

Term project

---

# SD-WAN Topology Viewer

---

Eastern Switzerland University of Applied Sciences  
Department of Computer Science

Period: 14.09.2020 - 18.12.2020

**Authors** Dominic Gabriel  
Lars Barmettler

<b>Supervisor</b>	Prof. Laurent Metzger
<b>Co-Supervisor</b>	Jessica Hilti
<b>Industry Partners</b>	KSAT Satellite Services Insoft Services

## Task description

### Description of the thesis: SD-WAN topology viewer

The Cisco Viptela SD-WAN solution builds by default a full-mesh of IPsec tunnels between all the sites of the inventory of the SD-WAN controller. This full-mesh topology is not going to scale in big networks and a policy is often used to restrict the tunnels between sites.

It is important to have a way to visualize at any time the topology of the IPsec tunnels built between sites. The goal of that SA is to build an application that displays the current IPsec tunnel topology in an SD-WAN network. This application is meant to be used in a NOC to monitor the tunnel health and alert when a specific tunnel is going down.

The following use cases should be addressed by the application:

Use Case	optional/mandatory
UC1: Monitoring the topology The IPsec tunnels should be displayed in a meaningful way	mandatory
UC1.1: View the node information The node information should be displayed in the GUI	optional
UC1.2: Display connection metrics The live connection metrics (Jitter, Packet loss, delay) should be displayed in the GUI	optional
UC1.3: Display bandwidth with IP-Sec Tunnels The current bandwidth used by a tunnel should be displayed in the GUI	optional
UC 1.4: Toggle full screen A toggle button allows the user to change the view to full screen.	optional
UC2: Apply customer filter It should be possible to select a customer in the gui and only the tunnels between sites where the VPN of that selected customer is present should be displayed	mandatory
UC3: Manage users for companies A customer can login to the application and should only see the tunnels between sites where the customer is present.	optional
UC4: Apply connection filter It should be possible to select a connection and to see only that specific connection in the GUI.	optional
UC5: Display of the path from router to destination Two WAN Edges can be selected and the path between the two selected devices should be displayed.	optional
UC6: Manage metric alarms When a tunnel goes down, an alarm should be raised in Vmanage and the tunnel should be highlighted in red. The tunnel should only be removed from the topology if an operator approves the removal.	optional
UC6.1: Send alarm to external syslog server When the jitter, the delay and the packet drops are higher than a certain threshold, an alarm should be raised and the link color should become orange.	optional

Rapperswil, 10.12.2020

Prof. Laurent Metzger



---

## Abstract

Software defined WAN (SD-WAN) is a trending new technology that is emerging fast. 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 provides a web application that is primarily designed for configuration. It provides 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 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 more emphasis on a visual representation of the topology. It not only displays both devices and IPSec tunnels, but also the states of both and the metrics of the IPSec tunnels. With a filter, the user is able to only display those resources they are interested in. Because of the simple user experience, a network administrator can quickly detect failures and anomalies in the topology in one view.

With an internal topology state representation based of the information gathered over Ciscos vManage API and stored in a PostgreSQL database, the Django backend offers a REST Application Programming Interface (API) for an improved user interface to a state-of-the-art single page application written in React. Depending on the size of the topology, the tool propagates vManage state changes to the frontend in under one minute and therefore can be rated as a live monitoring.

---

## Management Summary

### Baseline

Globally distributed companies are often faced with an increasing complexity of their 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 brings the advantage of managing the network by code and perform network segmentation and routing based on policies and rules. This not only reduces the operational effort but also makes automation and cloud-based management possible.

Our software has the goal to improve the monitoring experience of the network infrastructure based on Cisco's 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 Cisco SD-WAN solution. We fetch the data about the infrastructure from Cisco vManage and display the topology 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. Out of this we created a functional and non-functional requirement catalog, which was the baseline for our software.

In the second step we analyzed the system our software is built upon. Our software heavily relies on Cisco's vManage API. To figure out which data needs to be fetched 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 PostgreSQL database and is updated by scheduled tasks which fetches the latest data from the vManage API.

### Results

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

The user can limit the complexity of the topology to only see a subset of it using a filter. By clicking on a node or an edge the user can see detailed information about the resource. In under one minute the changes in the SD-WAN are detected and propagated to the user interface. If an error happens during the propagation or the fetching from vManage, a status bar, containing the error message, in the frontend will inform the user.

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 one week.



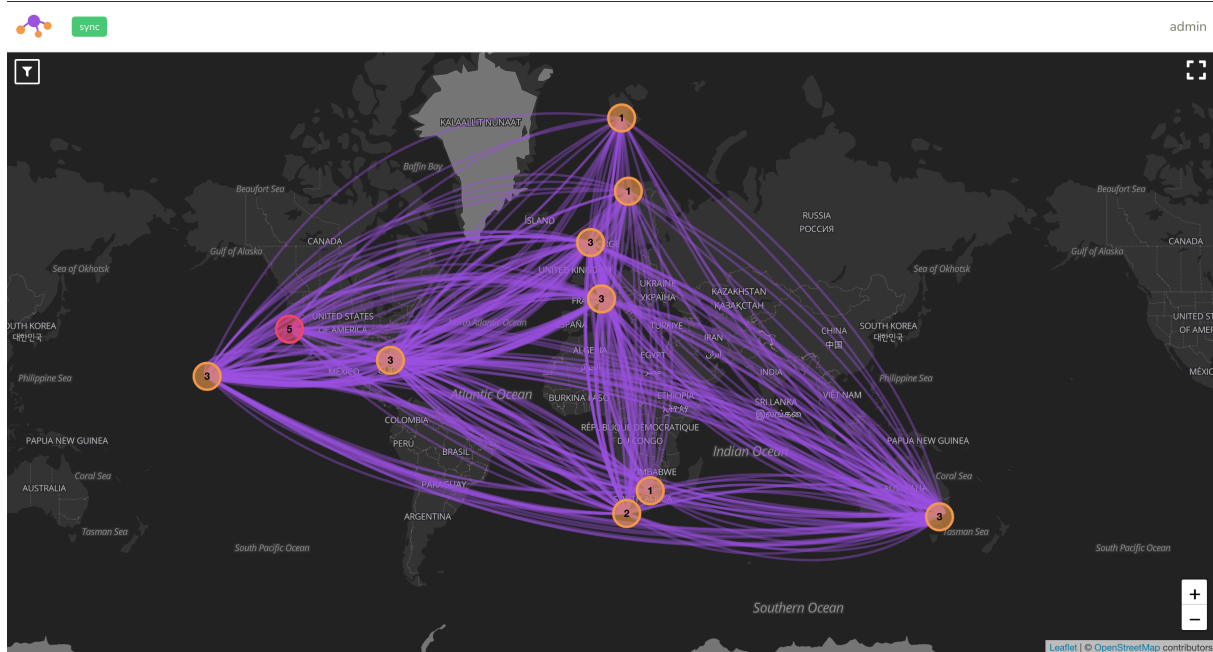


Figure 1: SD-WAN Topology Viewer

## Forecast

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

One of these features, which is interesting for companies with customers connected to their network, is a multitenancy functionality. As a result, it is possible to add new users to the system, who only have access to the subset of the topology they are allowed to see.

Next to the multitenancy support, more features could be added. For instance, it would be helpful if a notification system for metric data directly informs the responsible person or the path of the data flow from a source to a destination could be visualized.

If one would think even further, this tool could completely replace Cisco's vManage and provide a more user-friendly solution for the not only 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
- Technical support: INS Team Members
- Possibility for this thesis: KSAT, Thomas Torsteinsen
- Knowledge support: Insoft Services
- Technical support: Mikro Stocker (IFS)
- Architecture documentation review: Prof. Olaf Zimmermann
- Proofreading: AnneMarie O'Neill

---

# Contents

---

<b>Glossary and List of Abbreviations</b>	<b>xi</b>
<b>List of Figures</b>	<b>xiii</b>
<b>List of Tables</b>	<b>xv</b>
 <b>I Technical Report</b>	 <b>1</b>
<b>1 Technical Report</b>	<b>2</b>
1.1 Introduction and Overview . . . . .	2
1.1.1 Problem . . . . .	2
1.1.2 Goals . . . . .	2
1.1.3 Limitations . . . . .	3
1.1.4 Work structure . . . . .	3
1.2 Evaluation . . . . .	3
1.2.1 Information acquisition . . . . .	3
1.3 Concept . . . . .	4
1.4 Solution . . . . .	5
1.4.1 Implementation . . . . .	6
1.5 Conclusion . . . . .	6
1.6 Forecast . . . . .	7
1.6.1 Non-technical improvements . . . . .	7
1.6.2 Technical improvements . . . . .	7
1.6.3 Technical debt . . . . .	7
 <b>II Project Documentation</b>	 <b>9</b>
<b>2 Requirements Specification</b>	<b>10</b>
2.1 Thesis Requirements . . . . .	10
2.2 Actors . . . . .	10
2.2.1 Administrator . . . . .	10
2.2.2 Customer . . . . .	10
2.2.3 System . . . . .	10
2.3 Use Cases . . . . .	11
2.3.1 UC1: Monitoring the topology . . . . .	11
2.3.2 UC1.1: View the node information . . . . .	12
2.3.3 UC1.2: Display connection metrics . . . . .	12

2.3.4	UC1.3: Display bandwidth of IPsec Tunnels . . . . .	12
2.3.5	UC1.4: Toggle full screen . . . . .	12
2.3.6	UC2: Apply customer filter . . . . .	13
2.3.7	UC3: Manage users for companies . . . . .	13
2.3.8	UC4: Apply Connection filter . . . . .	13
2.3.9	UC5: Display of the path from router to the destination . . . . .	14
2.3.10	UC6: Manage Metric Alarms . . . . .	14
2.3.11	UC6.1: Send alarm to the external syslog server . . . . .	14
2.4	Non functional requirements . . . . .	15
2.4.1	Functionality . . . . .	15
2.4.2	Reliability . . . . .	15
2.4.3	Usability . . . . .	15
2.4.4	Efficiency . . . . .	15
2.4.5	Supportability . . . . .	15
2.4.6	Portability . . . . .	16
2.4.7	Scalability . . . . .	16
<b>3</b>	<b>Analysis</b>	<b>17</b>
3.1	Domain model . . . . .	17
3.2	Data model . . . . .	18
3.2.1	auth_user . . . . .	18
3.2.2	auth_group . . . . .	18
3.2.3	edge . . . . .	19
3.2.4	metric . . . . .	19
3.2.5	node . . . . .	19
3.2.6	sync . . . . .	19
3.2.7	company . . . . .	19
3.3	Cisco vManage API analysis . . . . .	19
3.3.1	List devices . . . . .	20
3.3.2	OMP Services . . . . .	21
3.3.3	IPsec connections . . . . .	22
3.3.4	Events . . . . .	23
3.3.5	Metrics . . . . .	24
<b>4</b>	<b>Architecture &amp; Design Specification</b>	<b>25</b>
4.1	Scope . . . . .	25
4.2	Design . . . . .	25
4.2.1	Container Diagram . . . . .	26
4.2.2	Goals . . . . .	26
4.2.3	Twelve Factors . . . . .	27
4.3	Design Decisions . . . . .	30
4.3.1	Django Backend . . . . .	30
4.3.2	React Frontend . . . . .	30
4.3.3	PostgreSQL Database . . . . .	30
4.4	Software Architecture . . . . .	31
4.4.1	Frontend . . . . .	31
4.4.2	Communication . . . . .	33
4.4.3	Backend . . . . .	33
4.5	Sequence Diagrams . . . . .	34
4.5.1	Fetch topology . . . . .	34
4.5.2	Fetching metrics . . . . .	35
4.6	Deployment . . . . .	36

4.6.1	Client & SDWANTV Frontend . . . . .	36
4.6.2	Traefik . . . . .	36
4.6.3	Frontend Server . . . . .	36
4.6.4	Backend Server . . . . .	37
4.6.5	PostgreSQL Database . . . . .	37
4.6.6	Beat . . . . .	37
4.6.7	Redis . . . . .	37
4.6.8	Celery Task Engine . . . . .	37
4.6.9	vManage API . . . . .	37
4.6.10	Deployment with Docker-compose . . . . .	37
4.7	Tools & Frameworks . . . . .	38
4.7.1	Frontend . . . . .	38
4.7.2	Backend . . . . .	38
4.7.3	Communication . . . . .	39
4.7.4	Deployment . . . . .	39
4.8	UI-Design . . . . .	39
4.8.1	Tools . . . . .	39
4.8.2	Mock-up . . . . .	39
4.8.3	MVP design . . . . .	40
4.8.4	Prototype end of construction . . . . .	40
<b>5</b>	<b>Implementation &amp; Testing</b>	<b>41</b>
5.1	Implementation . . . . .	41
5.1.1	Python Django Backend . . . . .	41
5.1.2	Frontend . . . . .	47
5.2	Automated Testing . . . . .	50
5.2.1	Unit Tests . . . . .	50
5.2.2	Integration Tests . . . . .	50
5.2.3	Test Coverage . . . . .	50
5.3	Manual Testing . . . . .	51
5.3.1	System Tests . . . . .	51
5.3.2	Non functional Requirements Tests . . . . .	51
<b>6</b>	<b>Project Management</b>	<b>52</b>
6.1	Project organization . . . . .	52
6.2	Project Meetings . . . . .	52
6.3	Process Model . . . . .	53
6.4	Software Development Process . . . . .	53
6.5	Releases . . . . .	56
6.6	Milestones . . . . .	56
6.7	Project Plan . . . . .	57
6.8	Risk Analysis . . . . .	58
6.9	Logging . . . . .	59
6.10	Time Report . . . . .	59
6.11	Quality Control . . . . .	60
6.11.1	Linting . . . . .	60
6.11.2	Definition of Done . . . . .	61
6.11.3	Coding Guidelines . . . . .	61
6.12	MVP . . . . .	61
<b>7</b>	<b>Project monitoring</b>	<b>62</b>
7.1	Project reporting . . . . .	62

7.1.1	Working times	62
7.1.2	Project phases	63
7.1.3	Task types	64
7.1.4	Milestones	65
7.2	Code statistics	66
<b>III</b>	<b>Appendix</b>	<b>67</b>
<b>A</b>	<b>User Manual</b>	<b>68</b>
A.1	Installation	68
A.2	Deployment	69
A.3	Deployment configuration	69
A.4	Configure https certificates	71
A.5	Operational tasks	71
A.6	Docker configuration	73
A.7	Termination	73
A.8	Docker-compose Yaml	74
A.9	Traefik config file	76
<b>B</b>	<b>Systemtest protocol</b>	<b>77</b>
B.1	UC1: Monitoring topology: Test 1	78
B.2	UC1: Monitoring topology: Test 2	80
B.3	UC1: Monitoring topology: Test 3	82
B.4	UC1: Monitoring topology: Test 4	83
B.5	UC1: Monitoring topology: Reported bugs	84
B.6	UC1.1: View node information: Test 1	85
B.7	UC1.2: Display connection metrics: Test 1	86
B.8	UC1.4: Toggle fullscreen: Test 1	87
B.9	UC2: Apply customer filter: Test 1	88
B.10	UC4: Apply connecton filter: Test 1	89
<b>C</b>	<b>Non functional requirement testprotocol</b>	<b>90</b>
C.1	Security	91
C.2	Fault tolerance, user data	92
C.3	Fault tolerance, vManage data	93
C.4	Maturity	93
C.5	Understandability	93
C.6	Failure management	94
C.7	Time behaviour	95
C.8	Response time	96
C.9	Supportability	97
C.10	Portability	97
C.11	Scalability	97
<b>D</b>	<b>Mockup &amp; Wireframe</b>	<b>98</b>
D.1	First Mockup	98
D.2	MVP Design Wireframe	100
<b>E</b>	<b>Risk analysis</b>	<b>102</b>
E.1	Risk Analysis Table	102
<b>F</b>	<b>vManage API Request &amp; Responses</b>	<b>104</b>

F.1	Devices list response . . . . .	104
F.2	IPsec inbound response . . . . .	105
F.3	Event aggregation query . . . . .	106
F.4	Metrics aggregation query . . . . .	107
F.5	Metrics response . . . . .	108

---

## Glossary and List of Abbreviations

---

**API** Application Programming Interface. ii, xi, 6, 15, 19–21, 26, 28, 30, 33, 35–37, 44, 46, 47, 50, 58, 78, 92, *see* Application Programming Interface

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

**CI** Continuous Integration is the practice of merging all developers' working copies to a shared mainline several times a day.. 29, 39, 50, 51, 53, 55, 60, 61

**Cisco** Cisco is the worldwide leader in IT, networking, and cybersecurity solutions.. ii–iv, 2, 3, 5, 19, 25, 26, 28, 29, 37, 42, 46

**GitLab** Git repository server to host our code on. 27–29, 39, 53, 55, 60, 66, 69

**IFS** Abbreviation for Institute for Software Rapperswil. 53

**INS** Abbreviation for Institute for Networked Solutions Rapperswil. 25, 90

**Insoft Services** Insoft Services is the other industry partner who may wants to form a whole product out of our provided application and extend it to support multiple SD-WAN solutions. 10

**JSX** Template language of React. 50

**KSAT** Industry partner KSAT who will use to product for their daily business. 10

**MVP** Abbreviation for Minimum viable Product, defines a minimal set of features an application has to provide.. 40, 61

**REST API** This is a common used set of rules, on which application interface are designed.. 2, 6

**Scrum** Agile project management method which breaks goals into small work packages which are completed in sprints. 53

**SD-WAN** Technology to manage wide area networks (WAN) via software.. ii–iv, 2, 3, 5, 10, 25, 26, 37

**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, iv, xii, 3, 6, 7, 10, 19, 25–27, 29, 30, 38, 41, 45, 59, 68, 69, 73, 77, 93, 94, *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.. 29
- Single Page Application** SPA are a way to implement web application. SPA is coded mainly in Javascript. 30, 34–36
- SonarCube** Tool to measure Code metrics. 55
- Stdout** Standard output, which in a docker container is the container log. 15, 29, 59, 73, 91
- Unified Process** Agile project management method breaks a project into 4 phases (Inception, Elaboration, Construction, Transition). 53
- VCS** Version Control System. xii, 27, *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. 27
- vManage** The API of Ciscos SD-WAN solution vManage. The API can be queried to gather information about the SD-WAN infrastructure.. ii–iv, 2, 4, 7, 15, 20, 25, 26, 46, 47, 78, 80, 83, 93, 95
- vManage API** The RESTful API of the vManage.. ii, iii, 2–7, 15, 18, 19, 21, 28, 29, 35, 37, 38, 41–44, 46, 47, 50, 56, 78, 84, 93, 94
- YouTrack** Issue tracking tool from JetBrains. 55

---

## List of Figures

---

1	SD-WAN Topology Viewer . . . . .	iv
1.1	Deployment diagram of the application . . . . .	4
1.2	Screenshot of the SDWAN Topology Viewer . . . . .	5
2.1	UseCase Diagram . . . . .	11
3.1	Domain model . . . . .	17
3.2	Data model . . . . .	18
4.1	SDWANTV Context Diagram . . . . .	25
4.2	SDWANTV Container Diagram . . . . .	26
4.3	System overview . . . . .	31
4.4	Sequence diagram . . . . .	34
4.5	Sequence diagram fetching metric . . . . .	35
4.6	Deployment Diagram . . . . .	36
4.7	Mockup design . . . . .	39
4.8	MVP design . . . . .	40
4.9	End of construction design . . . . .	40
5.1	OpenAPI Documentation Overview . . . . .	44
5.2	OpenAPI Documentation Request . . . . .	44
5.3	OpenAPI Documentation Response . . . . .	44
5.4	JWT mechanism if access token is outdated . . . . .	48
6.1	YouTrack Sprint Planning . . . . .	54
6.2	YouTrack Issue Overview . . . . .	55
6.3	Slack . . . . .	55
6.4	Project Plan . . . . .	57
6.5	Risk analysis matrix . . . . .	58
7.1	Hours spent per component per team member . . . . .	62
7.2	Cake diagram issues per phase . . . . .	63
7.3	Cake diagram hours spent per Issue type . . . . .	64
7.4	Cake diagram issues per milestone . . . . .	65
A.1	Html friendly backend healthcheck . . . . .	72
B.1	Topology view . . . . .	78
B.2	Edge down . . . . .	79
B.3	API Edge down . . . . .	79

B.4	Edge is gone . . . . .	79
B.5	Miami-a up . . . . .	80
B.6	API Miami-a up . . . . .	80
B.7	Miami-a down . . . . .	80
B.8	API Miami-a down . . . . .	80
B.9	Miami-a red and edges gone . . . . .	81
B.10	Miami-a up again . . . . .	82
B.11	API Miami-a up again . . . . .	82
B.12	Full-mesh topology view . . . . .	83
B.13	Policy applied . . . . .	84
B.14	Edges cleaned after 5 minutes . . . . .	84
B.15	Full topology view . . . . .	85
B.16	View node information . . . . .	85
B.17	Full topology view . . . . .	86
B.18	Selected edge displays metrics . . . . .	86
B.19	Selected edge updated metrics . . . . .	86
B.20	Normal fullscreen . . . . .	87
B.21	Fullscreen view enabled . . . . .	87
B.22	Full topology view . . . . .	88
B.23	Customer filter applied . . . . .	88
B.24	Full topology view . . . . .	89
B.25	Site-id filter applied . . . . .	89
C.1	Password stored in DB . . . . .	91
C.2	Sample JWT payload . . . . .	91
C.3	Logs of the stdout stream . . . . .	91
C.4	Login failure message . . . . .	92
C.5	Sync state if system runs successful . . . . .	94
C.6	Sync is in error state . . . . .	94
C.7	Sync information could not be fetched . . . . .	94
C.8	Full match without policy applied . . . . .	95
C.9	Policy applied 50s later . . . . .	95
C.10	Render performance . . . . .	96
C.11	Full match fetching speed . . . . .	97
C.12	Applied policy fetching speed . . . . .	97
D.1	Topology Viewer for User . . . . .	98
D.2	Admin panel . . . . .	99
D.3	Add new user . . . . .	99
D.4	Login page . . . . .	100
D.5	Register page . . . . .	100
D.6	Full screen view . . . . .	100
D.7	Zoomed in view . . . . .	101
D.8	Click on connection . . . . .	101

---

## List of Tables

---

1.1	Use Case states . . . . .	5
2.1	Specification Use Case 1 . . . . .	11
2.2	Specification Use Case 1.1 . . . . .	12
2.3	Specification Use Case 1.2 . . . . .	12
2.4	Specification Use Case 1.3 . . . . .	12
2.5	Specification Use Case 1.4 . . . . .	12
2.6	Specification Use Case 2 . . . . .	13
2.7	Specification Use Case 3 . . . . .	13
2.8	Specification Use Case 4 . . . . .	13
2.9	Specification Use Case 5 . . . . .	14
2.10	Specification Use Case 6 . . . . .	14
2.11	Specification Use Case 6.1 . . . . .	14
3.1	vManage list devices response . . . . .	20
3.2	vManage OMP services response . . . . .	21
3.3	vManage IPsec connections response . . . . .	22
3.4	vManage events response . . . . .	23
3.5	vManage metrics response . . . . .	24
4.1	12 Factors . . . . .	27
6.1	Team Members and Responsibilities . . . . .	52
6.2	SDWANTV Releases . . . . .	56
6.3	Risk analysis . . . . .	58
6.4	Quality control measures . . . . .	60
7.1	Working times per team member . . . . .	62
7.2	Time spent per project phase . . . . .	63
7.3	Issues per phase . . . . .	63
7.4	Time spent per task type . . . . .	64
7.5	Time spent per milestone . . . . .	65
7.6	Issues per milestone . . . . .	66
7.7	Code statistics . . . . .	66
B.1	Systemtest protocol . . . . .	77
C.1	NFR test protocol . . . . .	90

# **Part I**

# **Technical Report**

# Technical Report

---

## 1.1 Introduction and Overview

With the trend to software defined networks (SDN) more and more globally operating companies choosing a SD-WAN solution for their infrastructure. SD-WAN makes the management of the network easier because of features like network by code, network segmentation and routing based on policies and rules.

Another big advantage of this approach is the monitoring capabilities. Since all routers are connected with a service plane to the central controller, they can also send state information and metrics to a 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.

