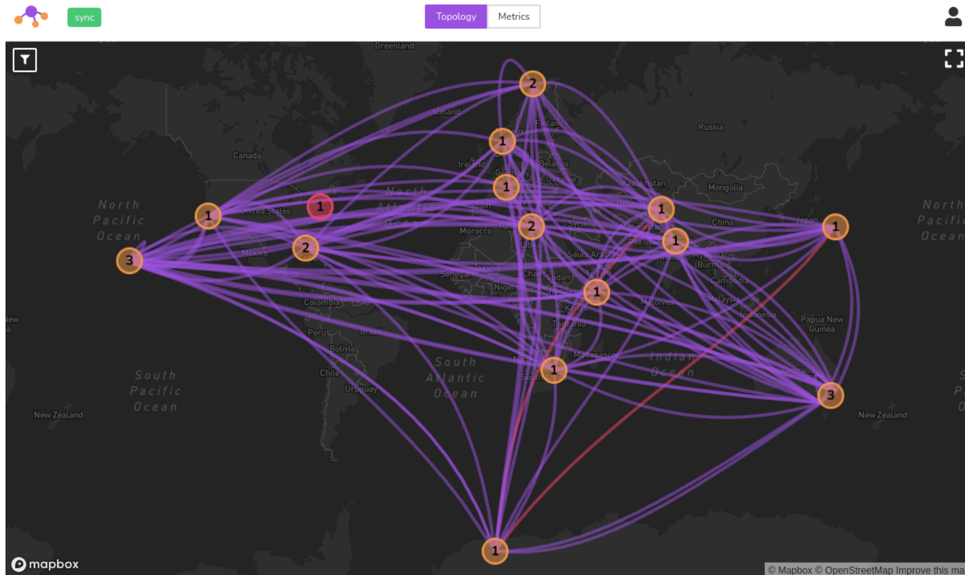

Figure B.43: Topology overview

Test procedure

The user selects a tunnel from the topology map which opens a popup and displays the information about the selected Tunnel. It shows up that the selected tunnel is between Hawaii and RS-Sydney-a and through the transport layer biz-internet. In the middle of the popup it is possible to select MPLS as the transport layer which will display the metrics for the MPLS IPSec tunnel. Tunnels between the same nodes are aggregated together for all transport layers.

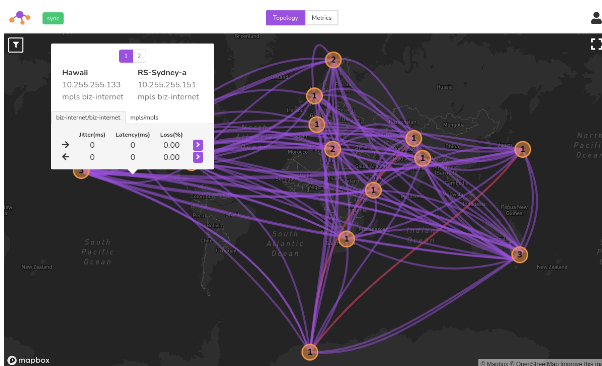

Figure B.44: Biz-internet

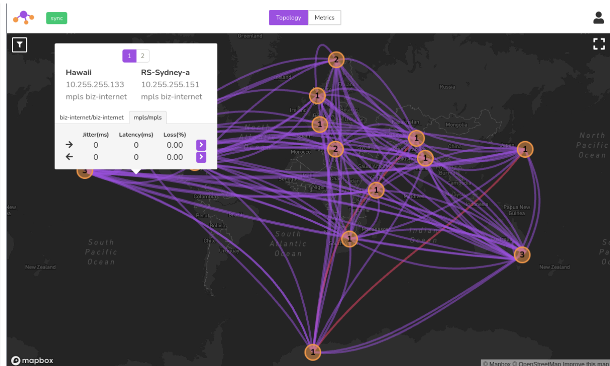

Figure B.45: MPLS

The user can select the second card in this popup by clicking on the number that is located in the top area of the popup. This will show the connection Hawaii to RS-Sydney-B. The nodes RS-Sydney-a and RS-Sydney-b are located on the exact same coordinates and have the same site. Tunnels that are between Nodes with the same coordinates are aggregated. This means

for all these tunnels there is only one connection displayed on the world map.

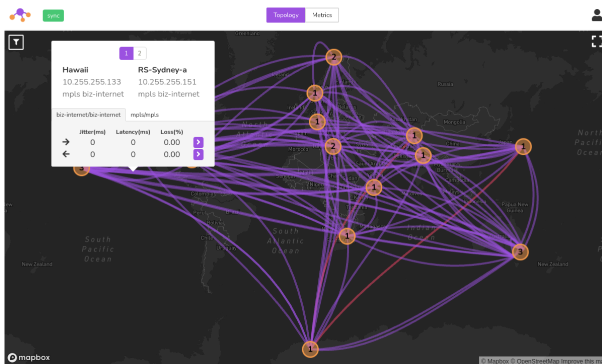

Figure B.46: Card 1

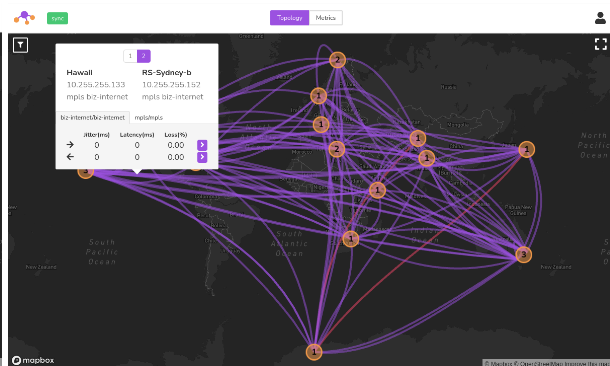

Figure B.47: Card 2

Test results

Tunnels were successfully aggregated and the information of the underlying IPSec tunnels are able to retrieved with cards and different transport layers.

The test was partly successful!

B.14 S1: Monitoring topology: Test 1

Pull out the ethernet cable of Miami-a

Initial state

All tunnels and nodes, except the single node that is marked red, are up and available.

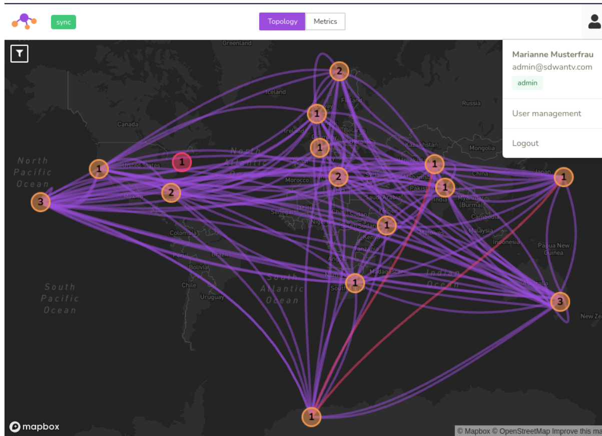

Figure B.48: Topology admin

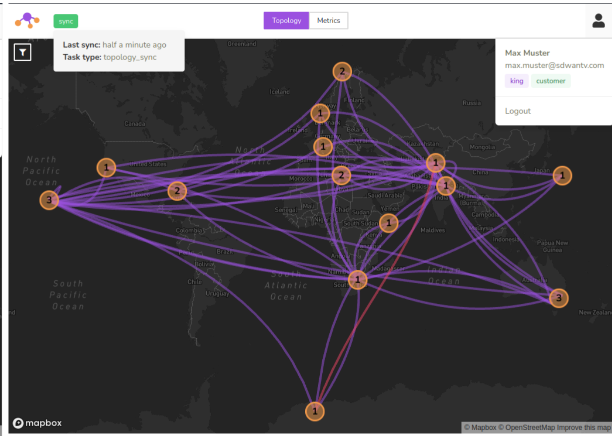

Figure B.49: Topology customer

Test procedure

Pulling the cable from all ports of node Miami-a simulates a network connection failure and brings the port down. As soon as a new API call to the vManage API occurs, the backend realizes that the tunnels that start or end in Miami-a are gone and report them as down in the database. The frontend is informed by websocket messages and marks the down tunnels red.

Test results

The frontend reports the down tunnels and marks them as red. Because all ethernet cables were unplugged from node Miami-a it was reported as down in the vManage API and therefore also marked down. The test was successful!

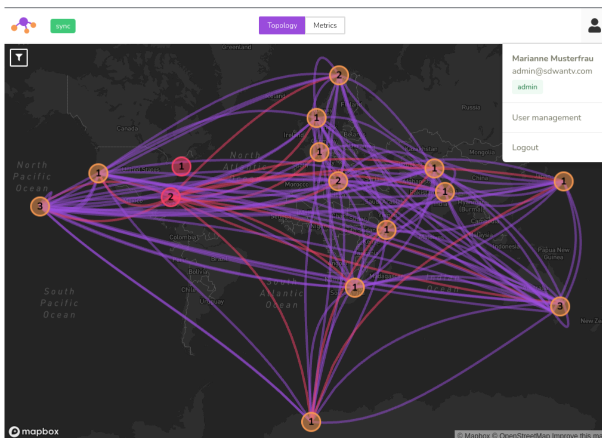

Figure B.50: Topology admin

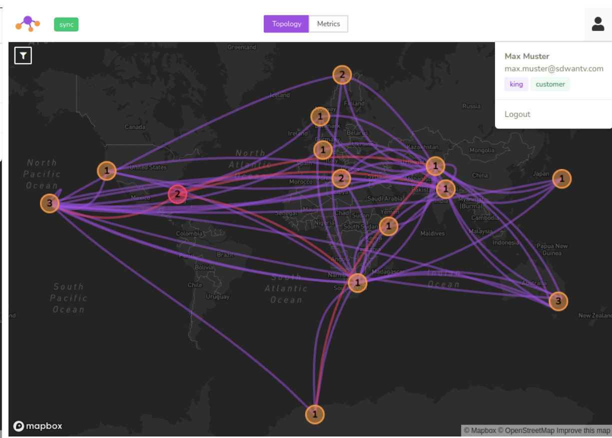

Figure B.51: Topology customer

B.15 S1: Monitoring topology: Test 2

Power outage of node Miami-a.

Initial state

All tunnels and nodes, except the single node that is marked red, are up and available.

