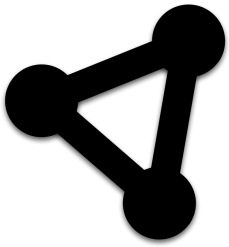
Bachelor's Thesis

Model-Based Generation of Service Provider Network Topologies

Spring Term 2023



Authors:	Lukas Ribi
	Pascal Christen
Advisor:	Prof. Laurent Metzger
Co-Advisor:	Severin Dellsperger
External Co-Examiner:	Marcel Witmer
Internal Co-Examiner:	Prof. Dr. Olaf Zimmermann
Project Partner:	Cisco Systems - François Clad

OST
Eastern Switzerland
University of Applied Sciences

CISCO

Department of Computer Science
OST - Eastern Switzerland University of Applied Sciences
Campus Rapperswil-Jona

Abstract

Objective Researchers and network engineers working for service providers need to validate network technologies to see how they scale on networks of varying sizes. Currently, these test networks are often created based on random graphs. Due to their randomness, they often lack characteristics found in real service provider networks. The goal was to develop an approach to generate network topologies with specific characteristics of these service provider networks. Furthermore, the existing Graph Analyzer system created during the previous term project had to be expanded to include edge weights in graph property calculations. In addition, a new graph property indicating robustness needed to be implemented.

Approach The MOST-Model (**M**esh-**O**riented **S**ervice Provider **T**opology) has been developed to facilitate the generation of network topologies that structurally resemble service provider networks. The model consists of eight steps and is parameterised to provide flexibility in the generation process. The model is inspired by existing methodologies and uses approaches that lead to characteristics matching service provider networks. A Gabriel graph is used to create the topology with additional optimisations to ensure redundancy. In other published papers, it has been documented that this type of graph reflects backbone networks well. The newly created Graph Generator application uses the MOST-Model to generate topologies. An API provides the ability to interact with it. The Frontend of the Graph Analyzer system has been extended to integrate the Graph Generator. It allows the seamless generation and visualisation of topologies using adjustable model parameters. The generated result can subsequently be imported into the Graph Analyzer system. Graph property calculations have been extended by incorporating edge weights from imported topology data. An approach using a targeted attack on high-degree nodes was chosen to calculate the robustness of a graph.

Conclusion During the bachelor's thesis, the MOST-Model was developed to generate network topologies with service provider network characteristics. Due to the underlying use of the Gabriel graph, the number of edges is minimised while still providing redundancy through alternative paths. This property reflects service provider network requirements from a cost and availability perspective. Known issues of the Gabriel graph, such as stub links, are mitigated through applied optimisations. The model has been integrated into the Graph Analyzer system as a separate application. This separation allows the straightforward integration of the MOST-Model via the Graph Generator application into other systems and contexts. In addition, the results can be imported into other applications by providing the generated topologies in the commonly used GEXF and GraphML file formats. The resulting Graph Analyzer system provides an easy way to examine how networks are structured and can generate new topologies through the Graph Generator application. The source code is publicly available and published under the permissive MIT license.

Management Summary

Initial Situation

In network research and engineering, an important part of validating network technologies is the evaluation of them on networks of various sizes. However, the often-used method of generating random graphs for such assessments fails to capture the characteristics of service provider networks. This thesis aimed at developing an approach to generate network topologies that resemble service provider networks. In addition, the existing Graph Analyzer system, created during the previous term project, had to be extended with new features.

Procedure and Technology

In the initial inception and elaboration phases, the setup for the following phases was done. Requirements were discussed with the advisors and evaluated. Based on them, research was done into the subject area of topology generation. Potential solutions were implemented as proof of concepts and presented to the advisors. Based on their feedback it was decided what approach should be taken.

The following construction and transition phases contained the implementation of a separate Python application to handle topology generation. In addition, the existing Go and TypeScript-based applications of the Graph Analyzer system were extended to support edge weights and a robustness metric.

Results

The developed MOST-Model (**M**esh-**O**riented **S**ervice Provider **T**opology) defines a process to generate network topologies with service provider network characteristics. The model minimises the number of edges and incorporates optimisations resulting in redundancy through alternative paths. This reflects the cost and availability requirements of service provider networks. Figure 1 shows a simplified visualisation of the MOST-Model.

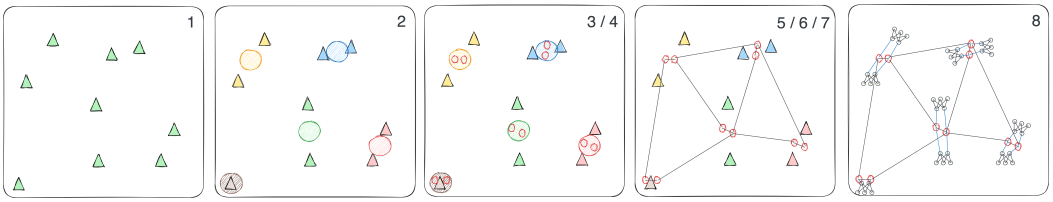
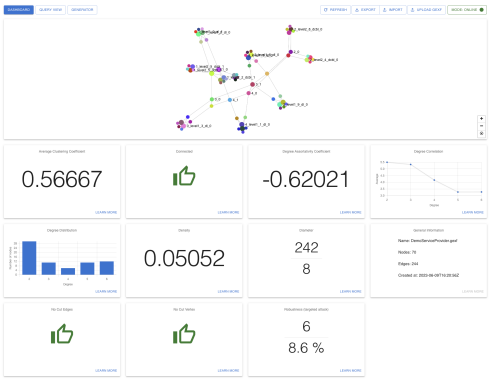
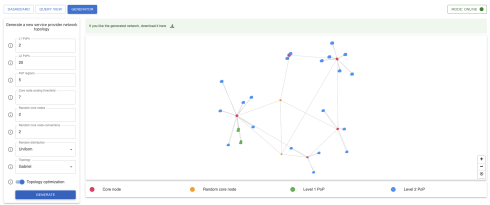


Figure 1: Simplified MOST-Model visualisation

A new application called Graph Generator was created to implement the MOST-Model. This application provides an API for interaction and seamless generation of topologies using adjustable model parameters. The Frontend of the Graph Analyzer system was extended to integrate the Graph Generator as seen in figure 2.



Dashboard



Generator

Figure 2: Integration into the Graph Analyzer Frontend

This enables the visualisation and export of generated topologies. Topologies can subsequently be imported into the Graph Analyzer system for analysis. By providing the generated topologies in the commonly used GEXF or GraphML file formats, they can be used in other contexts too.
By including edge weights in graph property calculations and the newly added robustness property, the accuracy and depth of the analysis are enhanced.

Overall, the thesis has achieved its objectives by developing a network topology gen-

eration approach and expanding the capabilities of the Graph Analyzer system. The MOST-Model and Graph Analyzer system provides researchers and network engineers with tools to gather insights into network structures. This supports the efficient validation and assessment of network technologies.

Outlook

The developed MOST-Model and the integration of the Graph Generator into the Graph Analyzer system provide a solid foundation for future enhancements. In the next step, the model should be refined based on real service provider networks.

Acknowledgments

Prof. Laurent Metzger

We want to thank Prof. Laurent Metzger for being our advisor. His guidance and expertise in IT networks were especially appreciated, and his initial ideas and support played an essential role during the research process.

Severin Dellsperger

We would like to thank our co-advisor Severin Dellsberger for his support, guidance and feedback during our bachelor's thesis. His assistance regarding organisational matters was especially appreciated.

Prof. Dr. Olaf Zimmermann

We want to thank Prof. Dr. Olaf Zimmermann for his helpful and extensive feedback during the interim presentation.

Prof. Oliver Augenstein

We would like to thank Prof. Oliver Augenstein for taking the time to discuss our chosen approach regarding PoP placement and clustering. The insights and feedback gathered from the discussion were extremely helpful.

Contents

Glossary and Abbreviations

Backbone Also called a core network is part of a service provider network, which consists of core routers, where no customers are directly connected, and very high bandwidths are available.

CI/CD Continuous Integration/Continuous Deployment: It is an approach that is used in software development to automate testing, deployment and be able to carry out fast and reliable deployments.

Cut Vertex Also known as an articulation point is a node in a graph. Removing it disconnects the graph.

Degree The number of edges connected to a node.

GDS Graph Data Science Library - A Neo4j graph database plugin that implements common graph algorithms.

Gephi A popular application for the visualisation and exploration of graphs.

GEXF Graph Exchange XML Format - A graph file format based on XML that is used to store graph structures.

GraphML A graph file format based on XML that is used to store graph structures.

gRPC Google Remote Procedure Calls - A remote procedure call framework.

Helm A package manager for Kubernetes that simplifies deployments.

iBGP Interior Border Gateway Protocol - Usage of the BGP protocol within an autonomous system.

IGP Interior Gateway Protocol - Protocol used to exchange routing table information.

INS Institute for Network and Security - An institute at the Eastern Switzerland University of Applied Sciences.

Jalapeño An open-source infrastructure platform for network services developed by Cisco.

Kubernetes An open-source container orchestration system that automates the deployment, scaling, and management of containerized applications.

Minimum spanning tree A subset of a graph that contains all nodes without any cycles with the minimal amount of edge weights.

Neo4j A popular graph database.

NP-hard Describes computer science problems that have no solution that can be found in polynominal-time.

OpenAPI OpenAPI (formerly known as Swagger) is a specification for building APIs that allows developers to describe the structure and behaviour of APIs in a standardized way, enabling automated documentation and code generation.

PoP Point of Presence is a network location where service providers have equipment and provide connectivity to customers.

Pydantic Python library that provides data validation functionality.

RTK Query A JavaScript tool that facilitates the fetching and caching of data in a web application. It is designed to simplify the process of loading data, reducing the need for manual coding of data fetching and caching logic.

TSP Travelling salesman problem - A popular problem in computer science.

Bibliography

[1] F. Menczer, S. Fortunato, and C. A. Davis, *A First Course in Network Science*, 1st edition. Cambridge University Press, 2020-01-30, 274 pp.

[2] V. Balakrishnan, *Schaum's Outline of Graph Theory: Including Hundreds of Solved Problems*, 1st edition. New York: McGraw Hill, 1997-02-22, 288 pp., ISBN: 978-0-07-005489-9.

[3] B. Quoitin, V. Van den Schrieck, P. Francois, and O. Bonaventure, "IGen: Generation of router-level Internet topologies through network design heuristics," in *2009 21st International Teletraffic Congress*, 2009-09, pp. 1–8.

[4] "Backbone - Infrastructure - SWITCHlan - SWITCH." (), [Online]. Available: `https://www.switch.ch/network/infrastructure/backbone/` (visited on 2023-06-14).

[5] S. Freitas, D. Yang, S. Kumar, H. Tong, and D. H. Chau, "Graph Vulnerability and Robustness: A Survey," *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–1, 2022, ISSN: 1041-4347, 1558-2191, 2326-3865. DOI: `10.1109/TKDE.2022.3163672`. arXiv: `2105.00419 [cs]`. [Online]. Available: `http://arxiv.org/abs/2105.00419` (visited on 2023-05-10).

[6] P. Christen and L. Ribi, "Graph properties of a telecommunication network," other, OST Ostschweizer Fachhochschule, 2023. [Online]. Available: `https://eprints.ost.ch/id/eprint/1081/` (visited on 2023-05-24).

[7] M. E. J. Newman, "The structure and function of complex networks," *SIAM Review*, vol. 45, no. 2, pp. 167–256, 2003-01, ISSN: 0036-1445, 1095-7200. DOI: `10.1137/S003614450342480`. arXiv: `cond-mat/0303516`. [Online]. Available: `http://arxiv.org/abs/cond-mat/0303516` (visited on 2023-03-02).

[8] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, vol. 393, no. 6684, pp. 440–442, 6684 1998-06, ISSN: 1476-4687. DOI: `10.1038/30918`. [Online]. Available: `https://www.nature.com/articles/30918` (visited on 2023-06-12).

[9] A.-L. Barabási and M. Pósfai, *Network Science*, 1st edition. Cambridge, United Kingdom: Cambridge University Press, 2016-08-05, 475 pp., ISBN: 978-1-107-07626-6.

[10] A. Medina, A. Lakhina, I. Matta, and J. Byers, "BRITE: An approach to universal topology generation," in *MASCOTS 2001, Proceedings Ninth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2001-08, pp. 346–353. DOI: 10.1109/MASCOT.2001.948886.

[11] M. Naldi, "Connectivity of Waxman topology models," *Computer Communications*, vol. 29, no. 1, pp. 24–31, 2005-12-01, ISSN: 0140-3664. DOI: 10.1016/j.comcom.2005.01.017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0140366405000630 (visited on 2023-05-15).

[12] E. K. Çetinkaya, M. J. F. Alenazi, Y. Cheng, A. M. Peck, and J. P. G. Sterbenz, "A comparative analysis of geometric graph models for modelling backbone networks," *Optical Switching and Networking*, Special Issue on RNDM 2013, vol. 14, pp. 95–106, 2014-08-01, ISSN: 1573-4277. DOI: 10.1016/j.osn.2014.05.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1573427714000356 (visited on 2023-03-10).

[13] H. Haddadi, M. Rio, G. Iannaccone, A. Moore, and R. Mortier, "Network topologies: Inference, modeling, and generation," *IEEE Communications Surveys & Tutorials*, vol. 10, no. 2, pp. 48–69, 2008, ISSN: 1553-877X. DOI: 10.1109/COMST.2008.4564479.

[14] F. M. Dekking, C. Kraaikamp, H. P. Lopuhaä, and L. E. Meester, *A Modern Introduction to Probability and Statistics: Understanding Why and How*, First Edition. London: Springer, 2005-06-15, 504 pp., ISBN: 978-1-85233-896-1.

[15] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, ser. KDD'96, Portland, Oregon: AAAI Press, 1996-08-02, pp. 226–231.

[16] L. Kaufman and P. J. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley & Sons, 2009-09-25, 369 pp., ISBN: 978-0-470-31748-8. Google Books: YeFQHiikNo0C.

[17] M. Krechetov. "Generating Gabriel graphs.," Mikhail Krechetov. (2022-07-14), [Online]. Available: https://mkrechetov.github.io/gabriel_graphs (visited on 2023-06-06).

[18] F. Goulart, *Python TSP Solver*, 2023-06-04. [Online]. Available: https://github.com/fillipe-gsm/python-tsp (visited on 2023-06-05).

[19] V. Gill and J. Mitchell, "AOL Backbone OSPF-ISIS Migration," *NANOG29*, 2003-10. [Online]. Available: https://archive.nanog.org/meetings/nanog29/presentations/gill.pdf.

[20] "GEXF File Format." (), [Online]. Available: https://gexf.net/ (visited on 2023-06-10).

[21] "Introduction — Jalapeño API Gateway." (), [Online]. Available: https://jalapeno-api-gateway.github.io/jagw/docs/introduction (visited on 2023-06-10).

[22] “swissBOUNDARIES3D,” Bundesamt für Landestopografie swisstopo. (), [Online]. Available: https://www.swisstopo.admin.ch/de/geodata/landscape/boundaries3d.html (visited on 2023-05-31).

List of Figures

List of Tables

Listings

Part I

Technical Report

Chapter 1

Introduction

This chapter gives an introduction to the thesis. The thesis assumes that the reader has existing knowledge of essential computer science topics such as computer networks, algorithms, application architecture and distributed systems.

1.1 Thesis Structure

The following sections describe how the thesis is structured.

1.1.1 Technical Report

Three chapters make up the Technical Report. The Introduction provides a foundation regarding the topic of this thesis and its goals. The Results chapter documents the research and implementation of this thesis. Finally, the Conclusion chapter reflects on what has been achieved. It also discusses known limitations and an outlook with possible future steps.

1.1.2 Project Documentation

The Project Documentation consists of four chapters. The requirements that the final result of the thesis needs to fulfil are documented in the Requirements chapter. Decisions that shaped the implementation of the result are detailed in the Design Decisions chapter. The Architecture chapter deals with the implementation structure and how quality is maintained. The final Project Management chapter contains details regarding the project management processes used while working on this thesis.

1.2 Terms and Techniques

In the following sections, several concepts are introduced in order to provide context for this thesis.

1.2.1 Graphs

The concept of graphs is introduced in a short form below and is based on [1], [2]. The intention is not to give a complete definition but a rough overview.
A graph can be defined as a set of elements that are connected in pairs. The elements are called nodes, while their connections are described as edges. The following three definitions define essential properties of a graph:

- Edges can be directed or undirected. Contrary to undirected edges, a graph with directed edges can have two edges between a pair of nodes.
- These edges can have properties assigned to them, for example, numbers that define a weight. However, what the weight represents depends on the context.
- A graph is a multigraph if multiple edges connect a pair of nodes.

Graphs and graph theory have applications in many areas, including computer networks. When talking about nodes in this thesis, they generally represent routers.

Figure 1.1 illustrates two types of graphs. An unweighted undirected simple graph and a weighted directed multigraph.