At the moment the biggest vendor of SD-WAN solution 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 a whole topology with the routers and the 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 is a missed opportunity by the existing monitoring tool.

### 1.1.2 Goals

In the context of a term project, we want to create a simple web application, which realizes the missed core features of the existing monitoring component of Cisco's SD-WAN solution, a topology view of the whole infrastructure.

Our application fetches in a regular interval the required information from the vManage API, processes it and finally displays it on a so-called progressive web application. On the web application itself, the user can view not only the status of the topology, but also the more

detailed information of the individual resources. For instance, if one wants to check the properties of an IPsec tunnel connection they can simply do it by clicking on the respective visual representation of the connection.

On top of the base use case, we want to implement further functionality which enriches the experience for a network administrator.

- To reassure that the topology is “live”, we will implement a sync status display, which gives the user a short visual feedback about the systems status.
- The user should be able to filter the topology resources by certain criteria.
- User Login to restrict access to disallowed users.
- Displaying metrics (jitter, latency and package loss) of IPsec tunnels.
- A full screen functionality for the usage on a monitoring screen at the office.

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

- Customer portal
- Notifications
- Display Routing paths

For a full overview of the requirements, we refer to the Use Case Section 2.3.

### 1.1.3 Limitations

Due to the term project the personal expenses are limited. We have approximately 480h distributed between two people at our disposal.

Our application is heavily dependent on the vManage API. Consequently, we can only fetch those resources which are accessible over this endpoint.

We also have a lot of technical limitations regarding the scalability 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.

### 1.1.4 Work structure

Our project is split into two parts. The first part of the project is to gather all required information together, becoming familiar with the SD-WAN technology and analysing the vManage API. The second part of the project is to develop the SDWANTV web application.

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 carried out. Especially Cisco had good online video tutorials [11], 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 the evaluation of the vManage API we designed the infrastructure for our application. First, we had to choose the architecture design type we wanted to use. Because of our professional background and the recommendation from domain experts we went for a React frontend and a Python backend using the Django Rest Framework.

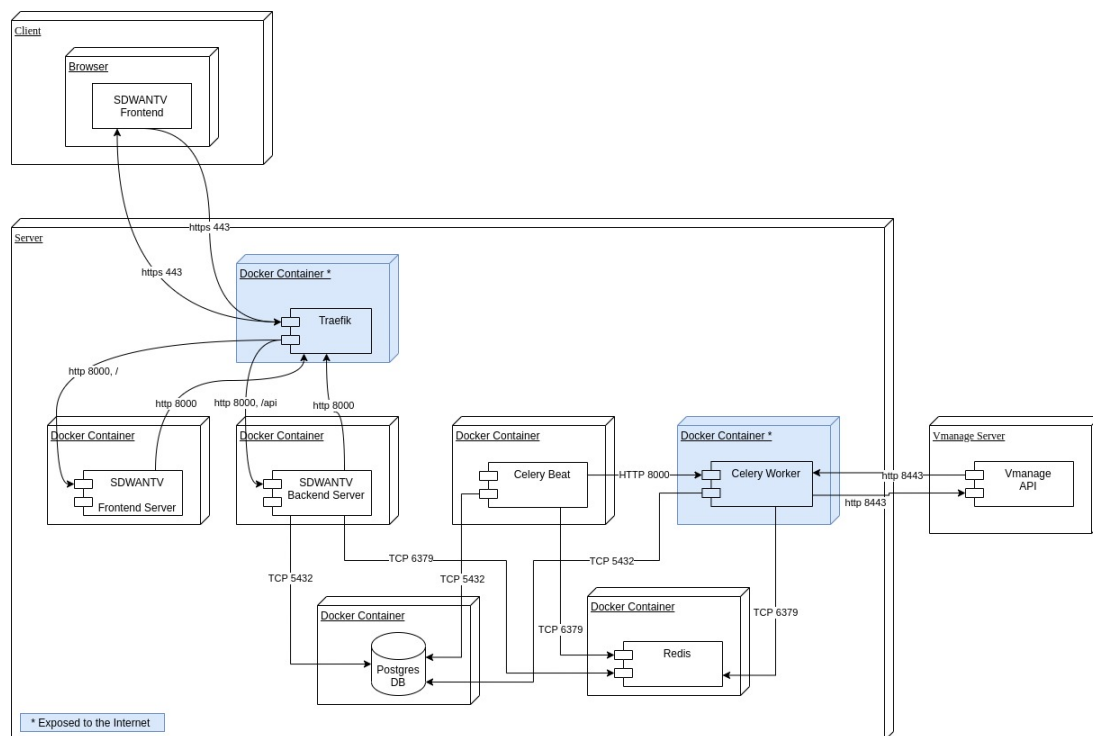


Figure 1.1: Deployment diagram of the application

Next to the frontend and the backend we had to fetch the information from the vManage API and store it in our internal database. We wanted to have this logic decoupled in our own error boundaries from the rest of the application and implemented separated containers for that. In the frontend we still wanted to get informed about the task execution. Due to that, we started to store the status of the tasks in the database and access them from the Django backend. In the header of the frontend the user can check the regularly updated status of the task.

Because we do not know how the environment, in which our application will run on, looks like, it was very important to have everything containerized. We even went one step further and applied the 12 factors that are essential to design a good cloud native web application. For more information about the 12 factors and how we implemented them refer to section the 12 factor section in the architecture and design chapter 4.2.3.



## 1.4 Solution

During the term project we implemented a functional web application, which is able to fetch the topology of a normal sized SD-WAN infrastructure in a relatively small interval. The system is able to map the Cisco vManage API to our own representation of the topology and render it to a frontend application in the browser.

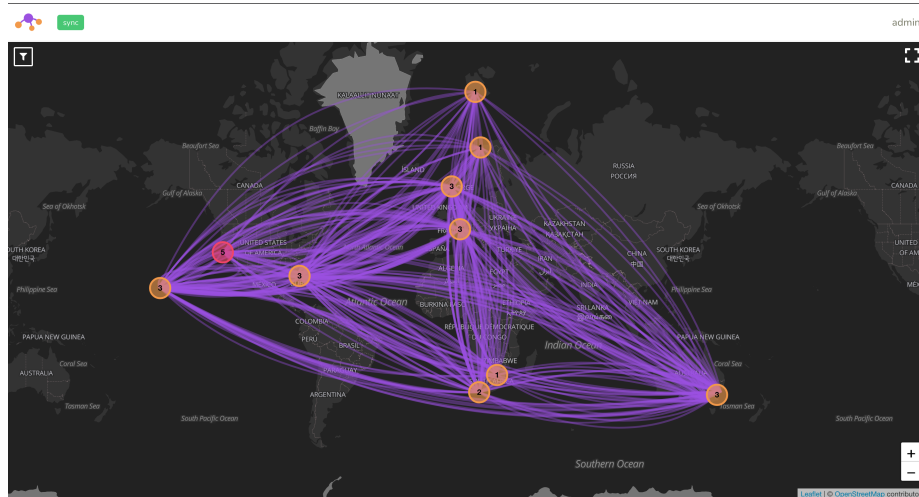


Figure 1.2: Screenshot of the SDWAN Topology Viewer

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 7 seconds and from the frontend to the backend in maximum 5 seconds.

In addition to the performance, the code quality was important as well, since the project might be the basis for a further bachelor thesis. This fact is represented by our test coverage which is more than 60% in the backend and in the frontend more than 80%.

The following table shows the state of the functional requirements. Further information about the use cases is in the project documentation section 2.3:

Use Case	State
UC1: Monitoring the topology	Done
UC1.1: View the node information	Done
UC1.2: Display connection metrics	Done
UC1.3: Display bandwidth with IP-Sec Tunnels	Open
UC1.4: Toggle full screen	Done
UC2: Apply Customer filter	Done
UC3: Manage users for companies	Open
UC4: Apply connection filter	Partly done
UC5: Display of the path from router to the destination	Open
UC6: Manage metric alarms	Open
UC6.1: Send alarm to external syslog server	Open

Table 1.1: Use Case states

### 1.4.1 Implementation

#### Fetching of the vManage API

Our premonition of the performance bottleneck became true as we used the vManage API in practice. The API for the IPsec tunnels needed to be queried for each device. As a result, we parallelized the fetching tasks as much as possible.

As we implemented the backend with Python, we quickly realized that this programming language is not the perfect fit for async tasks processing. Our first approach was to use the task manager celery. This, however, was not as performant as we hoped, since it can only parallelize up to eight tasks. The other, more performant approach was to use the python coroutine. Although it just runs in one single thread and therefore on only one processor, it is more performant if the resource fetching takes longer than the processing.

#### Rendering the topology

The core part of the frontend was the topology viewer itself. To achieve this, we used the leaflet plugin and OpenStreetMap combined with react.

One of the biggest obstacles was the rendering of the resources on the map. We took advantage of a react-leaflet library but ended up to implementing a big part on our own. Especially the edge and node popups were not available by default and needed a lot of customization in order to work.

#### Authentication

From the security perspective of SDWANTV using JWT token was appropriate. But how we should store the token was not clear. Two tokens needed to be stored, one for accessing the resources from the backend REST API and one refresh token. After some research and discussions with a professional, we ended up storing both of them in the local storage of the browser.

## 1.5 Conclusion

The terms of reference were clear and could be adapted according to the needs during the project. Both project members were able to bring a lot of previous knowledge to the project and so the overview was never lost. The cooperation in the team also worked very well, as we complemented each other wonderfully.

We consider the functional scope to be rather small, which however benefits the quality. We also invested a lot of time to understand the vManage API. Unfortunately, we found that the vManage API was not really well tailored to our needs and therefore scaled poorly. Also in the frontend we had problems with libraries that did not cover our needs and therefore had to make our own adjustments.

We consider the application itself to be successful. However, we can not say at this moment whether it can stand out from its competition. What we could well imagine, however, is that it could serve as a supplement to the existing systems. There are also many possibilities to expand the functionalities and make the application even more powerful.

## 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 UseCases could be implemented. Refer to the solutions section 1.4 to see which of the UseCases have not been implemented yet.

Probably the most interesting use case would be to implement the use case *manage user for companies UC3 2.3.7* and redesign SDWANTV to make it customer friendly. This would make it possible to register customers in the application and let them only see their own nodes and edges.

Another interesting idea to implement would be to enhance the filter possibilities. For instance new filters like VPN-id or up/down states of the nodes and edges could be added.

### 1.6.2 Technical improvements

#### WebGL

The frontend which runs on a browser engine is one major bottleneck in our application. 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. There are already proof of concepts for leaflet and WebGL.

#### WebSocket

The backend fetches the data for the topology from the vManage API. The frontend polls in an interval from the backend. It is a simple solution, but it reduces the liveness. Using WebSockets makes it possible to push our topology to the frontend whenever it changes.

#### 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 would still require a lot of analysis if it should really speed up the liveness time.

#### Metrics fetching

Another thing that might be of interest for users of SDWANTV would be to have the quality of an IPsec tunnel displayed by colours on the world map. So if a tunnel has a bad quality it will be displayed in a different kind of colour on the world map. This would require to regularly query all the metrics from the vManage API and update them in the frontend.

### 1.6.3 Technical debt

Although SDWANTV has a good quality codebase, there are three small things that need to be reported as technical debt. All of the bugs are addressed in the ticket management tool.

### **Admin panel without CSS**

Unfortunately the admin panel has no CSS because Django's static files are missing or the path to the static files is misconfigured. Because this was only detected in the transition phase, the admin panel is only needed to add or remove new users and because it can also be used without the CSS files, it was not fixed anymore.

### **Backend default log format without timestamp**

The second bug is a problem with the backend logging messages format. The configured log format is wrong and prints the log messages in the format as seen below instead of the timestamp.

```
sdwantv_celery_1 | {asctime} django INFO Sync started
```

Due to the fact that the log format can be configured through the docker container parameters, this can be fixed by setting DJANGO\_LOG\_FORMAT to  
`%(asctime)s %(name)-12s %(levelname)-8s %(message)s`.

### **Fullscreen not closing correctly**

After fullscreen mode was enabled and the fullscreen mode was left by pressing the ESC button on the keyboard, the application remains in the fullscreen mode, displays browser navigations again but not the applications header bar. After pressing on the fullscreen button in the top right corner of the applications screen, it will end the fullscreen mode and will display the header bar again but also bring up an error. This error can be safely ignored and the application can be used as normal.

## **Part II**

# **Project Documentation**

# Requirements Specification

---

## 2.1 Thesis Requirements

Working on SDWANTV should bring out an application that is able to display all SD-WAN IPsec tunnels and display their health. SDWANTV will be designed to run on a big shared TV screen in an office, so everybody can monitor the connections. The developed product should meet all the customer requirements from KSAT and Insoft Services.

## 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.

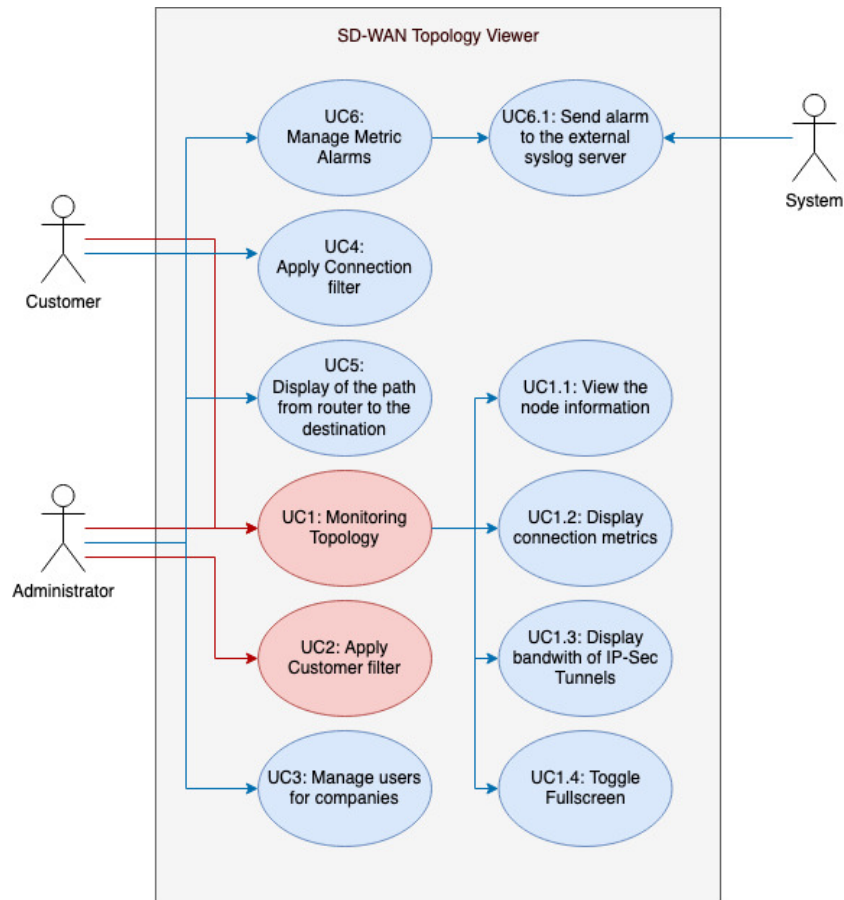


Figure 2.1: UseCase Diagram

### 2.3.1 UC1: Monitoring the topology

Use Case Section	Description
Main Actor	Customer & Administrator
Main Success Scenario	The <b>administrator</b> can monitor the nodes and edges of all companies.
Alternative Success Scenario	The <b>customer</b> can monitor the nodes and edges of their company infrastructure.
Success Guarantee	The information and the topology are updated on a regular basis and the system displays the last successful sync. The nodes are displayed on their geographic location on a world map. The user is able to navigate the view on the map (zoom out and zoom in).

Table 2.1: Specification Use Case 1

**2.3.2 UC1.1: View the node information**

Use Case Section	Description
Main Actor	Customer & Administrator
Main Success Scenario	The user can select a node and can see the name, reachability, IP-Address and Site ID on the node popup on the topology.

Table 2.2: Specification Use Case 1.1

**2.3.3 UC1.2: Display connection metrics**

Use Case Section	Description
Main Actor	Customer & Administrator
Main Success Scenario	The user can select an edge of the topology and sees following connection metrics: jitter, delay, loss.

Table 2.3: Specification Use Case 1.2

**2.3.4 UC1.3: Display bandwidth of IPsec Tunnels**

Use Case Section	Description
Main Actor	Customer & Administrator
Main Success Scenario	The user is able to see the bandwidth of each IPsec tunnel on the topology.

Table 2.4: Specification Use Case 1.3

**2.3.5 UC1.4: Toggle full screen**

Use Case Section	Description
Main Actor	Customer & Administrator
Main Success Scenario	The user is on the main page. The user toggles the full screen mode. The full screen mode makes the browser windows match the screen of the device and removes the header of the application. If the user toggles again the full screen mode the changes are reverted.

Table 2.5: Specification Use Case 1.4



### 2.3.6 UC2: Apply customer filter

Use Case Section	Description
Main Actor	Administrator
Main Success Scenario	The administrator sees the topology of all customers. The administrator can filter by customer.
Success Guarantee	The topology is updated according to the applied filter.

Table 2.6: Specification Use Case 2

### 2.3.7 UC3: Manage users for companies

Use Case Section	Description
Main Actor	Administrator
Main Success Scenario	The administrator can add or remove users. If he wants to add a user he needs to specify the company and the email address of the new user.
Success Guarantee	An email with a link including the register token is sent to the given e-mail address.
Alternative Scenario	A user with this email already exists. The administrator is notified about that. He needs to delete the existing user to assign him a new company.

Table 2.7: Specification Use Case 3

### 2.3.8 UC4: Apply Connection filter

Use Case Section	Description
Main Actor	Customer & Administrator
Main Success Scenario	The user can filter by the properties of their topology.
Success Guarantee	The topology is updated according to the applied filter.

Table 2.8: Specification Use Case 4

### 2.3.9 UC5: Display of the path from router to the destination

Use Case Section	Description
Main Actor	Customer & Administrator
Main Success Scenario	The user can select the router the destination IP address and the VPN to display the VPN tunnel, which is actually used for the data transfer.
Success Guarantee	The user can see the connection that is actually used, on the topology.
Alternative Scenario	The user input is incorrect. The system notifies the user about this.

Table 2.9: Specification Use Case 5

### 2.3.10 UC6: 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 and an IP address of a syslog server.
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 2.10: Specification Use Case 6

### 2.3.11 UC6.1: Send alarm to the external syslog server

Use Case Section	Description
Main Actor	The System
Preconditions	An metric threshold is getting violated
Main Success Scenario	The system sends a syslog message to the given 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 2.11: Specification Use Case 6.1

## 2.4 Non functional requirements

### 2.4.1 Functionality

**Security** The authorization should be OAuth [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 will keep a session in the frontend with a simple JWT [33]. The user can authenticate to the backend with this token. The token has a limited lifetime.

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.

**Efficiency 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 portable and not be dependent on the physical device it runs on.

### 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 be able to process twice as much data at the same time.

---

## Analysis

---

### 3.1 Domain model

The domain model gives us a simple overview about our problem domain. The detailed description of the entities can be found in the data model section 3.2 below.

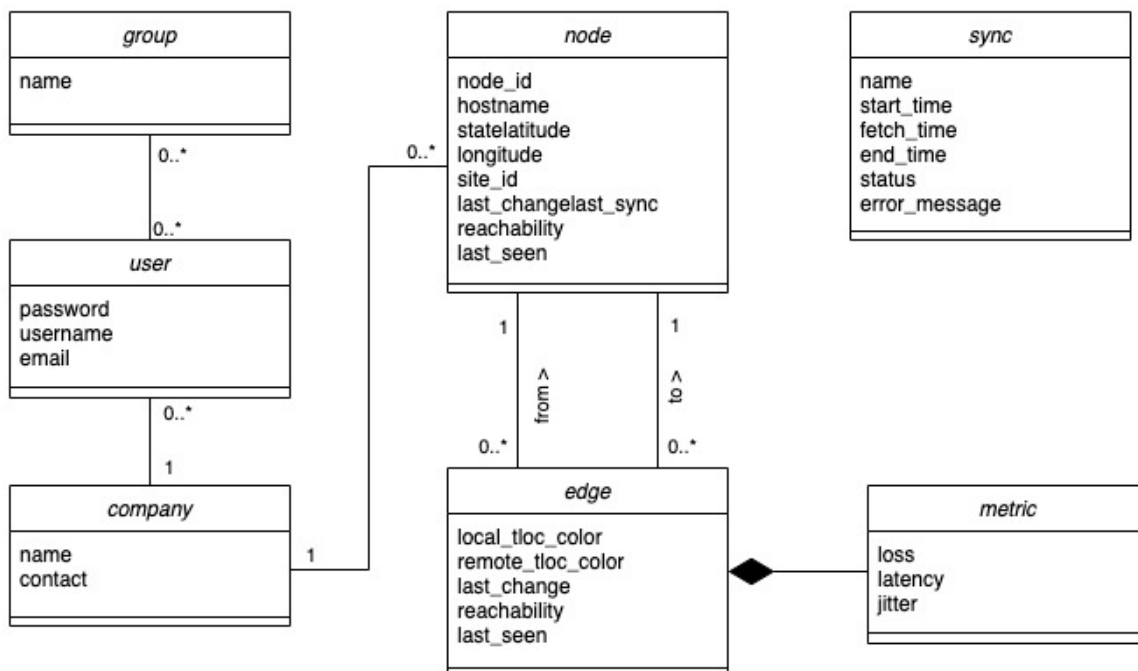


Figure 3.1: Domain model

## 3.2 Data model

The data model shows us how we represent the topology in the PostgreSQL database. The biggest part came from the Django default user management. However, we incorporated our logic into it. The most important entities are located in the square.

The entries are going to be updated quite often and therefore we want to keep the flexible parts as simple as possible.

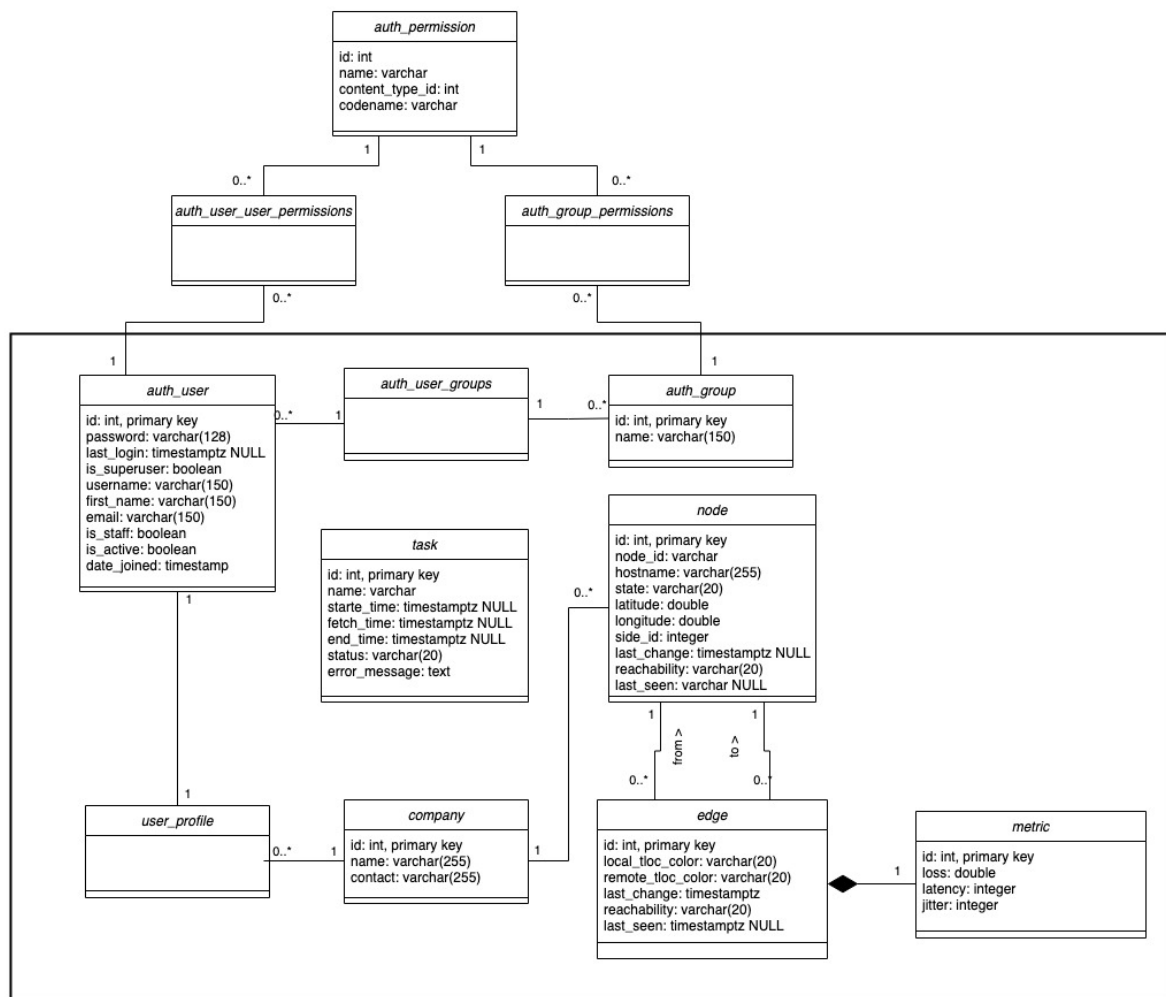


Figure 3.2: Data model

### 3.2.1 auth\_user

The user is an entity that is created by Django and enables us to get the user management out of the box.