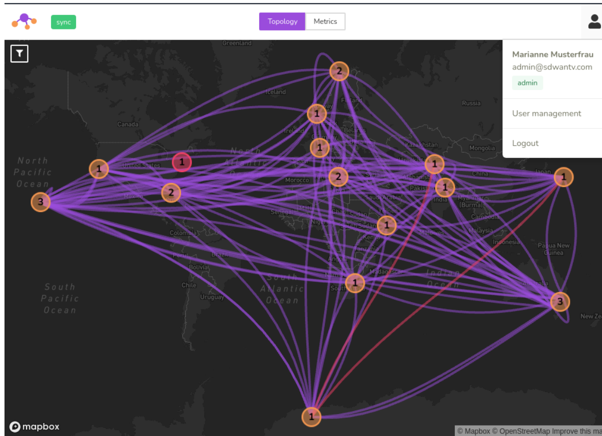

Figure B.52: Topology admin

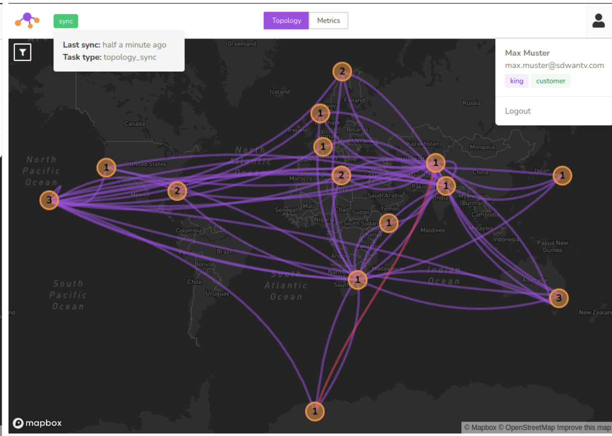

Figure B.53: Topology customer

Test procedure

Pulling the power cable of node Miami-a simulates a power outage. The node will be unavailable and unreachable for vManage.

Test results

Miami-a and all tunnels associated with it will be marked red.

The test was successful!

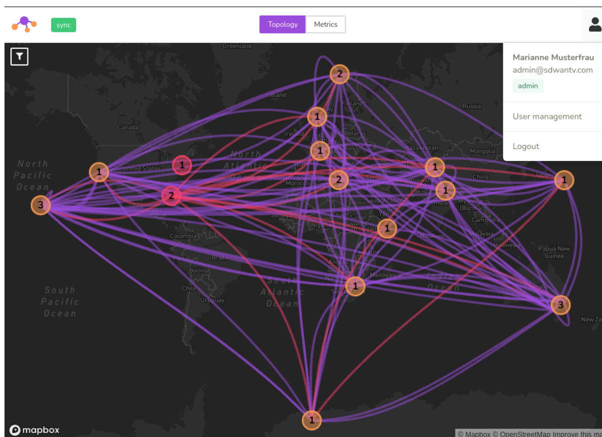

Figure B.54: Topology admin, Miami-a down

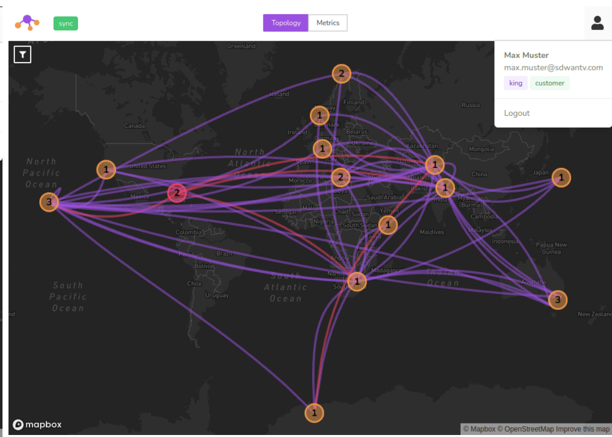

Figure B.55: Topology customer, Miami-a down

B.16 S1: Monitoring topology: Test 3

Give the power back to node Miami-a.

Initial state

Miami-a is down and marked as red. All tunnels from Miami-a are marked red.

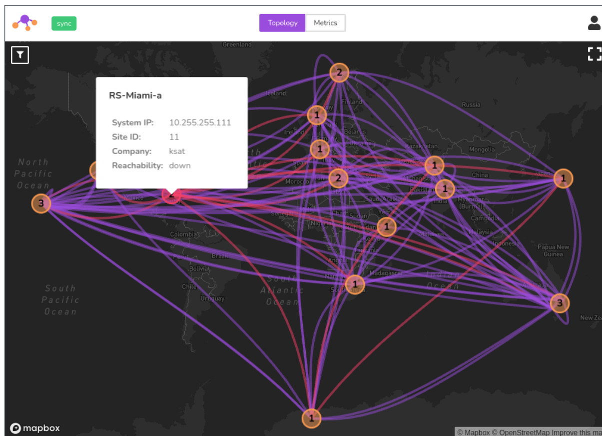

Figure B.56: Topology admin, Miami-a down

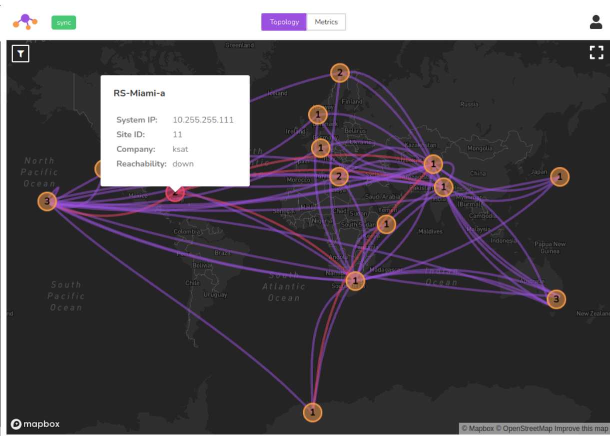

Figure B.57: Topology customer, Miami-a down

Test procedure

Turn on the node miami-a.

Test results

The node status is up again and Miami-a will be marked orange again.

The test was successful!

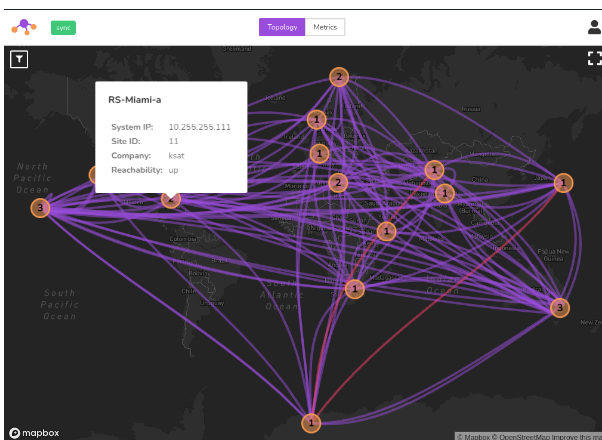

Figure B.58: Topology admin, Miami-a up

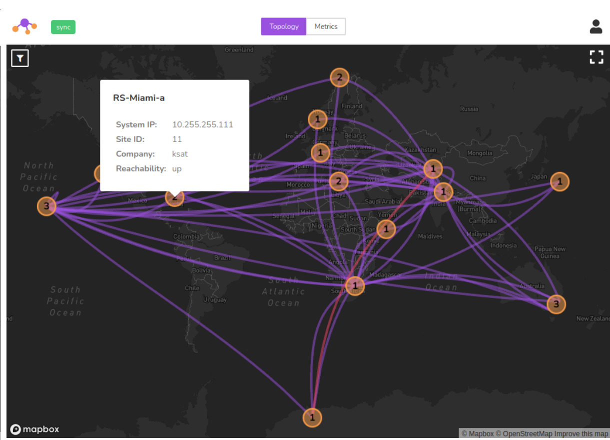

Figure B.59: Topology customer, Miami-a up

B.17 S1: Monitoring topology: Test 4

Apply a more restrictive policy.

Initial state

There is no policy active in vManage and therefore every node can speak to each other, which is represented in a full-mesh topology.

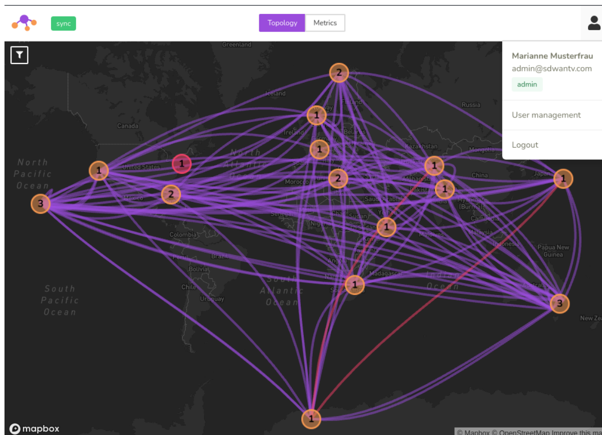

Figure B.60: Topology admin, full mesh

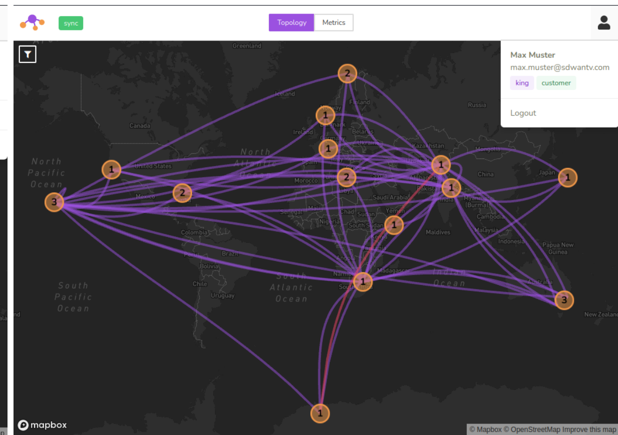

Figure B.61: Topology customer, full mesh

Test procedure

In vManage, the policy which prevents that each node can communicate with the others is applied. Customer nodes will only be able to communicate with a set of defined nodes. This results in a lot of tunnels being deleted.

Test results

Down tunnels are marked red in the frontend and not existing tunnels are removed from the map. The test was successful!

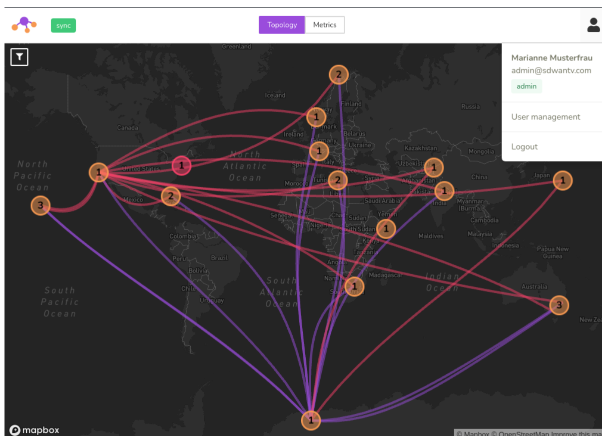

Figure B.62: Topology admin, policy active

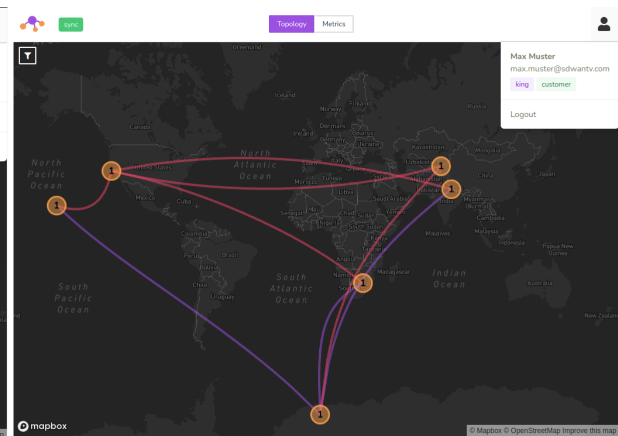

Figure B.63: Topology customer, policy active

B.18 S1: Monitoring topology: Reported bugs

Despite all 3 tests were successful we also encountered one minor error.