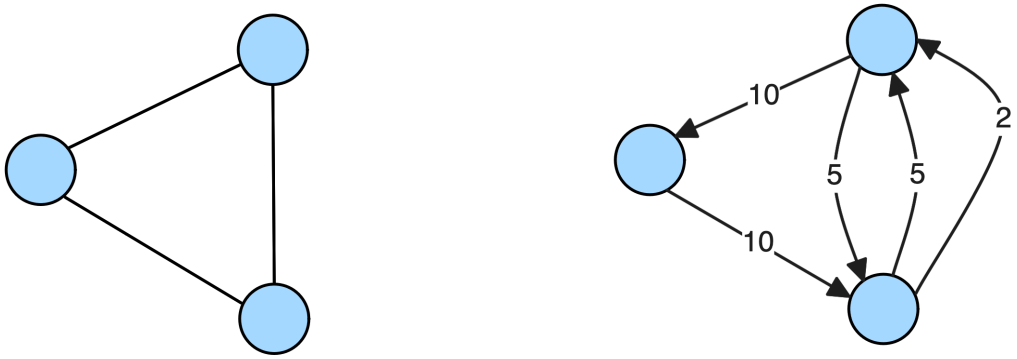


Figure 1.1: Graph examples

1.2.2 Service Provider Networks

Service providers offer internet or telecommunication services to customers. This requires a robust, economic and scalable network. The network can be divided into two parts.
The backbone, also known as the core network, is designed for large amounts of traffic with high bandwidths. Since a failure of the backbone affects the entire service provider network, it is usually designed with a high level of redundancy by using a mesh-like topology.
The access network is designed to connect customers to the backbone. The bandwidths are generally lower compared to the backbone. Customers connect via a point of presence, or PoP for short [3].

The SWITCHlan backbone that connects Swiss universities can be seen in figure 1.2.

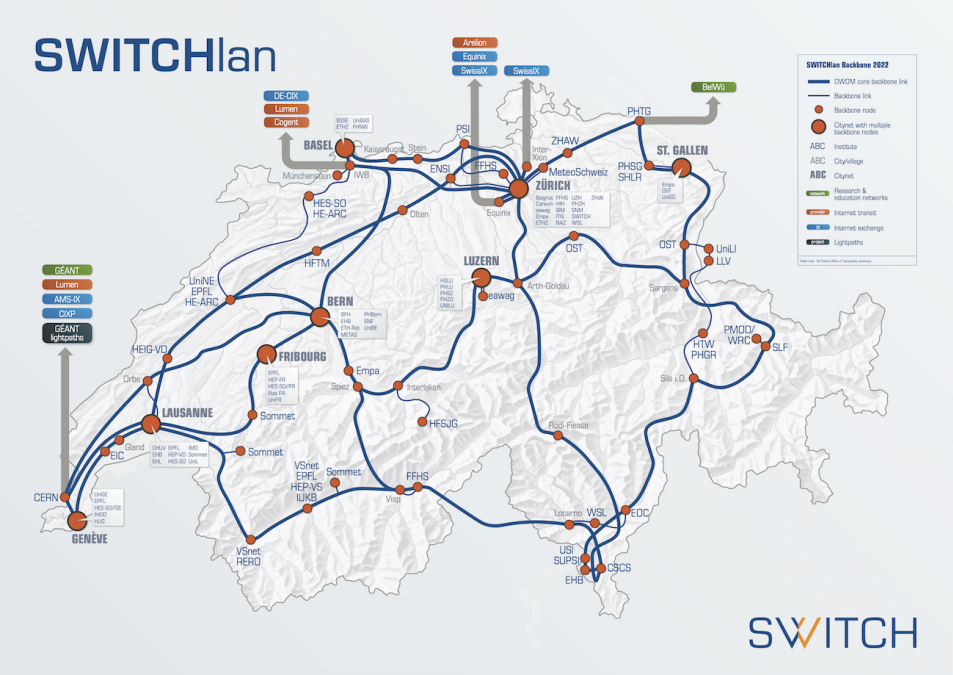


Figure 1.2: SWITCHlan backbone, Source: [4]

1.2.3 Network Robustness

Robustness is a term that describes how resistant a network is to disruptive factors. These can be natural failures of individual devices but also targeted attacks [5]. Often a balance must be found between how robust a network should be and the financial cost that it entails. Rating robustness is context-dependent. For example, a critical healthcare provider network has different requirements than a private home network.

1.3 Aims and Objectives

This section introduces the problems and related goals of this thesis.

1.3.1 Service Provider Network Topology Generation

Network engineers and researchers develop new network technologies. These developed methods must be tested in order to validate their functionality. While network engineers of service providers can use a network topology resembling a real network for simulation purposes, researchers often have other requirements. They need to test technologies on a multitude of different network topologies as they are targeted at more general usage. One such use case can be found in the area of software-defined networks. Software-defined networking applications need to be tested on large networks in order to see how they scale. In addition, service providers rarely disclose their network topology in detail to be used for research purposes due to security concerns.
Currently, these topologies are often randomly generated and do not have properties specific to service provider networks. The goal is to develop a process to create network topologies that incorporate characteristics found in service provider networks such as PoPs and backbone structure.

1.3.2 Integration into Graph Analyzer System

The existing Graph Analyzer system created in the term project [6] needs to be extended to provide the functionality to generate network topologies. It should be based on the solution developed for solving the problem described in section 1.3.1. It should be possible to export generated topologies in the form of a commonly used graph file format. Importing this file into the Graph Analyzer system should be possible in order to visualise and analyse it.

1.3.3 Weighted Support in Graph Analyzer System

The Graph Analyzer system currently does not consider the weight of edges when importing topology data. Thus, all calculated graph properties are based on an undirected unweighted simple graph. The topology data import needs to be adjusted in order to extract the edge weights. In addition, it should be evaluated which calculated graph properties are affected by this change.

1.3.4 Robustness Property

In addition to the calculated graph properties in the term project [6], a new property should be added. This property should quantify the robustness of a graph based on a suitable metric.

Chapter 2

Results

This chapter gives an overview of what has been achieved in this thesis and how it has been implemented.

2.1 Distinction

This section clarifies the scope of the work done in the previous term project. This is done to better distinguish the extent of what has been achieved in this thesis.

During the term project [6] in the fall semester of 2022, a system composed of multiple applications was created that analyses graphs and displays selected properties. It is composed of the Data Collector application that is responsible for importing topology data into a graph database. Calculation of graph properties and providing them through an interface is done by the API application. The graph properties are displayed in the Frontend application.

This thesis builds upon this existing system by adding a new application responsible for topology generation, that is integrated into the Frontend. In addition, adjustments to include edge weights in graph property calculations have been made. Further, a new graph property to indicate robustness was introduced. Changes related to this affect all parts of the existing Graph Analyzer system.

2.1.1 System Structure

Following, a brief overview of the Graph Analyzer system is given. A visual overview is shown in figure 2.1.

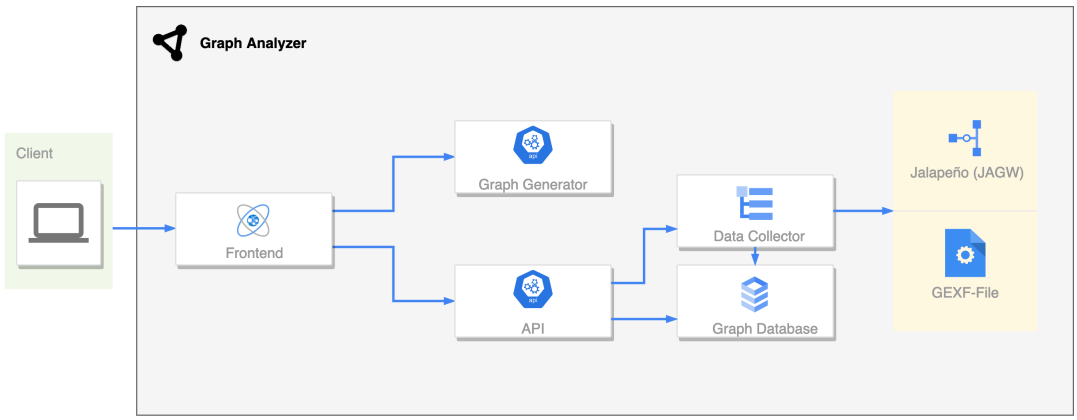


Figure 2.1: System overview

Frontend

The Frontend displays and explains graph properties. It communicates with the API in order to display the values of the properties. As part of this thesis, it has been extended to integrate the newly added Graph Generator.

API

The API analyses the graph persisted in the Graph Database and calculates graph properties based on it. The functionality to forward graph files to the Data Collector was added as part of this thesis.

Data Collector

The Data Collector is an independent application solely responsible for retrieving topology data from different data sources. Logic related to the import from various data sources is contained within it. The graph data is transformed into a defined structure and saved in the Graph Database.

Graph Database

The Graph Database contains the graph data that the Data Collector has collected. The API can retrieve the graph in order to calculate graph properties.

Graph Generator

The Graph Generator is a new application implemented as part of this thesis. It implements the generation of service provider network topologies. Converting them into graph file formats and delivering the results to the Frontend is also handled.

2.2 Existing Software and Limitations

A short investigation into existing software that provides service provider network topology generation did not lead to any results. Consulted network engineers at the INS also did not know of an existing product, their current approach is to generate the network topologies randomly. It can be also argued that the specific type of network topology that should be generated is too specific to have an existing software implementation.
In academic research, several approaches exist that try to solve a similar problem. More about these existing approaches can be found in section 2.4.
Based on conversations with network engineers, it is assumed that in an enterprise environment, test networks are typically created based on existing production networks.

2.3 Achievements

This section will provide an overview of the achieved results. Detailed information about their implementation can be found in section 2.5.

2.3.1 MOST-Model

To generate service provider network topologies, the MOST-Model was created. The name stands for **M**esh-**O**riented **S**ervice Provider **T**opology Model or can also be understood as a play on words with the Eastern Switzerland University of Applied Sciences, **OST** for short. Details about the model can be found in section 2.5.1.

Figure 2.2 illustrates the process of the model in a simplified form.

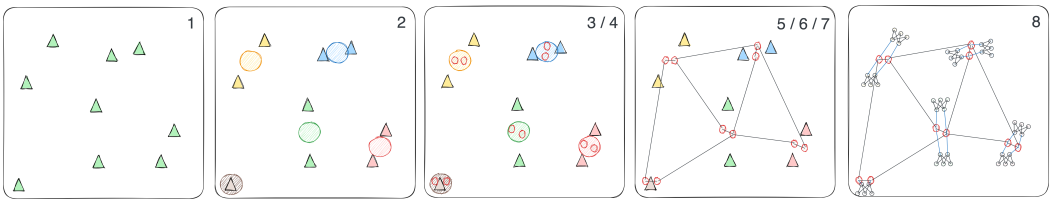


Figure 2.2: Simplified MOST-Model visualisation

2.3.2 Graph Generator

An application called Graph Generator has been developed to handle the topology generation. It implements the MOST-Model mentioned in section 2.3.1 and expands on it by parameterising it. In addition, a ring topology option is available as an option.
The decision to implement it separately from the existing applications as another microservice was made intentionally. The goal was to have the possibility to use the Graph Generator in other contexts and not only in conjunction with the Graph Analyzer system.
When generating a topology, it is returned in a simple proprietary JSON-based format. To provide easy interoperability, data in the proprietary format can be submitted in order to convert it to the broadly used GEXF or GraphML file formats.
The Graph Generator has been implemented as an HTTP API and uses the OpenAPI 3 specification to define interactions with it. The available endpoints can be seen in figure 2.3.

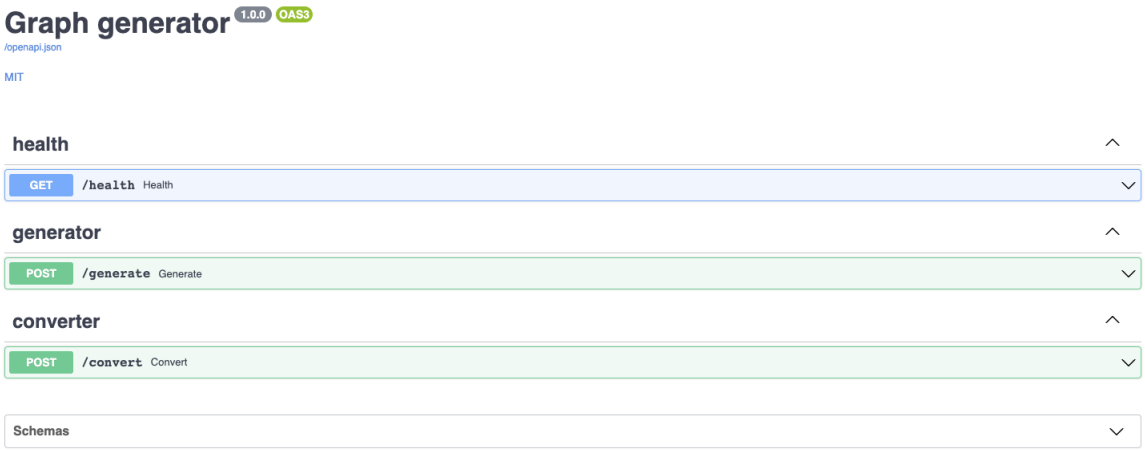


Figure 2.3: OpenAPI Specification - Graph Generator

2.3.3 Graph Generator Integration

The Graph Generator application was successfully integrated into the existing Frontend. The user has the option of adjusting parameters and generating topologies via a form that can be seen in figure 2.4. A preview of the generated topology is shown with the option to download it in the GEXF file format.

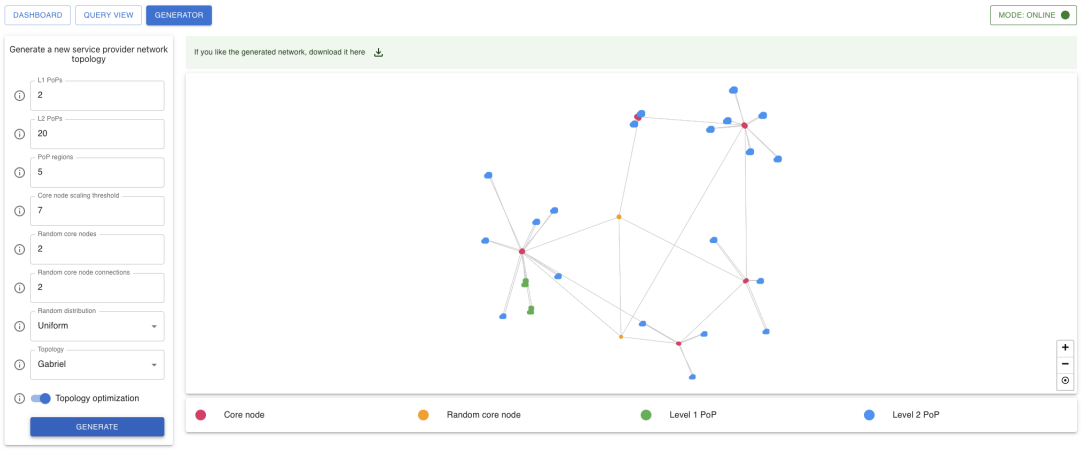
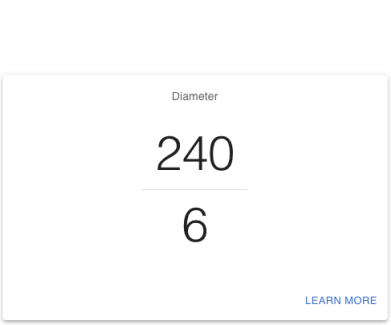


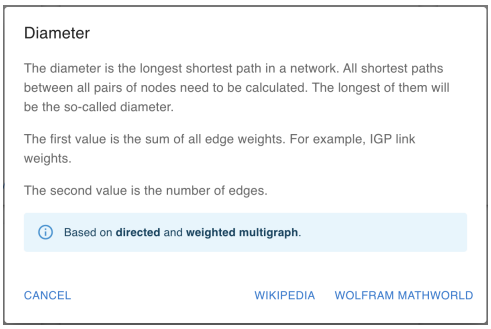
Figure 2.4: Integration into Graph Analyzer

2.3.4 Weight Support

The Graph Analyzer system has been adjusted to support edge weights. For both currently supported topology data providers, Jalapeño and GEXF, the processing was updated in order to persist the weight in the Graph Database.
Figure 2.5 shows the diameter property that has changed due to the inclusion of edge weights. Other properties are not affected by it.



Values



Description

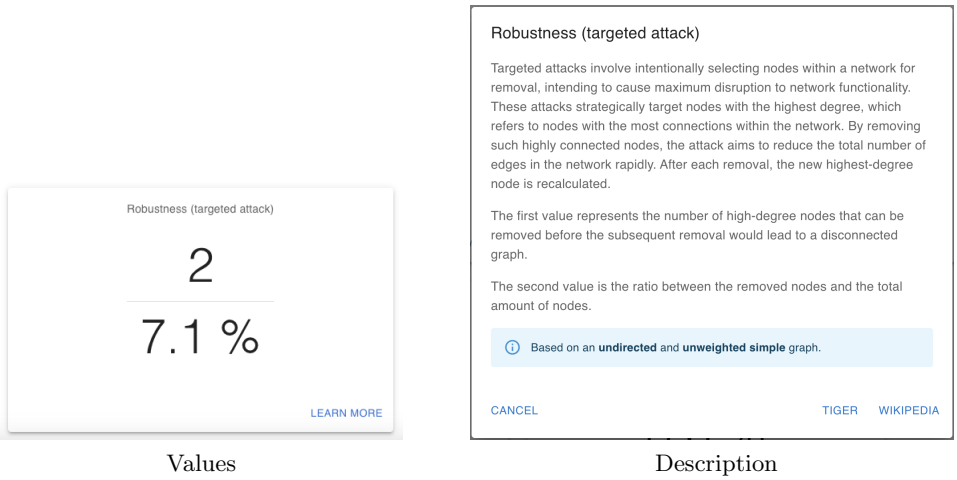
Figure 2.5: Diameter property

2.3.5 Data Import Improvements

The option to use GEXF files as the data source for the Graph Analyzer system has been reworked in order to be more flexible. Prior, the Data Collector only supported the Jalapeño API Gateway and locally available GEXF files as data sources.
The implementation of a gRPC API for the GEXF data source option adds support for importing custom GEXF files while the system is running.
To use this functionality, the Frontend and API have been adjusted to allow GEXF file uploads which then get forwarded to the Data Collector through the API. In conjunction with the integration of the Graph Generator into the Frontend mentioned in section 2.3.3, a generated network can be subsequently imported into the Graph Analyzer system.