### 3.2.2 auth\_group

As an entity it will be named auth\_group and it is also a default entity provided by the Django framework. We will use this to restrict access to only those edges and nodes the user belongs to. Those groups are fetched from the vManage API and are generated automatically.

### 3.2.3 edge

The edge represent the connection between two nodes. This representation does not exist in the vManage API. We artificially added this to suit our applications needs.

### 3.2.4 metric

A metric belongs to an edge and contains the package loss, the latency, and the jitter of the edge. Every time an edge object is created it will also automatically create an associated metric object.

### 3.2.5 node

A node is unique and only exists once in the topology. Every node belongs to one company.

### 3.2.6 sync

This entity is used for managing the fetching tasks. We want to know how long a task takes, its execution time and in which state it 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.

### 3.2.7 company

The company represents a customer, which owns some nodes on the topology. This field is fetched from the device-groups attribute on the device. In the frontend we can filter the topology based on this.

## 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 the it. The following tables displaying 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 **List devices**, **OMP Servies**, **IP Sec Inbound**, **Events** and the **Metrics** Endpoint.

The **Events** endpoint enables us to be close to live and therefore requires to be fetched at a regular interval.

### 3.3.1 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 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 customer. 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

#### Response body

**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 3.1: Devices sample response



### 3.3.2 OMP Services

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. We might be able to omit this request, if we ignore the vpn-id and group together the devices by the device-group field of the node.

#### Request

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

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

#### Response body

**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 3.2: OMP services sample response

### 3.3.3 IPsec connections

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

#### Request

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

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

#### Response body

**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.

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

Listing 3.3: IPsec inbound sample response

### 3.3.4 Events

The events endpoint provides a list of the most recent occurred events. This can be done with just one single request, is scalable and very quick. For this request the HTML POST method is used along with an aggregation query parameter. The aggregation query parameter can be found in the attachment section F.3.

#### Request

POST /dataservice/event/aggregation

Response attribute	Description
entry_time	Lets us decide if the event is outdated or relevant.
system_ip	Is the source node, from which the change happened.
details	This is the string that contains the information of the topology change.

Table 3.4: vManage events response

#### Response body

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

```
[
  {
    "system_ip": "10.255.255.254",
    "count": 1,
    "entry_time": "2020-10-15T13:58:04.000Z",
    "eventname": "bfd-state-change",
    "details": "
      host-name=SDWAN-1;
      src-ip=152.96.9.247;
      dst-ip=152.96.9.38;
      proto=ipsec;
      src-port=12426;
      dst-port=12386;
      local-system-ip=10.255.255.254;
      local-color=biz-internet;
      remote-system-ip=10.255.255.251;
      remote-color=biz-internet;
      new-state=up;
      deleted=false;
      flap-reason=na
    "
  },
  ...
]
```

Listing 3.4: Events sample response

### 3.3.5 Metrics

The metrics can be fetched through a statistics aggregation endpoint. This endpoint accepts an aggregation query which defines for which device and in which duration the statistics should be returned. The endpoint returns a list of connections, with each connection containing the connection metrics. A full example of the aggregation query can be found in the appendix F.4.

#### Request

POST /dataservice/statistics/approute/fec/aggregation

Response attribute	Description
loss_percentage	The percentage of lost packages on the connection.
latency	The latency metric on the connection.
jitter	The jitter metric on the connection.
name	The name of the connection that the metrics belong to.

Table 3.5: vManage metrics response

#### Response body

**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.5.

```
[
  {
    "loss_percentage":0,
    "latency":0,
    "jitter":0,
    "name":"10.255.255.133:mpls-10.255.255.255:mpls",
  },
  ...
]
```

Listing 3.5: Metrics sample response

---

## Architecture & Design Specification

---

### 4.1 Scope

This chapter describes the architecture and design of the SDWANTV application. However, 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 Design

The Software Architecture Design starts with an overview of the software components and their primary functions.

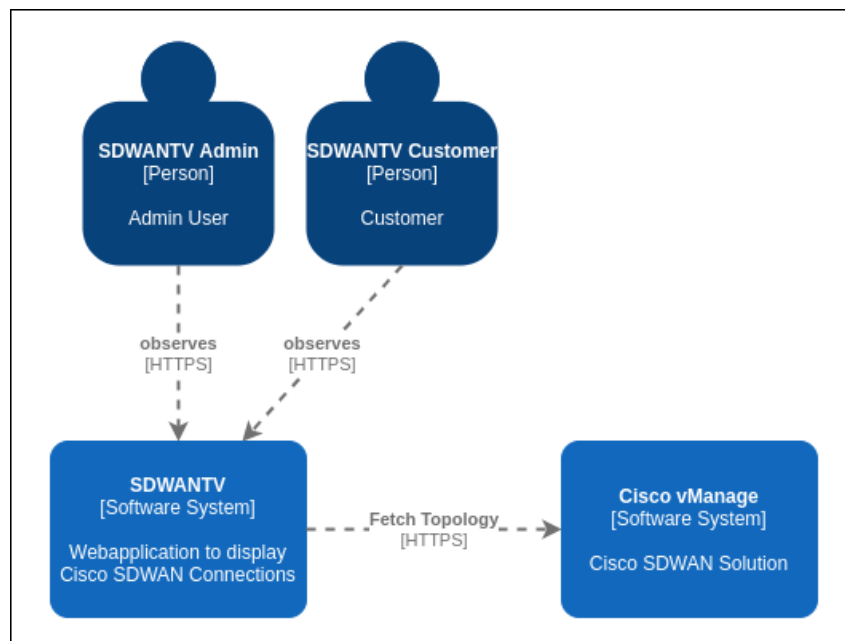


Figure 4.1: SDWANTV Context Diagram

**SDWANTV** 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 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.

### 4.2.1 Container Diagram

The container diagram provides a brief overview of how each container interacts with the other containers.

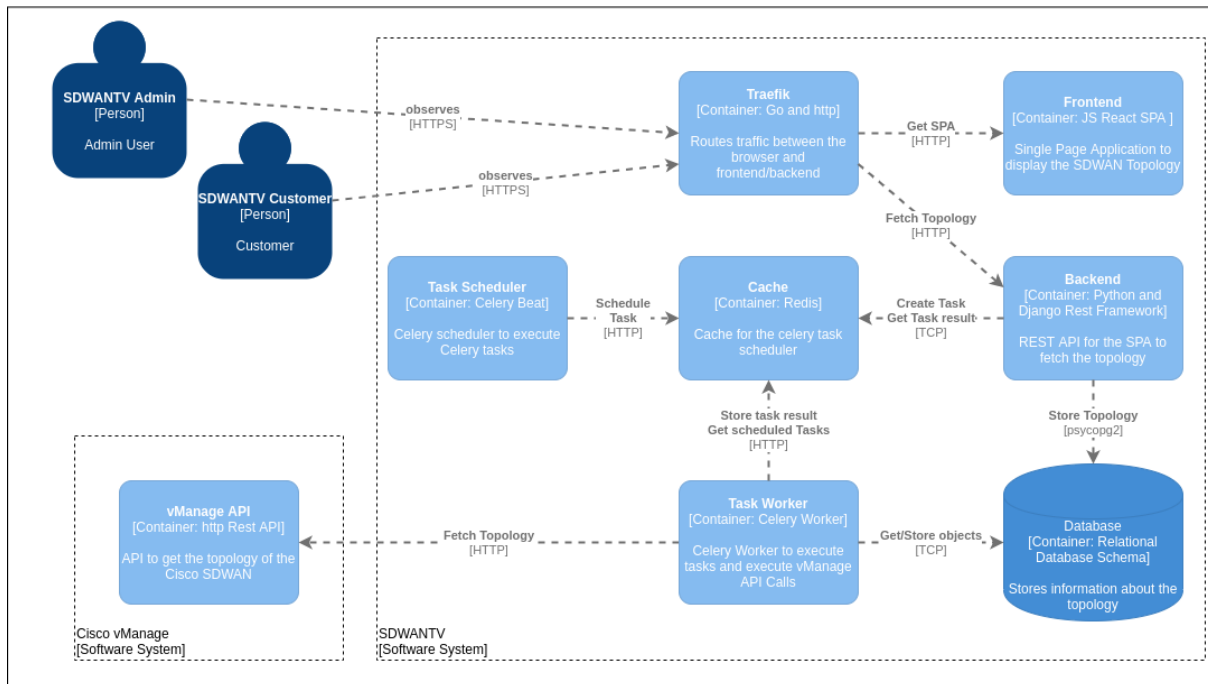


Figure 4.2: SDWANTV Container Diagram

### 4.2.2 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.

The dockerized SDWANTV frontend, backend, celery, redis and database will run on the same server, however, could also be distributed on multiple servers.

### 4.2.3 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)	4.2.3
II. Dependencies	Explicitly declare and isolate dependencies	4.2.3
III. Config	Store config in the environment	4.2.3
IV. Backing Services	Treat backing services as attached resources	4.2.3
V. Build, release, run	Strictly separate build and run stages	4.2.3
VI. Processes	Execute the app as one or more stateless processes	4.2.3
VII. Port binding	Export services via port binding	4.2.3
VIII. Concurrency	Scale out via the process model	4.2.3
IX. Disposability	Maximize robustness with fast startup and graceful shutdown	4.2.3
X. Dev/prod parity	Keep development, staging and production as similar as possible	4.2.3
XI. Logs	Treat logs as event streams	4.2.3
XII. Admin processes	Run admin/management tasks as one-off processes	4.2.3

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 docker 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 *docker-compose.yaml* file we put the container image and the needed configuration together. And finally, we execute the *docker-compose.yaml* file manually on the destination server. Although we have not fully automated the deployment process, we have strictly separated these 3 stages and therefore consider this factor to be 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 docker volume. If the app is started and no docker 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 docker 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. Unicorn, the production webserver we use in the backend, uses a master/worker model where it is possible to define the number of workers [24] that respond to http requests concurrently. The same is used in the frontend but with nodejs. 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 docker container receives the `docker stop` command, docker will send a SIGTERM signal to the containers root process (PID 1). Because we use Docker for the container management, this factor is considered to be fulfilled. Restarting the backend normally works, however it may be possible that there will be a leftover of a sync tasks present. For this purpose there is an admin task that can be executed within the backend docker container to clean this leftovers up. Restarting the frontend is possible at any times. Error handling is implemented in the core functionalities. Occured 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 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.

## 4.3 Design Decisions

Below we listed the most important design decisions during our development process.

### 4.3.1 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 it was a suggestion from the supervisor and also because we already had experience with 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, 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 an easy way to setup API authentication, it has built-in user management and it overall is easier to implement new features, we decided to use the Django Rest framework.

### 4.3.2 React Frontend

The decision to use a single page application written in React as the frontend was quite easy. Like Python also 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.

React also 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.

### 4.3.3 PostgreSQL Database

For the database we decided to go with PostgreSQL because it is OpenSource and widely used with Python applications. Due to the fact that we do not have any particular performance requirements, we were open to use what we are familiar with.

In addition, we do not expect such a huge amount of data that we would encounter database performance issues. But if this were the case we could still do some database tweaks or also switch to use TimescaleDB, which is a database plugin for PostgreSQL and can be used with docker as well.

## 4.4 Software Architecture

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.

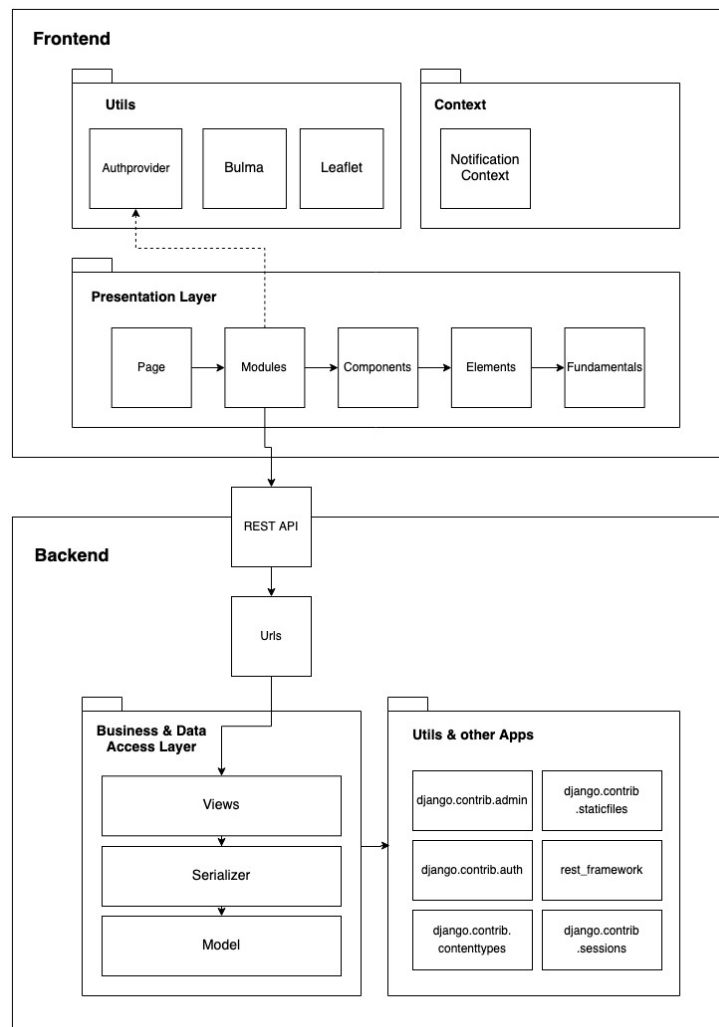


Figure 4.3: System overview

### 4.4.1 Frontend

#### Context

**Notification Context** The notification context is wrapped around the whole application. With this context it is possible to add a notification wherever we like in the code. If a notification has been set a notification consumer then reads the notification and displays it either as a warning or as an info banner.

#### 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 the 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

**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 [8] to get a beautifully aligned design out of the box.

**Leaflet[29]** Leaflet is a JS library to render elements on the Openstreetmap. We will use this especially for the topology, which needs to be rendered on the map.

### 4.4.2 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.

### 4.4.3 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 [17].

#### Urls

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

#### Business & Data Access Layer

**Views** The views not only contain the functionality to correctly render the result, it also contains the business logic. Depending on the complexity, we might outsource most of it into a separate service.

**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 object stored in the database. Only over the model will we access the database.

#### Utils & other Apps

The Django framework consists of several apps. Some of those app can run independently from each other and some of them are only working embedded in another app. For instance the *Django.contrib.admin* app, provides an out of the box admin panel. Again, 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 the documentations [15] [17].

**Django.contrib.admin** This gives us an out of the box admin and user management with roles.

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

**Django.conrib** *Django.contrib.auth* & *Django.contrib.staticfiles* & *Django.contrib.contenttypes* & *Django.contrib.sessions* are all helper apps that Django uses to enable the basic functionality of a modern web server.

## 4.5 Sequence Diagrams

### 4.5.1 Fetch topology

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

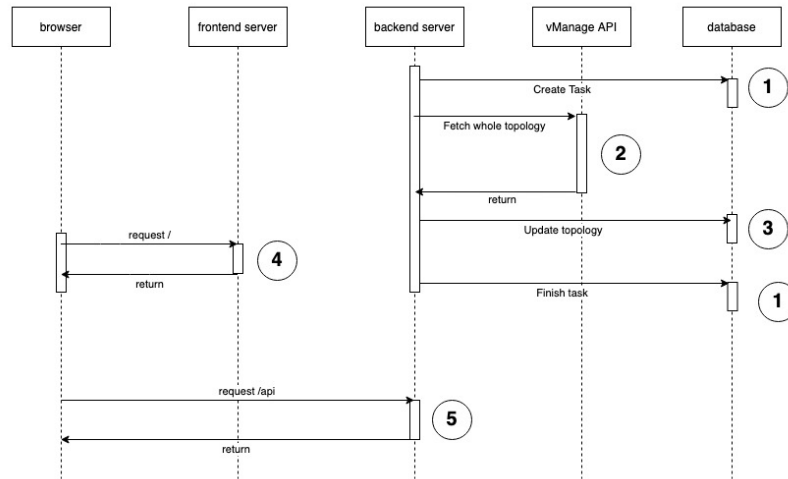


Figure 4.4: Sequence diagram

#### 1: Managing the Tasks

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

#### 2: Fetching and compiling the whole 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. The runtime separation of backend server and Task enables us to dynamically increase the resources for the task and does not influence the normal runtime.

#### 3: Update cached topology

After we fetched the resources, we will format the data in either edges 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 node server will serve all assets like JS, HTML, CSS and images and passes environment variables to the frontend. We might use code splitting to speed up the initial load.

#### 5: Django REST API

Eventually we will access our endpoint with the SPA. The backend server will access the required information from our PostgreSQL database and satisfy the request. It should not take much longer than two seconds to answer a request, if the topology is not too big.

### 4.5.2 Fetching metrics

The sequence diagram below visualizes the process flow of fetching the metrics for an edge.

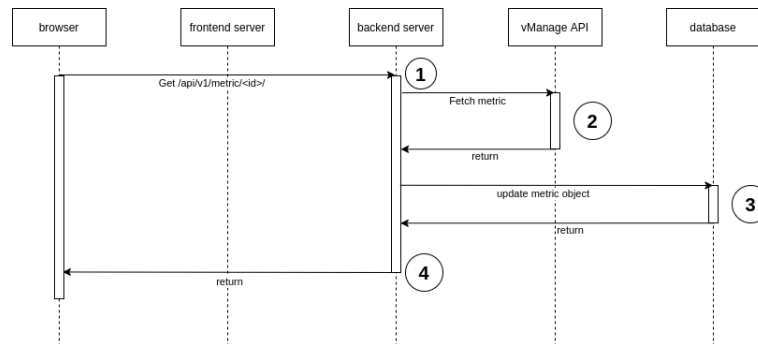


Figure 4.5: Sequence diagram fetching metric

#### 1: Get metric

Due to the fact that we use a Single Page Application there is no need to first send a query to the frontend server because the whole Single Page Application is already loaded into the browser. By selecting an edge on the world map, the browser will perform a request to the backend API metric endpoint and provide the id of the edge.

#### 2: Fetch metric

After the backend has received the request from the browser, it will send a request to the vManage API. The vManage API returns the requested metric which afterwards is processed by the backend server. More about fetching the metric can be found in the implementation section 5.1.1.

#### 3: Store in the database

The third step is to store the result of the requested metric into the database. For every edge there already is a matching metric object in the database. This metric object will be updated and return the result of the operation.

#### 4: Return metric to browser

In the end the requested metric will be returned to the browser which displays the result on the edge information window.

## 4.6 Deployment

The deployment diagram describes which part of our system runs on which tier. As can be seen, we took extensive use of the docker container. This decouples the underlying operation system to our software components.

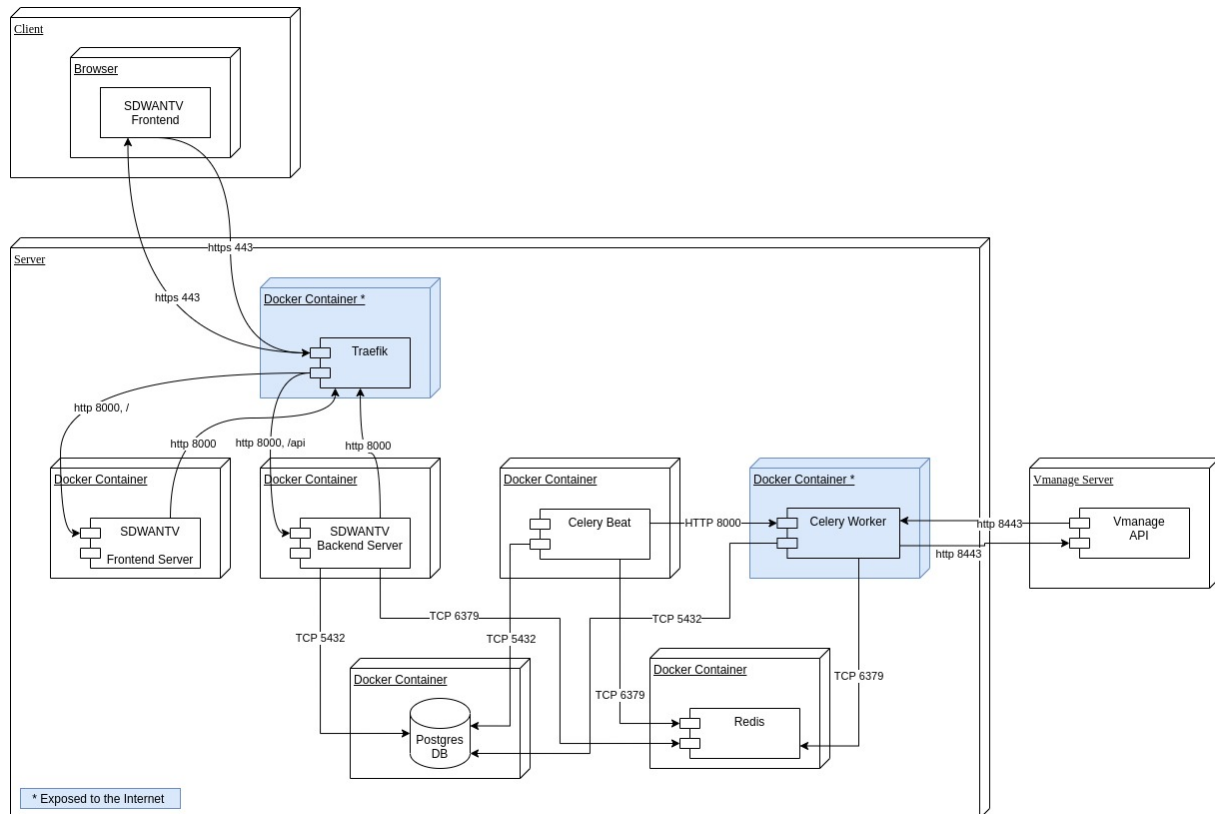


Figure 4.6: Deployment Diagram

### 4.6.1 Client & SDWANTV Frontend

The user first accesses the `"/`-Endpoint via https. The Traefik router forwards this kind of request to the our nodejs server. The nodejs server delivers the required assets for a Single Page Application application to run in the browser. From then on the client only accesses the `/api`-Endpoint, which provides over a REST API the topology data.

### 4.6.2 Traefik

The Traefik service is the entry point to our system. It will route the request to and from the services. The Traefik server also configures the SSL certificate to enable a secure communication over https with the frontend. It would also be possible to go without a Traefik reverse Proxy but this is significantly cleaner and nicer solution to have one single entypoint into the whole system. Traffic between Traefik and the backend and frontend are not secured and occur on http.

### 4.6.3 Frontend Server

As mentioned before, the frontend server delivers the assets, like html, css, graphics and js, to the browser. Depending on the performance we will be required to add the so called *dynamic*



*code splitting*. The coding splitting enables us to reduce the amount of assets loaded in the initial render.