1. Tunnel curves are not always calculated the same way and therefore the topology not always looks the same for every logged in user. The tunnel curve is calculated according to the coordinates and some other unique input that will result in every curve being unique and not overlapping.

The bug was reported and a respective task was opened to fix it as soon as possible.

B.19 S2: View node information: Test 1

Initial state

The user is logged into SDWANTV and the topology map is displayed. No node or tunnel is selected.

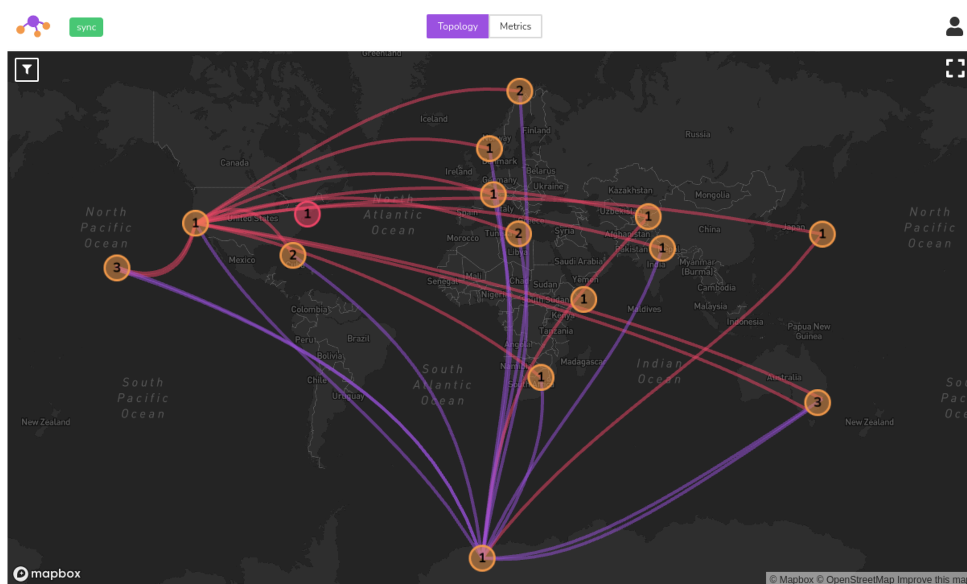

Figure B.64: Topology overview

Test procedure

The user selects a node on the topology world map by clicking on it.

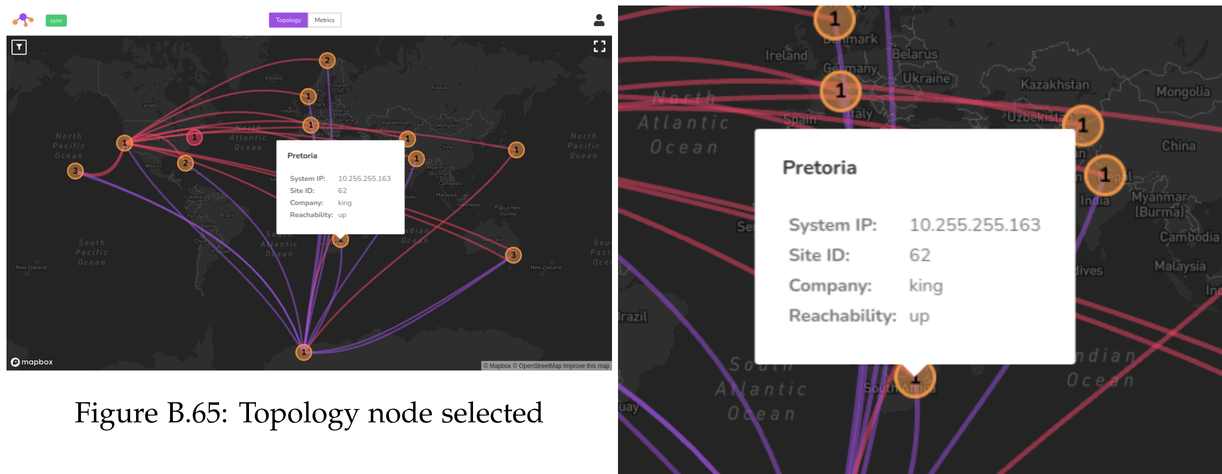

Figure B.65: Topology node selected

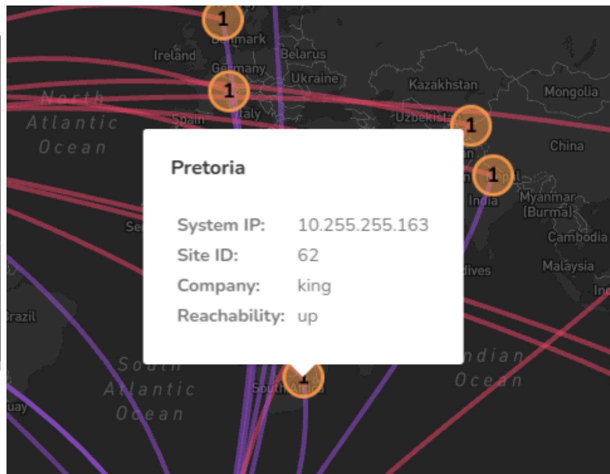

Figure B.66: Node popup

Test results

After a node is selected, a popup opens and displays the name and some other useful information of the node.

The test was successful!

B.20 S2: View node information: Test 2

Initial state

The user is logged into SDWANTV and the topology map is displayed. No node or tunnel is selected.

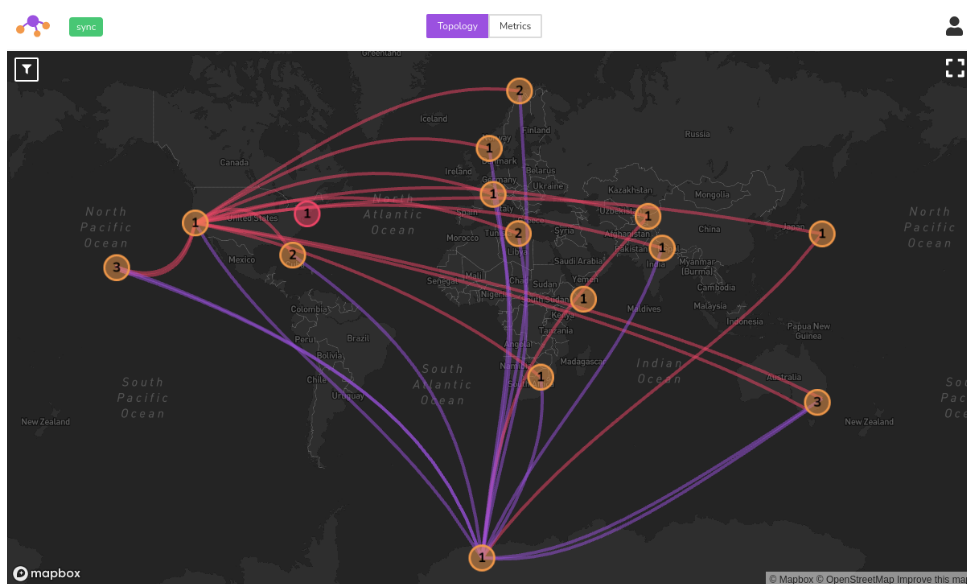

Figure B.67: Topology overview

Test procedure

The user selects a node group on the topology world map by clicking on it. If the node group contains nodes that have exactly the same coordinates, which is the case for sites that have multiple nodes, a popup opens with all the names of the nodes. The user can click on one node from the list and the node popup opens and displays the information about the selected node.

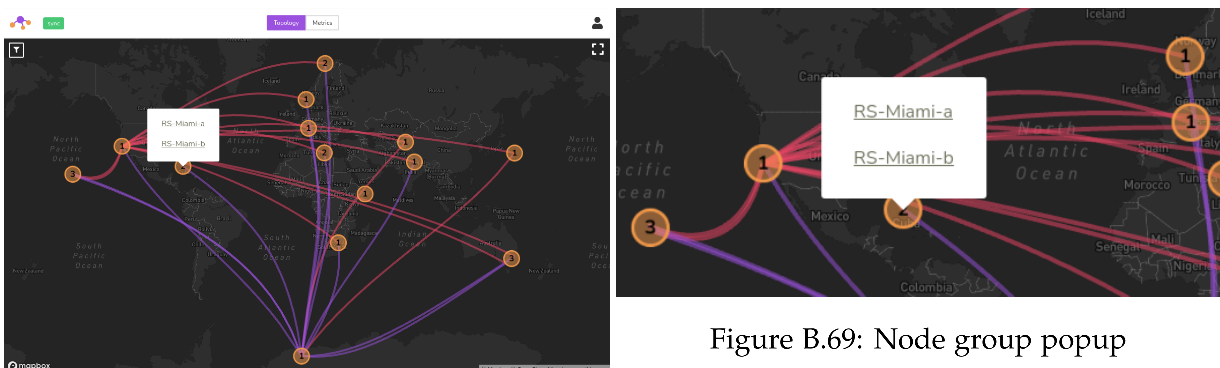

Figure B.68: Topology node group selected

Figure B.69: Node group popup

Test results

After a node from the list is selected, a popup opens and displays the name and some other useful information of the node. The test was successful!

B.21 S2: View node information: Test 3

Initial state

The user is logged into SDWANTV and the topology map is displayed. No node or tunnel is selected.

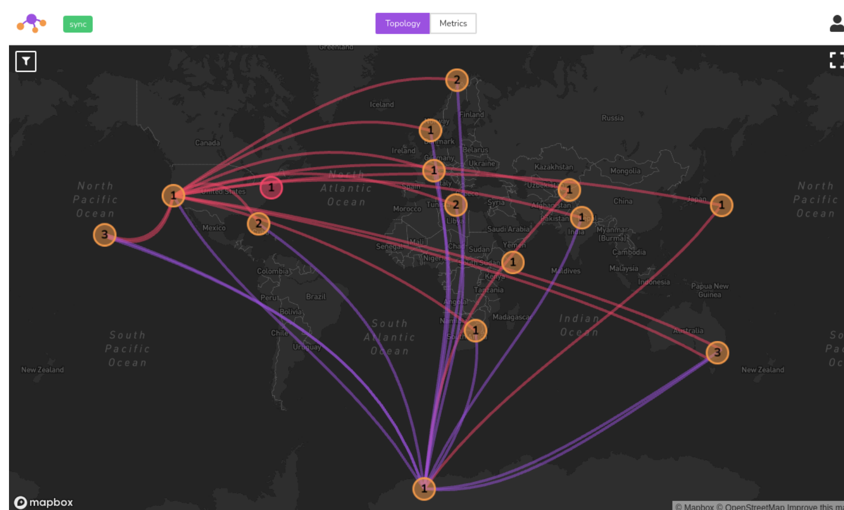

Figure B.70: Topology overview

Test procedure

The user selects a node group on the topology world map by clicking on it. If the node group contains nodes that do not have exactly the same coordinates, which is the case for nodes that are in the same city but on different sites, the map will automatically zoom in and divided into separate nodes.