2.3.6 Robustness Calculation

The API and Frontend of the Graph Analyzer system have been extended with a new property as seen in figure 2.6. The property quantifies the robustness of a network. How robustness is quantified is explained in section 2.5.6.



Values Description

Figure 2.6: Robustness property

2.3.7 Graph Analyzer System

All components of the Graph Analyzer System are publicly available on GitHub under an MIT license. The repositories are grouped in an organisation and are accessible here: `https://github.com/Graph-Analyzer`

2.4 Research

This section documents the research and results that led to the creation of the MOST-Model.

2.4.1 Random Graph Models

To generate random graphs, a multitude of different approaches exist. Three different models are introduced briefly in the following sections.

2.4.1.1 Erdős-Rényi Model

The Erdős-Rényi model is characterised as a random graph, in which edges are generated stochastically without adhering to any visible pattern.

Two closely related variants of the Erdős-Rényi random graph model exist [7].

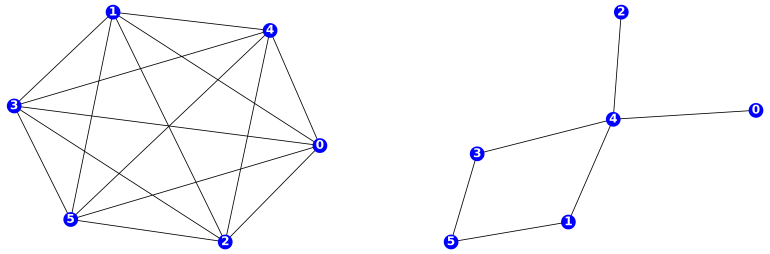
$$G(n, M)$$

n is the number of nodes, and M is the number of edges in a graph. $G(4, 2)$ for example means that there are in total 6 possible graphs where the order of edges does not matter. 6 is the result of the following calculation $\binom{4}{2} = 6$.

$$G(n, p)$$

n is the number of nodes, and p is the probability of an edge creation. p is in the interval $[0, 1]$. Such a graph will have an average of $\binom{n}{2}p$ edges.

Figure 2.7 shows two generated graphs using the $G(n, p)$ variant.



$G(6, 1)$ $G(6, 0.4)$

Figure 2.7: Graphs with the Erdős-Rényi model

The Erdős-Rényi model is not suitable for modelling service provider networks because it assumes that edges between nodes are created randomly. This does not reflect the complex structure of real-world networks. In these types of networks, the nodes are often

specialised and have different functions. Connections between them are not random but rather purposeful.

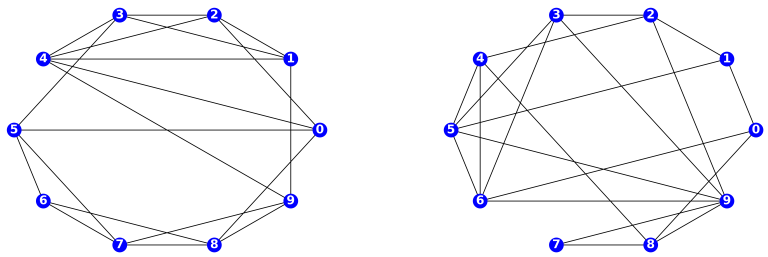
2.4.1.2 Watts-Strogatz Model

The Watts-Strogatz model is categorised as a random graph model. In comparison to the Erdős-Rényi model, it produces so-called small-world graphs with high clustering and short average path lengths [8].

$$G(n, k, p)$$

The graph to be constructed takes three parameters as arguments. The number of nodes n, mean degree k and the probability of rewiring each edge p. The creation process starts with a regular ring lattice of n nodes, where each node is connected to its k nearest neighbours. Next, each edge is rewired to a random node in the network with probability p. This rewiring process helps to capture the small-world phenomenon observed in real-world networks.

Two examples can be found in figure 2.8.



$G(10, 4, 0.2)$ $G(10, 4, 0.5)$

Figure 2.8: Graphs with the Watts-Strogatz model

The Watts-Strogatz model has some limitations regarding the representation of certain characteristics of service provider networks. For instance, it does not explicitly capture hierarchical structures and preferential attachment. In addition, the presence of hubs, which are common in many service provider networks, is neither captured [1].

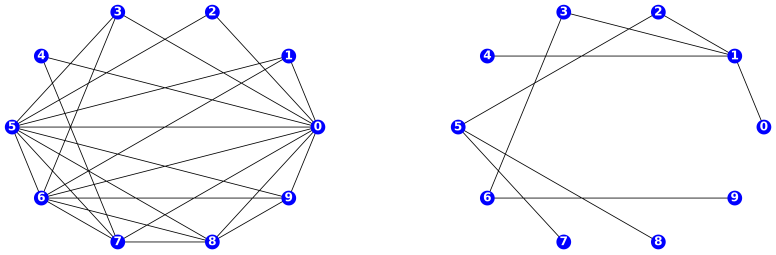
2.4.1.3 Barabási-Albert Model

The Barabási-Albert model is a network growth model that uses preferential attachment, where nodes with higher degrees have a greater probability of attracting new connections. It starts with a small number of nodes and adds new nodes iteratively, connecting each new node to existing nodes with a probability proportional to their degree. This process results in the emergence of scale-free networks with a power-law degree distribution, where a few nodes have a higher degree, so-called hubs [9].

$$G(n, m)$$

n is the number of nodes, and m denotes the number of edges to attach from a new node to existing nodes.

Figure 2.9 shows two examples.



$G(10, 4)$ $G(10, 1)$

Figure 2.9: Graphs with the Barabási-Albert model

Despite its prevalent use when generating graphs for computer networks or the internet, its use in an unmodified form is discouraged as intrinsics of a real network are not incorporated [9].

2.4.2 BRITE

An important work in the area of topology generation is BRITE [10]. BRITE defined an initial topology generation framework that was adapted in further works. It is a universal topology generation tool targeted at researchers in the network community doing research concerning internet topology. It incorporated various generation models such as the Waxman model explained in section 2.4.3. It also implemented the placement of the nodes as a separate step from connecting them, which is something that reappears in future topology generator implementations.

2.4.3 Waxman

A Waxman graph is a generalised random geometric graph. In contrast to random graphs, it not only relies on probabilities but incorporates the distances between nodes. In the first step, all nodes of the graph are uniformly distributed over a plane. Pairs of nodes are then connected based on a probability function that includes the distance between the two nodes [11].

An example of a Waxman graph can be found in figure 2.10.

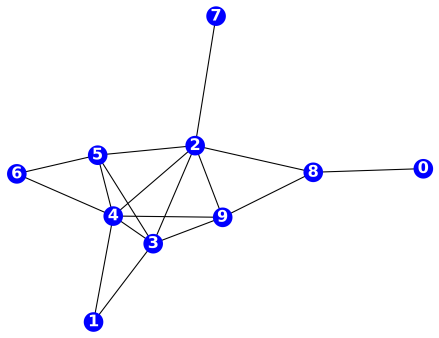


Figure 2.10: Graph with the Waxman model

It has often been used when generating topologies that should resemble telecommunication networks or to test routing algorithms. But more recent papers such as [12], [13] show that other methods lead to better results than a Waxman graph.

2.4.4 IGen

Another topology generator has been implemented in the form of IGen [3]. It specialises in generating topologies resembling service provider networks. It defines a network design methodology composed of six steps.

1. Node placing
2. PoP identification based on the placed nodes
3. PoP topology creation
4. Backbone topology creation
5. Capacity planning
6. iBGP topology design

The visualised process can be seen in figure 2.11.

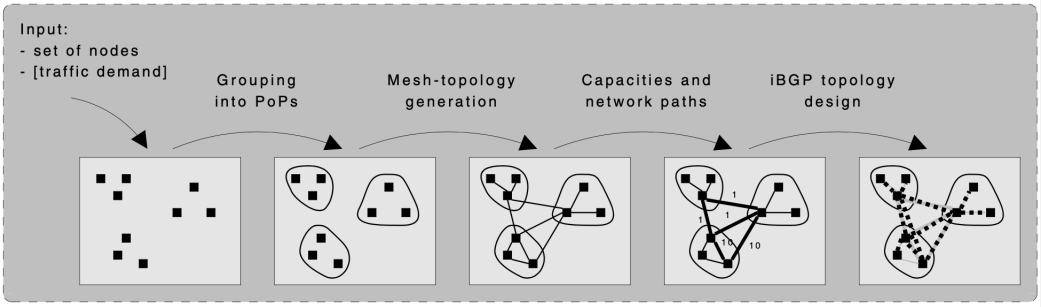


Figure 2.11: IGen process, Source: [3]

It defines concepts such as PoPs and a backbone in the generation process. Placement of the nodes is done similarly to BRITE [10], but the following steps that separate the nodes into specific service provider categories differentiate IGen. The logic of how nodes in a PoP are connected is based on operational practice. Further, the backbone creation step only targets nodes categorised as backbone nodes. In general, it is more opinionated compared to purely degree-based generators like BRITE due to its focus on service provider networks [13].

2.4.5 Delaunay and Gabriel

In IGen there are several backbone design heuristics present, the Delaunay triangulation is one of them. It is classified as a geometric spanner and leads to a topology with alternate paths between nodes. Due to its construction out of triangles, a node with only one edge is impossible and leads to a naturally redundant network. The minimum spanning tree is a subset of the Delaunay triangulation. The resulting topology is cost-effective and redundant [3].

A Delaunay triangulation with 8 nodes is shown in figure 2.12.

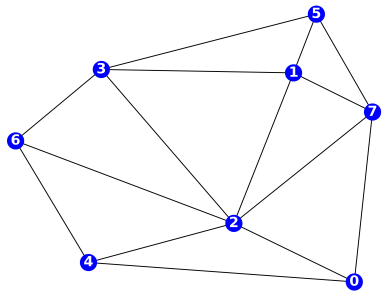


Figure 2.12: Delaunay triangulation

Another type of graph is the Gabriel graph, which is a subset of the Delaunay triangulation. It has been shown in [12] that it is one of the best options when generating backbone network structures. A Gabriel graph is a subset of the Delaunay triangulation. Each edge is only kept if there are no other nodes present in the circle with the start and end node on the circumference. This leads to a reduction in the number of edges. There have been several papers about the application of Gabriel graphs in network topologies. It generates networks at a small cost but tends to generate grid-like structures and leaves stubs that negatively affect redundancy [12].

A Gabriel graph with 8 nodes is shown in figure 2.13.

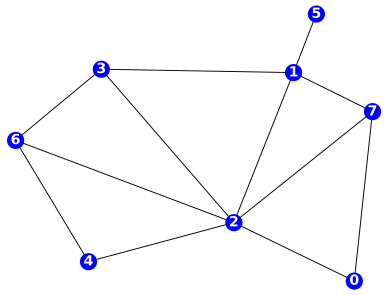


Figure 2.13: Gabriel graph

2.4.6 Robustness

Quantifying network robustness is a field of active research. There are many options to define a metric, and it is difficult to find a generally applicable one. One of these metrics is based on the removal of nodes until the graph splits. There are options on how the to-be-removed nodes are chosen. One option is a random selection, and another is a targeted attack based on certain properties, node degree for example [5].

2.5 Implementation

This section documents the implementation of features and explains major decisions that had to be made.

2.5.1 MOST-Model

The Graph Generator's approach for topology generation is based on the methodology described by IGen [3]. However, several changes and additional steps have been introduced into the generation process. Some of these changes cover specific use cases, and others try to optimise and compensate for shortcomings of the topology.
The result should be a router-level topology composed of a backbone made up of core nodes. Core nodes should be connected in a fitting mesh-like structure. Attached to the core nodes are PoPs with predefined structures based on operational practice.

The result is the MOST-Model which is composed of 8 steps:

1. PoP placing
2. PoP region clustering
3. Add PoP region core nodes
4. Add random core nodes
5. Add core node edges
6. Add random core node edges
7. Topology optimisations
8. PoP structure creation

2.5.1.1 PoP Placing

In the initial step, the PoPs are placed onto a two-dimensional plane. This step is similar to what happens in IGen [3], with the significant difference that the placed points are not routers. This approach was taken as the goal of the model was to have specific PoP structures predefined. As these structures are composed of routers, placing routers and having predefined PoP structures can not be done simultaneously.

The coordinates are chosen randomly based on a defined probability distribution. Both currently available distributions can not fully represent all intrinsics of how PoPs are distributed geographically. Integrating population centres and geographical blockers, such as lakes, is a challenge that needs further research. An approach for better generating these random coordinates can be found in section 3.2.

- The uniform distribution is a fundamental statistical concept. It represents a probability distribution where all values within a specified interval have an equal likelihood of occurrence [14].
- The normal distribution is a widely studied statistical concept. It is characterised by a symmetrical bell-shaped curve determined by its mean and standard deviation [14].

Figure 2.14 compares the two distributions. 1'000'000 samples were taken. The intervals were set to $[-1, 1]$ for the uniform distribution. The normal distribution was created with a mean of 0 and a standard deviation of 1.

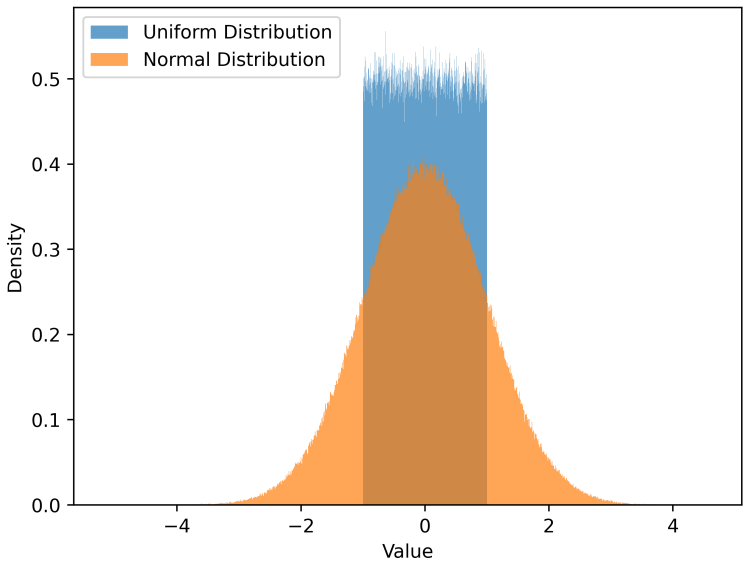
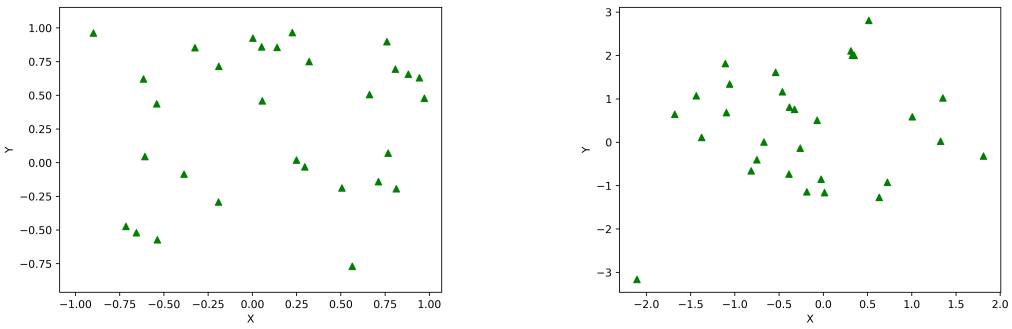


Figure 2.14: Probability distribution comparison

As mentioned, none of these distributions fully capture all aspects of how PoPs are spatially placed and distributed. Compared to each other, the uniform distribution is more suitable for distributing the PoPs. Generated topologies and feedback from network engineers have shown that the uniform distribution tends to be the better choice. Nevertheless, the normal distribution can also be a more appropriate choice depending on the intended usage. Based on this, the model does not explicitly define a probability distribution to be used.

A possible placement of 30 PoPs for both distributions can be seen in figure 2.15.



Uniform distribution

Normal distribution

Figure 2.15: Distribution comparison

2.5.1.2 PoP Region Clustering

The placed PoPs are grouped into so-called PoP regions in the next step. A parameter that determines the number of regions is part of the model. The regions should be selected so that nodes in the vicinity (distance-based) are clustered together. PoP regions are the base for the backbone of the generated topology and can be interpreted as geographical areas that service providers cover. IGen [3] contains a similar step but clusters routers into PoPs. Three clustering algorithms were evaluated.

- DBSCAN
- k-medoids
- k-means

DBSCAN was not considered suitable for the model. In contrast to the other two "k-algorithms", the number of desired clusters (PoP regions) cannot be specified and is determined as part of DBSCAN [15]. The model has the requirement to provide a parameter for the definition of the number of PoP regions.
While the k-medoids algorithm allows the definition of the number of clusters, it does not fit the model in another aspect. The k-medoids algorithm represents each cluster by an optimal data point within the cluster, known as the medoid. The medoid is usually the most centrally located point or the one that minimises the sum of dissimilarities to other points in the cluster.
K-means was chosen in the end due to the centroids it yields. These centroids are located in the centre of all cluster points and not based on a point in the cluster. They are more sensitive to outliers than medoids which is a desired property for the model [16].

Figure 2.16 shows how k-means divides the placed PoPs into 10 clusters. The arithmetic mean of the PoP locations defines the centre of each cluster.