#### 4.6.4 Backend Server

The backend server provides the API to the topology data and the peripheral data for extra functionalities such as user management. The backend server fetches the topology information from the vManage API in a regular interval and stores them in the PostgreSQL database. For a request from the client the backend then only needs to access the PostgreSQL database and can answer the request much faster.

#### 4.6.5 PostgreSQL Database

One database will represent our whole system. It will only be accessed by the backend server and can therefore ignore request from all other sources. The database will be updated quite often, because the topology metrics will change continuously. Hence we will not take advantage of PostgreSQL specific optimization for indexing or other speedups that require stable data.

#### 4.6.6 Beat

Beat is a part of celery and is responsible to periodically schedule tasks at a defined time or cron schedule.

#### 4.6.7 Redis

Redis is the cache that is being used by both Celery and Beat to temporarily store the tasks and the results.

#### 4.6.8 Celery Task Engine

**Fetching whole topology** It fetches from the vManage API the whole topology and gives us a full refresh. We need this to make sure that we are in sync with the main topology. Maybe this task is not scalable and the schedule needs to be adjusted if the topology is becoming more complex.

#### 4.6.9 vManage API

The vManage API is an external API. It is the gate to the SD-WAN solution [10] 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. In order to reduce the amount of request to the vManage API, we decided to fetch all the topology information in scheduled job and cache them afterwards in our PostgreSQL database.

#### 4.6.10 Deployment with Docker-compose

For the deployment of all containers we use the tool docker-compose. Docker-compose is a tool to deploy 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 builds starts the PostgreSQL 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 we can omit the docker file sync.

**Production** To run SDWANTV in production mode the file docker-compose-production.yaml is started. This starts the PostgreSQL database, backend python, celery, celery beat, redis and the frontend container in production mode. Additionally, the Traefik proxy will also be started which takes over the routing for all the containers and provides TLS encryption.

## 4.7 Tools & Frameworks

### 4.7.1 Frontend

In the frontend we decided to keep the tool chain as easy as possible and still have all the functionality we need.

**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.

**Jest/react-testing-library** Is our frontend testing setup. They are the most common and easy to use testing tools.

**Eslint** We use this tool to enforce code guidelines and improve the development experience.

**Prettier** This tool is used to maintain a clean code base, where indentation and order is the same all over to frontend. The rules for prettier are stored in the same config file as eslint.

### 4.7.2 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.

### 4.7.3 Communication

**HTTP Methods** For the user management and other non-monitoring related tasks, we will use simple http methods in the rest format.

**vManage API** Is an external REST endpoint where we gather all required data from.

### 4.7.4 Deployment

**GitLab CI** The automated testing, the linting, formatting and finally the deployment is executed in the GitLab CI.

**Docker** We deploy our artifacts in a dockerized form. Because of that, we have the flexibility to deploy wherever we want.

## 4.8 UI-Design

Similar to the agile principle we decided to work on the designs incremental. Therefore, we will show in this chapter the incremental development of the designs.

### 4.8.1 Tools

For the the user interface design we used figma[22].

### 4.8.2 Mock-up

In order to have a base for further design discussions, we created a mock-up to roughly outline our ideas visually. There are more screens in the attachment. D.1

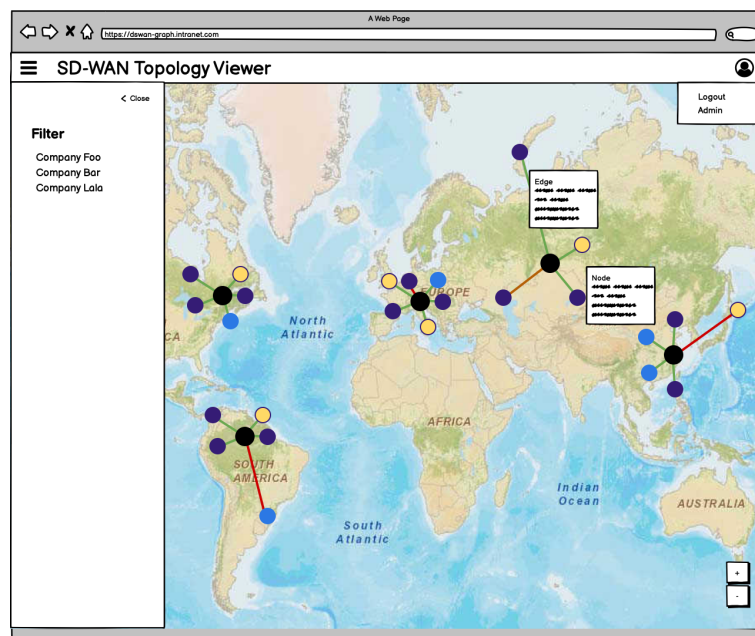


Figure 4.7: Mockup design

### 4.8.3 MVP design

The MVP is the result of the first construction iteration. The feature size is very small. D.2

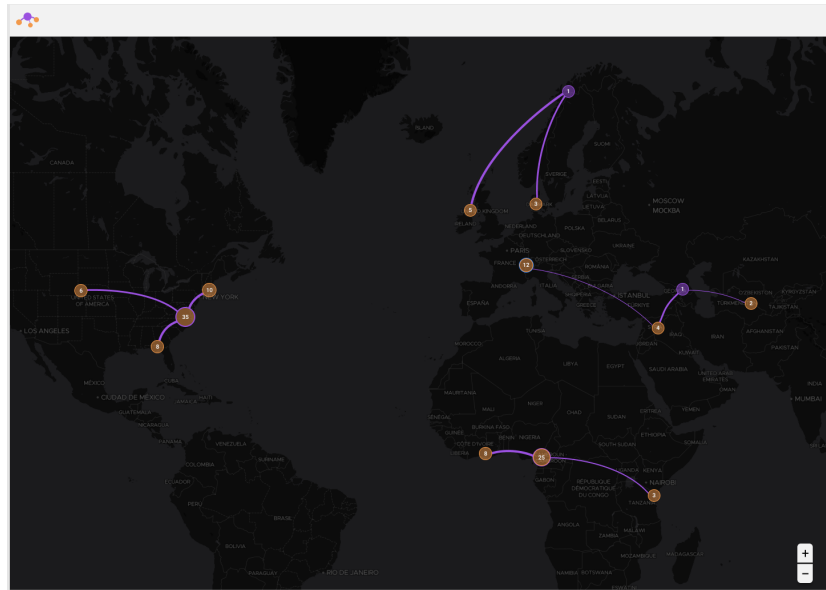


Figure 4.8: MVP design

### 4.8.4 Prototype end of construction

At the end of construction the topology viewer added several new graphical elements.

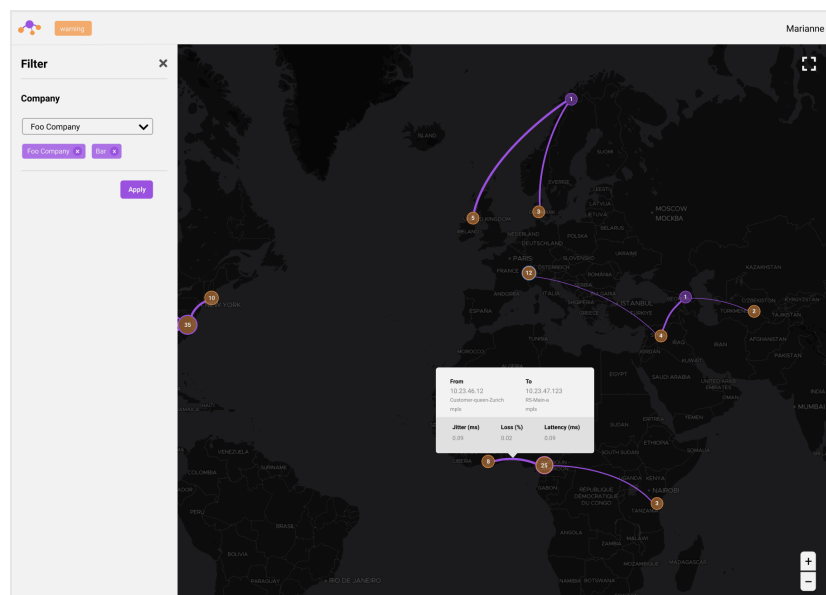


Figure 4.9: End of construction design

---

# Implementation & Testing

---

## 5.1 Implementation

As can be seen in the deployment diagram located in the next section, we separate the SD-WANTV 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 notable parts of the implementation and testing of the SD-WANTV application.

### 5.1.1 Python Django Backend

#### Fetching Nodes & Edges

The main purpose of the backend is to periodically query the Cisco vManage API and get all devices (represented as nodes) and IPsec tunnels (represented as edges). All fetched nodes and edges should then 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 edges 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 edges. The schedule interval can be configured from the environment variable `VMANAGE_FULL_SYNC_INTERVALL_SECONDS`.

```
full_sync_intervall_seconds = float(os.environ.get('VMANAGE_FULL_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 currently another task is already running and skips fetching the import of nodes and edges or if it should execute it. The `@shared_task` annotation marks this function as a task, so that Celery is able to detect and execute it. For reading purposes, logging lines were removed.

```

@shared_task
def sync_topology():
    existing_sync = existing_syncs()
    if existing_sync:
        no_sync_running = not sync_running()
        last_sync_older_than_sync_intervall = sync_older_than_sync_intervall()
    sync = Sync.objects.create(name='full_sync')
    sync.start()
    if not existing_sync:
        import_topology(sync)
        return
    if last_sync_older_than_sync_intervall:
        if no_sync_running:
            import_topology(sync)
        else:
            sync.skip('skipped because another sync is running')
    else:
        sync.skip('skipped because sync_intervall is not reached yet')

```

Listing 5.2: Sync topology Celery task

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

```

def import_topology(sync):
    session = create_session(sync)
    token_session = fetch_token(session, sync)
    fetched_devices, fetched_nodes = fetch_devices(token_session, sync)
    loop = asyncio.new_event_loop()
    fetched_edges, errors = loop.run_until_complete(fetch_device_edges(token_session,
        fetched_devices, sync))
    loop.close()
    if True in errors:
        sync = Sync.objects.get(id=sync.id)
        sync.warn()
    cleanup_topology(fetched_edges, fetched_nodes)
    sync = Sync.objects.get(id=sync.id)
    sync.end()

```

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. Asynchronously and in parallel getting all edges of every node and storing them in the database.
5. Cleanup old nodes and edges.

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

The function `fetch_device_edges` is executed asynchronously and executes the `get_device_edges` function one time for every node in the devices list passed as a parameter to the function. It then awaits the response of every concurrent execution and gathers the result together into the result variable.

```

async def fetch_device_edges(token_session, devices, sync):
    result = await asyncio.gather(*[get_device_edges(token_session, device, sync) for device in
    devices])
    fetched_edges = []
    errors = []
    for device in result:
        fetched_edges.append(device['fetched_edges'])
        errors.append(device['error'])
    return fetched_edges, errors

```

Listing 5.4: Asynchronously fetch edges for a device

The function `get_device_edges` creates a new `aiohttp ClientSession` and asynchronously gets the edges for the device. After those were fetched the data is validated against a JSON schema. If the data is valid the fetched edges 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_edges(session, device, sync):
    fetched_edges = []
    error = False
    device_url = vmanage_url + '/dataservice/device/ipsec/inbound?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_edge_schema)
            await client_session.close()
    except aiohttp.ClientConnectionError:
        error = True
    else:
        loop = asyncio.get_event_loop()
        fetched_edges = await loop.run_in_executor(None, process_fetched_device_edges, data,
        device)
    finally:
        result = {'fetched_edges': fetched_edges, 'error': error}
    return result

```

Listing 5.5: Asynchronously perform the vManage API request

## Sync objects

To keep track of fetching the topology we created a sync model. The model consists of the following properties:

1. start time
2. fetch time
3. end time
4. the status of the sync
5. an optional error message field

This makes it possible to manage the sync tasks in order to have just one sync task running at a time.

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

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

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

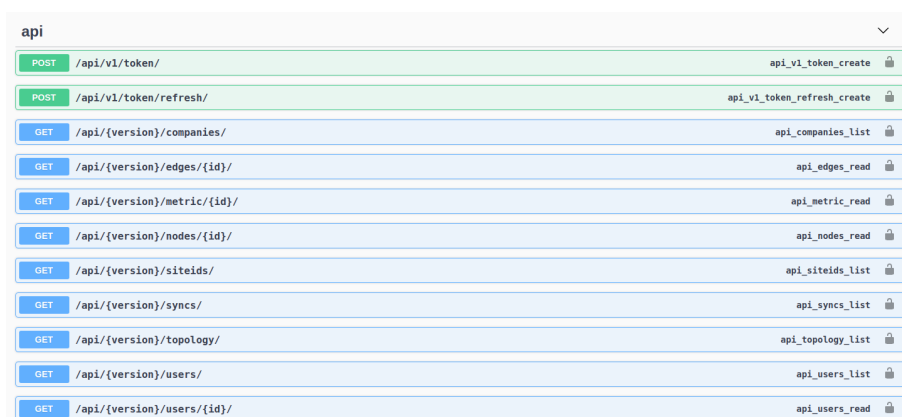
If a sync starts and another sync with the status running is still present in the database the new sync will get the status skipped and the sync is ended.

Those sync objects can also be queried by the frontend under the `api/v1/syncs/` 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.

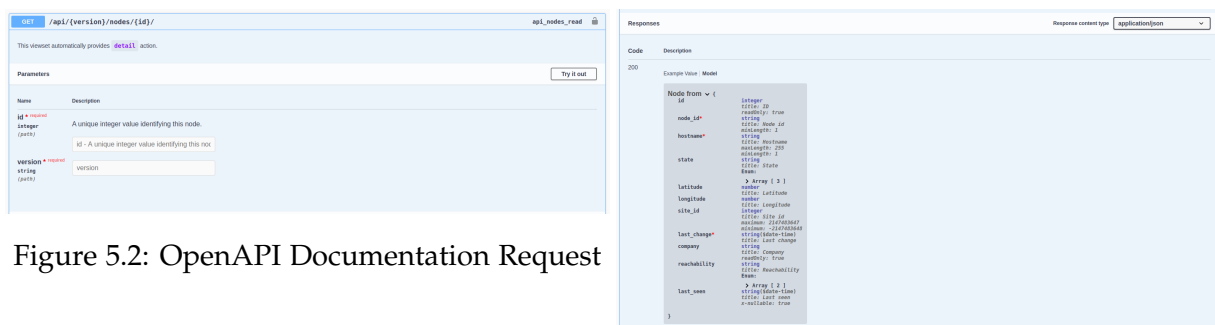
In a first step it is possible to see all available API endpoints.



Method	Endpoint	Action
POST	/api/v1/token/	api_v1_token_create
POST	/api/v1/token/refresh/	api_v1_token_refresh_create
GET	/api/{version}/companies/	api_companies_list
GET	/api/{version}/edges/{id}/	api_edges_read
GET	/api/{version}/metric/{id}/	api_metric_read
GET	/api/{version}/nodes/{id}/	api_nodes_read
GET	/api/{version}/siteids/	api_siteids_list
GET	/api/{version}/syncs/	api_syncs_list
GET	/api/{version}/topology/	api_topology_list
GET	/api/{version}/users/	api_users_list
GET	/api/{version}/users/{id}/	api_users_read

Figure 5.1: 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.



**Request Parameters:**

Name	Description
<b>id</b> (required)	A unique integer value identifying this node.
<b>version</b> (required)	A unique integer value identifying this node.

**Response:**

```

{
  "id": 1,
  "name": "Node 1",
  "description": "A unique integer value identifying this node.",
  "version": 1,
  "latitude": 1.0,
  "longitude": 1.0,
  "site_id": 1,
  "last_change": "2020-01-01T00:00:00Z",
  "company": "Company",
  "reachability": true,
  "last_seen": "2020-01-01T00:00:00Z"
}

```

Figure 5.2: OpenAPI Documentation Request

Figure 5.3: 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 [33], 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 *django-rest-framework-simplejwt* and include it in the Django Rest Framework settings.

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

Listing 5.6: DRF configuration in settings.py

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

```
urlpatterns = [
    path('api/v1/token/', TokenObtainPairView.as_view(), name='token_obtain_pair'),
    path('api/v1/token/refresh/', TokenRefreshView.as_view(), name='token_refresh'),
]
```

### Listing 5.7: 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 1 day and the refresh token lifetime to 7 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 username and password to the `api/v1/token/` endpoint. A request and response using curl could look like the following.

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

...
{
  "access": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX3BrIjoxLCJ0b2t1b190eXB1IjoieWYWNjZXNzIiwiaWF0IjE6IuKYgyISImV4cCI6MTIzNDU2LjQdGkioiOjMmZDZmOWQ1ZTFhNmOMmU4OTQ5MzVlMzYyYmNhOGJgYSJ9.NHlztMGER7UADHZJlxNGOWSi22a2KaYSfd1S-AuT7lU",
  "refresh": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyX3BrIjoxLCJ0b2t1b190eXB1IjoicmVmcmVzaCIsImVubGRfc3R1ZmYiOiLimIMiLCJleHAiOjIzNDU2NywiaWRpIjoizGUxMmY0ZTY3MDY4NDI3ODg5ZjElYWMyNzcwZGEwNTeifQ.eAoAYkSJjoWHlboshQAATkf8G3yn0kapko6HFRt7Rh4"
```

Listing 5.8: 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 without providing the username and password every time.

### Fetching metrics

An optional requirement was to be able to display metrics for edges. If we handled the metrics in the same fashion as we did for the other resources and load them in the scheduled task directly, it would require an enormous amount of Cisco vManage API request. Due to these performance reasons we decided not to fetch metrics periodically but rather requested it lazily as needed. To do so, we created the `api/v1/metric/` API endpoint which accepts a single integer parameter as the edge-id. When this endpoint is called, the request is being processed by the `MetricViewSet` class. Normally a view consists a queryset which just performs a query to the database and returns the corresponding objects. Since we decided to fetch the metrics of an Edge lazily we had to override the `get_queryset` function of the `MetricViewSet` class.

To strictly separate the fetching of the metric from the `MetricViewSet` class, we implemented these functions in another file and imported them into the `views.py` file.

```
from api.tasks import create_session, fetch_token
from api.views_helper import get_metric, process_metric

class MetricViewSet(DetailViewSet):
    """
    This viewset automatically provides 'detail' action.
    """
    permission_classes = [permissions.IsAuthenticated]

    def get_serializer_class(self):
        if self.request.version == 'v1':
            return MetricSerializerVersion1
        return MetricSerializerVersion1

    def get_queryset(self, *args, **kwargs):
        edge_id = int(self.kwargs.get('pk'))
        self.fetch_metric(edge_id)
        return Metric.objects.all()

    def fetch_metric(self, edge_id):
        edge = Edge.objects.get(id=edge_id)
        session = create_session()
        token_session = fetch_token(session)
        metrics = get_metric(token_session, edge)
        process_metric(edge, metrics)
```

Listing 5.9: MetricViewSet Class to return metrics for an Edge

The `get_queryset` function calls the `fetch_metric` function. This function gets the requested edge out of the database and updates the corresponding metrics object in the database after it has retrieved the latest metrics from the Cisco vManage API.

Retrieving the metrics for an edge is not as easy as it could be. vManage does not store the metrics for an edge (IPsec tunnel) on the IPsec tunnel itself but on the node (device) that it is connected to. This means that if we want to get the metrics for an edge we need to get all the metrics from a device and filter the metrics that relate to the requested edge.

After this function has done its job, the `get_queryset` function queries the database and returns the updated metrics for the requested edge. If it was not possible to get the requested metrics the endpoint will return an internal server error.

### 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 the vManage has the correct format, we decided to use a json schema validator. After some research we came up with the Python library `jsonschema`. It provides an easy way to define a json schema and validate the response from the vManage API against that schema. 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, the error is raised and the fetch of the topology will abort. 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 5.10: vManage response Validation

### User Management

The built-in user management [18] solution was one of the major criteria why we have chosen Django. It is already present in the list of installed Django apps by default when creating a Django project. It provides user, group, permissions and some other features which can be used out of the box.

```
INSTALLED_APPS = [
    'django.contrib.auth',
    'django.contrib.contenttypes',
    ...
]
```

Listing 5.11: User management configuration in settings.py

#### 5.1.2 Frontend

##### JWT client side management

As described in the backend authentication section 5.1.1, we are fetching both tokens but store either just one token (access) or two (access and refresh) at the client. If a refresh token exists and the access token is outdated, the frontend automatically queries a new access token and continues with the request of the resources.

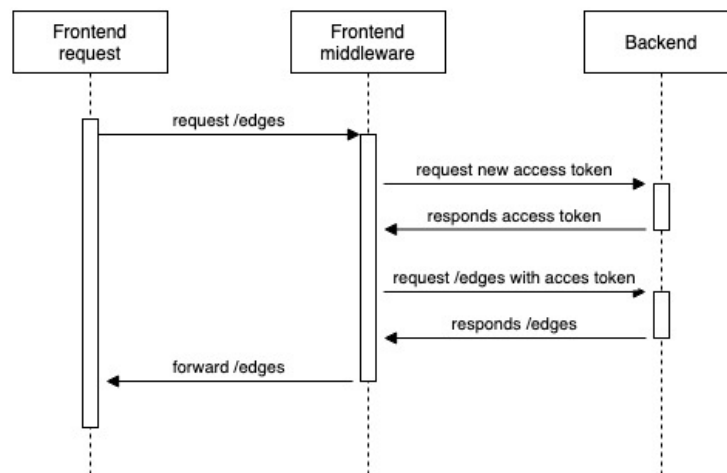


Figure 5.4: JWT mechanism if access token is outdated

With this approach the user can use the application without sensing an interruption.

Another aspect is security. What is the best way to store JWT [33] locally over a single page application like React? Mainly two attacks could harm us if we handle our authentication over JWT. Cross-side-request-forgery (CSRF) and Cross side scripting (XSS). While the danger from a cross-side scripting attack is limited due the built-in protection features by React, the CSRF is trickier.

To protect against CSRF attacks, the web application must prevent another website from accessing a stored JWT. Due to this, the question of how to protect against a CSRF attack boils down to how we store the JWT on the client. In a browser environment we can choose between the session storage, the local storage and a cookie. Based on some research [35] we came to the conclusion that the easiest and safest way is to use the local storage and omit the traditionally used CSRF-token.

### State management

A frontend consists of two type of states, a local state and a global state. The local state is to be preferred over a global state, because it reduces the complexity and improves the modularity. However, there is some data that is accessed all over the place and as a result needs to be stored globally. For this problem there are a lot of different ways how to solve this. Most of them include an external package to be installed. Nevertheless, we think a clean frontend solution only has a small global state and does not need an extra technology to manage it. Especially after the release of React 16.8, which includes react hooks[32], simple solutions are preferable.

In the frontend we have a global notification management, which is required by a lot of components scattered all over the application. In combination with react hooks we used a so-called context. The context is a singleton with a state and a set of functions to modify it. Components nested in such a context can now access the resources. This reduces the amount of parameters which would be needed to pass down to the respective component. On the other hand, a context complicates the testing, because an extra mock context needs to be added.

### Lazy loading on click

During the development of the application, we already encountered performance problems in the beginning. Especially during the initial render of the page. The cause of this problem is the amount of data fetched from the backend. Depending on the internet speed even 350KB of data can lead to disturbance of the user experience.

Inspired by the dynamic proxy pattern, which we learned in one of our lectures, we decided to create a trimmed version of the topology. This trimmed version only contains enough information to render the resources on the screen. If the user now clicks on an edge or node, he triggers a separate request to the backend to fetch the detailed information about the respective resource. The response payload to this request however is much smaller and does not interrupt the user experience.

After we fetched the resources once, we cache the result in memory so that we do not need to request the same details on the same resource again.

### Visualization of the edges

The render of the edges was one of the most complicated challenges in the frontend.

To make an edge visually appealing, we wanted to implement them as curves. Although we used a leaflet plugin which provides us with a Br zier curve [7], we still had to calculate the shifted middle point ourselves.

We have two points, the one from the origin node and the one from the destination node. In the first step we removed the global offset of the two coordinates. The distance between point A to point B could now be viewed as a vector.

```
const offsetX = to[1] - from[1]
const offsetY = to[0] - from[0]
```

Listing 5.12: Remove offset

Then we mapped the Cartesian vector to a polar representation.