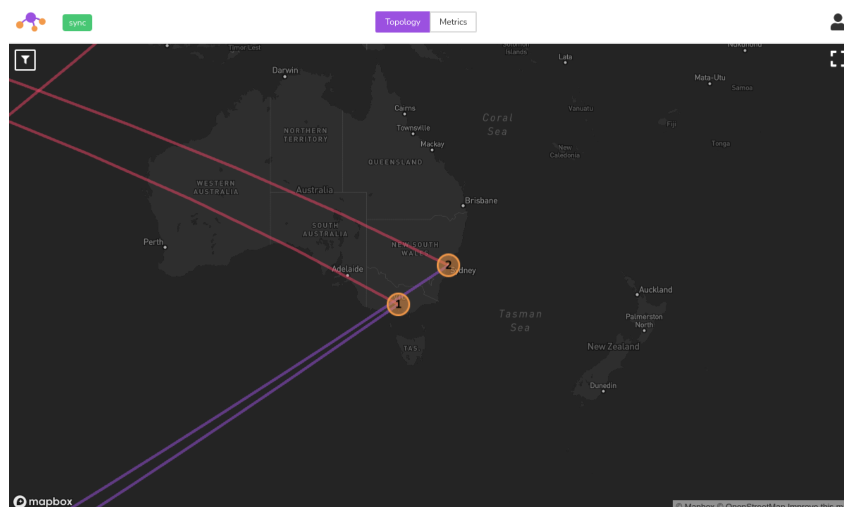

Figure B.71: Zoomed topology

Test results

After selecting a node group the nodes are divided and displayed at their location. The test was successful!

B.22 S3: Display connection metrics

Initial state

The user is logged into SDWANTV and the topology map is displayed. No node or tunnel is selected.

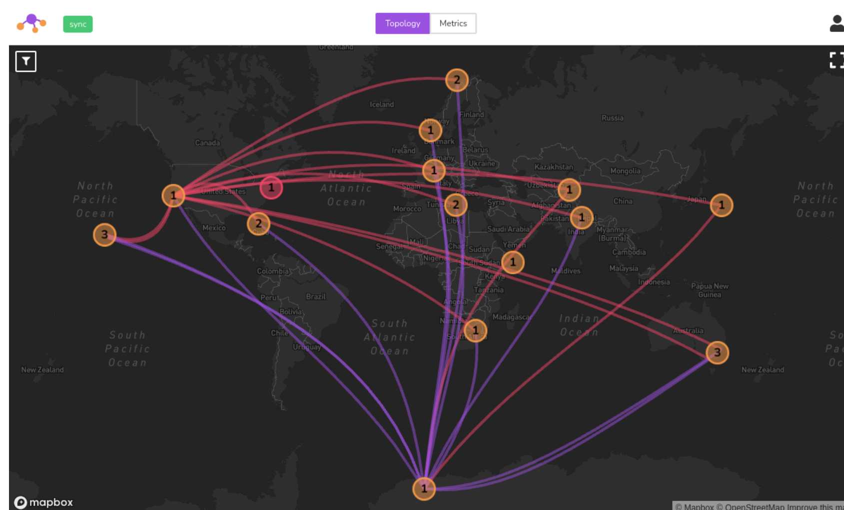

Figure B.72: Topology overview

Test procedure

The user selects a tunnel from the topology map which opens a popup and displays the latest metrics of that tunnel. Either in one or both directions.

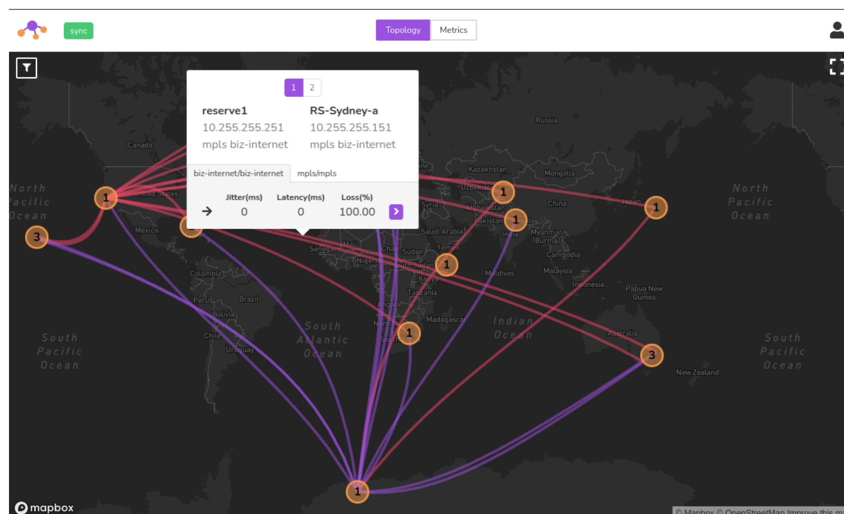

Figure B.73: Tunnel selected

Test results

The latest metric of the selected tunnel is displayed.

The test was successful!

B.23 S4: Apply customer filter

Initial state

The administrator is logged into SDWANTV and the topology map is displayed. No node or tunnel is selected. No filter is applied.

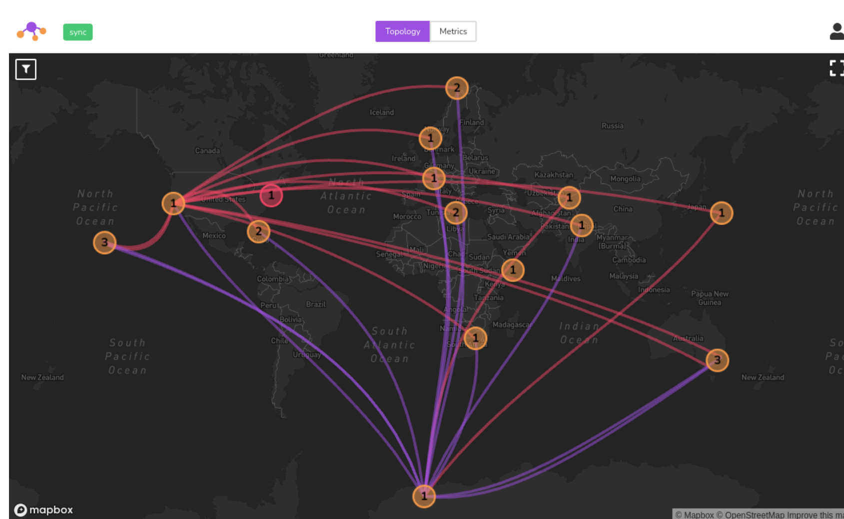

Figure B.74: Topology overivew, no filter

Test procedure

After opening the filter panel, by pressing on the button in the left top corner, a filter can be applied. A company is chosen from the list of available companies and the apply button is pressed.

Test results

After the filter is applied only nodes and tunnels that belong to this company are displayed.

The test was successful!

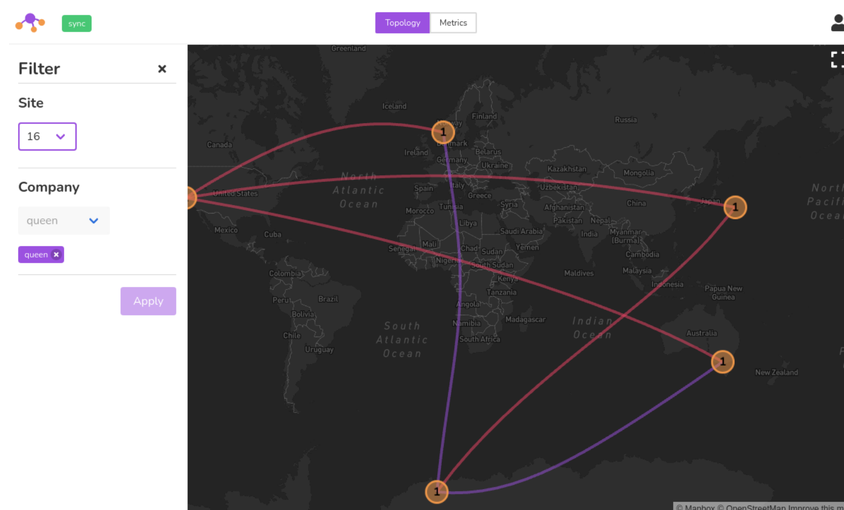

Figure B.75: Company filter applied

B.24 S5: Apply site filter

Initial state

The user is logged into SDWANTV and the topology map is displayed. No node or tunnel is selected. No filter is applied.

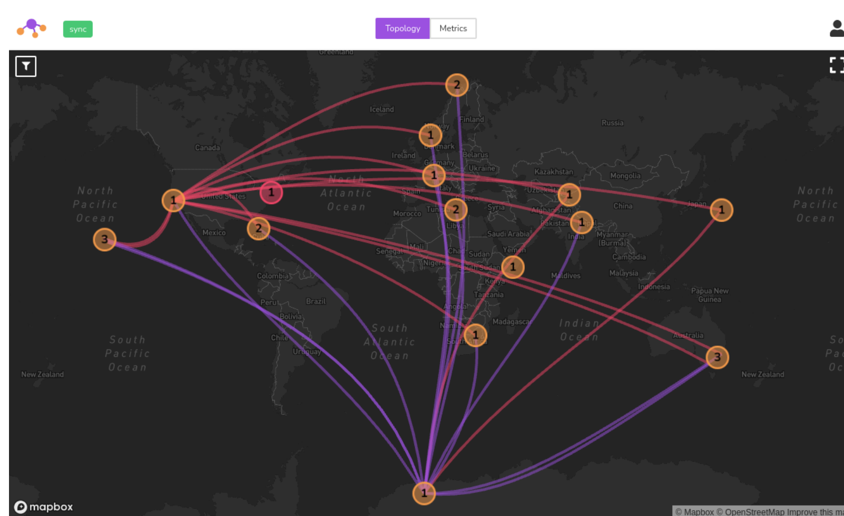

Figure B.76: Topology overivew, no filter

Test procedure

After opening the filter panel, by pressing on the button in the left top corner, a filter can be applied. A site is chosen from the list of available sites and the apply button is pressed.

Test results

After the filter is applied only nodes from this site and tunnels that go to the site or away from the site are displayed.