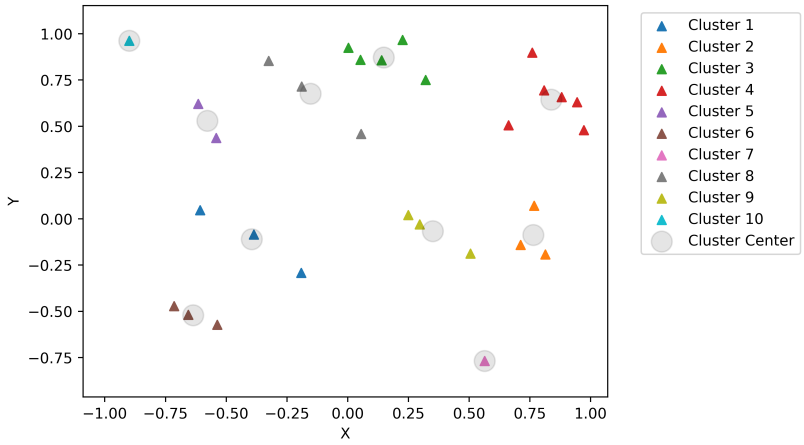


Figure 2.16: k-means clustering

2.5.1.3 Add PoP Region Core Nodes

Next, the core nodes are distributed around each PoP region's centre, each cluster's centroid. These core nodes represent routers that make up the backbone of the service provider network topology. The number of core nodes depends on the number of PoPs in a PoP region. PoP regions are assigned two core nodes as a base value.
Depending on a threshold parameter, additional nodes are placed. For example, when the threshold is set to 7, one core node is added for every seven PoPs in the region. The scaling is based on the circumstance that network utilisation increases with each PoP that is part of a PoP region. For larger PoP regions, the number of core nodes is increased to prevent network congestion.

The core nodes of each region are connected internally in a final step. Due to the possible scaling of the core nodes in a region, a method to connect them is needed. As the core nodes have been distributed around the PoP region centre, it is possible to use the Delaunay triangulation constrained to the PoP region core nodes to create a mesh structure. The Delaunay triangulation was chosen due to its property to create naturally redundant structures.

Figure 2.17 shows how core nodes (red dots) are placed, scaled if necessary and connected.

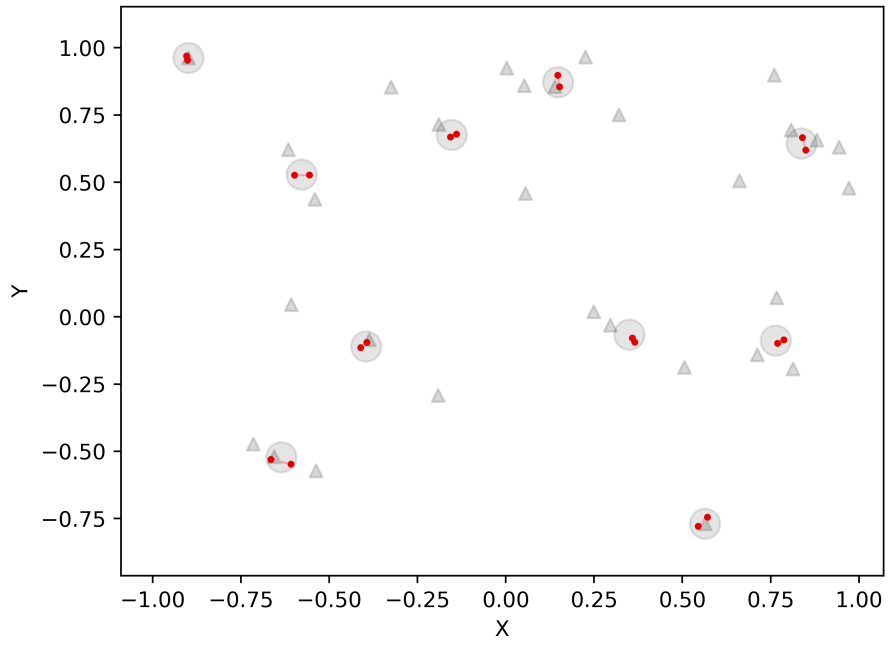


Figure 2.17: Adding core nodes to PoP regions

2.5.1.4 Add Random Core Nodes

Additional core nodes are added based on a model parameter. These random core nodes are defined by their own PoP regions and have no attached PoPs. They aim to improve the topology by introducing additional edges and potentially alternative paths. They are randomly distributed in the coordinate system using the same approach as in section 2.5.1.1.

Figure 2.18 shows how random core nodes (orange dots) are placed.

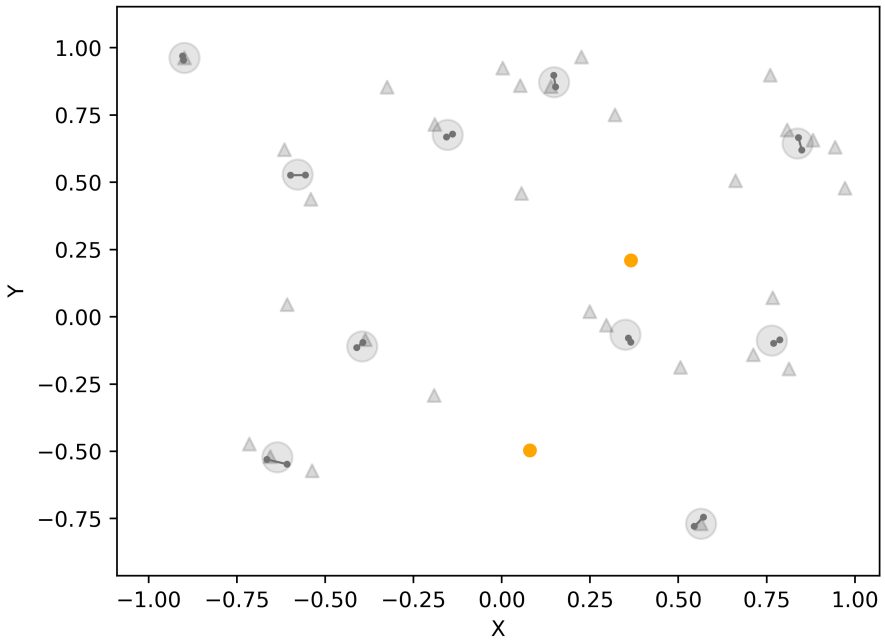


Figure 2.18: Adding random core nodes

2.5.1.5 Add Core Node Edges

After placing all core nodes, edges are added to connect them. How these edges are placed defines the resulting topology. The model sets these edges based on a Gabriel graph. Two topologies are implemented in the model and are selected by a parameter. It is important to mention that the model is intended to be used with the Gabriel topology, as ring topologies do not scale very well and are thus not representative of service provider network topologies. Regardless, the ring topology is available and implemented as an option for smaller topologies.

Gabriel Based on the research in section 2.4.5, the topology that fits a service provider network best is based on the Gabriel graph. There are multiple different approaches to creating a Gabriel graph. The chosen approach is more sophisticated than a brute force method but not as optimised as using a Voronoi diagram. The current approach creates the Gabriel graph based on a Delaunay triangulation. This approach was chosen as it optimises the calculation while still being understandable. It is based on and adapted from [17]. Since the Gabriel graph is a subgraph of the Delaunay triangulation, first the Delaunay triangulation is created based on the core nodes. Edges in a Delaunay triangulation can be determined by applying the following simplified process:

- Select three nodes.
- Draw a circle that places all three nodes on its circumference.
- If no other node is present in the circle area, add the edges that connect the three nodes and form a triangle.
- Go over all node combinations.

With the Delaunay triangulation present, the Gabriel graph can be extracted from it. The following simplified process illustrates how the subgraph is determined:

- Select two nodes that are connected by an edge.
- Draw a circle that places both nodes on its circumference.
- Remove the edge if another node is in the circle area.
- Go over all edges.

The resulting topology will not entirely resemble a Gabriel graph. Additional edges are present due to the added edges PoP region internally mentioned in section 2.5.1.3. In addition, the following two optimisations further influence the topology.

An optimisation that is always applied is that only one edge will be added between core nodes in separate PoP regions. Parallel edges that connect the same PoP regions are avoided through this. Further, fewer edges are present when compared to the Gabriel

graph. This optimisation prevents parallel edges, leading to a more cost-efficient network.

In section 2.5.1.7, an optimisation adds additional edges to the topology to make it robust. The addition of the edges in this current step of the model gets optimised to reduce the later addition of edges in section 2.5.1.7. Note that this optimisation is applied only in conjunction with the latter one. As the addition of the edges is based on the coordinates of the core nodes, it is possible that core nodes in a PoP region do not have any edges that connect to other PoP regions. In the worst case, one core node in the PoP region has all edges assigned that connect to other PoP regions.
This situation occurs naturally as the Gabriel graph removes the property of the naturally redundant triangles of the Delaunay triangulation and is a known issue in a Gabriel graph. Nodes are not biconnected as a consequence which reduces redundancy [12]. In addition, the limitation of one edge between PoP regions also plays into it as edges are removed.
To counteract this, when an edge is added, it is only guaranteed that the PoP regions get connected by an edge. The core node with the lowest degree in the PoP region will be selected to create the edge. If multiple core nodes with the same degree are present, the core node that leads to the shortest edge is selected by calculating the distances. The edges are distributed equally between the core nodes in the PoP region and create redundancy.

This optimisation does not affect the number of edges. The structure of the edges is affected but will match those of the Gabriel graph when all core nodes in a PoP region are contracted. A contracted Gabriel graph can be seen in figure 2.19.

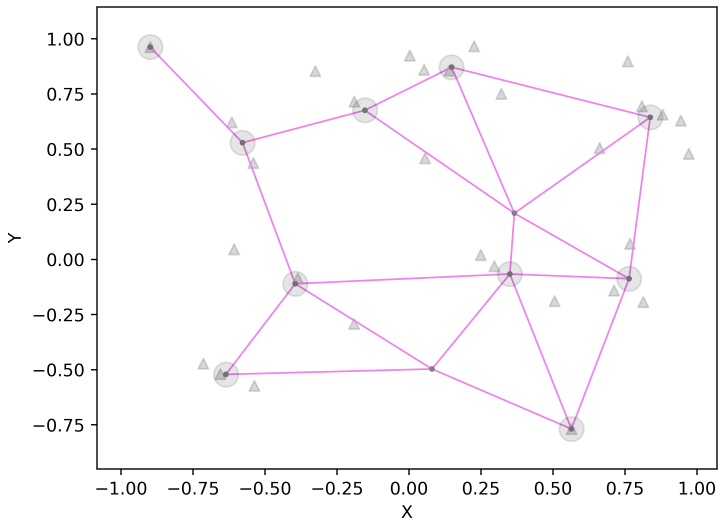
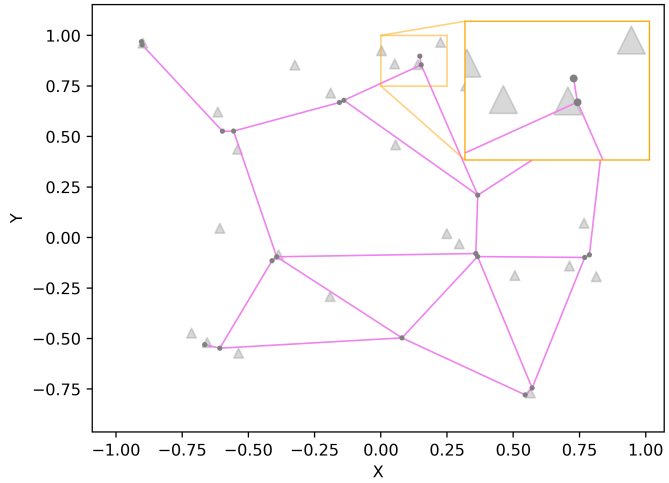
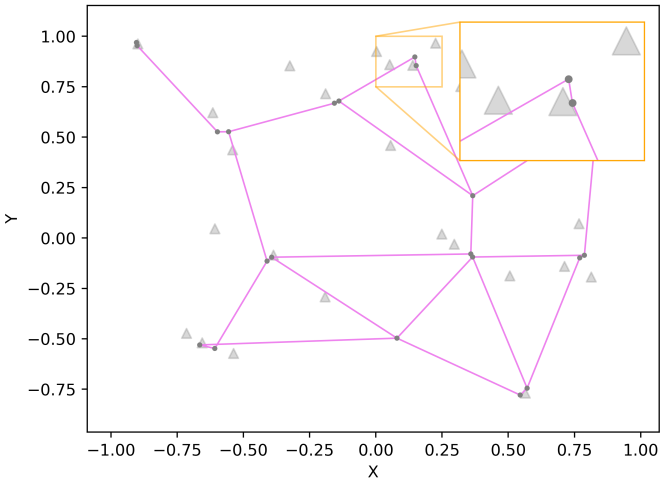


Figure 2.19: Node contraction

The optimisation is visualised in figure 2.20. The top image shows how the upper core node in the triangle got both edges assigned while the second core node in the PoP region has no edges connecting to other PoP regions. The bottom image shows the same situation but with the optimisation applied. The edges are distributed between the two available core nodes.



Optimisation disabled



Optimisation enabled

Figure 2.20: Gabriel optimisation comparison

The resulting topology with enabled optimisation is shown in figure 2.21.

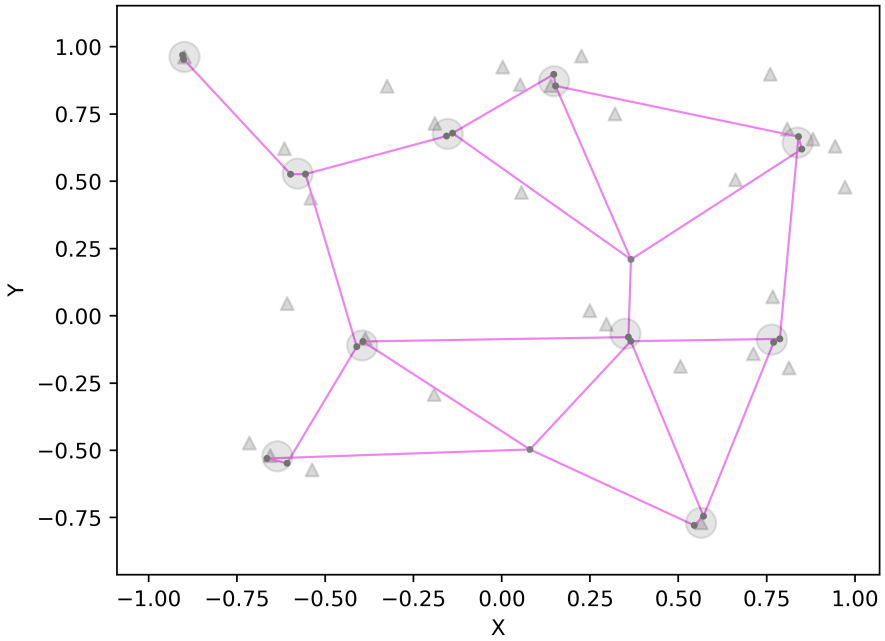


Figure 2.21: Gabriel result

Ring An approach based on solving the travelling salesman problem is used to create the ring topology. As that is a classic problem categorised as NP-hard, an approximated solution creates the cycle. Regardless, even the approximated solution does not scale well with more nodes. This limitation was accepted as the ring topology's use case is targeted at smaller topologies. In addition, a non-optimal solution satisfies the use case of the ring topology. As this topology was not the focus of this thesis, the implementation is done using the python-tsp library [18].

A ring topology should have exactly two edges per node. Due to the added edges in the PoP regions mentioned in section 2.5.1.3, the topology does not exactly match a ring. However, if the core nodes in the PoP regions are contracted, the ring is going to match. The resulting topology is shown in figure 2.22.

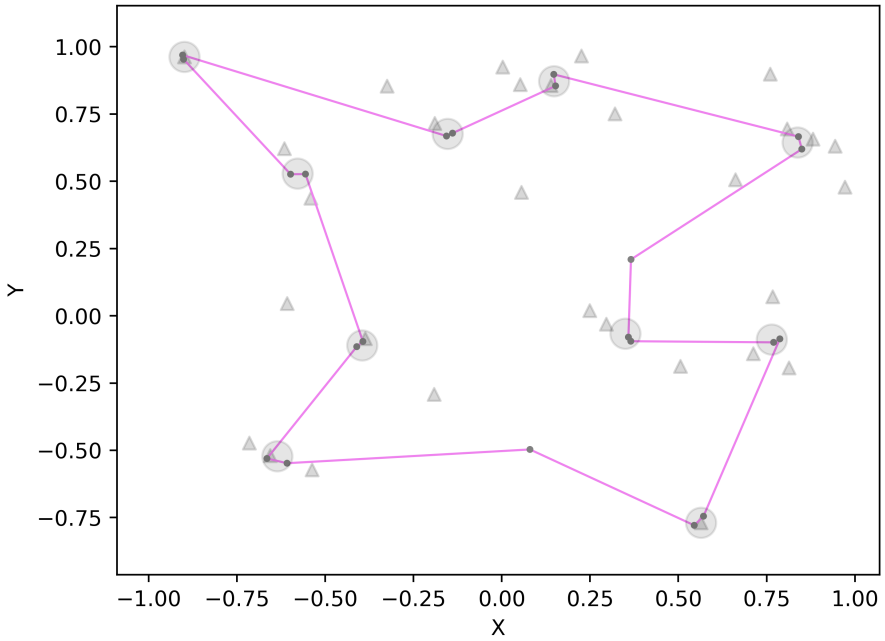


Figure 2.22: Ring result

2.5.1.6 Add Random Core Node Edges

On top of the edges added based on the topology, an amount of random core node edges are added to the topology. The amount is a parameter of the model. Through the introduction of these additional edges, redundancy is increased. The edges are added by selecting two random core nodes and adding an edge between them. If an edge already exists, another combination is tried.