```
const r = Math.sqrt(offsetX ** 2 + offsetY ** 2)
const theta = Math.atan2(offsetY, offsetX)
```

Listing 5.13: Map to polar representation

If we now select the middle of the polar vector, add a random offset to the angle and calculate the new sinus and cosines, we will receive the coordinates of the shifted middle point. Finally, we only needed to convert the polar vector back to the Cartesian and add the global offset to the middle point.

```
const thetaOffset = 3.14 / 10
const r2 = r / 2 / Math.cos(thetaOffset)
const theta2 = theta + thetaOffset
const midpointX = r2 * Math.cos(theta2) + from[1]
const midpointY = r2 * Math.sin(theta2) + from[0]
```

Listing 5.14: Add offset and map back to Cartesian space

Since multiple edges share the same nodes, we had the problem of overlaying edges. We did not realize this issue until we did the system tests at the end of the term project.

The solution was now to multiply the middle point by a random factor, so that the edges spread a little bit and do not share the exact same path. But the randomness needed a seed in order to produce the same curve for the same edge. Otherwise, every new topology render

would cause a change to the path for all curves. This is not feasible since this is really irritating for the user.

Eventually we had built a function which calculates the middle point of the Brézier curve for all edges. We are aware that this function could be enhanced, but we did not have the resources to apply the improvements.

```
const midPoint = (from: [number, number], to: [number, number], seed: string) => {
  const randomGenerator = seedrandom(seed)
  const offsetX = to[1] - from[1]
  const offsetY = to[0] - from[0]

  const radius = Math.sqrt(offsetX ** 2 + offsetY ** 2)
  const theta = Math.atan2(offsetY, offsetX)
  const thetaOffset = 3.14 / 10
  const radius2 = radius / 2 / Math.cos(thetaOffset)
  const theta2 = theta + thetaOffset
  const midpointX = radius2 * Math.cos(theta2) + from[1]
  const midpointY = radius2 * Math.sin(theta2) + from[0]
  return [midpointY + midpointY * (randomGenerator() - 0.5) * 0.1, midpointX + midpointX *
    (randomGenerator() - 0.5) * 0.7]
}
```

Listing 5.15: Full function for edge path

## 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* [27] testing framework. On top of that we use the *react-testing-library* [36], which simplifies the testing with JSX. Similar to the *hot-module-replacement* of *nodejs* [30] we can use the **watch** functionality of *jest*, which speeds up the building 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 [16], which is *pytest*. 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 tests suites easily and enables us to clearly differentiate between unit and integration tests.

### 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 will be 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 systemtest 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 being done 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.

---

# 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.
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 Tuesday 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. For the meetings in which the advisor participates we will create minutes.

Besides the official meeting with the advisor we decided to have a sprint planning and review meeting on Tuesday 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: 15.09.2020 - 29.09.2020 (2 weeks)
- Elaboration: 29.09.2020 - 20.10.2020 (3 weeks)
- Construction: 20.10.2020 - 08.12.2020 (7 weeks)
- Transition: 08.12.2020 - 18.12.2020 (1.5 weeks)

Sprints:

- Sprint 1: 15.09.2020 - 29.09.2020
- Sprint 2: 29.09.2020 - 13.10.2020
- Sprint 3: 13.10.2020 - 27.10.2020
- Sprint 4: 27.10.2020 - 10.11.2020
- Sprint 5: 10.11.2020 - 24.11.2020
- Sprint 6: 24.11.2020 - 08.12.2020
- Sprint 7: 08.12.2020 - 18.12.2020

## 6.4 Software Development Process

To make the development process as easy as possible, we decided to use the provided GitLab instance by the IFS. We created two Git repositories to separate the frontend (React) and backend (Python) 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 [23] 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.

Because of our negative experiences with time tracking in the engineering project using GitLab issues and GTT, we decided to go with YouTrack. 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.

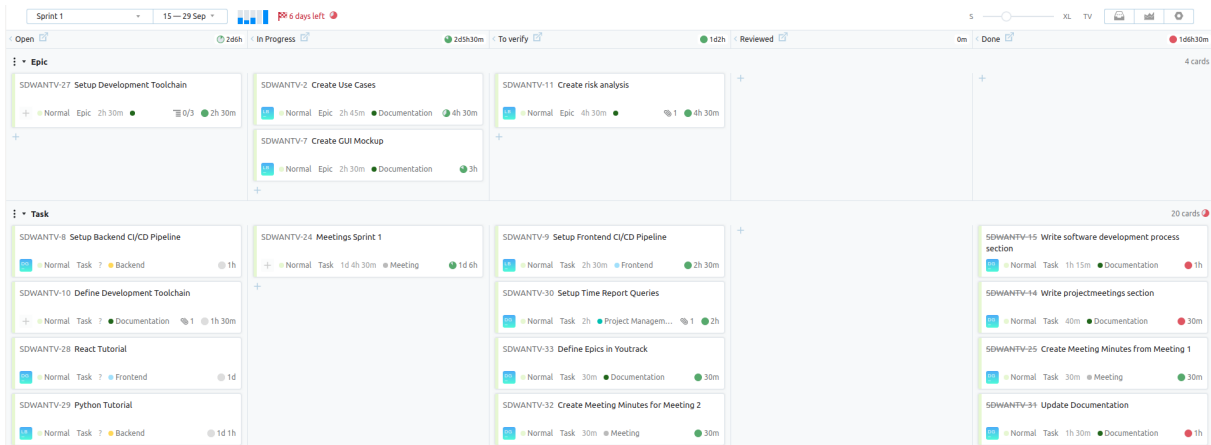


Figure 6.1: YouTrack Sprint Planning

To create our issues we decided to use the 3 issue types *Bug*, *Task* and *Epic*. We use issue type task to create simple issues and issues that belong to a higher level epic. Furthermore, we use bugs to document problems occurring 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.

The screenshot shows a YouTrack issue titled "SD-WAN Topology Viewer" created by Dominic Gabriel. The issue is visible to all users and has a normal priority. It is a task that is currently done, assigned to Dominic Gabriel, and is part of Sprint 1. The estimation is 4 hours, and the milestone is M1. The spent time is 4 hours and 15 minutes. The phase is Inception, and the component is Documentation. The issue has two activity entries: one by Dominic Gabriel on 16 Sep 2020 for 4h 00m of documentation work, and another by Lars Barmettler on 21 Sep 2020 for 15m of documentation review.

Field	Value
Project	SD-WAN Topology Viewer
Priority	Normal
Type	Task
State	Done
Assignee	Dominic Gabriel
Sprints	Sprint 1
Estimation	4h
Milestone	M1
Spent time	4h 15m
Phase	Inception
Component	Documentation

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 have all messages at one location, we use a Slack Channel. Messages from YouTrack about issue tracking and from GitLab about CI pipeline runs and merge request are sent directly to Slack which informs us about new events.

The screenshot shows a Slack channel named "#gitlab" with a theme "Thema hinzufügen". A message from the GitLab app, dated Wednesday, 30. September, 20:13 Uhr, reports that Dominic Gabriel pushed to the master branch of the SA-SDWAN / Backend repository. The pipeline #44641 has passed with warnings in 03:02. The commit is "Update dockerfile to use a custom user". The failed stage is linting, and the failed job is linting.

Figure 6.3: Slack

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 statictic 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
P1	Prototype	0.0.1	14.10.2020
R1	Alpha Release	0.1.0	04.11.2020
R2	Beta Release	0.2.0	01.12.2020
R3	Final Release	1.0.0	08.12.2020

Table 6.2: SDWANTV Releases

## 6.6 Milestones

**M1 - End of Inception - 29.09.2020** Project plan is created, basic requirements are written down, a mockup and risk analysis are created. Moreover the toolchain is defined.

**M2 - End of Elaboration - 20.10.2020** Requirements fine-tuned, software architecture design defined, C4 and deployment diagrams are generated, domain analysis is completed, toolchain is setup 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 - Alpha Release - 10.11.2020** Submission of the MVP including Usability Tests.

**M4 - Beta Release - 01.12.2020** Submission of the Beta release and feature freeze which means that no additional features are going to be added to the software.

**M5 - End of Construction - 08.12.2020** All open bugs are fixed and the software is ready for the transition phase.

**M6 - Project Closure - 18.12.2020** All documents are finalized and ready to be handed in.

## 6.7 Project Plan

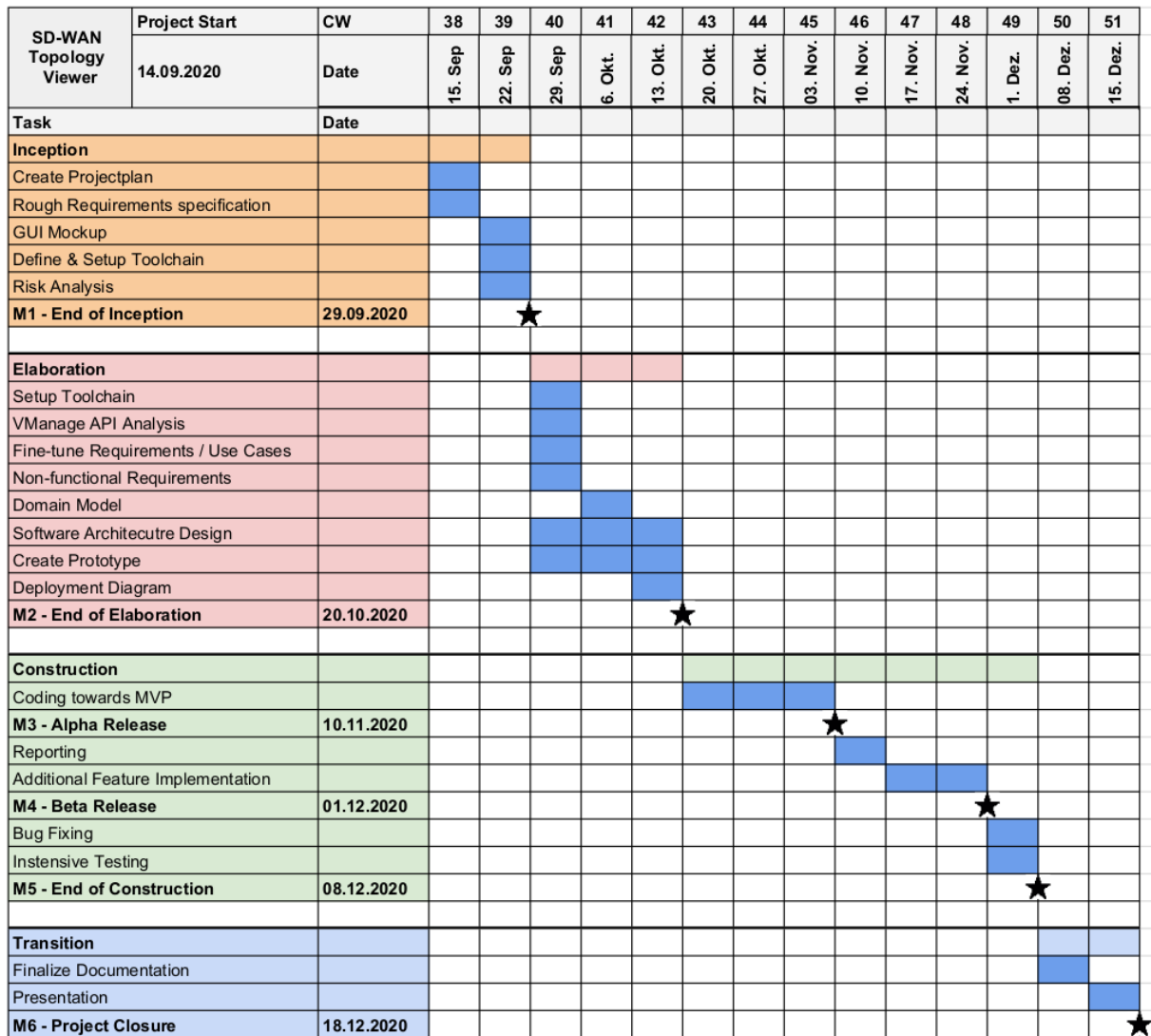


Figure 6.4: Project Plan

## 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.

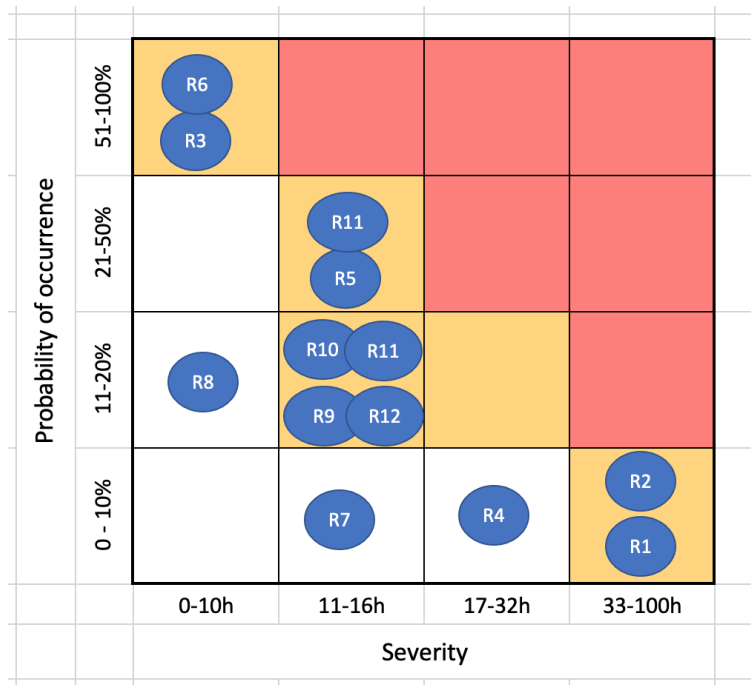


Figure 6.5: Risk analysis matrix

The full list with the preventions and the behaviour on entry can be found in the attachment E.

Nr.	Title
R1	Interpersonal conflicts
R2	Outage of team member
R3	Not testable
R4	Customer requirements not met
R5	Overstrain of the technical complexity
R6	Lifecycle of dependencies
R7	Lack of security
R8	Repository not accessible
R9	Vmanage API is not deterministic
R10	Software is too slow
R11	vManage is not scaleable
R12	Faulty State of Topology
R13	Our time getting out of sync

Table 6.3: Risk analysis

## 6.9 Logging

SDWANTV consists of multiple docker containers for which logging needs 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 loglevel 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*.

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

**Traefik reverse proxy** The Traefik reverse proxy inspects the incoming traffic and forwards it to the frontend or backend server. These requests are logged. Moreover, the Traefik proxy needs to be configured to write those access and error requests 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 container's 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 re-views	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 [31] - 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-Guidelines [4].

## 6.12 MVP

The MVP will include all the features of Use Case 1 defined in the requirements section 2.3.1. This will bring the following features:

- The whole topology can be monitored on the website after a successful login. All connections will be displayed with simple lines between the customer location and the remote stations of KSAT.
- The displayed connections will be updated on a regular basis.
- It is possible to zoom in on the map to get a closer look.

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

## 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

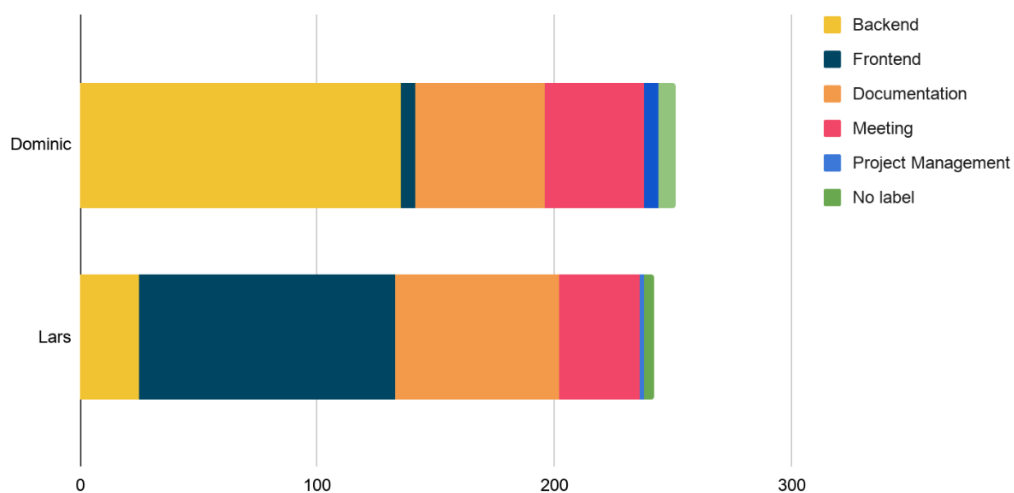


Figure 7.1: Hours spent per component per team member

Member	Time spent
Dominic Gabriel	252h
Lars Barmettler	241h
<b>Total</b>	<b>494h</b>

Table 7.1: Working times per team member

### 7.1.2 Project phases

#### Time spent per phase

Phase	Time estimated	Time spent
Inception	63h	63h
Elaboration	124h	145h
Construction	234h	247h
Transition	45h	58h

Table 7.2: Time spent per project phase

#### Issues per phase

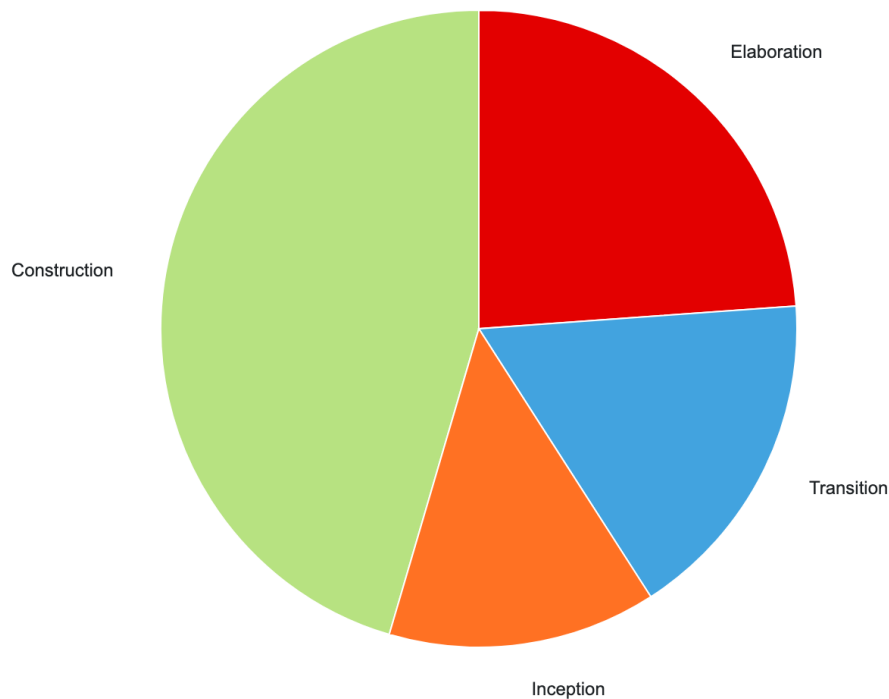


Figure 7.2: Cake diagram issues per phase

Phase	Number of issues
Inception	24
Elaboration	42
Construction	80
Transition	30

Table 7.3: Issues per phase

### 7.1.3 Task types

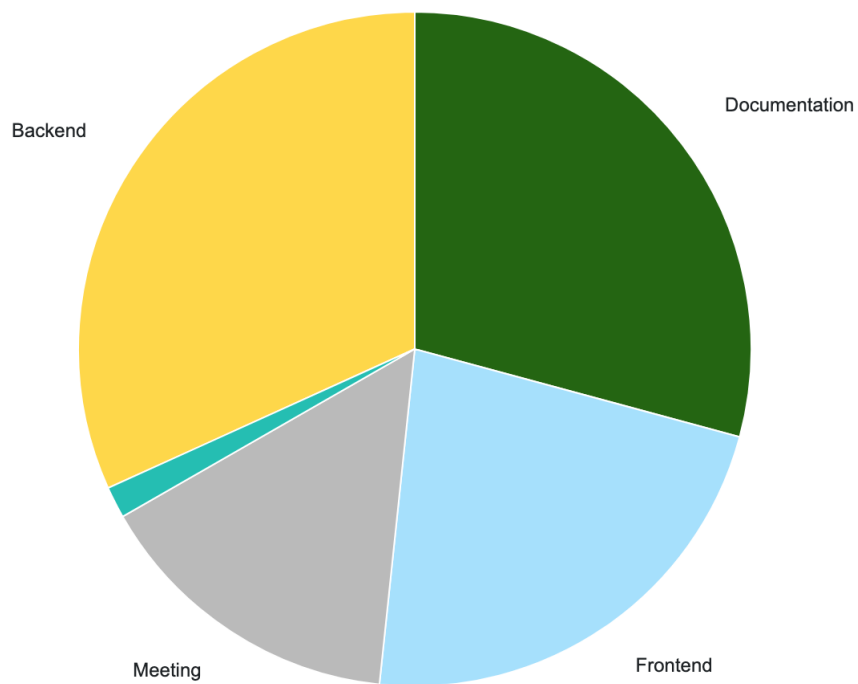


Figure 7.3: Cake diagram hours spent per Issue type

Type	Time estimated	Time spent
Backend	154h	161h
Frontend	105h	114h
Documentation	103h	125h
Meetings	87h	76h
Project Management	6h 30m	7h 40m
No type	11h	10h

Table 7.4: Time spent per task type

### 7.1.4 Milestones

#### Time spend per milestone

Type	Time estimated	Time spent
M1	63h	63h
M2	113h	111h
M3	103h	123h
M4	110h	101h
M5	30h	30h
M6	41h	58h

Table 7.5: Time spent per milestone

#### Issues per Milestone

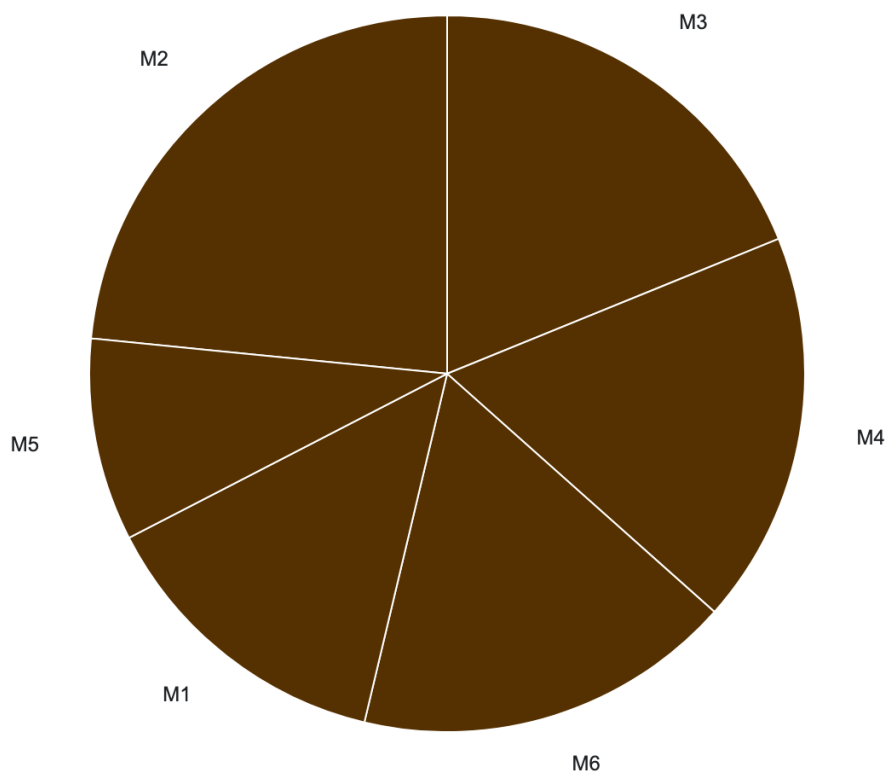


Figure 7.4: Cake diagram issues per milestone

Milestone	Number of issues
M1	24
M2	41
M3	33
M4	31
M5	16
M6	30

Table 7.6: Issues per milestone

## 7.2 Code statistics

The code metrics are obtained with Sonarqube [34] and GitLab repository statistics.

Tier/Metric	Backend	Frontend	Total
Lines of Code	1500	2700	4200
Statements	867	835	1702
Comments (%)	7.8%	3.1%	5.45%
Commits	103	66	169
Testing coverage	64%	81%	72.5%
Docker image size	35.82 MiB	42.00 MiB	77.82 MiB
Cyclomatic complexity	135	419	554
LoC Docker configuration	799	83	882

Table 7.7: Code statistics

# **Part III**

# **Appendix**

## Appendix A

---

# User Manual

---

The installation guide provides information about how to install and configure SDWANTV. SDWANTV can be installed on every system that has docker [19] and docker-compose [20] installed. Furthermore it describes some operational tasks that may be necessary.

### A.1 Installation

The installation section describes the steps how to install docker [19] and docker-compose [20] on the system, as the requirements to run SDWANTV. The installation guide is based on a Ubuntu 20.04 server.

#### Install docker

The installation guide is based on the official docker installation guide [25].

Update the system.