The test was successful!

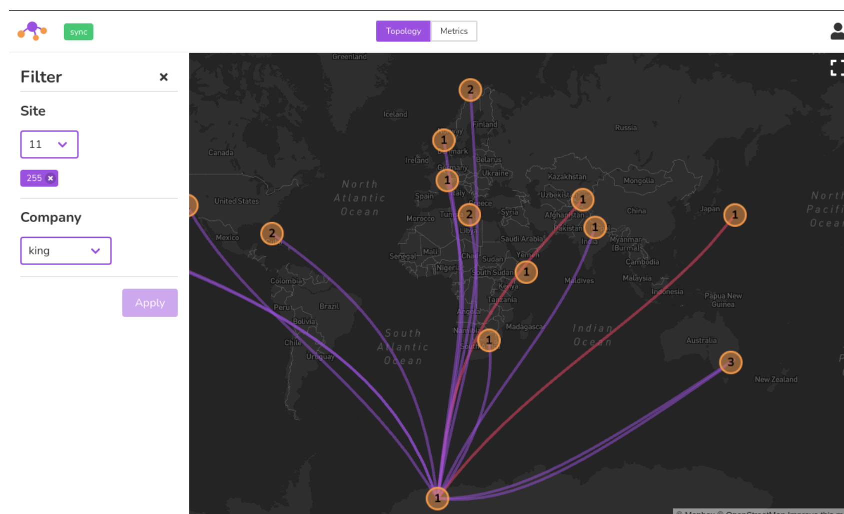

Figure B.77: Site filter applied

Appendix C

Non functional requirement test protocol

All test are executed on a Macbook Pro in the network of the lab environment of the INS.

Non functional requirement	**Implemented**	**Result**	**Status**
Security	yes	result C.1	passed
Fault tolerance, user data	yes	result C.2	passed
Fault tolerance, vManage data	yes	result C.3	passed
Maturity	yes	result C.4	passed
Understandability	yes	result C.5	passed
Failure management	yes	result C.6	passed
Time behaviour	yes	result C.7	passed
Efficiency compliance	maybe	no result	could not be tested
Response time	yes	result C.8	passed
Supportability	yes	result C.9	passed
Portability	yes	result C.10	passed
Scalability	yes	result C.11	partly passed

Table C.1: NFR test protocol

C.1 Security

To be tested

1. Passwords are never stored in plain text.
2. JWT token has limited lifetime.
3. Websocket connection do not log JWT token
4. System logs relevant information to the Stdout stream.

Test result

Password is encrypted The python default user management uses the PBKDF2 algorithm with SHA256 hashes to encrypt passwords.

password ↓ =
pbkdf2_sha256$216000$MACoTMDZf009$0Hp8LfIf90Uso36H96slLmdwUuF3KwtzBjpGSmj8J0o=

Figure C.1: Password stored in DB

JWT has limited lifetime The lifetime of the JWT token is embedded in the token body itself. We let the system run for over a day and tested the token devaluation. It was successful.

```
{
  "token_type": "access",
  "exp": 1607862643,
  "jti": "28dbbc148f4744f6ab3ef2c05783992e",
  "user_id": 1
}
```

Figure C.2: Sample JWT payload

Websocket connection do not log JWT token The logging output upon a newly connection does not contain any sensible information.

```
{
  "token_type": "access",
  "exp": 1607862643,
  "jti": "28dbbc148f4744f6ab3ef2c05783992e",
  "user_id": 1
}
```

Figure C.3: JWT token logging

System logs all relevant infromation The backend displays all requests made from the frontend into the Stdout. This is sufficient for security checking at the current state of the project.

```
ode node cedge195 with system-ip {device["system-ip"]} because it is not reachable
sdwantv_backend_local_1  | django.server INFO     "OPTIONS /api/v1/token/ HTTP/1.1" 200 0
sdwantv_backend_local_1  | django.server INFO     "POST /api/v1/token/ HTTP/1.1" 200 524
sdwantv_backend_local_1  | django.server INFO     "OPTIONS /api/v1/syncs/ HTTP/1.1" 200 0
sdwantv_backend_local_1  | django.server INFO     "OPTIONS /api/v1/companies/ HTTP/1.1" 200 0
sdwantv_backend_local_1  | django.server INFO     "OPTIONS /api/v1/siteids/ HTTP/1.1" 200 0
sdwantv_backend_local_1  | django.server INFO     "OPTIONS /api/v1/topology HTTP/1.1" 200 0
sdwantv_backend_local_1  | django.server INFO     "OPTIONS /api/v1/users/1 HTTP/1.1" 200 0
sdwantv_backend_local_1  | django.server INFO     "GET /api/v1/syncs/ HTTP/1.1" 200 272
sdwantv_backend_local_1  | django.server INFO     "GET /api/v1/companies/ HTTP/1.1" 200 35
sdwantv_backend_local_1  | django.server INFO     "GET /api/v1/siteids/ HTTP/1.1" 200 61
sdwantv_backend_local_1  | django.server INFO     "OPTIONS /api/v1/topology/ HTTP/1.1" 200 0
sdwantv_backend_local_1  | django.server INFO     "OPTIONS /api/v1/users/1/ HTTP/1.1" 200 0
sdwantv_backend_local_1  | django.server INFO     "GET /api/v1/users/1/ HTTP/1.1" 200 72
sdwantv_celery_1         | [2020-12-12 12:30:45,041: WARNING/ForkPoolWorker-1] /usr/local/lib/py$
```

Figure C.4: Logs of the stdout stream

C.2 Fault tolerance, user data

To be tested

1. Frontend is not able to bring the system into failed state.
2. JWT token has limited lifetime.
3. User receives feedback if he makes an invalid input.

Test result

No failed state During the bachelor thesis, we introduced the user management as HTTP REST endpoints. The user management contains some frontend actions which send a POST and UPDATE HTTP method to the backend and whereby modifies the backend state. To prevent any unwanted user input we implemented an input validation in the frontend and in the backend. Consequently, we can sanitize the input at the border of our system and prevent any user to bring our system into a failure state.

Similar to the HTTP endpoint, the Websocket endpoint does some input validation. In contrast to the HTTP request however, only a part of the system will crash if a websocket connection fails on cause of a wrong user input. The error boundary is in this case only limited on a single connection and will not be able to incapacitate other services.

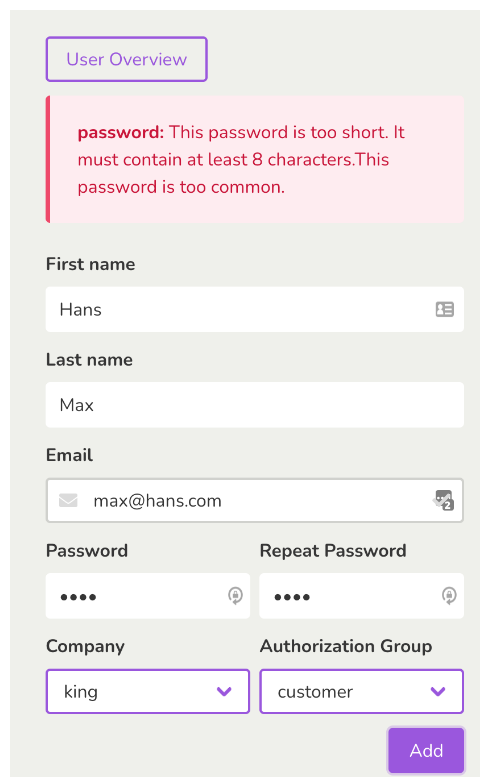

Figure C.5: Backend validates password

Feedback if input is invalid One input that needs to be checked is on the login page. We created two kind of inputs. A direct feedback if a field is missing or an authentication failed notification if the credentials are wrong.

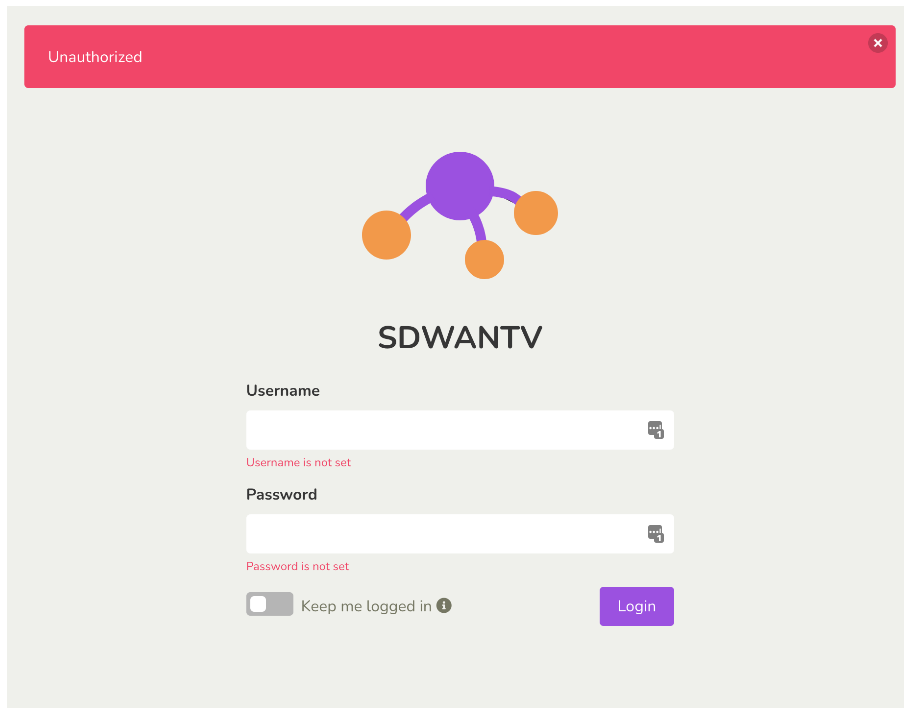

Figure C.6: Login failure message

C.3 Fault tolerance, vManage data

To be tested

1. vManage response validation.
2. Exception management during the vManage fetch.

Test result

vManage response validation The response body of vManage API calls is validated in the code with the Python jsonschema [29] library before we access the property.

Exception management All exceptions thrown are handled directly in a multiple except statements. We ensure with a final block, that the task is aborted in a proper manner.

```
...code
   except aiohttp.ClientConnectionError:
   except aiohttp.ClientResponseError:
   except ValueError:
   except ValidationError:
   finally:
   return result
```

Listing C.1: Exception management backend tasks

C.4 Maturity

To be tested

1. Ratio of successful requests.

Test result

Ration of successful requests If vManage is correctly configured and SDWANTV has access to the vManage API endpoint, we did experience in our measurements a success rate of 99,37% of all running tasks. Therefore our goal to reach a 90% success rate has been achieved.

count	status
2	skipped
24	failed
30	warning
8864	successful

Figure C.7: Statistics of all task runns

C.5 Understandability

To be tested

1. A user not familiar with the application can understand it without a tutorial.

Test result

New user understands it A person not familiar with the application and the context the application is built for had no problem to use all the provided features.