In figure 2.23, the addition of a random core node edge (red) is illustrated.

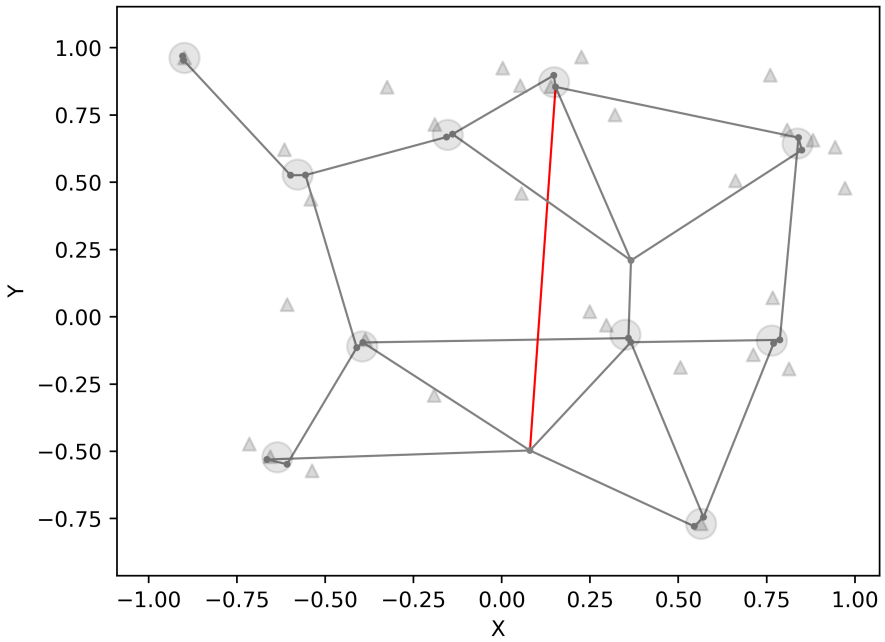


Figure 2.23: Add random core node edges

2.5.1.7 Topology Optimisations

At this stage, a final optimisation is done on the topology to mitigate cut vertices that negatively affect the robustness of the topology. As mentioned in section 2.5.1.5, the first step of the process happens already in advance to reduce the needed amount of added edges in this step. This optimisation step mitigates the shortcomings of using the Gabriel graph to create a topology and is an essential part of the model. The following process is applied until no cut vertices are present anymore:

1. Select a cut vertex.
2. Create a copy of the graph with the cut vertex removed. The graph will be composed of at least two components.
3. The nearest node to the cut vertex of each component is determined. Nodes in PoP regions that are not yet directly connected to the previously mentioned nearest node are prioritised.
4. Edges are added to connect these nodes in the original graph. They provide an alternative path between the potential components.

Figure 2.24 illustrates how a cut vertex is mitigated by adding an additional edge (green).

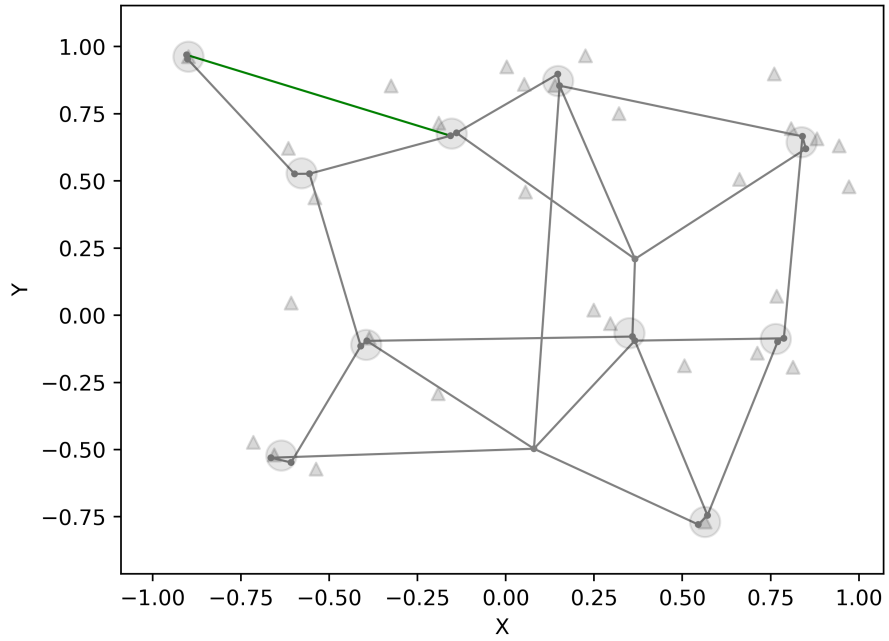


Figure 2.24: Optimisations

2.5.1.8 PoP Structure Creation

Each node represents a router, as the topology is on a router level. Until now, PoPs have been referred to in an abstract form and are not yet actually part of the topology. The PoPs will be transformed into their router structure and connected to the core nodes of their assigned PoP region. Two types of structures get assigned to the PoPs. The structures have been defined in collaboration with network engineers at the INS and are based on operational practices [19].

Level 1 A level 1 PoP represents a large PoP. Larger PoPs are placed where it is suspected that a higher data volume must be expected. These can be large cities or other densely populated areas. It is divided into a PoP Core Border Layer, PoP Distribution Layer and a PoP Access Layer. Using the PoP Access Layer customers connect to the PoP while routers in the Core Border Layer connect to a core node from the region. The structure is static for each level 1 PoP. It is composed of a total of 7 routers and is divided into:

- PoP Core Border layer: 2 Routers
- PoP Distribution layer: 2 Routers
- PoP Access layer: 3 Routers

Figure 2.25 visualises the structure of a level 1 PoP.

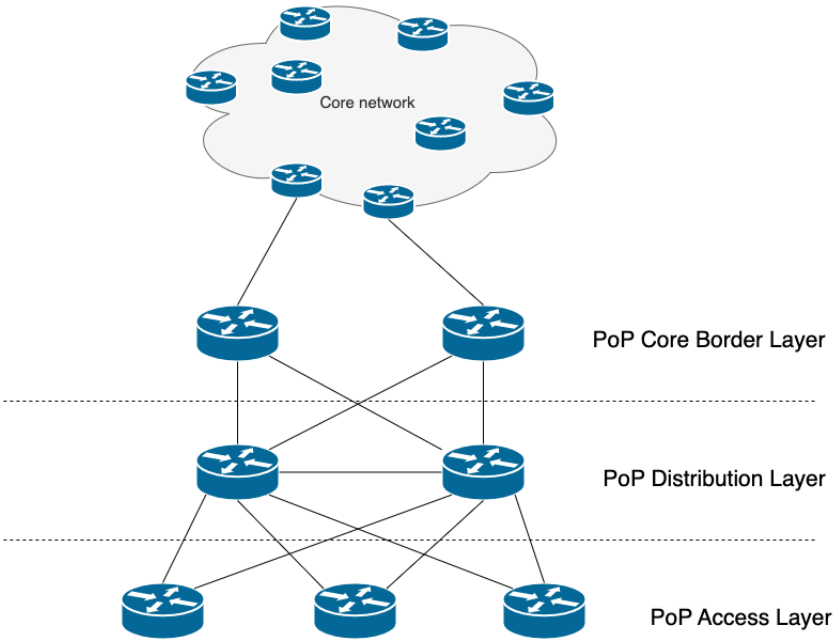


Figure 2.25: Level 1 PoP

Level 2 Level 2 PoPs are smaller compared to level 1 PoPs. This type of PoP is assigned to less dense areas. The PoP Core Border and PoP Distribution layers are combined when compared to level 1 PoPs. It is composed of a total of 5 routers and is divided into:

- PoP Core Border layer / PoP Distribution layer: 2 Routers
- PoP Access layer: 3 Routers

Figure 2.26 visualises the structure of a level 2 PoP.

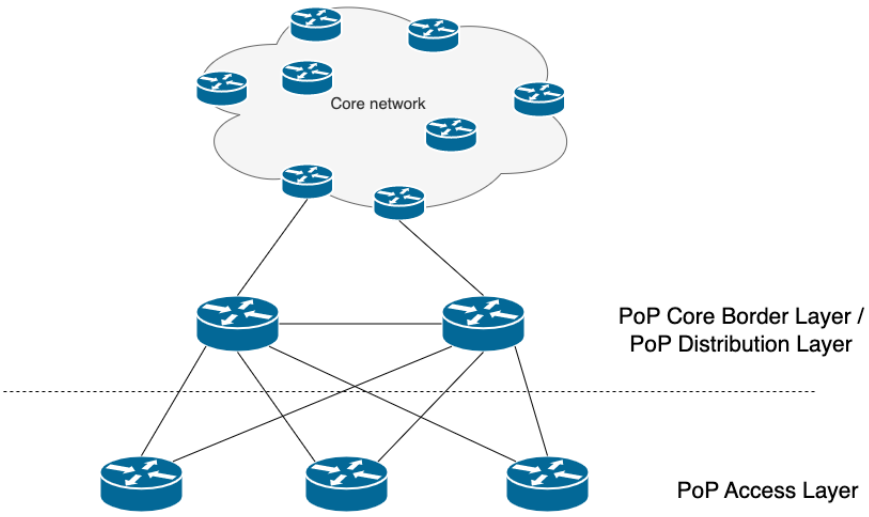


Figure 2.26: Level 2 PoP

The number of level 1 and level 2 PoPs generated can be adjusted using parameters. The distribution of level 1 and level 2 PoPs takes place as follows.
First, the PoP regions (where the core nodes are located) are sorted in descending order of degree. This sorting ensures that PoPs in high-degree PoP regions are prioritised when assigning level 1 PoPs. Thus, the available Level 1 PoPs are allocated to the high-degree PoP regions. After the limit of level 1 PoPs is reached, the rest of the PoPs are assigned level 2 structures.

The added PoPs can be seen in figure 2.27. Level 1 PoPs are coloured black, and level 2 PoPs are green. The zoomed boxes make it possible to show the structure of level 1 and level 2 since their spacing is otherwise minimal.

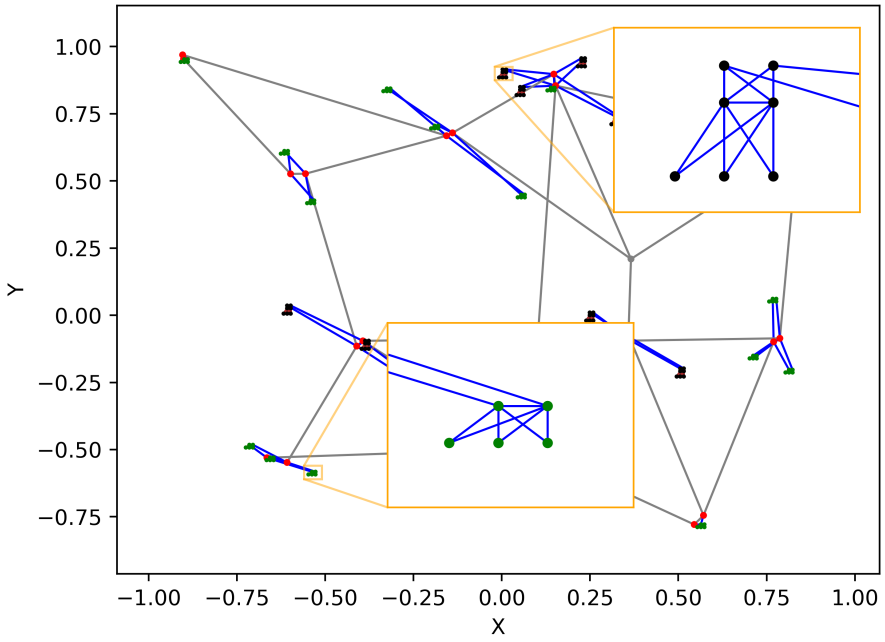


Figure 2.27: Adding PoP structures to Gabriel graph topology

2.5.1.9 Result

Based on IGen [3], the MOST-Model is able to create a multitude of network topologies that incorporate properties present in service provider networks.
The introduction of additional core nodes in section 2.5.1.4 adds additional edges to the topology that lead to alternative connections. Adding additional core node edges in section 2.5.1.6 has the same effect.

By using the Gabriel graph to create the backbone of the network in section 2.5.1.5, the topology is created based on a synthetic graph that yields low costs [12].
The Gabriel graph issue leading to stub links, mentioned in [12] is mitigated by optimisation steps described in section 2.5.1.5 and section 2.5.1.7. Biconnected nodes provide reliability and resistance against link failures. The topology minimises the costs associated with edges while still providing reliability.

The added PoP structures in section 2.5.1.8 are based on best practices [3]. They get assigned based on their structure sizes to provide PoP regions with high degrees with larger PoP structure sizes.

The final generated topology can be seen in figure 2.28.

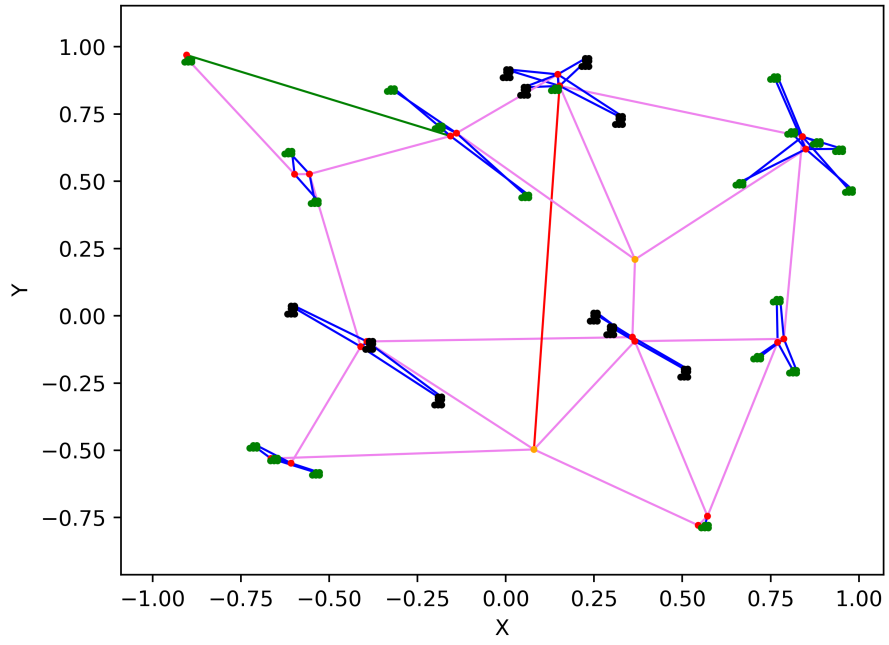


Figure 2.28: Final topology

2.5.2 Graph Generator

The Graph Generator is a Python-based application that uses the FastAPI framework to implement an HTTP API.
The MOST-Model is implemented as part of it using the NumPy, SciPy, scikit-learn and NetworkX libraries. The generation has been implemented in a way that allows easy extension concerning other clustering and topology options in the future. This has been achieved by splitting the logic into separate Python packages and defining shared function signatures.

The following three endpoints are available and can be interacted with:

- GET - `health` indicates the healthiness of the application. It is used in the Kubernetes deployment to monitor the status of the container.
- POST - `generate` provides the functionality to generate topologies. It accepts parameters based on the underlying MOST-Model implementation and returns a list of nodes and edges. More details concerning the parameters can be found in section 2.5.3.
- POST - `convert` accepts the structure returned by the `generate` endpoint as an input and converts it into the GEXF or GraphML file format.

The Graph Generator uses the OpenAPI 3 specification to define interactions with it and allows other applications to integrate it based on this quickly.

2.5.2.1 Graph Generator - Generate

The `generate` endpoint accepts multiple parameters. The following ones are available directly as part of the implemented MOST-Model.

- Amount of level 1 and 2 PoPs. See section 2.5.1.1.
- Number of PoP regions in order to cluster the PoPs as described in section 2.5.1.2.
- Scaling threshold that increases the number of core nodes in a PoP region based on the assigned PoPs. This is detailed in section 2.5.1.3.
- Amounts of random core nodes and core node edges to add. See section 2.5.1.4 and section 2.5.1.6.

In addition to the parameters defined as part of the MOST-Model mentioned in section 2.5.1, further parameters are available in the generation process.

- As introduced in section 2.5.1.1, the normal probability distribution, based on a standard normal distribution, is available as an alternative to the uniform distribution. The MOST-Model uses a uniform probability distribution.

- As an alternative to the Gabriel graph that is part of the MOST-Model, the ring topology can be chosen as described in section 2.5.1.5.
- The MOST-Model defines optimisation steps as described in section 2.5.1.5 and section 2.5.1.7. For experimental purposes, it is possible to disable them.

The result of the generation is returned as a custom JSON data structure composed of arrays for both nodes and edges. Listing 2.1 shows a shortened version of it.

```
{
    "nodes": [
        {
            "node_id": "0_1",
            "attributes": {
                "pop_region": "0",
                "node_type": "core",
                "coordinate_x": 0.4671831947233605,
                "coordinate_y": 0.36467550076790156
            }
        },
        ...
    ],
    "edges": [
        {
            "node_from": "0_1",
            "node_to": "0_2",
            "attributes": {
                "weight": 1.0,
                "edge_type": "core_to_core"
            }
        },
        ...
    ]
}
```

Listing 2.1: Generator output

Nodes get an identifier and attributes based on the MOST-Model assigned:

- node_id - Unique identifier of the node.
- pop_region - Defines to which PoP region this node is assigned.
- node_type - The type of the node. Can have the following types:
  - core
  - core_random
  - core_border_layer
  - distribution_core_border_layer

    - distribution_layer
    - access_layer
  - coordinate_x - Location that the node occupies on the x-axis.
  - coordinate_y - Location that the node occupies on the y-axis.