```
sudo apt update
```

Install all docker dependencies.

```
sudo apt install apt-transport-https ca-certificates curl software-properties-common
```

Download and add the docker repository key.

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

Add the docker repository.

```
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu  
$(lsb_release -cs) stable"
```

Update the package list.

```
sudo apt update
```

Check which version of docker is installed or will be installed.

```
apt-cache policy docker-ce
```

Install docker

```
sudo apt install docker-ce
```



Add the current logged in user to the docker group. This will allow the user to run docker commands without sudo.

```
sudo usermod -aG docker ${USER}
```

After the user was added to the group, re-login to make it effective. Check if the user was added to the user group. This should list the docker group.

```
id -nG
```

At this point docker was successfully installed.

### Install docker-compose

The installation guide is based on the DigitalOcean docker-compose installation guide [26].

Check the latest release of docker-compose on the docker-compose release GitHub page [21]. At the time of this writing the latest version was 1.27.4.

Download the docker-compose binary and place it in /usr/local/bin/docker-compose.

```
sudo curl -L "https://github.com/docker/compose/releases/download/1.27.4/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
```

Make the docker-compose binary executable.

```
sudo chmod +x /usr/local/bin/docker-compose
```

Check the version of docker-compose.

```
docker-compose --version
```

At this point all requirements for running SDWANTV were successfully installed.

## A.2 Deployment

To deploy SDWANTV first login to the container registry of the GitLab server.

```
docker login gitlab.dev.ifs.hsr.ch:45023
```

After the login succeeded get the docker-compose.yaml file from the docker compose yaml section A.8 and place it in a file on the server. Customize the docker containers if needed with the parameters from the deployment configuration section A.3.

Start SDWANTV in the daemon mode.

```
docker-compose up -d
```

If this finishes successfully, SDWANTV is available on the systems hostname or ip address on port 80 and 443. However, port 80 will be redirected to 443. The single entrypoint is served by a traefik container, which by default creates a default traefik certificate for serving https traffic. To see how to use own ssl certificates see section configure https certificates A.4.

## A.3 Deployment configuration

Both SDWANTV images, backend and frontend can be configured with environment variables. The environment variables are described in this section.

## Backend

The following table lists the configurable parameters of the backend container. Also configure the same parameters for the `sdwantv\_beat` and `sdwantv\_celery` container.

**DEBUG** Activate debug mode for django. Only for development!

Default: 0

**DB.CONNECTION.STRING** Connection string to connect to the database.

Format: <db-type>://<username>:<password>@<host>:<port>/<database-name>

Example: postgres://sdwantv:sdwanTV2020@sdwantv\_db\_local:5432/sdwantv

Default: nil

**CELERY\_BROKER\_URL** URL for the celery broker backend. Typically redis or local.

Example: redis://sdwantv\_redis:6379/0

Default: nil

**SECRET\_KEY** Secret key to be used for Django.

Default: changeme

**WEBSERVER\_ARGS** Additional Gunicorn webserver arguments to start the webserver with.

Example: --log-level info

Default: nil

**ALLOWED\_HOSTS** Comma separated string of allowed hosts which are able to interact with the backend API.

Example: 127.0.0.1,localhost

Default: nil

**DJANGO\_LOG\_LEVEL** Change the Log Level.

Default: INFO

Possible values: DEBUG, INFO, WARNING, ERROR, CRITICAL

**DJANGO\_LOG\_FORMAT** Define another Log Format for log messages.

Default: '{asctime} %(name)-12s %(levelname)-8s %(message)s'

**VMANAGE\_URL** URL to the Cisco vManage API.

Example: https://vmanage:8443

Default: nil

**VMANAGE\_USERNAME** Username to authenticate at Cisco vManage API.

Default: nil

**VMANAGE\_PASSWORD** Password to authenticate at Cisco vManage API.

Default: nil

**VMANAGE\_FULL\_SYNC\_INTERVAL\_SECONDS** How often to query the Cisco vManage API for topology changes.

Default: 30 seconds

**KEEP\_DOWN\_NODES\_EDGES** Set to False if nodes and edges should be removed from the topology immediately when `reachability: down`. Otherwise they will be present in the topology for `KEEP_DOWN_NODES_EDGES_TIME_SECONDS` seconds and after that deleted.

Default: True

Possible values: True, False

**KEEP\_DOWN\_NODES\_EDGES\_TIME\_SECONDS** Define how long nodes and edges with `reachability: down` should be present in the topology until removed.

Default: 3600 seconds

## Frontend

The following table lists the configurable parameters of the frontend container.

**PORT** Port on which the frontend runs

Default: 3000

**TOPOLOGY\_UPDATE\_INTERVAL** Interval how often the backend is queried and therefore the topology updated. Note, that defining this value too low may cause delay on the backend and therefore negatively influence the performance of the backend.

Default: 30000 milliseconds

**SYNC\_UPDATE\_INTERVAL** Interval how often the sync status field should be updated.

Default: 10000 milliseconds

**SYNC\_THRESHOLD\_WARNING** Threshold after how many milliseconds without a successful sync the sync status will display a warning.

Default: 120000 milliseconds

**API\_URL** URL of the backend API. The URL needs to be accessible from the browser where the frontend is accessed.

Example: `https://localhost/api/v1` Default: `http://localhost:8000/api/v1`

## A.4 Configure https certificates

If no ssl certificates are provided, traefik creates self-signed certificates. Traefik will create a folder with the same name in the directory where the `docker-compose.yaml` is located and one level deeper two subfolders called `config` and `certs`.

Add the own certificate and the key to that certificate into the folder `traefik/certs/` and place the `certs.toml`, which can be found in the traefik config file A.9 section, into the folder `traefik/config/`.

```
sdwantv/
├── docker-compose.yaml
└── traefik/
    ├── certs/
    │   ├── cert.crt
    │   └── privkey.key
    └── config/
        └── certs.toml
```

To create an own self-signed certificate use the command below.

```
openssl req -newkey rsa:4096 -nodes -sha256 -keyout certs/privkey.key \
-x509 -days 365 -out certs/cert.crt
```

## A.5 Operational tasks

### Change docker image parameters

To change the configuration of the docker images the application needs to be stopped. See section termination A.7 how to stop the application. After the application was stopped and the environment parameters of the docker images have been adjusted, it can be started again.

## Cleanup sync tasks

If the application was stopped because docker environment parameters needed to be changed or because of a server restart, it may happen that the backend was in the middle of a topology fetch. This will leave the created sync object in a running state. Execute the following command in the backend docker container to cleanup all running sync objects.

```
docker exec -it be_sdwan_tv_backend_1 sh
~/app $
~/app $ python manage.py cleanup_syncs
Info: Using postgresql Database on Host sdwan_tv_db
Info: Using Celery broker redis://sdwan_tv_redis:6379/0

Successfully cleaned up running syncs
```

## Backend healthcheck

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

<https://localhost/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.1: Html friendly backend healthcheck

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

<https://localhost/api/ht?format=json>

This produces the json output below.

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

## User management

To change a users password or to add new users, use the Django administration endpoint `/admin` can be used. Only administrative users are able to login and add new users.

Unfortunately, because of a bug, there is no CSS available at the moment. For more information refer to the technical debt section 1.6.3.

### 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.6 Docker configuration

Using docker as docker engine is great. But without also using an application orchestration engine like Kubernetes it is hard to use in production. To make docker usable in production it is possible to configure docker.

### Logging

By default container logs are written to the Stdout stream and collected by docker. Container logs can be check with the `docker logs <container-name>` command. For further investigation, monitoring or also alerting this is not enough. Due to this it is possible to forward the container logs to another log system. This could be for example syslog or also into a file to be read by a monitoring system. The docker logging website [13] provides more information about how to configure logging for docker.

### Backup

Docker uses docker volumes to store data persistent on the file system and make them available again after a container has crashed or restarted. However, these volumes should also get backedup. This blogpost [6] describes how to backup docker volumes.

### Systemd

On Ubuntu the docker daemon is managed by systemd. To make sure docker and containers are automatically started if the system reboots configure the docker systemd daemon like described here [14] [12].

## A.7 Termination

To stop the application the `docker-compose.yml` file is required to be present on the system. If the file is available simply run the command below. This will stop all running containers of SDWANTV but not removing the containers, images, networks or volumes.

```
docker-compose -f docker-compose.yml down
```

However if it is required to not stop the containers but also delete the containers, images, networks and also volumes, this can be achieved with the command below.

```
docker-compose -f docker-compose.yml down --rmi all -v
```

## A.8 Docker-compose Yaml

This is the `docker-compose.yaml` that can be used for a production environment.

```
version: '3.7'

services:
  sdwantv_traefik:
    container_name: sdwantv_traefik
    image: traefik:v2.3.1
    restart: always
    ports:
      - 80:80
      - 443:443
      - 8080:8080
    command: >
      --api.insecure=true
      --log.level=INFO
      --providers.docker=true
      --providers.docker.exposedByDefault=false
      --providers.file.directory=/config
      --providers.file.watch=true
      --entrypoints.http.address=:80
      --entrypoints.websecure.address=:443
      --entrypoints.http.http.redirections.entryPoint.to=websecure
      --entrypoints.http.http.redirections.entryPoint.scheme=https
      --entrypoints.http.http.redirections.entrypoint.permanent=true
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - ./traefik/config/:/config:ro
      - ./traefik/certs/:/certs:ro
    networks:
      - transit
      - transit_sdwantv

  sdwantv_frontend:
    image: gitlab.dev.ifs.hsr.ch:45023/sa-sdwan/fe:v1-0-0
    restart: always
    ports:
      - "3000:3000"
    environment:
      - TOPOLOGY_UPDATE_INTERVAL=20000
      - API_URL=https://10.20.1.16/api/v1
    labels:
      - traefik.enable=true
      - traefik.docker.network=transit_sdwantv
      - traefik.http.services.frontend.loadbalancer.server.port=3000
      - traefik.http.routers.frontend.entrypoints=websecure
      - traefik.http.routers.frontend.tls=true
      - traefik.http.routers.frontend.rule=PathPrefix('/')
    networks:
      - intern_sdwantv
      - transit_sdwantv
    depends_on:
      - sdwantv_backend

  sdwantv_backend:
    image: gitlab.dev.ifs.hsr.ch:45023/sa-sdwan/be/backend:latest
    restart: always
    ports:
      - "8000:8000"
    environment:
      - DB_CONNECTION_STRING=postgres://sdwantv:sdwanTV2020@sdwantv_db:5432/sdwantv
      - CELERY_BROKER_URL=redis://sdwantv_redis:6379/0
```

```

- ALLOWED_HOSTS=10.20.1.16
- DJANGO_LOG_FORMAT=%(name)-12s %(levelname)-8s %(message)s
- VMANAGE_URL=https://152.96.9.236:8443
- VMANAGE_USERNAME=sdwan
- VMANAGE_PASSWORD=ins@sdwan
labels:
- traefik.enable=true
- traefik.docker.network=transit_sdwantv
- traefik.http.services.backend.loadbalancer.server.port=8000
- traefik.http.routers.backend.entrypoints=websecure
- traefik.http.routers.backend.tls=true
- traefik.http.routers.backend.rule=PathPrefix('/api/v1') || PathPrefix('/api/ht') ||
  PathPrefix('/admin') || PathPrefix('/swagger')
networks:
- intern_sdwantv
- transit_sdwantv
- transit
depends_on:
- sdwantv_db
- sdwantv_redis
- sdwantv_celery

sdwantv_db:
  image: postgres:13.0-alpine
  ports:
    - 5432:5432
  restart: always
  environment:
    - POSTGRES_USER=sdwantv
    - POSTGRES_PASSWORD=sdwanTV2020
    - POSTGRES_DB=sdwantv
  volumes:
    - postgres_data:/var/lib/postgresql/data
  networks:
    - intern_sdwantv

sdwantv_redis:
  image: redis
  restart: always
  ports:
    - 6379:6379
  networks:
    - intern_sdwantv

sdwantv_celery:
  image: gitlab.dev.ifs.hsr.ch:45023/sa-sdwan/be/celery:latest
  restart: always
  volumes:
    - celery-state:/var/run/celery/
  environment:
    - DB_CONNECTION_STRING=postgres://sdwantv:sdwanTV2020@sdwantv_db:5432/sdwantv
    - CELERY_BROKER_URL=redis://sdwantv_redis:6379/0
    - ALLOWED_HOSTS=10.20.1.16
    - DJANGO_LOG_FORMAT=%(name)-12s %(levelname)-8s %(message)s
    - VMANAGE_URL=https://152.96.9.236:8443
    - VMANAGE_USERNAME=sdwan
    - VMANAGE_PASSWORD=ins@sdwan
  depends_on:
    - sdwantv_redis
    - sdwantv_db
  networks:
    - transit
    - intern_sdwantv

```

```

sdwantv_beat:
  image: gitlab.dev.ifs.hsr.ch:45023/sa-sdwan/be/beat:latest
  restart: always
  environment:
    - DB_CONNECTION_STRING=postgres://sdwantv:sdwanTV2020@sdwantv_db:5432/sdwantv
    - CELERY_BROKER_URL=redis://sdwantv_redis:6379/0
    - ALLOWED_HOSTS=10.20.1.16
    - DJANGO_LOG_FORMAT=%(name)-12s %(levelname)-8s %(message)s
    - VMANAGE_URL=https://152.96.9.236:8443
    - VMANAGE_USERNAME=sdwan
    - VMANAGE_PASSWORD=ins@sdwan
  depends_on:
    - sdwantv_redis
    - sdwantv_celery
    - sdwantv_db
  networks:
    - intern_sdwantv

volumes:
  postgres_data:
  celery-state:

networks:
  transit:
    name: transit
  transit_sdwantv:
    name: transit_sdwantv
    internal: true
  intern_sdwantv:
    name: intern_sdwantv
    internal: true

```

Listing A.1: docker-compose.yaml

## A.9 Traefik config file

This is the `certs.toml` that can be used for a production environment.

```

[[tls.certificates]]
  certFile = "/certs/cert.crt"
  keyFile = "/certs/privkey.key"

[tls.stores]
[tls.stores.default]
  [tls.stores.default.defaultCertificate]
    certFile = "/certs/cert.crt"
    keyFile = "/certs/privkey.key"

```

Listing A.2: Traefik config certs.toml



## Appendix B

---

# Systemtest protocol

---

The systemtest 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 in the appendix and is linked in the result column in the table.

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

UseCase	Implemented	Result	Status
UC1: Monitoring topology	yes	result B.1	passed
UC1.1: View node information	yes	result B.6	passed
UC1.2: Display connection metrics	yes	result B.7	passed
UC1.3: Display bandwidth of IP-sec tunnels	no	no result	not tested
UC1.4: Toggle Fullscreen	yes	result B.8	passed
UC2: Apply customer filter	yes	result B.9	passed
UC3: Manage Users for companies	no	no result	not tested
UC4: Apply connection filter	yes	result B.10	passed
UC5, UC6, UC6.1	no	no result	not tested

Table B.1: Systemtest protocol

## B.1 UC1: Monitoring topology: Test 1

Pull out the ethernet cable of Miami-a

### Initial state

All connections and all nodes, except the 4 nodes that are not connected to vManage, are up and available.

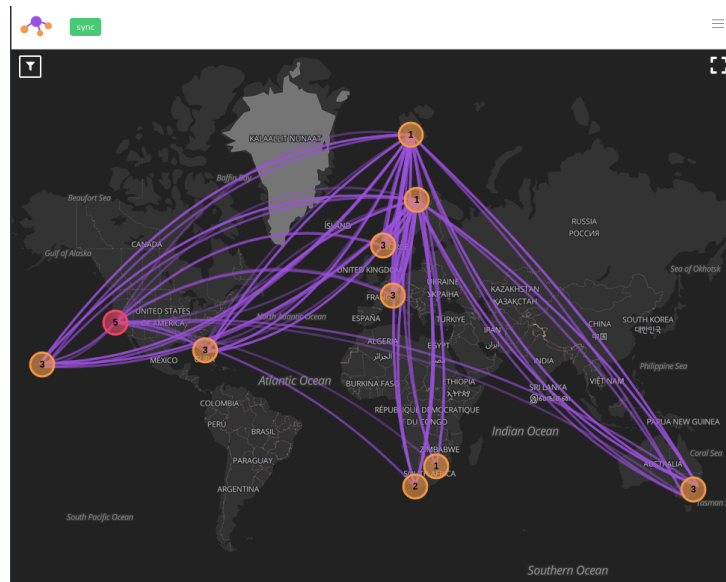


Figure B.1: Topology view

### Test procedure

Pulling the cable from port XXX 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 connection is gone and mark it as down in the database. The backend represents the gone connection in the API with a reachability set to down. The frontend marks the down connection red.

## Test results

The frontend reports the down edges and marks them as red. After 5 minutes the edges are deleted from the backend which shows up in the logs and afterwards also in the frontend.

The test was successful!

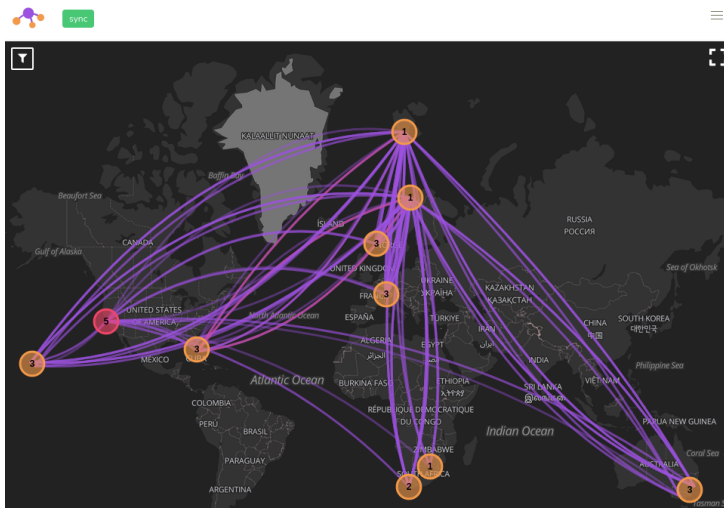


Figure B.2: Edge down

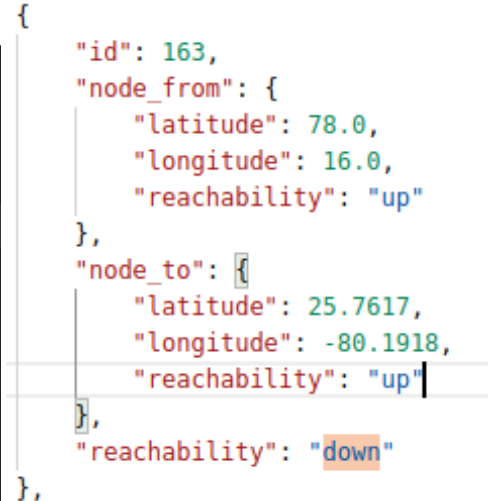


Figure B.3: API Edge down

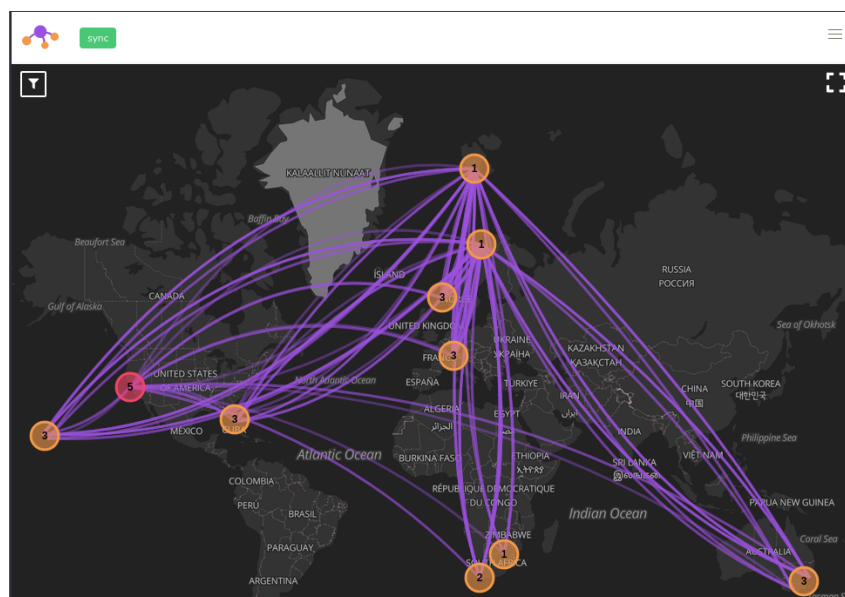


Figure B.4: Edge is gone

## B.2 UC1: Monitoring topology: Test 2

Power outage of node Miami-a.

### Initial state

All connections and all nodes, except the 4 nodes that are not connected to vManage, were up and available.

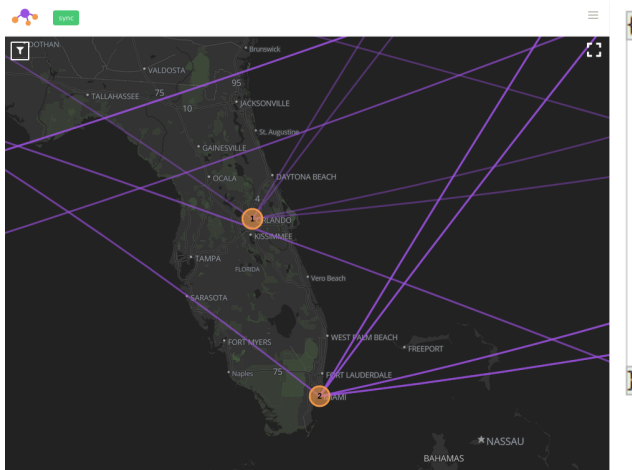


Figure B.5: Miami-a up

```
"id": 16,  
"node_id": "10.255.255.111",  
"hostname": "RS-Miami-a",  
"state": "green",  
"latitude": 25.7617,  
"longitude": -80.1918,  
"site_id": 11,  
"last_change": "2020-12-03T10:39:31.519000Z",  
"company": "KSAT",  
"reachability": "up",  
"last_seen": null
```

Figure B.6: API Miami-a up

### Test procedure

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

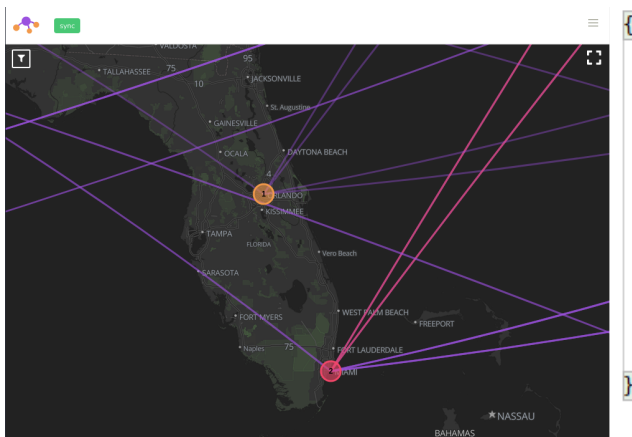


Figure B.7: Miami-a down