To really understand the different elements on the page however the person, who tested the application, needed some background knowledge on the domain.

We consider the test as successful.

C.6 Failure management

To be tested

1. User is getting notified if an exception happens.

Test result

User is getting notified In a first step the system runs under normal circumstances. The status display in the frontend shows the correct state.

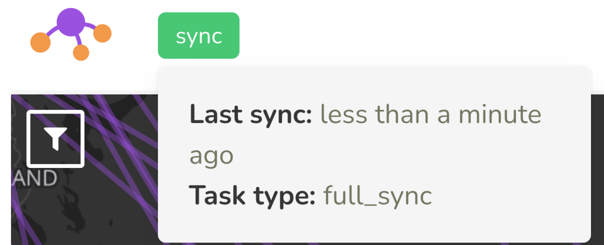

Figure C.8: Sync state if system runs successful

If we remove the internet connection to prevent SDWANTV to fetch from the vManage API the sync state changes.

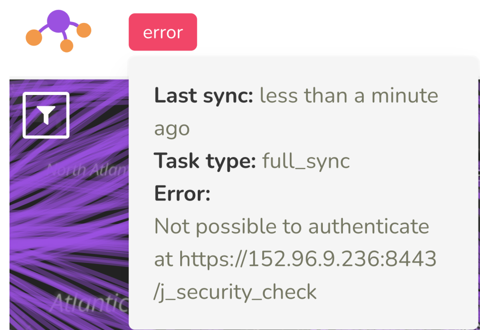

Figure C.9: Sync is in error state

If we shutdown the backend we will receive a no sync state.

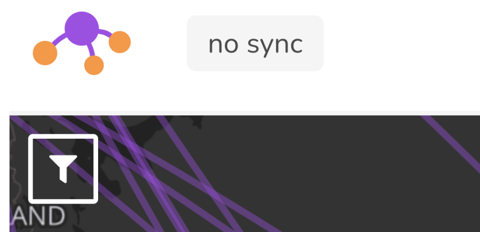

Figure C.10: Sync information could not be fetched

C.7 Time behaviour

To be tested

1. Applying policy.

Test result

Applying policy The initial state is a full mesh topology and no policy is applied.

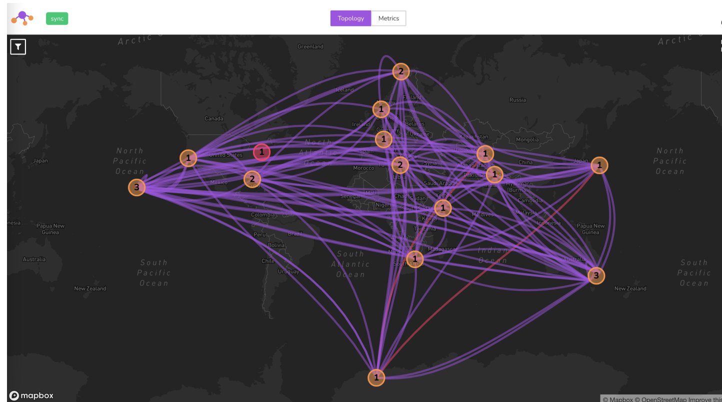

Figure C.11: Full match without policy applied

Over the vManage user interface we applied a policy. After the policy was applied to the nodes. We fetch each 30 seconds the topology from vmanage and ingest the data to the database. After that the messages are propagated directly to the frontend. The practical test confirms the the runtime calculation and we receive the first updates within already in 30 seconds range. Nevertheless, this can vary depending on the occurrence on the change in relation with our 30 seconds fetch interval.

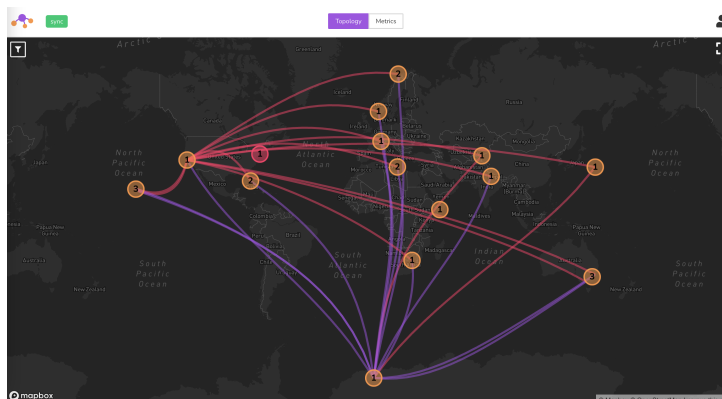

Figure C.12: Policy applied 30s later

The 30s is below the required two minute threshold and therefore this test is successful.

C.8 Response time

To be tested

1. Initial render duration.

Test result

Initial render We measured the loading time with the chrome performance measuring tool [11]. The result shows that the performance is far below the required threshold. The web application shows the first render in only 100ms and after 3s we have rendered the whole topology. So the initial render is much faster than the render of the topology. If the caching is active, the first render would be even faster than 100ms.

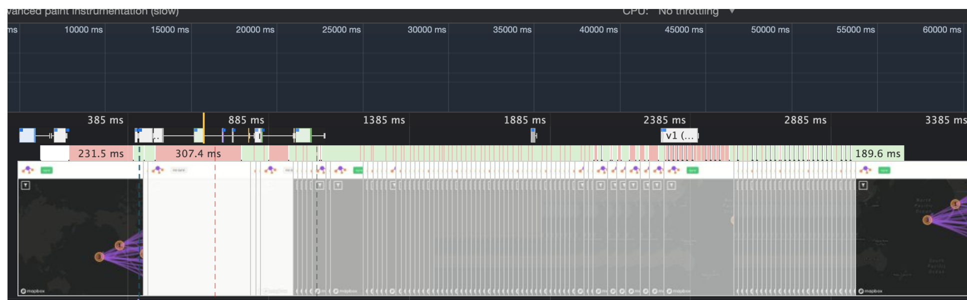

Figure C.13: Render performance

C.9 Supportability

To be tested

1. Complexity of the application.
2. Sonarqube evaluation

Test result

Complexity of the application Our project has combined only 10k to 20k lines of code and a test coverage is at least 75%. Because of these properties we can assume that a single developer still has an overview over the whole code base and is perfectly able to fix a bug in under 48h.

Sonarqube evalutation We used sonarqube to evaluate our code base for maintainability. The result should be enjoyed with a grain of salt because sonarqube does only check certain rules and does not cover everything.

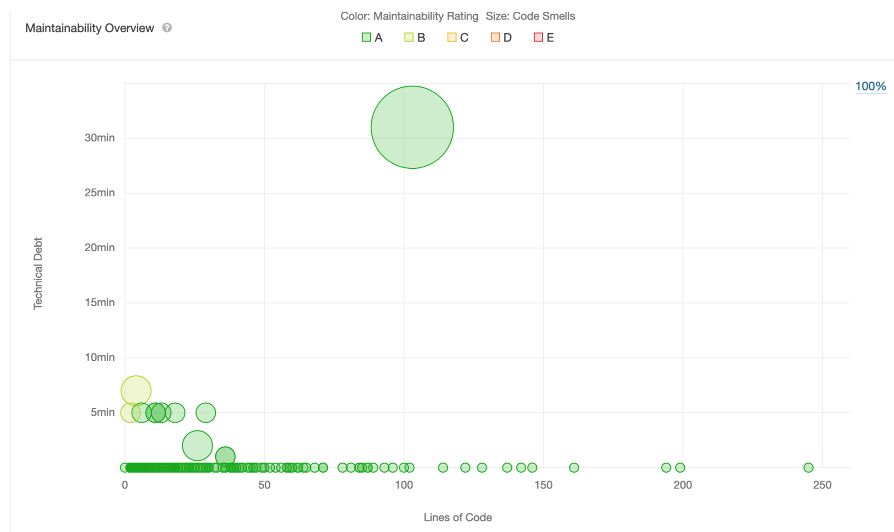

Figure C.14: Sonarqube evaluation frontend

Figure C.15: Sonarqube evaluation backend

C.10 Portability

To be tested

1. 12 Factor are applied.
2. Kubernetes deployment and Helm chart

Test result

12 Factor are applied In section 4.4.1 we listed how we made our application cloud ready and therefore portable.

Kubernetes deployment and Helm chart During the bachelor thesis we put emphasis on an easy deployment and a cloud ready application. We used technologies like Kubernetes and Helm to make it easier to deploy and run the software system as many environments as possible.

C.11 Scalability

To be tested

1. Time difference if more resources are in the topology.
2. Kubernetes deployment
3. Scales in O(n)

Test result

Time difference We let our tasks run under two different precondition. The first run we will do with a full-mesh topology. The full-mesh will have 25 nodes and approximately 750 tunnels. The second run has a policy applied, which will have 25 nodes and approximately 350 tunnels. As we can see a full mesh takes approx. 18 seconds to run and a topology with only the half of the resources takes only the half of the time to fetch with approx. 8 seconds.

edit	1951	full_sync	2020-12-12 13:21:54.497697+00	2020-12-12 13:21:54.777181+00	2020-12-12 13:22:11.952644+00	successful	*NULL*
edit	1952	full_sync	2020-12-12 13:22:24.49496+00	2020-12-12 13:22:24.748268+00	2020-12-12 13:22:42.246791+00	successful	*NULL*

Figure C.16: Full match fetching speed

edit	2151	full_sync	2020-12-12 15:02:26.248738+00	2020-12-12 15:02:26.399588+00	2020-12-12 15:02:34.277993+00	successful	*NULL*
edit	2152	full_sync	2020-12-12 15:02:56.249915+00	2020-12-12 15:02:56.367721+00	2020-12-12 15:03:03.487339+00	successful	*NULL*

Figure C.17: Applied policy fetching speed

Kubernetes deployment Since we use Kubernetes it is easy to add more instances of the components to cope with the growing number of requests. We could even go one step further and apply auto scaling, which will spin up extra pods if the application passes some thresholds.

Scales in O(n) To find a adequate explanation of our scaleability we need to differentiate between the fetching tasks and our backend service. While we are quite confident that our backend technologies smoothly scales in O(n), it is not possible to scale the fetching tasks of a full match topology in O(n).

If we have **n** nodes and every nodes has a connection to every other one, we will end up with $\frac{n*(n-1)}{2}$ connections. In Big O notation this would scale in O(n^2). There is no way that we can bypass the mathematical limitations and reduce it to O(n). On top of to the logical boundaries, does our system dependent on the scale ability of vManage itself. So the best what we could do, and what we did, is to detect the bottlenecks on our system and reduce the computing time as much as possible.