Edges have the following attributes:

  - node_from - Defines the node that is the start of the edge.
  - node_to - Defines the node that is the end of the edge.
  - weight - The weight of the edge. The following weights are currently assigned:
    - 1 - Edges between core nodes.
    - 10 - Edges between nodes that are part of a PoP.
    - 100 - Edges between nodes that are part of a PoP connecting to core nodes.
  - edge_type - The type of edge. Can have the following types:
    - core_to_core
    - core_to_core_random
    - core_to_core_random_optimization
    - core_to_pop
    - pop_to_pop

2.5.2.2 Graph Generator - Convert

The `convert` endpoint accepts the data as previously described in section 2.5.2.1 and converts them into another format. Currently, the GEXF and GraphML file formats are available as options. These two formats were chosen due to their common usage. The functionality to export the generated topology into one of these file formats allows their further use in external applications such as Gephi.

2.5.3 Integration into Graph Analyzer - Generator

Due to the microservice architecture, it is possible to use the Graph Analyzer Frontend without the new Graph Generator integration. This subsection describes how the Graph Generator was integrated into the existing Graph Analyzer system.

The Generator page is only available in the Frontend if the Graph Generator is available. This is checked by calling the `health` endpoint of the Graph Generator.

When the Graph Generator indicates that it is available, a new navigation item is displayed in the navigation bar that leads to the Generator page, as seen in figure 2.29.

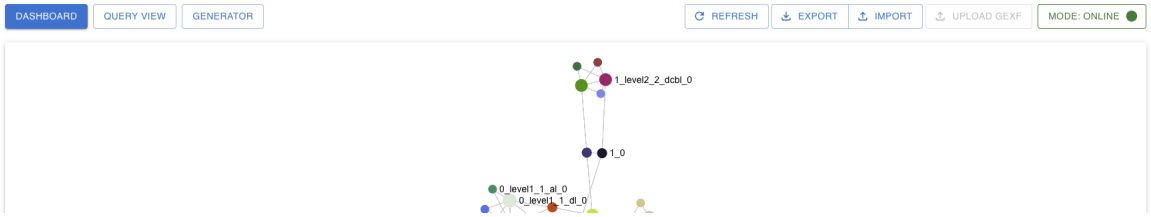


Figure 2.29: Frontend Dashboard - Generator navigation

On the Generator page, the parameters for the generation can be specified as described in section 2.5.2. In the Frontend, the Graph Generator is defined as an additional RTK Query API by using the OpenAPI 3 specification. The Frontend validates the input and enforces value limits before initiating the topology generation. All fields are required for the input validations in the Frontend and are validated as follows:

- L1 PoPs:
  - Must be an integer
  - Must be between 0 - 10'000
- L2 PoPs:
  - Must be an integer
  - Must be between 0 - 10'000
- PoP regions:
  - Must be an integer
  - Must be between 1 - 20'000
- Core node scaling threshold:
  - Must be an integer
  - Must be between 1 - 1'000

- Random core nodes:
  - Must be an integer
  - Must be between 0 - 10’000
- Random core node connections:
  - Must be an integer
  - Must be between 0 - 10’000
- Random distribution:
  - Must be ’uniform’ or ’normal’
- Topology:
  - Must be ’gabriel’ or ’ring’
- Topology optimization:
  - Must be a boolean

The validation is visualised in figure 2.30.

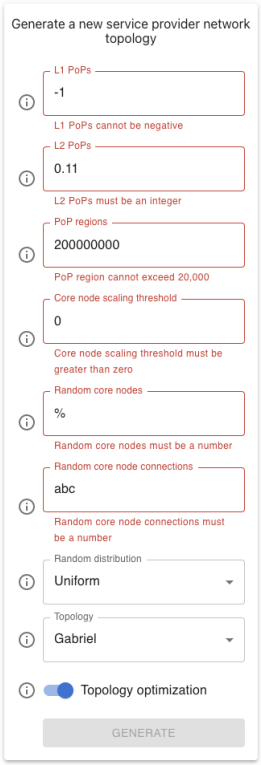


Figure 2.30: Frontend Generator - Validation

The Graph Generator also performs the same validation checks to ensure that input validation is not solely reliant on client-side validation. In addition to the previously mentioned basic input validation, the Graph Generator also performs extended validation checks. Context-dependent checks such as the total amount of edges exceeding that of a complete graph or the number of PoP regions being higher than the total number of PoPs. The situation mentioned first can be triggered by defining a value for the "Random core node connections" parameter that is too high in combination with the chosen topology. This and other errors are displayed in the Frontend, as seen in figure 2.31.

Figure 2.31: Frontend Generator - Error

Upon the successful generation of the topology using the `generate` endpoint, it is displayed as a preview in the Frontend, as can be seen in figure 2.32. The topology can be explored using the same component already used for the graph visualisation in the Frontend. Nodes are coloured based on their type and display their identifier while edges show what weight they have assigned.

Figure 2.32: Frontend Generator - Preview

Figure 2.32 also shows the option to download the generated topology. Using the `convert` described in section 2.5.2.2, a GEXF file is provided as a download.

2.5.4 Integration into Graph Analyzer - Upload

In order to allow the import of generated topologies in the form of GEXF files into the Graph Analyzer system via the Frontend, the existing Data Collector and API needed to be adjusted.

The Data Collector GEXF data import option has been extended to provide a GEXF listener option in addition to local files. When this option is active, a gRPC API is started and accepts the contents of GEXF files in order to be imported.
The API has been extended with two new endpoints.

- GET - `upload-status` indicates if the upload functionality is supported.
- POST - `upload` accepts GEXF files for import.

These two new endpoints reflect the gRPC API of the Data Collector. They act as intermediaries between the Frontend and Data Collector. The `upload-status` endpoint is required as the Data Collector only accepts GEXF files when specifically configured. When another topology data provider is selected, such as Jalapeño, GEXF imports are not possible. The added endpoints are shown in figure 2.33.

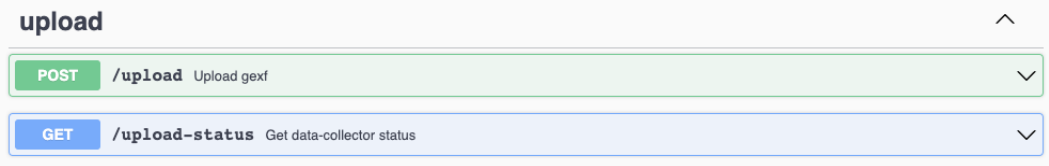
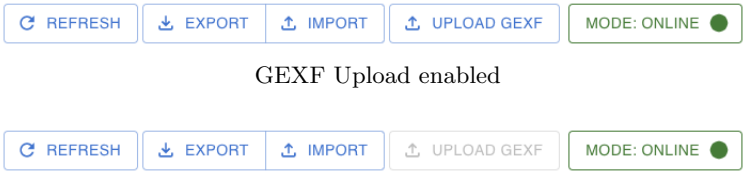


Figure 2.33: API upload endpoints

Figure 2.34 shows how the Frontend handles the availability of the GEXF upload.



GEXF Upload enabled

GEXF Upload disabled

Figure 2.34: Frontend Dashboard - GEXF upload

2.5.5 Weighted Support

An additional requirement of the thesis was to extend the existing Graph Analyzer system to support weighted graphs. To achieve this, the Data Collector was adjusted to include the weight data when processing topology information. The GEXF processing only needed minor adjustments as the weight is part of the GEXF file specification [20]. As a consequence of this change, the weight is now a required property when importing GEXF files.
Contrasting this, the Jalapeño processing required some more work. The used `NodeEdge` endpoints in the initial request and further subscription-based processing do not include the weight as part of the response. The weight corresponds to the `igp_metric` property, which is only available as part of `LsLink` [21].
When adding or updating an edge, an additional request is made to retrieve the `LsLink` with the identifier provided by `LsNodeEdge`. This allows the retrieval of the weight when using Jalapeño as the data source.

The weight from the data sources needs to be persisted in the Neo4j graph database in order to process it in the API. The weights were solely added to the edges persisted in the Neo4j graph database. The GDS projection, used to simplify the calculation of certain graph properties in the API, was kept as is.
The requirements specified that the newly added weights should only affect the calculated diameter property. Other properties should keep being calculated based on an unweighted undirected simple graph.
The diameter is newly calculated based on a directed weighted multi-graph. The shortest longest path in the graph is based on the cumulative weight of all edges that are part of the path. In addition, the amount of edges, or hops, is returned as a second value of the diameter calculation.

The visualisation of the diameter in the Frontend displays both the cumulative weights and the according hop count. See figure 2.35
To provide clarity, the API endpoints and graph property visualisations in the Frontend have been extended to detail which type of graph they are calculated based upon.

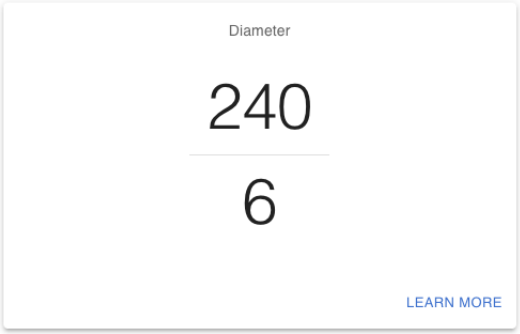


Figure 2.35: Frontend - Diameter property

2.5.6 Robustness

In order to find a robustness measure for a graph, approaches described in [5] were consulted. It describes two types of failures that affect a graph and thus defines how robust it is.

- Isolated and cascading failures
  - This type broadly covers malfunctions that occur frequently and can be characterised as random. Failures of individual devices due to hardware defects or defective links due to a fibre cut are examples of this.
- Targeted attacks
  - In a targeted attack, nodes or edges are specifically selected in order to try to disrupt a network with the lowest effort and biggest effects. This makes a lot more sense in the context of a large network and was therefore chosen as the method of choice.

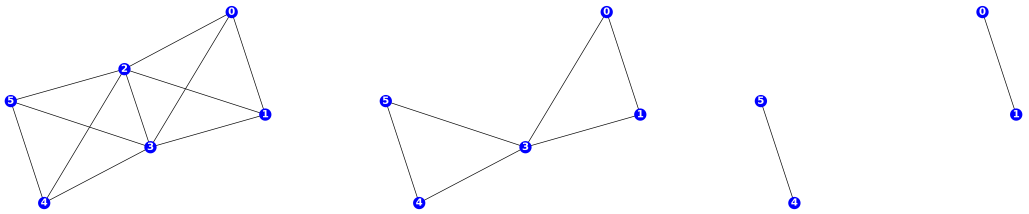
In large networks, the amount of low-degree nodes usually outweighs high-degree nodes. For isolating and cascading failures, the chances to remove a low-degree node are thus higher. Removal of a low-degree node usually quickly leads to a disconnected graph, but the resulting components are not necessarily proportional. For example, a disconnected graph with two components, one being a multiple of the size of the other, has a lesser effect on the operation of a network compared to equally sized components. The measure is also not reproducible in a usable way due to its random nature.
Due to these reasons, an approach using targeted attacks was chosen. Two degree-based types of targeted attacks are described in [5].

- Initial degree removal
  - In the initial degree removal method, all node degrees are initially calculated and subsequently removed in descending order until the network is disconnected. The degrees are only calculated once at the start. As a consequence changes in degrees on node removals are not taken into account in the following removals.
- Recalculated degree removal
  - With recalculated degree removal, the list of node degrees is recalculated after each removal. This introduces additional complexity but would reflect a more realistic attacker that re-assesses the next target for each step.

Based on the reasons given above, in the end, a targeted attack with recalculated degree removal was chosen.
The calculation works as follows. Before each removal, a list of nodes with descending degrees is created. Based on it the node with the highest degree is removed and checked

if the graph is still connected. This is repeated until the graph is no longer connected. In the case of multiple nodes with the same degrees, all removal combinations are evaluated in order to determine the minimal number of needed node removals.

Figure 2.36 shows how the robustness is calculated. The graph starts with a total of 6 nodes and the graph is connected as long as no more than 1 high-degree node is removed. This corresponds to 16.66% of all present nodes.



Initial Graph - Graph connected

Node 2 removed - Graph connected

Node 3 removed - Graph disconnected

Figure 2.36: Targeted attack with recalculated degree removal

The mentioned measures for robustness are visualised in the Frontend as can be seen in figure 2.37.

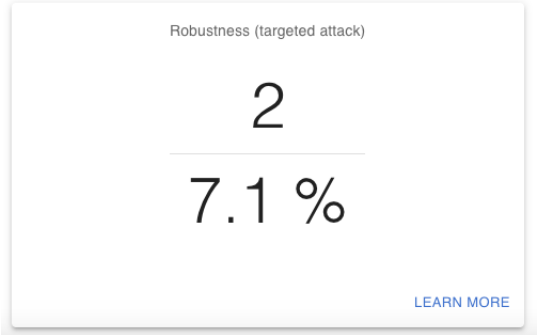


Figure 2.37: Frontend - Robustness property

Chapter 3

Conclusions

The upcoming sections cover accomplished use cases. Subsequently, possible enhancements are explored, and alternative approaches discussed.

3.1 Retrospective

3.1.1 Functional Requirements

Table 3.1 lists all defined use cases from section 4.1 and indicates via traffic light colours if they have been fulfilled.

ID	Use case
FR-01	Required: Generate topology
FR-01.a	Required: Generate topology - Choose the number of level 1 PoPs
FR-01.b	Required: Generate topology - Choose the number of level 2 PoPs
FR-01.c	Required: Generate topology - Choose the number of PoP regions
FR-01.d	Required: Generate topology - Choose the core node scaling threshold
FR-01.e	Required: Generate topology - Choose the number of random core nodes
FR-01.f	Required: Generate topology - Choose the number of random connections
FR-01.g	Required: Generate topology - Choose the random distribution
FR-01.h	Required: Generate topology - Choose the topology
FR-01.i	Required: Generate topology - Toggling optimisation
FR-01.j	Required: Generate topology - Weighted edges
FR-02	Required: Usage of generated topology
FR-02.a	Required: Usage of generated topology - Download
FR-02.b	Optional: Usage of generated topology - Graph Analyzer
FR-03	Optional: Extend Graph Analyzer to support weighted edges
FR-04	Optional: Extend Graph Analyzer to support a robustness property

Table 3.1: Fulfilment of use cases

3.1.2 Non-Functional Requirements

Table 3.2 contains the results of the NFR validation process defined in table 4.1. Traffic light colours indicate if an NFR has been met.

ID	Result
NFR-01	The API response time is logged by using a simple Curl command and the output can be found in listing 3.1. A total of 2802 nodes were generated in 6.40 seconds
NFR-02	To test the self-healing capability of the application, it was tested if the Graph Generator restarts after it has been killed. As can be seen in listing 3.2, the Graph Generator was up and running again after about 10 seconds without any manual intervention.
NFR-03	The input validation was described in detail in section 2.5.3 and was successful.
NFR-04	Despite the likely usage of the Graph Analyzer system on desktop-sized devices, the Frontend has been programmed in a responsive way that adapts to smaller devices and viewports. Figure 3.1 shows the Generator on a mobile device.
NFR-05	All applications were created cloud-native right from the start. Thus, all applications run in containers and are created and validated by pipelines. Logging takes place directly after stdout/stderr and the logic of the import and the backend is divided into its services.
NFR-06	Each part of the system has its pipeline that performs linting, testing, building and tagging the container images.
NFR-07	All components were successfully deployed using a Helm chart in Kubernetes.

Table 3.2: NFR validation

```
curl -o /dev/null -s -w 'Total: %{time_total}s\n'  -X 'POST' \
'https://graph-generator-api-jagw.stu.network.garden/generate' \
-H 'accept: application/json' \
-H 'Content-Type: application/json' \
-d '{
"l1_pops": 200,
"l2_pops": 200,
"pop_regions": 200,
"core_node_scaling_threshold": 7,
"random_core_nodes": 2,
"random_core_node_connections": 2,
"random_distribution": "uniform",
"topology": "gabriel",
"no_cut_edges_and_nodes": true
}'
Total: 6.405085s
```

Listing 3.1: Generating a large network with at least 1'000 nodes

```
while true; do date; curl -s -o /dev/null -w "%{http_code}" "https://
    graph-generator-api-jagw.stu.network.garden/health";  sleep 1; echo;
    done
Mon Jun  5 20:57:02 CEST 2023
200
Mon Jun  5 20:57:04 CEST 2023
200
Mon Jun  5 20:57:05 CEST 2023
404
Mon Jun  5 20:57:06 CEST 2023
404
Mon Jun  5 20:57:07 CEST 2023
404
Mon Jun  5 20:57:08 CEST 2023
404
Mon Jun  5 20:57:10 CEST 2023
404
Mon Jun  5 20:57:11 CEST 2023
404
Mon Jun  5 20:57:12 CEST 2023
404
Mon Jun  5 20:57:14 CEST 2023
404
Mon Jun  5 20:57:15 CEST 2023
404
Mon Jun  5 20:57:16 CEST 2023
200
Mon Jun  5 20:57:17 CEST 2023
200
```

Listing 3.2: Self healing test output

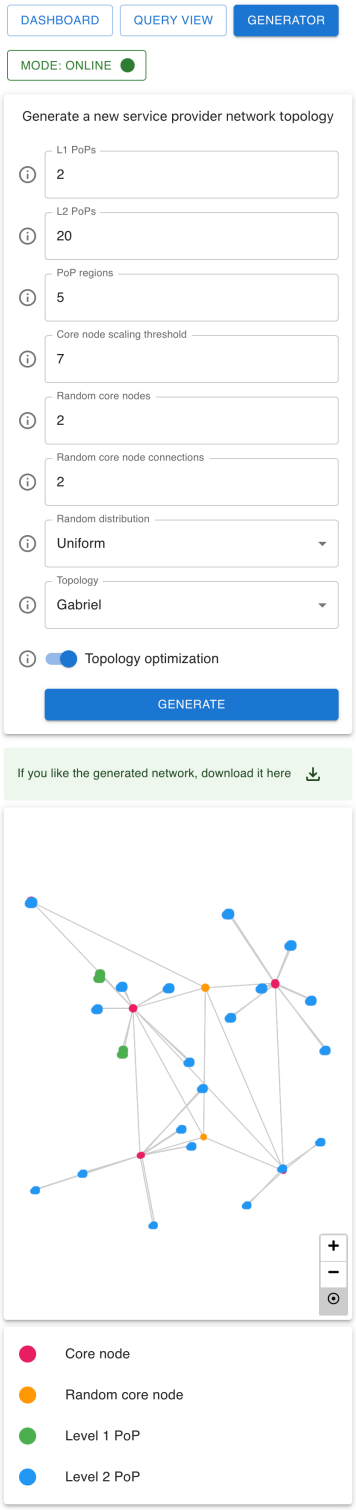


Figure 3.1: Responsive Frontend on a smaller viewport

3.1.3 Discussion

All functional requirements, as well as non-functional requirements, have been met in this thesis. The creation of the MOST model showed a possible approach to generate service provider-like networks. However, validating the generated results is challenging since publicly available data is sparse. However, early feedback from network engineers and visual comparisons with publicly available data have shown that the generated topologies come close to service provider network topologies. Further validation in depth is needed regardless.