```
"id": 16,  
"node_id": "10.255.255.111",  
"hostname": "RS-Miami-a",  
"state": "green",  
"latitude": 25.7617,  
"longitude": -80.1918,  
"site_id": 11,  
"last_change": "2020-12-03T10:39:31.519000Z",  
"company": "KSAT",  
"reachability": "down",  
"last_seen": null
```

Figure B.8: API Miami-a down

## Test results

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

After 5 minutes the edges associated with node Miami-a will be removed from the map. But Miami-a will remain on the map and marked red.

The test was successful!

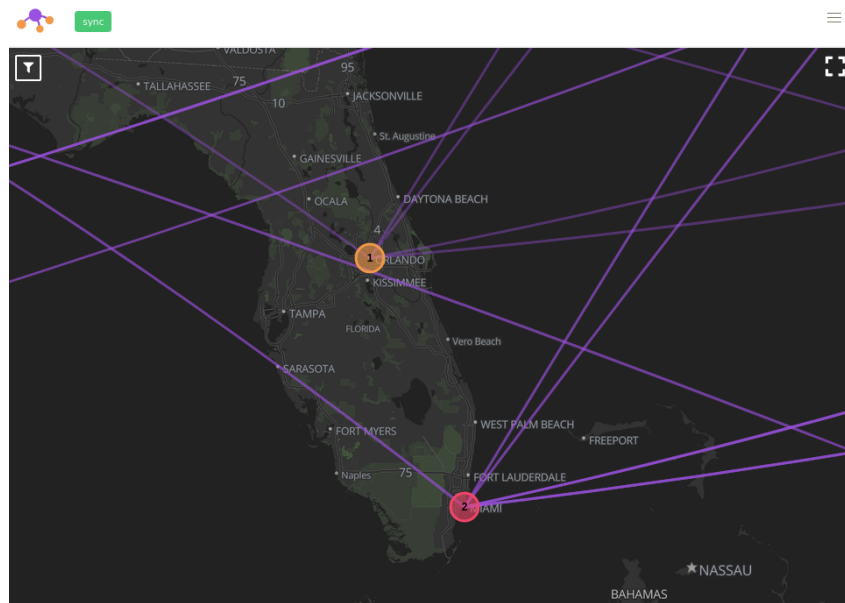


Figure B.9: Miami-a red and edges gone

## B.3 UC1: Monitoring topology: Test 3

Give the power back to node Miami-a.

### Initial state

Miami-a is down and marked as red. All connections are already removed from the map.

### 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!

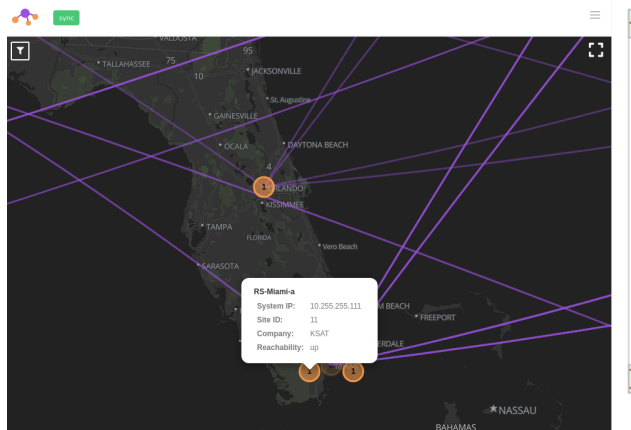


Figure B.10: Miami-a up again

```
"id": 16,  
"node_id": "10.255.255.111",  
"hostname": "RS-Miami-a",  
"state": "green",  
"latitude": 25.7617,  
"longitude": -80.1918,  
"site_id": 11,  
"last_change": "2020-12-03T10:58:05.267000Z",  
"company": "KSAT",  
"reachability": "up",  
"last_seen": null
```

Figure B.11: API Miami-a up again

## B.4 UC1: 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.

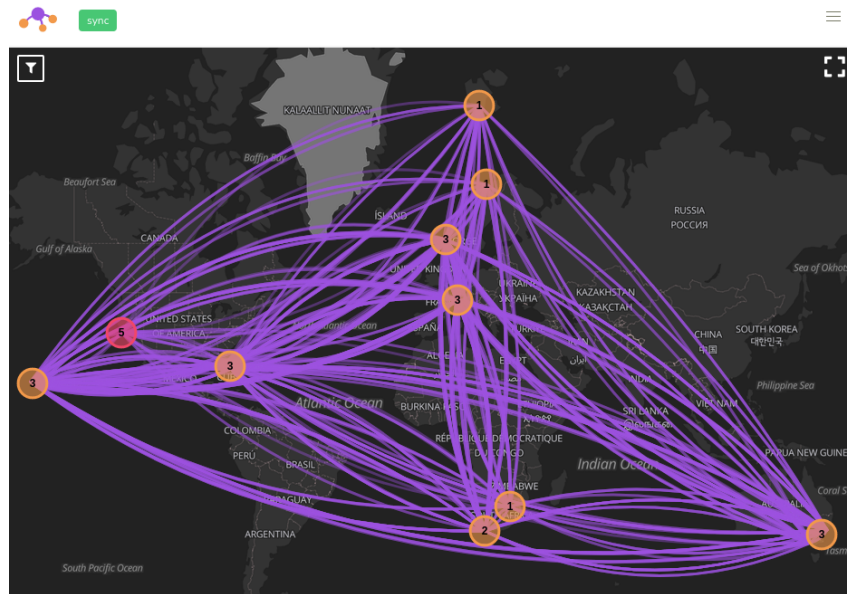


Figure B.12: Full-mesh topology view

### 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 edges being deleted and therefore in the status down on those.

## Test results

Down edges are marked red in the frontend.

After 5 minutes all red edges that were reported down are removed.

The test was successful!



Figure B.13: Policy applied

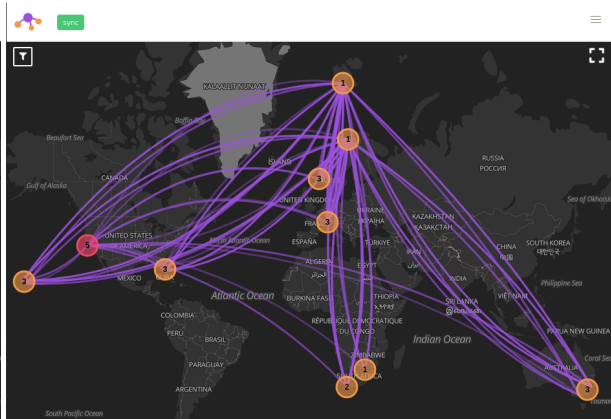


Figure B.14: Edges cleaned after 5 minutes

## B.5 UC1: Monitoring topology: Reported bugs

Despite all 3 tests were successful we also encountered 2 errors.

1. The frontend sometimes displays mpls and biz-internet edges from and to the same nodes over each other. This means that one edge is covered by the other and therefore is invisible and unclickable.
2. The second bug we reported was an uncaught ServerTimeout exception that occurred when the backend was querying the vManage API for edges of node Miami-a.

Both bugs were reported and a respective task was opened to fix them as soon as possible. Both of these bugs do not exist anymore and were successfully fixed.



## B.6 UC1.1: View node information: Test 1

### Initial state

All nodes and edges are rendered. No node or edge is selected.

### Test procedure

Selecting a node.

### 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!

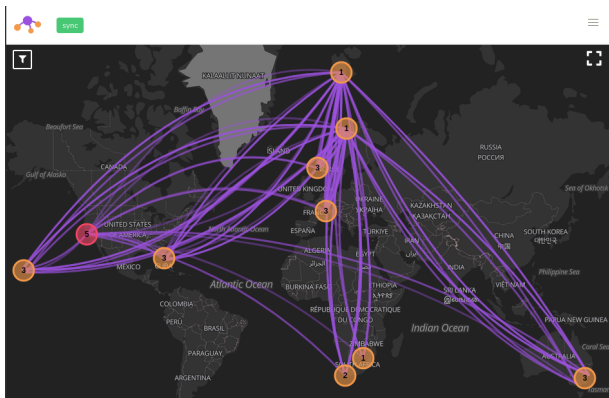


Figure B.15: Full topology view

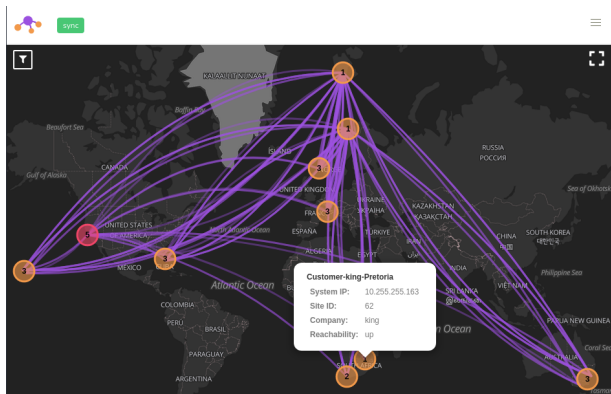


Figure B.16: View node information

## B.7 UC1.2: Display connection metrics: Test 1

### Initial state

All nodes and edges are rendered. No node or edge is selected.

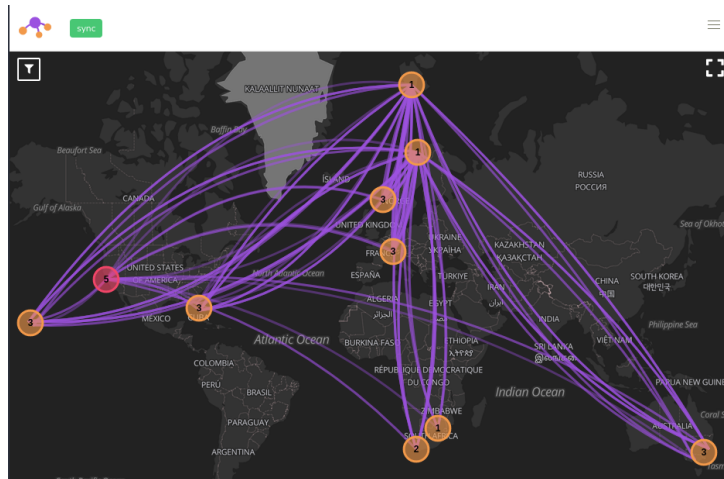


Figure B.17: Full topology view

### Test procedure

Selecting an edge.

### Test results

After an edge is selected a popup opens and displays the name and some other useful information of the edge. When selecting the same edge again the metrics will be queried again and updated. The new metric values are displayed.

The test was successful!

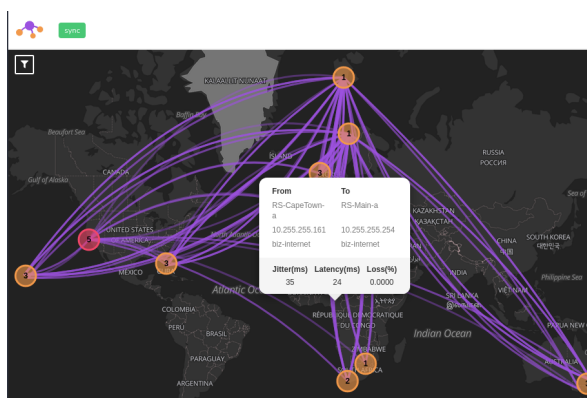


Figure B.18: Selected edge displays metrics

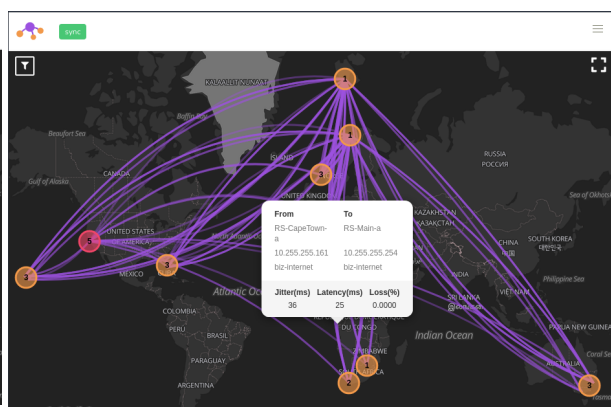


Figure B.19: Selected edge updated metrics

## B.8 UC1.4: Toggle fullscreen: Test 1

### Initial state

All buttons and the application header as well as the browser navigation are displayed.

### Test procedure

Click on the fullscreen button in the top right corner the application.

### Test results

The application will switch into fullscreen mode and hide the browser navigations and the application header.

After clicking again on the fullscreen button in the top right corner, the application will end the fullscreen mode. Browser navigations and the application header are back again.

The test was successful!

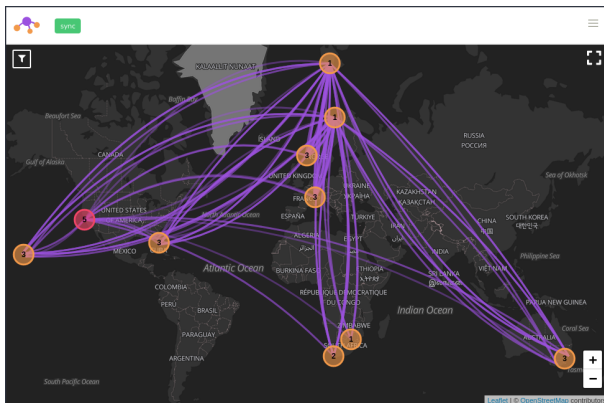


Figure B.20: Normal fullscreen

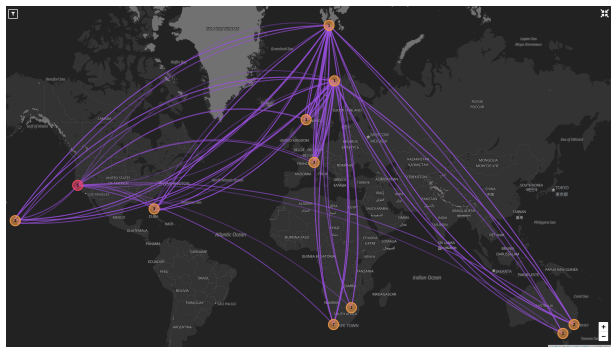


Figure B.21: Fullscreen view enabled

## B.9 UC2: Apply customer filter: Test 1

### Initial state

All edges and nodes are displayed and no filter is applied.

### 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 edges that belong to this company are displayed.

The test was successful!

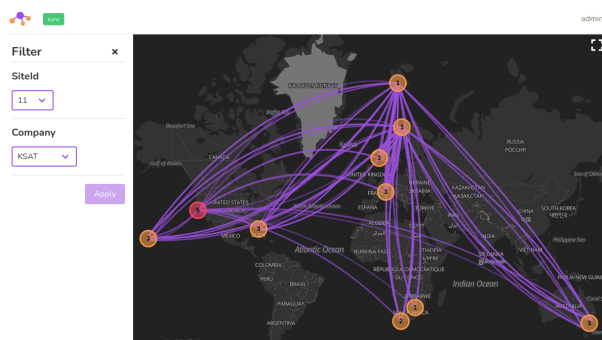


Figure B.22: Full topology view

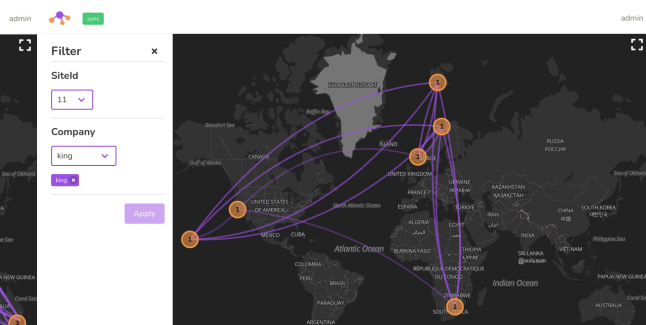


Figure B.23: Customer filter applied

## B.10 UC4: Apply connecton filter: Test 1

### Initial state

The initial state is that all edges and nodes are displayed and no filter is applied.

### Test procedure

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

### Test results

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

The test was successful!

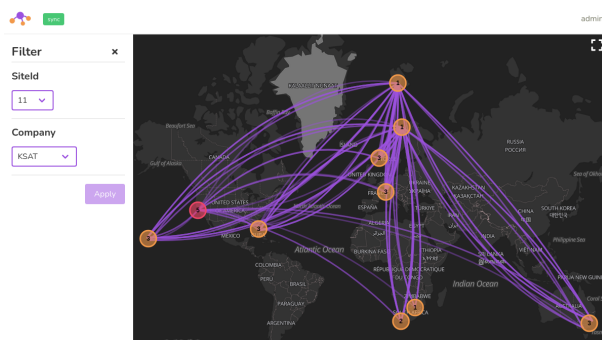


Figure B.24: Full topology view

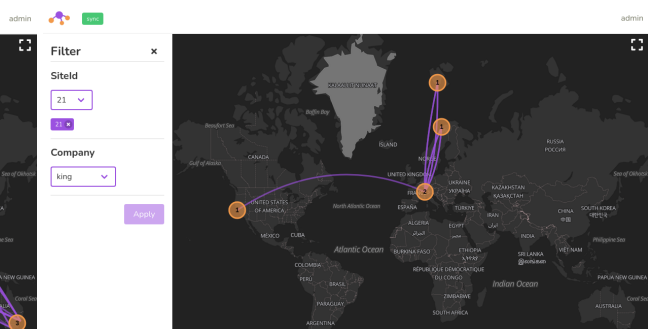


Figure B.25: Site-id filter applied

## Appendix C

---

# Non functional requirement testprotocol

---

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	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. 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\$0Hp8Lf90Uso36H96sllmdwUuF3KwtzBjpGSmj8J0o=

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

**System logs all relevant information** 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
sdwntv_backend_local_1 | django.server INFO      "OPTIONS /api/v1/token/ HTTP/1.1" 200 0
sdwntv_backend_local_1 | django.server INFO      "POST /api/v1/token/ HTTP/1.1" 200 524
sdwntv_backend_local_1 | django.server INFO      "OPTIONS /api/v1/synchs/ HTTP/1.1" 200 0
sdwntv_backend_local_1 | django.server INFO      "OPTIONS /api/v1/companies/ HTTP/1.1" 200 0
sdwntv_backend_local_1 | django.server INFO      "OPTIONS /api/v1/siteids/ HTTP/1.1" 200 0
sdwntv_backend_local_1 | django.server INFO      "OPTIONS /api/v1/topology HTTP/1.1" 200 0
sdwntv_backend_local_1 | django.server INFO      "OPTIONS /api/v1/users/1 HTTP/1.1" 200 0
sdwntv_backend_local_1 | django.server INFO      "GET /api/v1/synchs/ HTTP/1.1" 200 272
sdwntv_backend_local_1 | django.server INFO      "GET /api/v1/companies/ HTTP/1.1" 200 35
sdwntv_backend_local_1 | django.server INFO      "GET /api/v1/siteids/ HTTP/1.1" 200 61
sdwntv_backend_local_1 | django.server INFO      "OPTIONS /api/v1/topology/ HTTP/1.1" 200 0
sdwntv_backend_local_1 | django.server INFO      "OPTIONS /api/v1/users/1/ HTTP/1.1" 200 0
sdwntv_backend_local_1 | django.server INFO      "GET /api/v1/users/1/ HTTP/1.1" 200 72
sdwntv_celery_1 | [2020-12-12 12:30:45,041: WARNING/ForkPoolWorker-1] /usr/local/lib/py8
```

Figure C.3: 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** There is no frontend feature that modifies backend data because the backend only provides read-only API endpoints. Therefore, the backend can not fall into a failed state based on the frontend input.

**Feedback if input is invalid** The only 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.

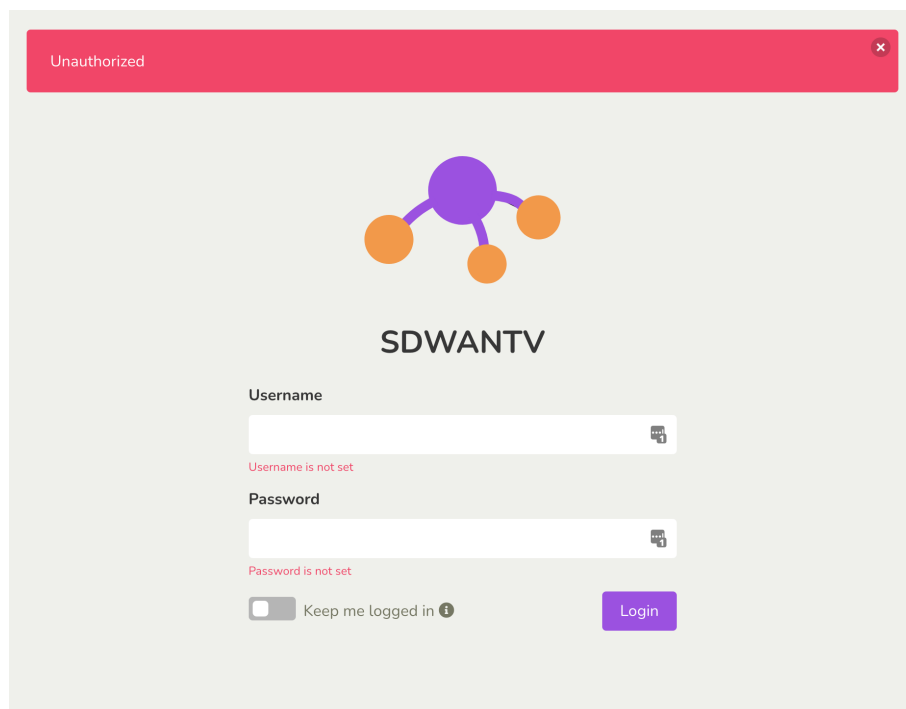


Figure C.4: Login failure message



## 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 [28] library before we access the property.

**Exception management** All exceptions thrown are handled directly in a multiple except statements. We ensure with a finally 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 not experience one failed test on our side. However we only tested the system for 24 hours.

## 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.

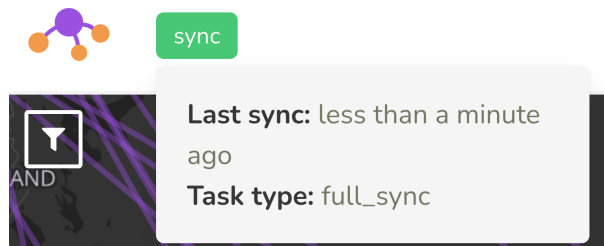


Figure C.5: Sync state if system runs successful

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

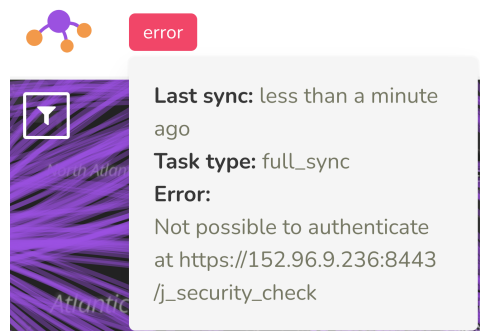


Figure C.6: Sync is in error state

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

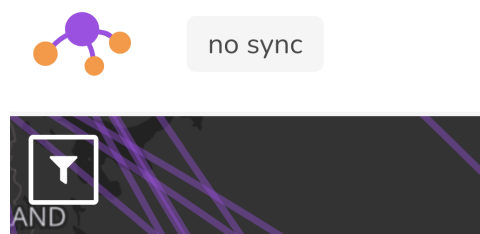


Figure C.7: Sync information could not be fetched

## 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.

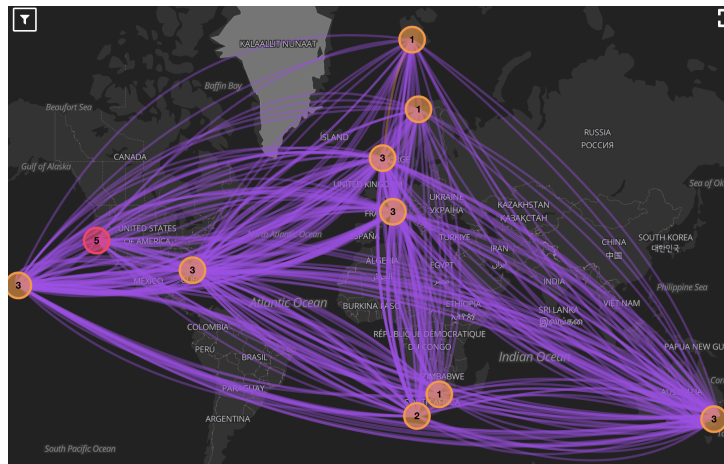


Figure C.8: Full match without policy applied

Over the vManage user interface we applied a policy. After the policy was applied to the nodes 50s later the change was visible in our frontend.