Appendix D

Wireframe & Prototypes

D.1 Wireframes

D.1.1 UC2: User management

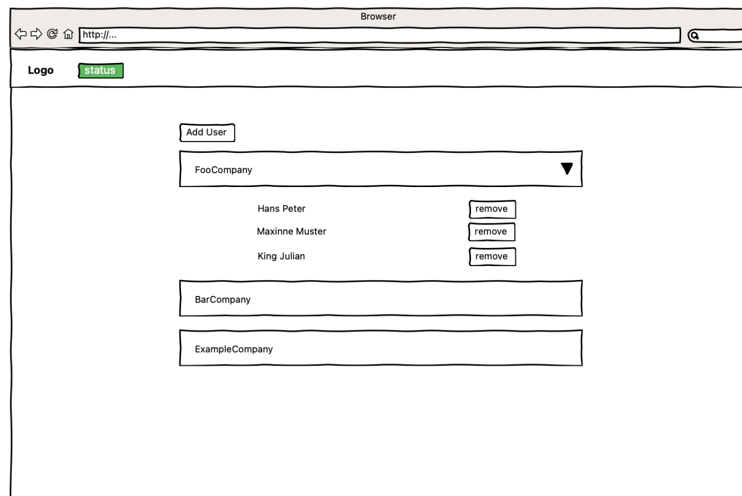

Figure D.1: UC2 wireframe

D.1.2 UC2.1: Update customer

Figure D.2: UC2.1 wireframe

Figure D.3: UC2.1 wireframe 2

D.1.3 UC3: Add filter between two sides

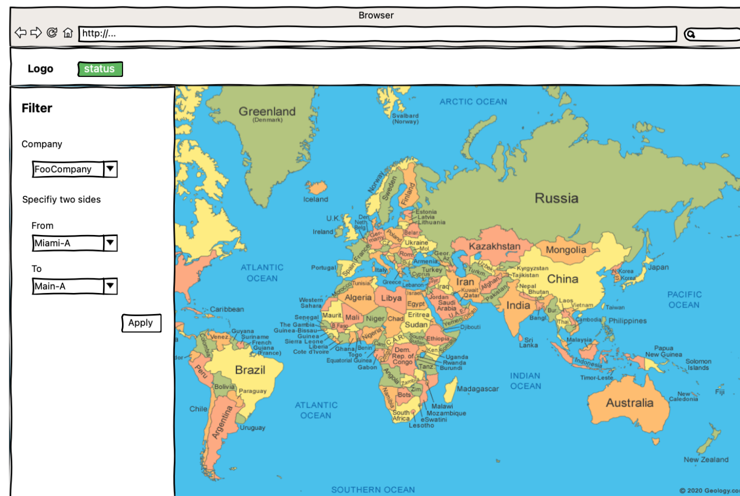

Figure D.4: UC3 wireframe

D.1.4 UC4: Acknowledge tunnel down changes

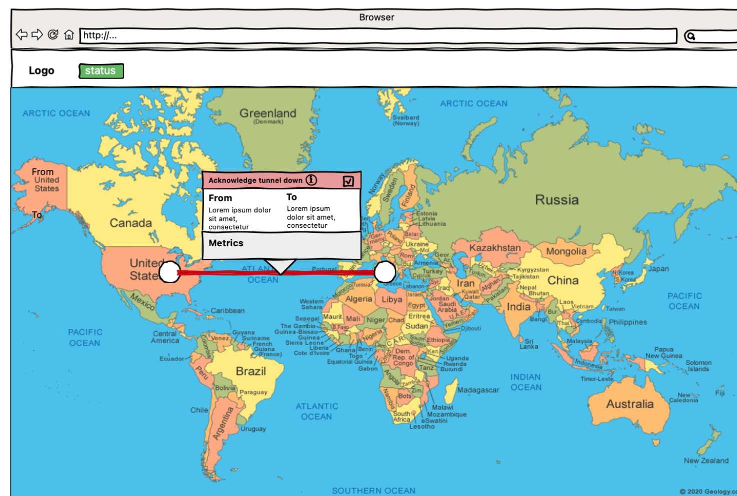

Figure D.5: UC4 wireframe

D.1.5 UC5: Display the metrics of each tunnel

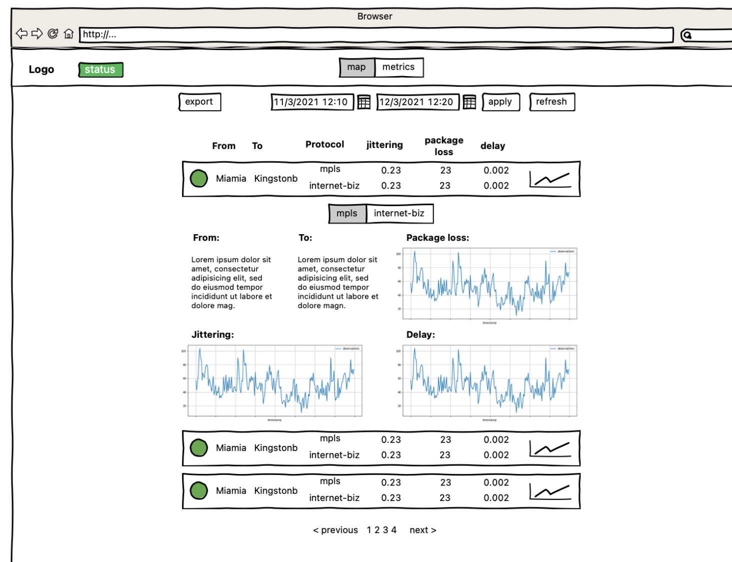

Figure D.6: UC5 wireframe

D.1.6 UC8: Global Filtering

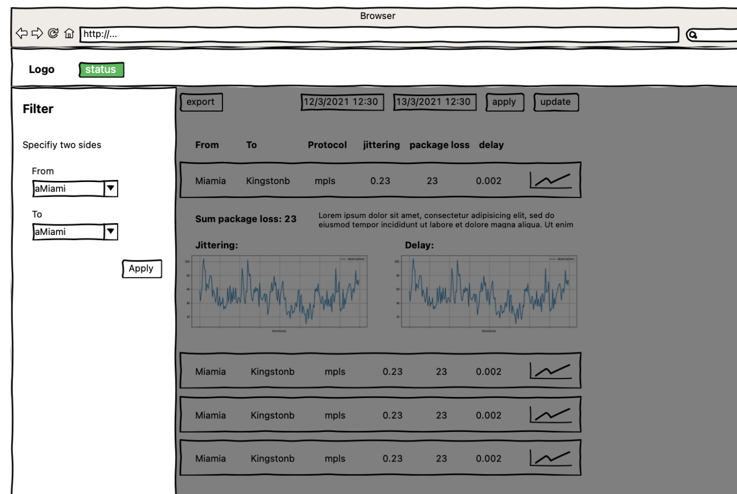

Figure D.7: UC8 wireframe

Figure D.8: UC8 wireframe 2

D.2 Prototypes

D.2.1 UC2: User management

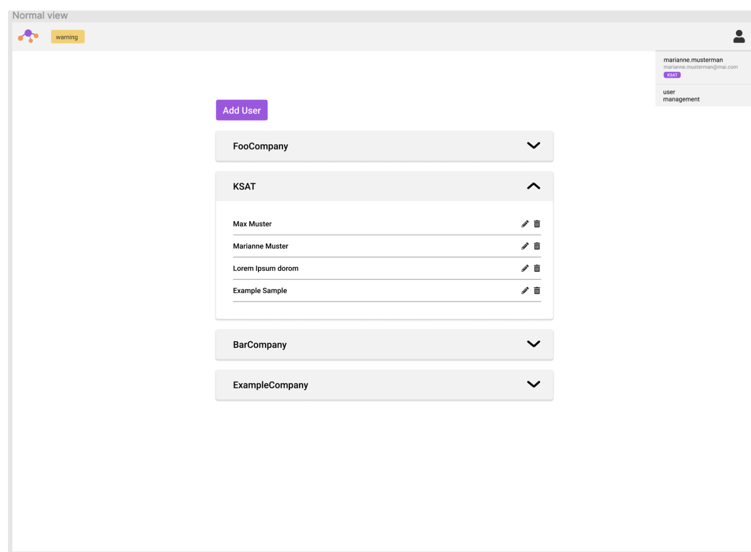

Figure D.9: UC2 User overview

D.2.2 UC2.1: Update user

Figure D.10: UC2.1 Update user

D.2.3 UC5: Display the metrics of each tunnel

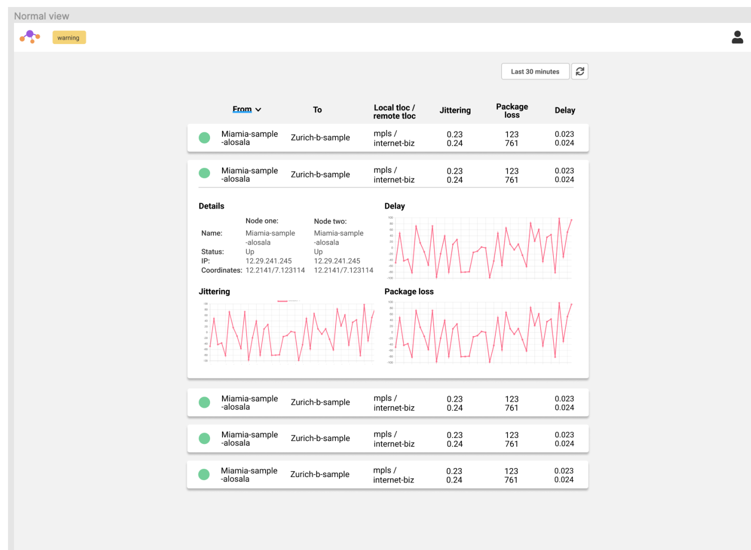

Figure D.11: UC5 display metrics

D.2.4 UC5.1: Specify historical metrics time range

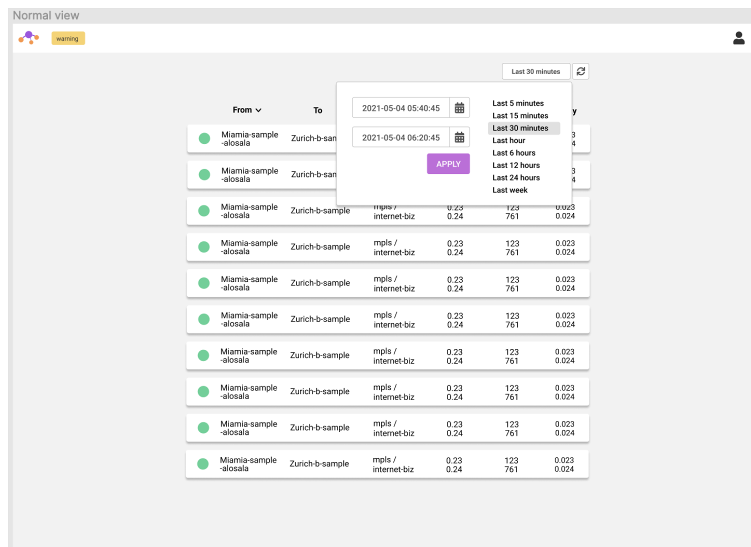

Figure D.12: UC5 display metrics

Appendix E

Risk analysis

E.1 Risk Analysis Table

#	Title	Description	Max damage [h]	Probability of occurence	Weighted damage (h * prob)	Prevention	Behaviour on entry	had entered	What we did on entry
R1	Interpersonal conflicts	Disputes and different opinions can worsen the communication and lead to a bad teamwork	64h	2%	1.28	Clarify the goals and expectation of each team member for this project, keep the work structured and the feature growth small	Asking the supervisor to help us minimizing the conflict.	FALSE	
R2	Outage of team member	Long-term outage of team member because of an accident or quarantine	80h	2%	1.6	Enable remote meetings over Microsoft Teams. Compliance with the regulation of the federal office of public health	Check with the supervisor if we can reduce the feature size of our project.	FALSE	
R3	Not testable	Parts of our application are not testable or the effort to test it is to high	8h	50%	4	We try to isolate the non testable components from the rest of our application	We test the components with manual system tests.	TRUE	Because some components from frontend applications are not testable, we were able to test them over a manual system test
R4	Customer requirements not met	We deliver a different product than was expected	32h	10%	3.2	We've chosen a user centered design approach. The customer is involved at every stage of the project and will give us regular feedback. Mockups and wireframes to visualize our ideas in a clearer way.	Check with the supervisor and the customer, what action can be done on our side to come back on track	FALSE	
R5	Overstrain of the technical complexity	If we are getting problem with a new technology, we need more time than anticipated	16h	25%	2	Allocating some time to get familiar with unknown technology. Ask the advisor for technical help	Get help of external experts.	TRUE	Implementing WebSockets and DeckGL was difficult. We had to invest more time than planned. This led to a delay of one week. For DeckGL we needed the helf of an expert.
R6	Lifecycle of dependencies	We are dependent on many third-party technologies, which's APIs might change over time	0	100%	0	Because our project only takes one semester, we will stick with a certain version set. We will never do a major library update.	None	TRUE	We sticked to the same version and did not upgrade versions althought newer stable versions were available
R7	Lack of security	We might need to store some security relevant information like passwords or tokens. This can lead to a problem in a live environment with sensible data.	16h	10%	1.6	We will stick to the best practice, that are recommended by the community.	Get help of external experts.	FALSE	

#	Title	Description	Max damage [h]	Probability of occurence	Weighted damage (h * prob)	Prevention	Behaviour on entry	had entered	What we did on entry
R8	vManage API is not deterministic	Our system heavily depends on the vManage API. If the API suddenly changes we will face problems.	16h	15%	2.4	Carefully read the vManage API documentation in order to differentiate between intentional behaviour and unintentional behaviour.	Asking the supervisor to help us to bring the vManage API back in a predictive state.	FALSE	
R9	vManage outage	If upgrading vManage fails this could lead to an outage of vManage.	24h	20%	4.8	Discuss with the supervisor when is the best time to upgrade so we wouldn't get blocked heavily.	Perform tasks that do not depend on vManage first.	TRUE	vManage was down for nearly 2 weeks. We solved tasks that don't depend on vManage first
R10	vManage is not scaleable	The topology can increase and we still want to provide a fast software	16h	30%	4.8	Understand the vManage API as good as possible.	Ask technical experts for help.	TRUE	We implemented coroutine fetching to fetch API data parallel and speed up the task run time.
R11	Faulty topology state	Because we deliver partial updates via Websocket this could lead to updates being lost and a wrong topology on the client side	12h	30%	3.6	Writting tests for Websocket behaviour, to prevent faulty behaviour	We will change the behaviour of the update functionality	TRUE	It was not possible to find out the state that is on the client, which required to implement an in-memory state in the Websocket consumer.
R12	Time format incompatibility	We use timestamps in order to fetch the resources from the vManage API	4h	20%	0.8	Using timezone aware timestamps.	Asking the supervisor to help us.	FALSE	
R13	No real-time metrics data available	It might be that there are no real-time metrics data available in the vManage API	8h	40%	3.2	Perform a proper analysis of the vManage API and check what metrics data are available	Define an alternative scenario that will be used to bring near real time	FALSE	
R14	Heavy deployment process	The newly created Continuous Deployment process is heavy and takes a long time to complete	8h	20%	1.6	Check best-practices how to use GitLab CI/CD to enable a Continuous Deployment process.	Increase the runner resources or ask someone for help who has experiences with CI/CD.	FALSE	
R15	vManage API version incompatibility	When upgrading the vManage to a newer version it could be that there are incompatible API changes	6h	15%	0.9	With every vManage API call we check if the response contains all the data as we defined it in a json schema.	Investigate what the problem is and check if we can solve it or create a workaround	FALSE	

Appendix F

vManage API Request & Responses

F.1 Devices list response


```
{
   "header":{