3.2 Outlook

This section contains possible improvements and ideas for how it could be further developed.

3.2.1 Improvements

Below are the areas that could be improved in the future.

3.2.1.1 Robustness

Robustness is a big topic and could be a separate work integrating other factors into the Graph Analyzer. Therefore, the current robustness integration is considered one of many possibilities for integrating and calculating robustness.

3.2.1.2 Gabriel Graph

The Gabriel graph used for creating the backbone in the MOST-Model has several undesired properties as mentioned in [12]. Its tendency to leave stub links with a negative influence on redundancy has been addressed in the model with optimisation steps.
However, the tendency to strictly create grid-like structures disregarding star-like structures are open issues that could be improved in potential following works.

3.2.1.3 PoP Placement

Instead of randomising the placement of the PoPs, they could be based on other data sources or even user input. An example can be seen in figure 3.2. Swiss cities with a population of more than 10,000 were randomly selected in the graphic on the left. In the graphic on the right, random cities and municipalities were selected without population restriction. For simplicity, only the backbone is shown without PoPs.

Swiss cities with a population $\geq$ 10'000 Random Swiss cities

Figure 3.2: Possible topology of Switzerland, Source (Map): [22]

The currently used uniform probability distribution is not ideal for representing cities or population centres that would lead to a PoP. An approach to refine this was discussed briefly but not implemented due to time constraints. In a first step, city sizes would have been defined based on a Pareto distribution. The cities would have then been placed uniformly on a 2D plane with constraints to prevent bordering large cities. Based on the city sizes, PoPs would then be assigned.

Part II

Project Documentation

Chapter 4

Requirements

This chapter describes the various requirements that the final system created as part of the thesis needs to fulfil.

4.1 Functional Requirements

4.1.1 Actor - User

The system has only one actor, a regular user. His goal is to generate a network topology that resembles a service provider network topology. The generated topology should be downloadable through the existing Graph Analyzer Frontend in the form of a commonly used file format for graph representation.

4.1.2 Use Cases

The use cases are shown in figure 4.1 and are specified in a brief form.

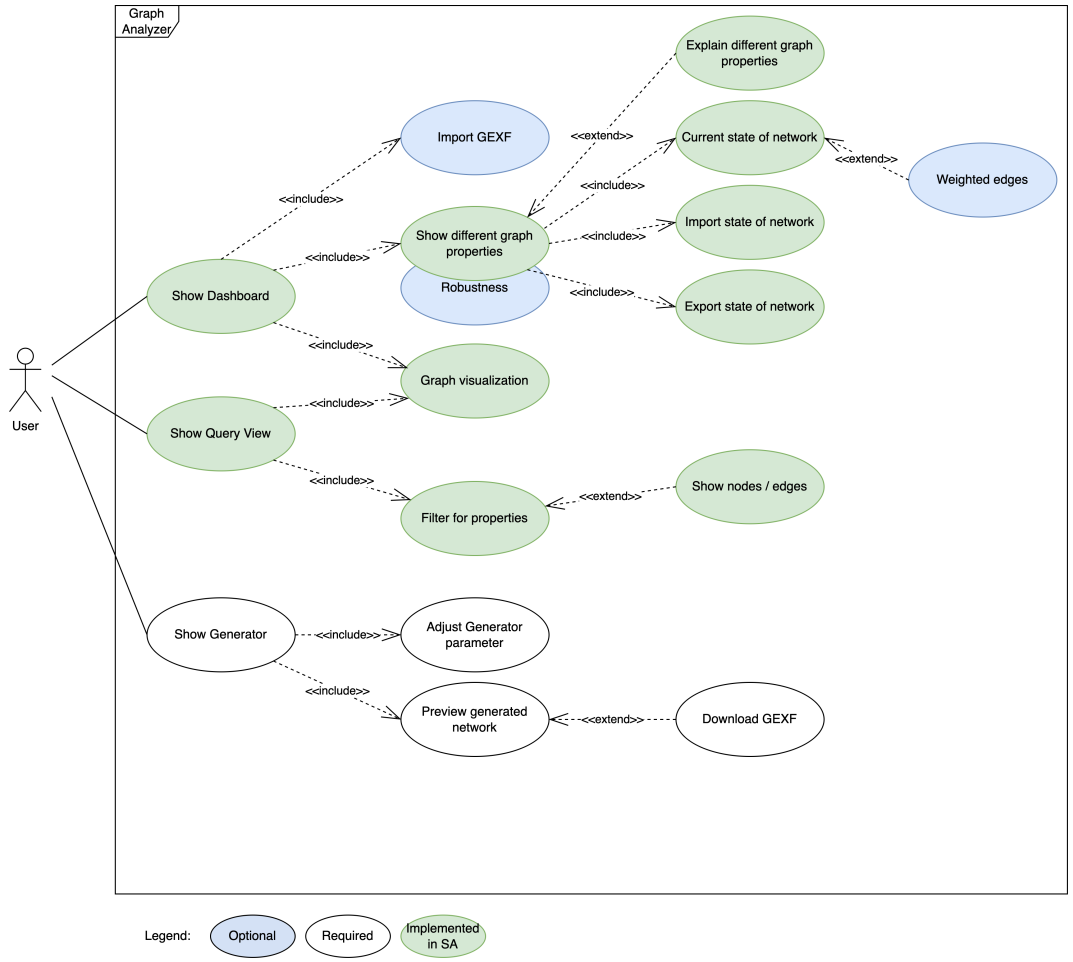


Figure 4.1: Use Case diagram

FR-01	
Use case:	Generate topology
Brief:	User accesses the application via the Graph Analyzer Frontend and is shown a generator page that can generate a new network.
Electivity:	Required

FR-01.a	
Use case:	Generate topology - Choose the number of level 1 PoPs
Brief:	The number of level 1 PoPs can be chosen by the user.
Electivity:	Required

FR-01.b	
Use case:	Generate topology - Choose the number of level 2 PoPs
Brief:	The number of level 2 PoPs can be chosen by the user.
Electivity:	Required

FR-01.c	
Use case:	Generate topology - Choose the number of PoP regions
Brief:	The number of generated PoP regions can be chosen by the user.
Electivity:	Required

FR-01.d	
Use case:	Generate topology - Choose the core node scaling threshold
Brief:	The threshold of how core nodes in a PoP region scale can be chosen by the user.
Electivity:	Required

FR-01.e	
Use case:	Generate topology - Choose the number of random core nodes
Brief:	The number of random core nodes can be chosen by the user.
Electivity:	Required

FR-01.f	
Use case:	Generate topology - Choose the number of random connections
Brief:	The number of random connections can be chosen by the user.
Electivity:	Required

FR-01.g	
Use case:	Generate topology - Choose the random distribution
Brief:	The type of random distribution that is used can be chosen by the user.
Electivity:	Required

FR-01.h	
Use case:	Generate topology - Choose the topology
Brief:	The type of the generated topology can be chosen by the user (ring, mesh).
Electivity:	Required

FR-01.i	
Use case:	Generate topology - Toggling optimisation
Brief:	The optimisation of the graph can be enabled or disabled by the user.
Electivity:	Required

FR-01.j	
Use case:	Generate topology - Weighted edges
Brief:	The edges must be weighted based on the provided ruleset.
Electivity:	Required

FR-02	
Use case:	Usage of generated topology
Brief:	After the network has been generated the user must be able to interact with the network.
Electivity:	Required

FR-02.a	
Use case:	Usage of generated topology - Download
Brief:	The user can download the generated topology in a commonly used file format for graph representation.
Electivity:	Required

FR-02.b	
Use case:	Usage of generated topology - Graph Analyzer
Brief:	The user can use the generated topology by importing it into the Graph Analyzer system.
Electivity:	Optional

FR-03	
Use case:	Extend Graph Analyzer to support weighted edges
Brief:	Extending the existing Graph Analyzer system to support weighted edges and reflect them in the calculated graph properties.
Electivity:	Optional

FR-04	
Use case:	Extend Graph Analyzer to support a robustness property
Brief:	Extending the existing Graph Analyzer system to support a new robustness property.
Electivity:	Optional

4.2 Non-Functional Requirements

The non-functional requirements that the result of the thesis needs to meet are specified below.

NFR-01	
Category:	Performance
Brief:	The system should be able to generate topologies with at least 1000 nodes.
Electivity:	Required

NFR-02	
Category:	Availability
Brief:	In case of a software fault the system should be self-healing within 30 seconds without manual interaction.
Electivity:	Required

NFR-03	
Category:	Usability
Brief:	The system should validate the input data of the user.
Electivity:	Required

NFR-04	
Category:	Usability
Brief:	The system should be usable on mobile devices by using a responsive design.
Electivity:	Optional

NFR-05	
Category:	Maintainability
Brief:	The system should be developed cloud-natively.
Electivity:	Required

NFR-06	
Category:	Maintainability
Brief:	The system should be tested and built through CI/CD.
Electivity:	Required

NFR-07	
Category:	Scalability
Brief:	The solution should be deployable to Kubernetes.
Electivity:	Optional

4.2.1 Validating NFRs

Table 4.1 shows how and when the NFRs are validated.

ID	How	When
NFR-01	Generating a large network with at least 1'000 nodes.	Before releasing the Alpha version.
NFR-02	Kill a container and measure the time until a new container has been spun up.	Before releasing the Alpha version.
NFR-03	Test the system's resistance regarding invalid user input and ensure that it displays helpful error messages.	Before releasing the Beta version.
NFR-04	Browser tools are used to check whether the website is responsive and can be used on devices with different screen sizes.	Before releasing the Beta version.
NFR-05	With the help of available cloud-native concepts, it is checked whether the architecture meets the requirements.	Before releasing the Alpha version.
NFR-06	It is checked whether a sufficient CI/CD pipeline exists for each part of the system.	Before releasing the Alpha version.
NFR-07	It is checked whether a method has been defined for how deployment to Kubernetes works.	Before releasing the Beta version.

Table 4.1: How NFRs are validated

Chapter 5

Design Decisions

This chapter contains design decisions that were made before and during the implementation. It focuses solely on the newly added Graph Generator application. Design decisions concerning the other parts of the existing Graph Analyzer system can be found in [6].

5.1 Graph Generator

This section describes the design decisions concerning the Graph Generator.

5.1.1 Programming Language

In the initially given task for this thesis, no programming language was given that had to be used for the implementation. Due to the application's scientific nature, performance was not a major concern. Based on the existing experience and knowledge, Go and Python were evaluated.
Python stands out compared to Go with its countless libraries. Extensive libraries are available that support the implementation of the Graph Generator. The used libraries are detailed in section 5.1.3. During the term project [6] it turned out that Go had fewer useful libraries in this area and a lot had to be implemented manually. Because of the reasons above, Python was chosen as the programming language for the Graph Generator.

5.1.2 Architecture

A microservice architecture was used so that the Graph Generator can be used independently from the existing Graph Analyzer system. This decision brings the following advantages:

- Can be used independently.
- Individual components can be scaled better.
- Clear separation of the different used programming languages.

5.1.3 Libraries

All libraries and frameworks this thesis uses are published under an open-source license. The list below shows the most important used libraries in the Graph Generator.

- NetworkX [1]
  - Used for the data structure and graph operations. It is also used for conversion to common graph file formats, specifically GEXF and GraphML.
- NumPy [2]
  - Used for mathematical operations such as generating random points or distance calculations.
- SciPy [3]
  - Used for calculating the Delaunay triangulation.
- scikit-learn [4]
  - Used for clustering using k-means.
- python-tsp [5]
  - Used in the creation of the ring topology by using its implementation of an approximated travelling salesman problem solver.

[1] `https://networkx.org`
[2] `https://numpy.org`
[3] `https://scipy.org`
[4] `https://scikit-learn.org`
[5] `https://github.com/fillipe-gsm/python-tsp`

5.1.4 API Framework

To implement the HTTP API for the Graph Generator the following Python frameworks were analysed:

- FastAPI [6]
- Flask [7]

Framework Comparison

A short comparison of the two frameworks can be found in table 5.1.

	FastAPI	**Flask**
Github Stars	58'000	63'000
Built-in API documentation	Yes	No
Built-in data validation	Yes	No
License	MIT	BSD 3-Clause
Maintained	Yes	Yes

Table 5.1: Framework comparison

The short comparison did lead to a clear result, since these two are very frequently used API frameworks and both meet the requirements. In the end, FastAPI was chosen because it offers features out-of-the-box, such as built-in docs (OpenAPI) or an integrated validation using Pydantic.

[6] `https://fastapi.tiangolo.com`
[7] `https://flask.palletsprojects.com`

Chapter 6

Architecture

This chapter explains the architecture of the implemented system. Decisions from chapter 5 influence the chosen architecture.

6.1 Architecture Model

In order to visualise and conceptualise the system and application architecture, the C4 model was used. Only the first three levels are shown, as it was decided that the fourth level is unnecessary. It can be automatically generated on-demand with tools in modern IDEs. Large parts and main decisions have been taken from the previous term project [6]. The following sections will focus on new and adjusted parts of this thesis.

6.1.1 System Context Diagram

The System Context diagram can be seen in figure 6.1. A user interacts with the Graph Analyzer system, which depends on the data provided by network topology providers.

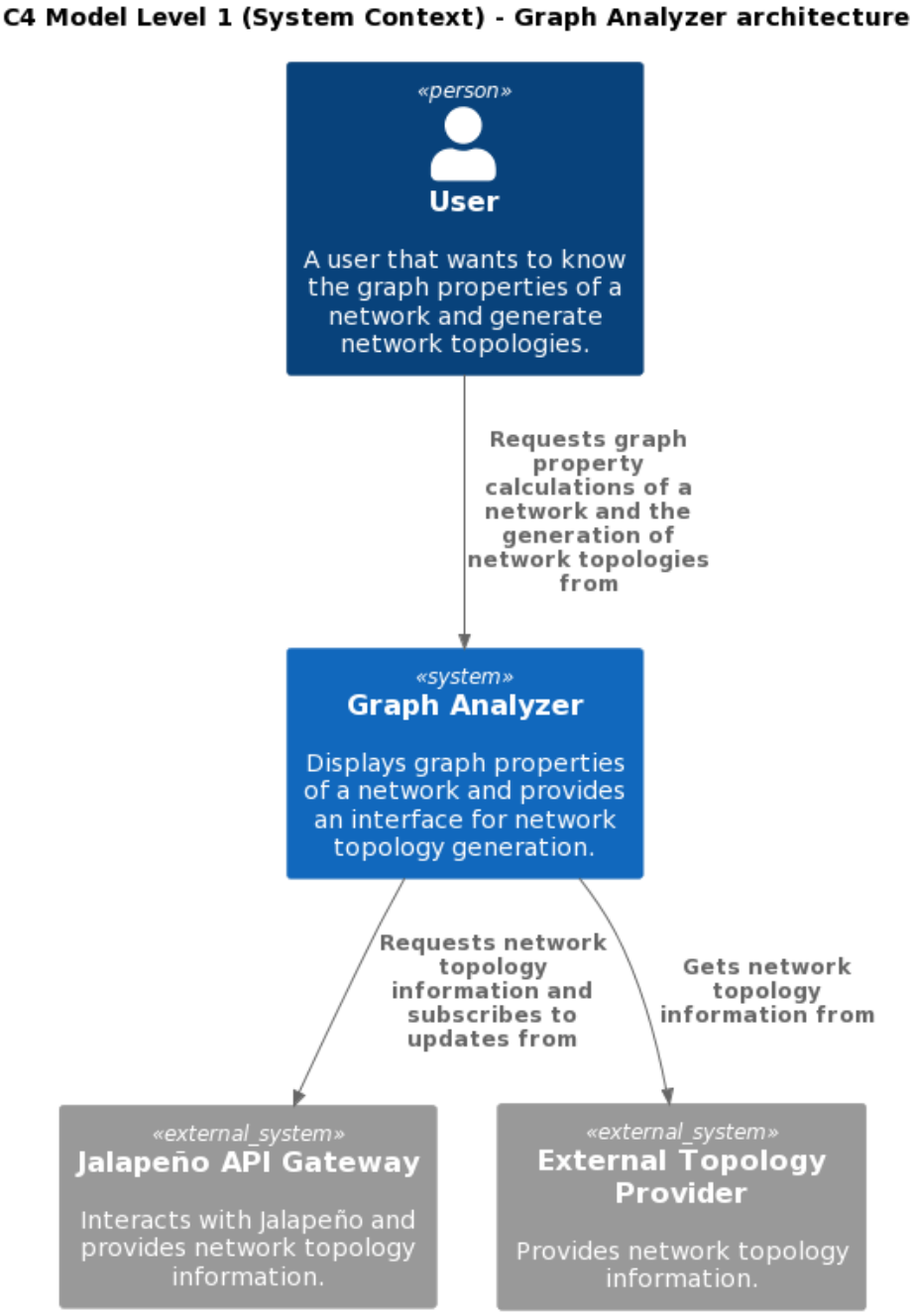


Figure 6.1: C4 Model Level 1 - System context diagram

6.1.2 Container Diagram

Figure 6.2 shows the Container diagram of the Graph Analyzer System. The user only interacts with the Single-Page Application directly, which is responsible for data visualisation and providing the interface for network topology generation. The Single-Page Application interacts with the API Application and the Topology Generator Application. The Topology Generator Application is intentionally not part of the API Application, intending to provide the possibility to integrate it into other systems. The API Application is mainly responsible for calculating graph properties from a graph stored in the Graph Database. In addition, it is able to interact with the Data Collector Application to trigger new imports. The Data Collector Application implements the logic for processing network topology data and persists it in the Graph Database.