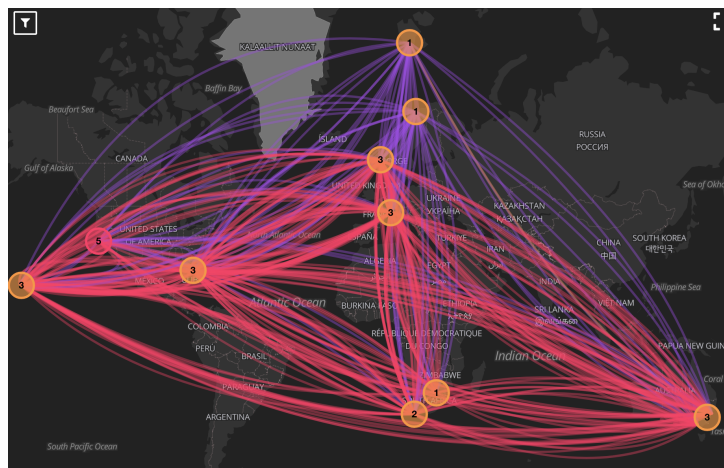


Figure C.9: Policy applied 50s later

The 50s 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 [9]. The result shows that the performance is far below the required threshold. The web application shows the first render in only 100ms and after 1.6s we have rendered the whole topology. So the initial render is much faster than the render of the topology.

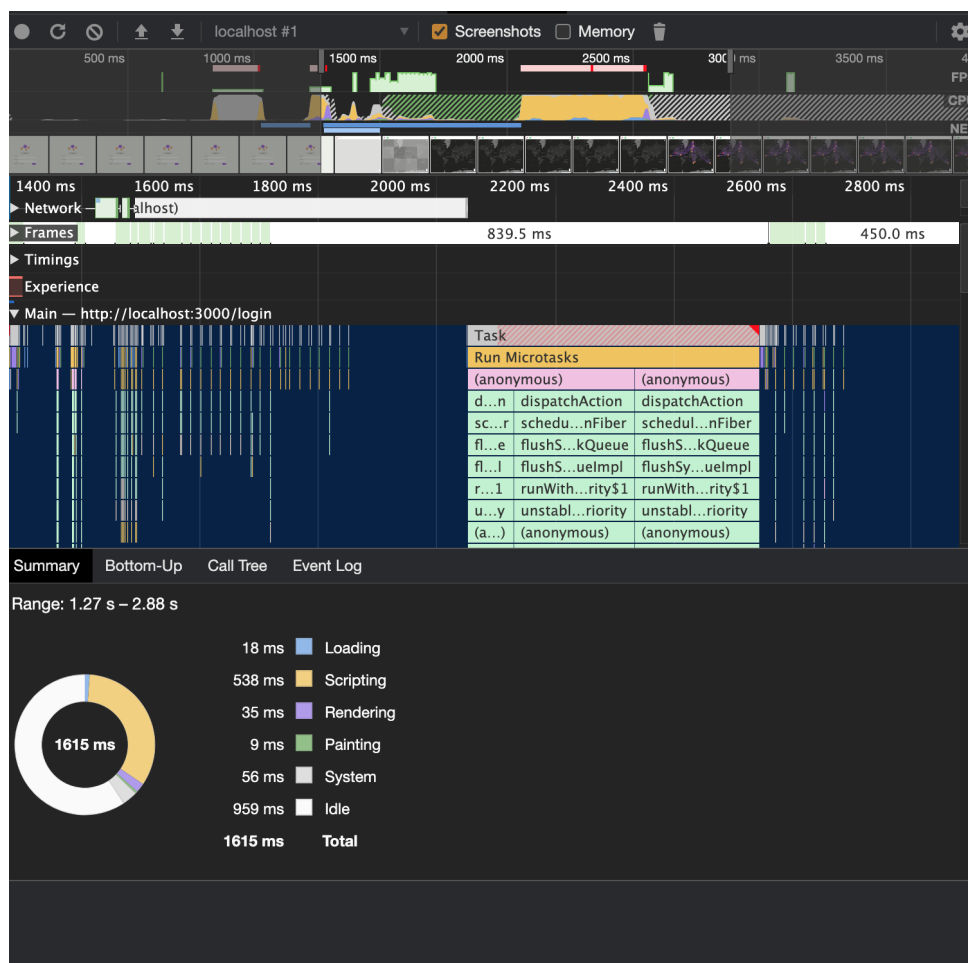


Figure C.10: Render performance

## C.9 Supportability

### To be tested

1. Complexity of the application.

### Test result

**Complexity of the application** Because our project has only 4200 lines of code and a test coverage of 72.5% we can assume that a developer still has the overview over the whole code base. This enables an easy and fast support.

## C.10 Portability

### To be tested

1. 12 Factor are applied.

### Test result

**12 Factor are applied** In section 4.2.3 we listed how we made our application cloud ready and therefore portable.

## C.11 Scalability

### To be tested

1. Time difference if more resources are in the topology.

### 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 edges. The second run has a policy applied, which will have 25 nodes and approximately 350 edges. 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.

<input type="checkbox"/> <a href="#">edit</a>	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
<input type="checkbox"/> <a href="#">edit</a>	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.11: Full match fetching speed

<input type="checkbox"/> <a href="#">edit</a>	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
<input type="checkbox"/> <a href="#">edit</a>	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.12: Applied policy fetching speed

## Mockup & Wireframe

### D.1 First Mockup

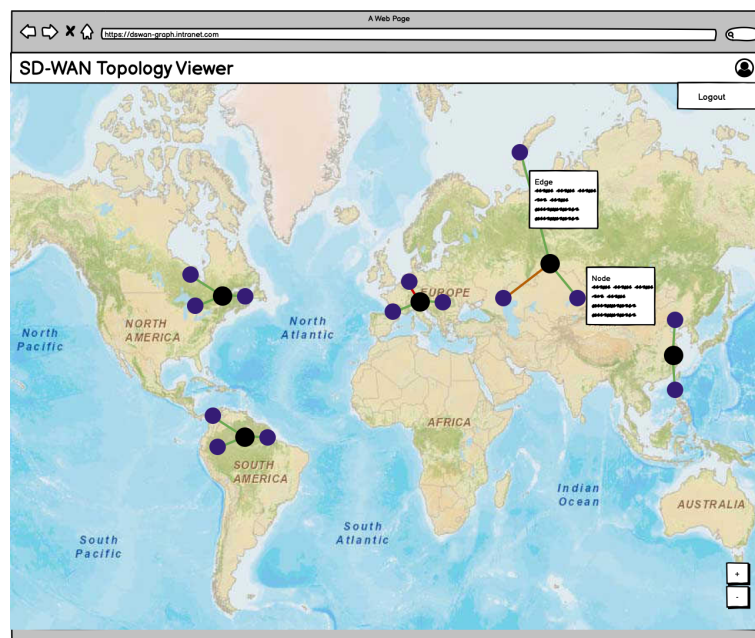


Figure D.1: Topology Viewer for User

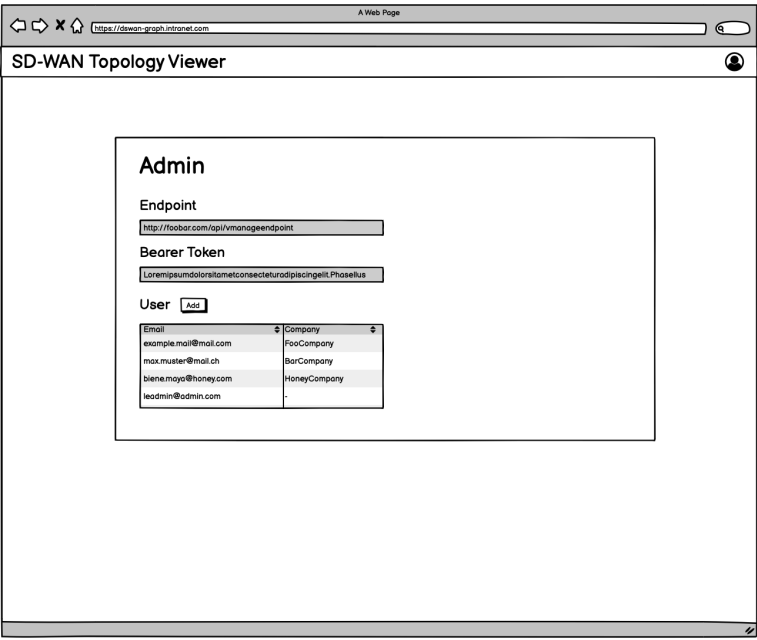


Figure D.2: Admin panel

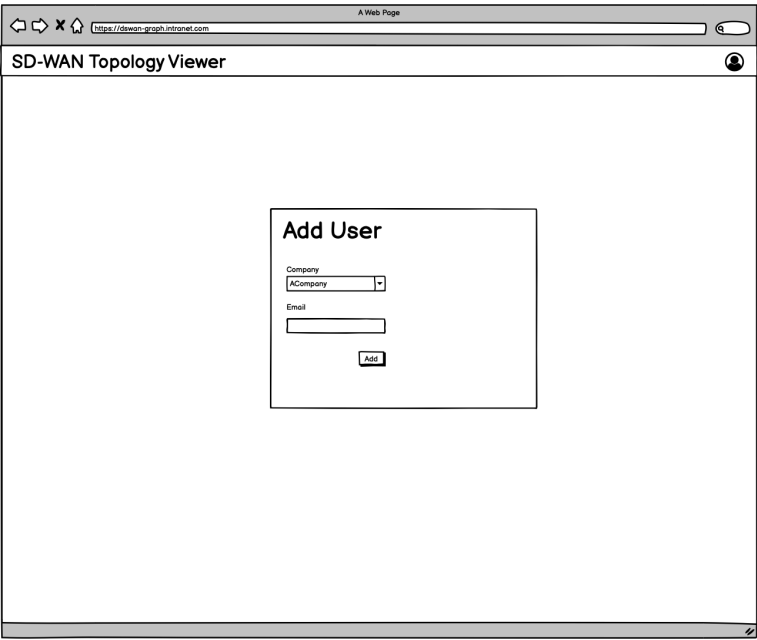


Figure D.3: Add new user

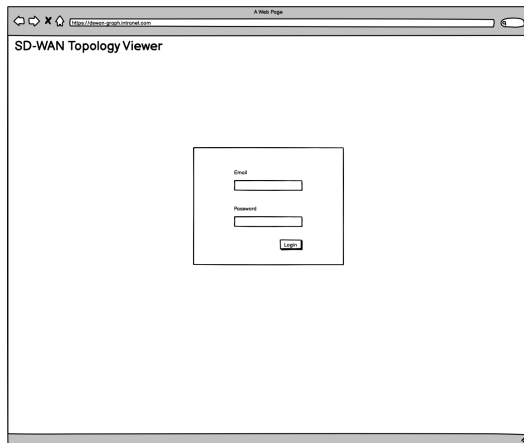


Figure D.4: Login page

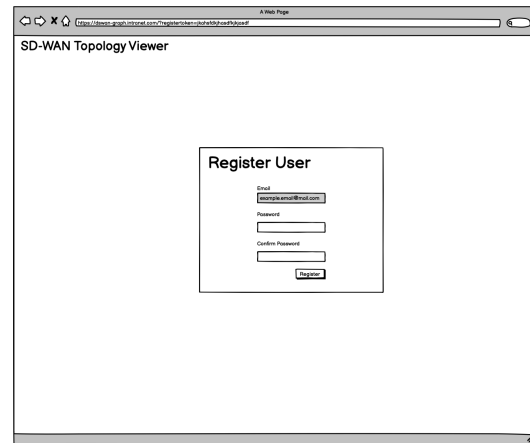


Figure D.5: Register page

## D.2 MVP Design Wireframe

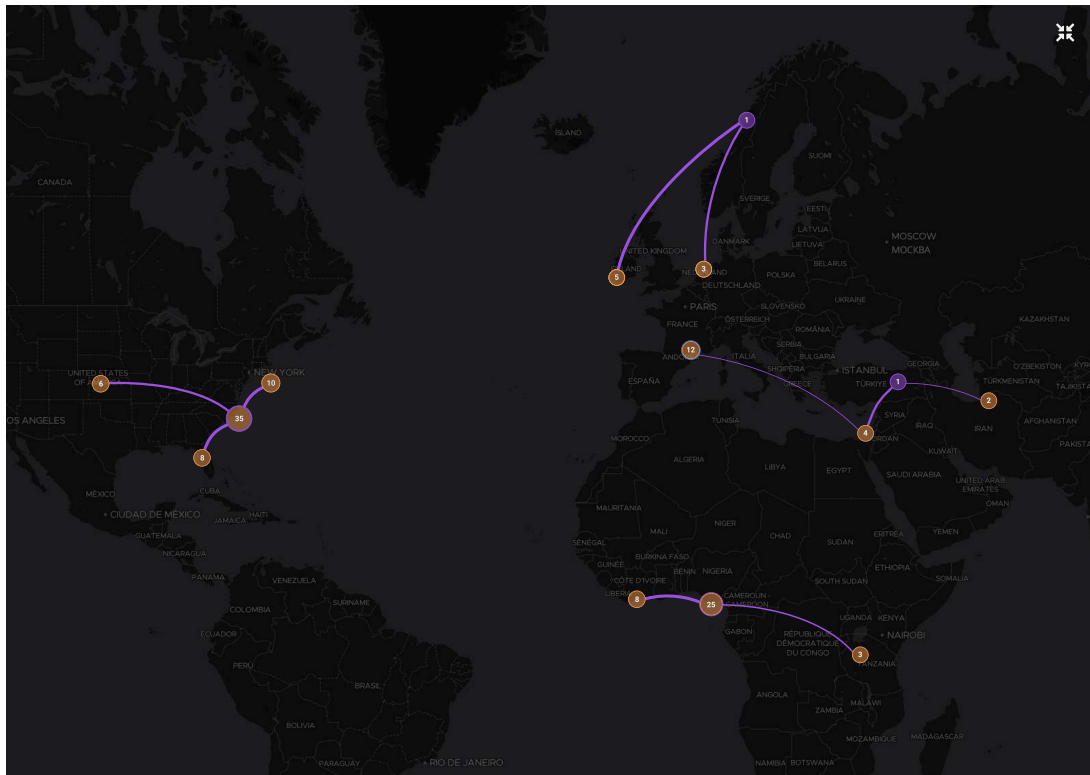


Figure D.6: Full screen view



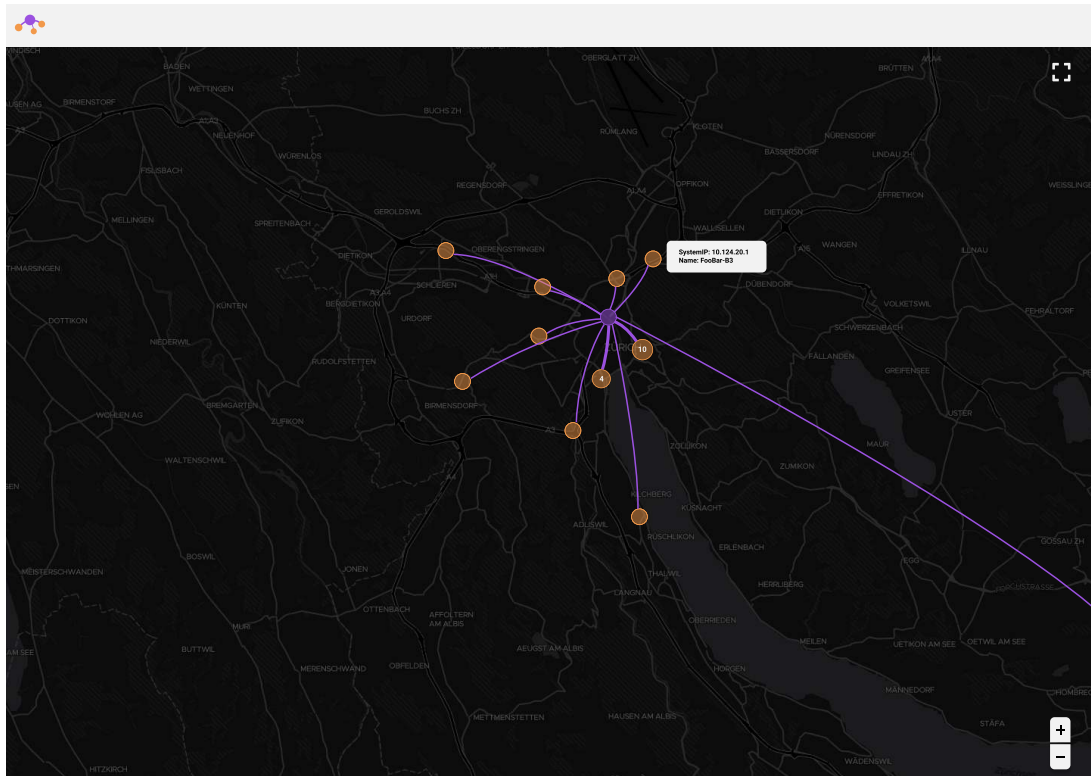


Figure D.7: Zoomed in view

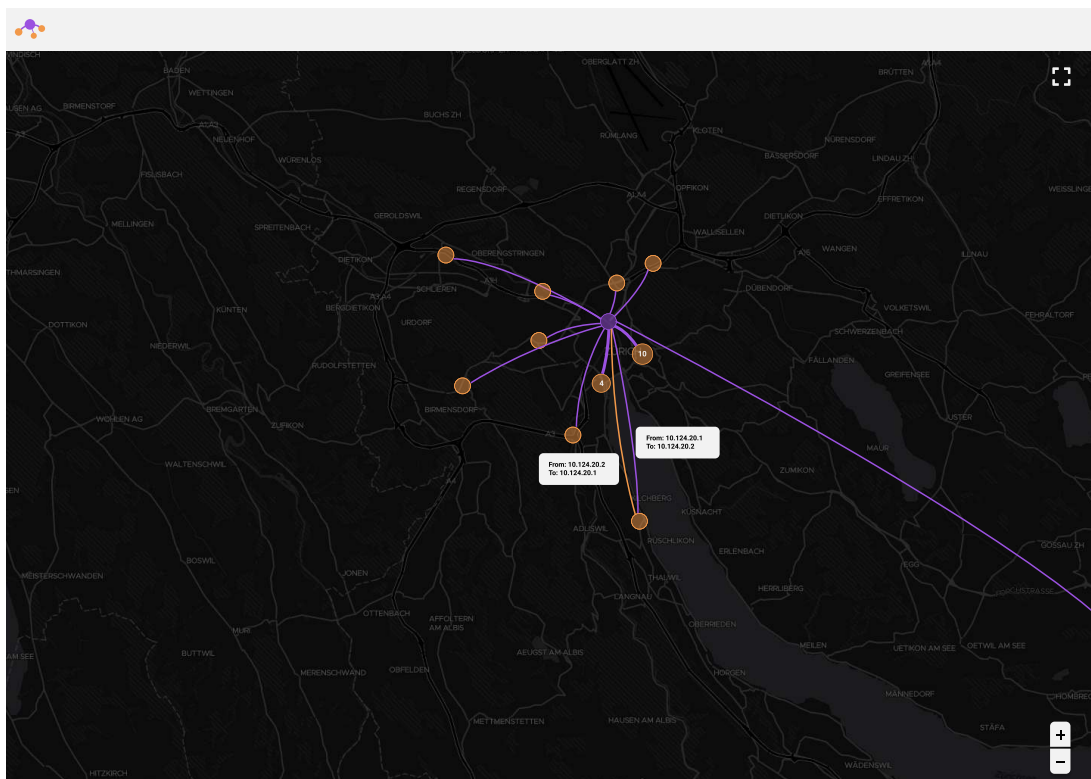


Figure D.8: Click on connection

## Appendix E

# Risk analysis

## E.1 Risk Analysis Table

#	Title	Description	Max damage [h]	Probability of occurrence	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 is not testable or the effort to test it is to high	8h	50%	4	We try to isolate the testable component from the non testable	We test the components over end to end tests.	TRUE	Because some components from frontend applications are not testable, we were able to test them over a manual end-to-end test
R4	Customer requirements not met	We deliver a different product than was expected	32h	10%	3.2	We've chosen a user centred design approach. The customer is involved at every stage of the project and will give us regular feedback. We also created mockups and wireframes to visualise 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.	FALSE	
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 stuck to the old versions, where it made sense
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	

## E.1. RISK ANALYSIS TABLE

#	Title	Description	Max damage [h]	Probability of occurrence	Weighted damage (h * prob)	Prevention	Behaviour on entry	had entered	What we did on entry
R8	Repository not accessible	We are currently using the gitlab from HSR. Sooner or later a transfer to the OST will happen. If an error occurs during the transfer, we could lose our repository.	8h	15%	1.2	We keep our local master branch up to date.	We use our local Gitlab data to create a new repository somewhere else.	TRUE	The hsr gitlab repository is getting replaced the ost repo. Luckily there is a transition phase and we don't need to change the gitlab during the term thesis
R9	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	
R10	Software is too slow	Depending on the amount of data we receive from Vmanage we can't render the page in a reasonable time.	16h	20%	3.2	In the backend implementation we will stick to the best practices.	Check which data can be omitted on initial load. All the other data can be loaded lazily.	TRUE	We implemented lazy loading solution for detailed edge and node informations
R11	vManage is not scalable	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 had technical meetings with our supervisor, to discuss different alternative solutions. We ended with tracking the changes via the event distribution.
R12	Faulty State of Topology	Because we updating only those resources that changed, we could run into a fault state which never gets updated	12h	30%	3.6	Writing test for event celery tasks, to prevent faulty behaviour	We will change the behaviour of the update functionality	FALSE	
R13	Our time getting out of sync	We use timestamps in order to fetch the resources from the vManage API	12h	30%	3.6	Using the same time server in vManage API than in our system. Using the same timezone	Asking the supervisor to help us.	FALSE	

---

## 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 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.2: Devices list sample response

### F.3 Event aggregation query

```
{
  "query": {
    "condition": "AND",
    "rules": [
      {
        "value": [
          "2020-10-15T13:10:00 UTC",
          "2020-10-15T14:21:00 UTC"
        ],
        "field": "entry_time",
        "type": "date",
        "operator": "between"
      },
      {
        "value": [
          "BFD"
        ],
        "field": "component",
        "type": "string",
        "operator": "equal"
      }
    ]
  },
  "aggregation": {
    "field": [
      {
        "property": "system_ip",
        "order": "asc",
        "sequence": 1
      },
      {
        "property": "entry_time",
        "order": "asc",
        "sequence": 1
      },
      {
        "property": "eventname",
        "order": "asc",
        "sequence": 1
      },
      {
        "property": "component",
        "order": "asc",
        "sequence": 1
      },
      {
        "property": "details",
        "order": "asc",
        "sequence": 1
      }
    ]
  }
}
```

Listing F.3: Events aggregation query

## F.4 Metrics aggregation query

```
{
  "query":{
    "condition":"AND",
    "rules":[
      {
        "value":[
          "24"
        ],
        "field":"entry_time",
        "type":"date",
        "operator":"last_n_hours"
      },
      {
        "value":[
          "10.255.255.133"
        ],
        "field":"vdevice_name",
        "type":"string",
        "operator":"in"
      }
    ]
  },
  "aggregation":{
    "field":[
      {
        "property":"name",
        "sequence":1,
        "size":50
      },
      {
        "property":"proto",
        "sequence":2
      }
    ],
    "metrics":[
      {
        "property":"loss_percentage",
        "type":"avg"
      },
      {
        "property":"vqoe_score",
        "type":"avg"
      },
      {
        "property":"latency",
        "type":"avg"
      },
      {
        "property":"jitter",
        "type":"avg"
      },
      {
        "property":"rx_octets",
        "type":"sum"
      },
      {
        "property":"tx_octets",
        "type":"sum"
      }
    ]
  }
}
```

```
}
}
```

Listing F.4: Metrics aggregation query

## F.5 Metrics response

```
{
  "header":{
    ...
  },
  "data":[
    {
      "loss_percentage":0,
      "latency":0,
      "count":141,
      "tx_octets":0,
      "jitter":0,
      "rx_octets":3661,
      "proto":"IPSEC",
      "name":"10.255.255.133:mpls-10.255.255.255:mpls",
      "fecLossRecovery":"-",
      "vqoe_score":10
    },
  ]
}
```

Listing F.5: Metrics sample response