      ...
   },
   "data":[
      {
         "deviceId":"10.255.255.133",
         "system-ip":"10.255.255.133",
         "host-name":"Customer-king-Hawaii",
         "reachability":"reachable",
         "status":"normal",
         "personality":"vedge",
         "device-type":"vedge",
         "timezone":"UTC +0000",
         "device-groups":[
            "\"king\""
         ],
         "lastupdated":1604841099143,
         "bfdSessionsUp":38,
         "domain-id":"1",
         "board-serial":"01DB3829",
         "certificate-validity":"Valid",
         "max-controllers":"0",
         "uuid":"C1111X-8P-FGL2346L61N",
         "bfdSessions":"38",
         "controlConnections":"3",
         "device-model":"vedge-C1111X-8P",
         "version":"16.12.02r.0.23",
         "connectedVManages":[
            "\"10.255.255.1\""
         ],
         "site-id":"32",
         "ompPeers":"2",
         "latitude":"19.5429",
         "longitude":"-155.6659",
         "isDeviceGeoData":True,
         "platform":"x86_64",
         "uptime-date":1597931460000,
         "statusOrder":4,
         "device-os":"next",
         "validity":"valid",
         "state":"green",
         "state_description":"All daemons up",
         "model_sku":"None",
```

```
        "local-system-ip":"10.255.255.133",
        "total_cpu_count":"4",
        "linux_cpu_count":"4",
        "testbed_mode":False,
        "layoutLevel":4
     },
  ]
}
```


Listing F.1: Devices list sample response

F.2 Tunnel list response


```
{
  "header":{
     ...
  },
  "data":[
     {
          "src-ip": "152.96.9.247",
          "dst-ip": "152.96.9.11",
          "vdevice-name": "10.255.255.254",
          "color": "biz-internet",
          "src-port": 12346,
          "createTimeStamp": 1617817314401,
          "system-ip": "10.255.255.251",
          "dst-port": 12406,
          "site-id": 251,
          "transitions": 0,
          "vdevice-host-name": "RS-Main-a",
          "local-color": "biz-internet",
          "detect-multiplier": "7",
          "vdevice-dataKey": "10.255.255.254-biz-internet-10.255.255.251-biz-internet-ipsec",
          "@rid": 948,
          "local-if-desc": "",
          "vmanage-system-ip": "10.255.255.254",
          "local-remote-ifname": "GigabitEthernet2-GigabitEthernet0/0/0",
          "proto": "ipsec",
          "lastupdated": 1621008895791,
          "remote-if-desc": "",
          "tx-interval": 1000,
          "state": "up",
          "uptime-date": 1620730740000
      },
   ]
}
```


Listing F.2: Tunnel list response

F.3 IPsec inbound response


```
{
   "header":{
      ...
   },
   "data":[
      {
         "dest-ip":"10.8.0.162",
         "source-port":12406,
         "vdevice-name":"10.255.255.162",
         "vdevice-host-name":"RS-CapeTown-b",
         "remote-tloc-address":"10.255.255.111",
         "negotiated-encryption-algo":"AES-GCM-256",
         "dest-port":12386,
         "vdevice-dataKey":"10.255.255.162-10.255.255.162",
         "local-tloc-address":"10.255.255.162",
         "lastupdated":1604841086657,
         "source-ip":"10.8.0.111",
         "remote-tloc-color":"mpls",
         "local-tloc-color":"mpls"
      },
   ]
}
```


Listing F.3: Devices list sample response

Appendix G

vManage API Metrics Request & Responses

G.1 Metric history request aggregation query


```
{
    "query": {
        "condition": "AND",
        "rules": [
            {
                "value": [
                    "168"
                ],
                "field": "entry_time",
                "type": "date",
                "operator": "last_n_hours"
            },
            {
                "value": [
                    "10.255.255.122"
                ],
                "field": "local_system_ip",
                "type": "string",
                "operator": "in"
            },
            {
                "value": [
                    "10.255.255.131"
                ],
                "field": "remote_system_ip",
                "type": "string",
                "operator": "in"
            },
             {
                "value": [
                    "mpls"
                ],
                "field": "local_color",
                "type": "string",
                "operator": "in"
            },
            {
                "value": [
                    "mpls"
                ],
                "field": "remote_color",
                "type": "string",
                "operator": "in"
            }
```

```
        ]
    },
    "aggregation": {
        "field": [
            {
                "property": "name",
                "sequence": 1
            }
        ],
        "histogram": {
            "property": "entry_time",
            "type": "hour",
            "interval": 1,
            "order": "asc"
        },
        "metrics": [
            {
                "property": "jitter",
                "type": "avg"
            },
            {
                "property": "loss_percentage",
                "type": "avg"
            },
            {
                "property": "latency",
                "type": "avg"
            }
        ]
    }
}
```


Listing G.1: Metric history request query

G.2 Metric history response


```
{
    "header": {
       ...
    },
    "entryTimeList": [
        1615280400000,
        ...
    ],
    "data": [
        {
            "entry_time": 1615280400000,
            "count": 1,
            "jitter": 0,
            "loss_percentage": 0,
            "latency": 2
        },
        ...
    ]
}
```


Listing G.2: Metric history response

G.3 Live Metric response


```
{
    "header": {
        ...
    },
    "data": [
        {
            "src-ip": "10.8.0.121",
            "dst-ip": "10.8.0.111",
            "tx-data-pkts": "0",
            "vdevice-name": "10.255.255.121",
            "src-port": 12406,
            "dst-port": "12366",
            "remote-color": "mpls",
            "remote-system-ip": "10.255.255.111",
            "mean-latency": 2,
            "total-packets": "664",
            "loss": "0",
            "mean-jitter": 0,
            "ipv6-rx-data-pkts": "0",
            "average-latency": "2",
            "index": "0",
            "sla-class-index": "0",
            "vdevice-host-name": "RS-Bern-a",
            "local-color": "mpls",
            "mean-loss": 0,
            "vdevice-dataKey": "10.255.255.121-0",
            "proto": "ipsec",
            "lastupdated": 1615286246756,
            "average-jitter": "0",
            "rx-data-pkts": "0",
            "ipv6-tx-data-pkts": "0"
        },
        ...
    ]
}
```


Listing G.3: Live metric response