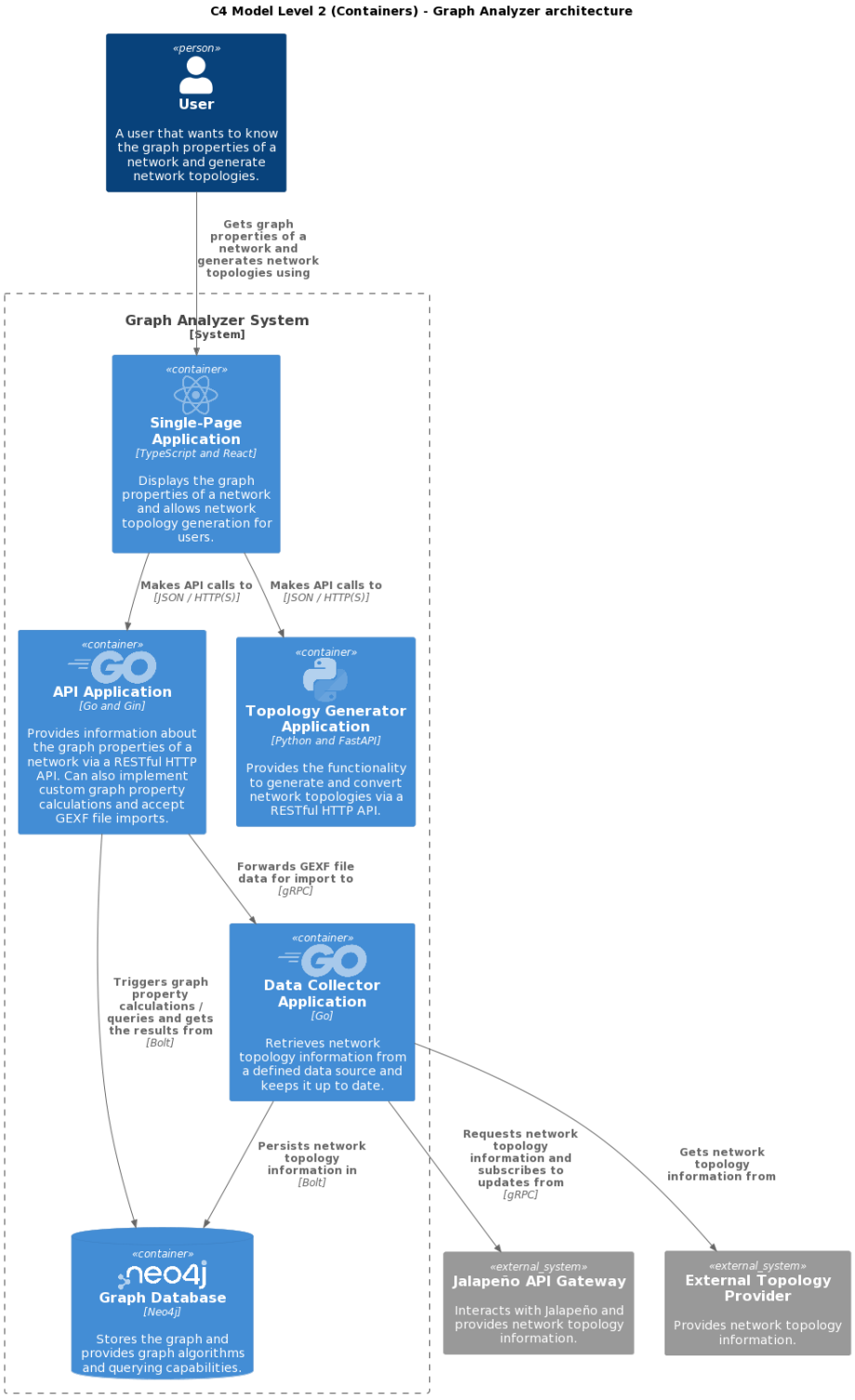


Figure 6.2: C4 Model Level 2 - Container diagram

6.1.3 Component Diagram - Single-Page Application

The Single-Page Application's Component diagram can be found in figure 6.3. The only change compared to the previous version in the term project [6] is the additional communication to the newly added Topology Generator Application.

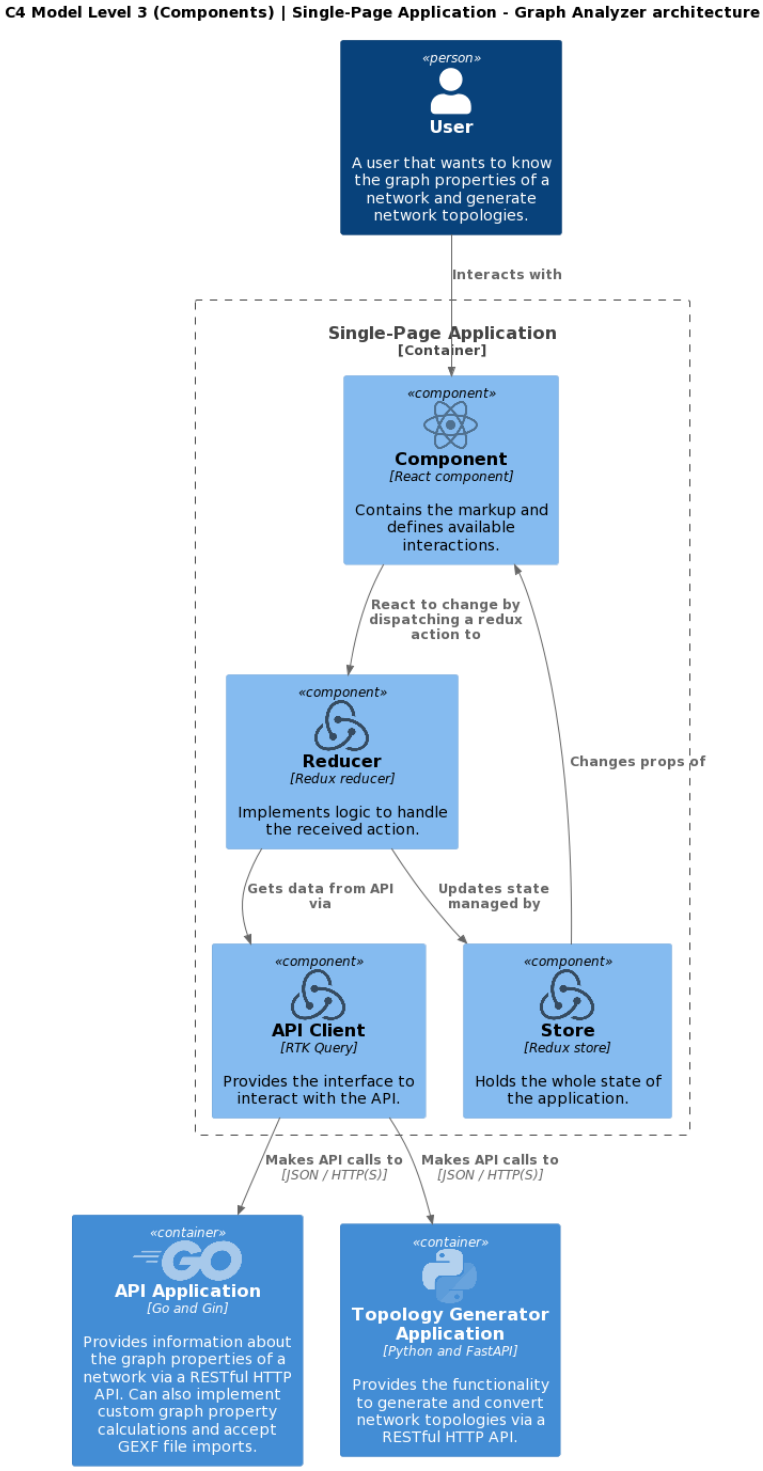


Figure 6.3: C4 Model Level 3 - Component diagram - Single-page application

6.1.4 Component Diagram - API Application

The Component diagram of the API Application can be seen in figure 6.4. Excluding the newly added interaction with the Data Collector Application, it stayed the same as in the term project [6]. The added interaction is present due to the possibility of uploading GEXF files in the Single-Page Application to import them by forwarding them to the Data Collector Application.

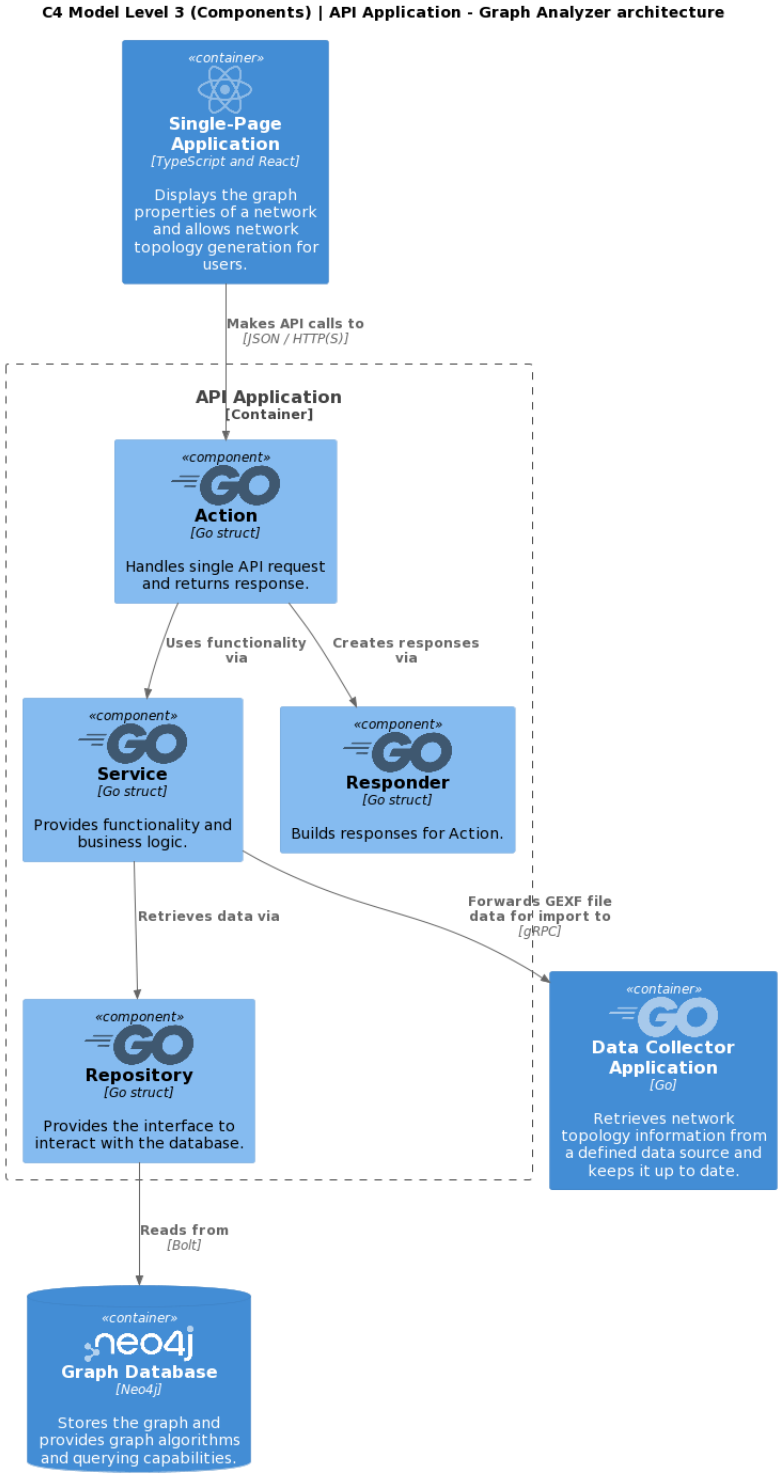


Figure 6.4: C4 Model Level 3 - Component diagram - API application

6.1.5 Component Diagram - Data Collector Application

The same interaction as described in section 6.1.4 can be seen in figure 6.5 as the only difference compared to the term project [6]. The GEXF file data is forwarded by the API Application and then processed by the Data Collector Application.

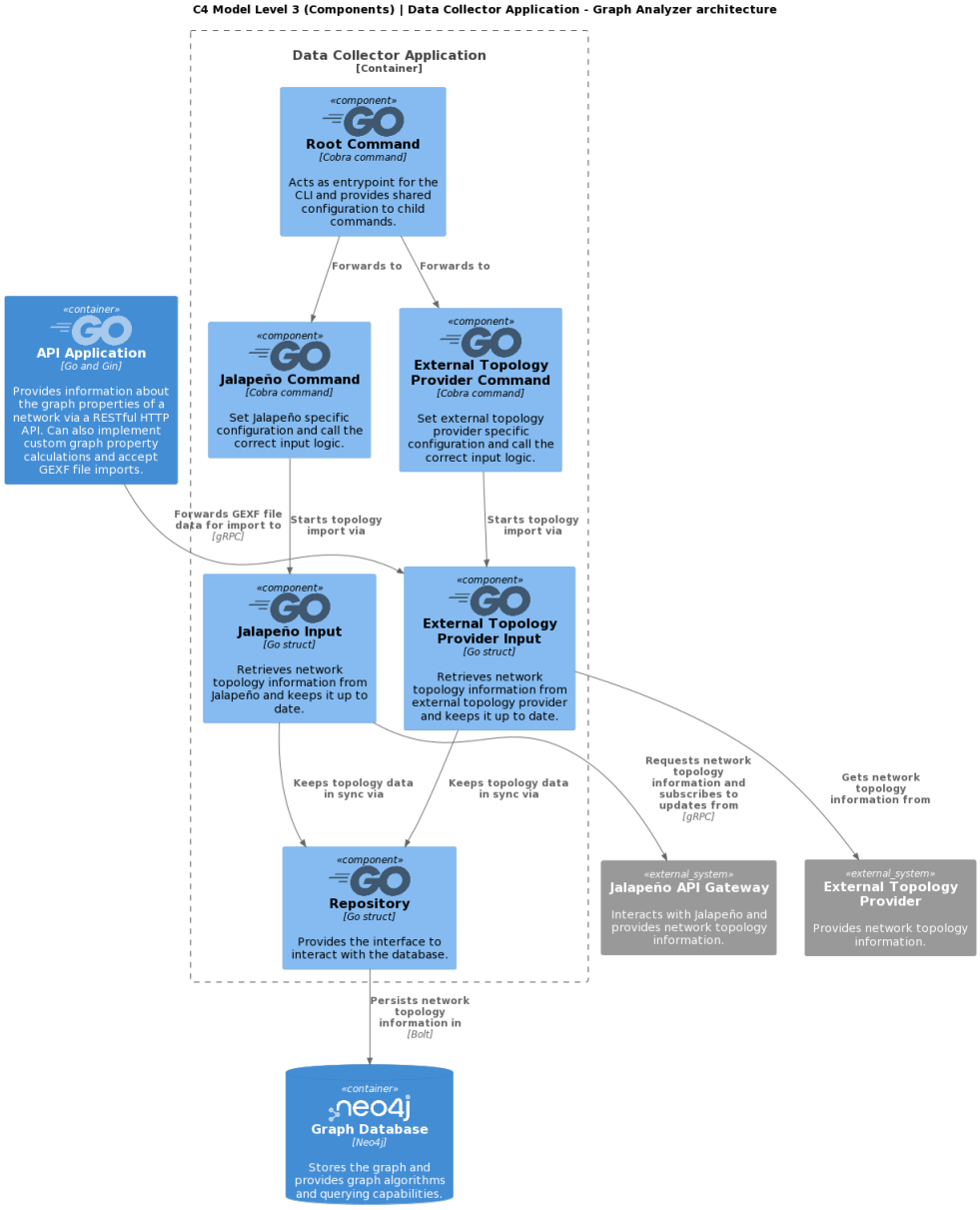


Figure 6.5: C4 Model Level 3 - Component diagram - Data Collector application

6.1.6 Component Diagram - Topology Generator Application

The Topology Generator Application implements the MOST-Model and provides an interface to interact with the Single-Page Application. In addition, it provides the functionality of converting the generated topology data into the form of a commonly used graph file format. This can be seen in figure 6.6.

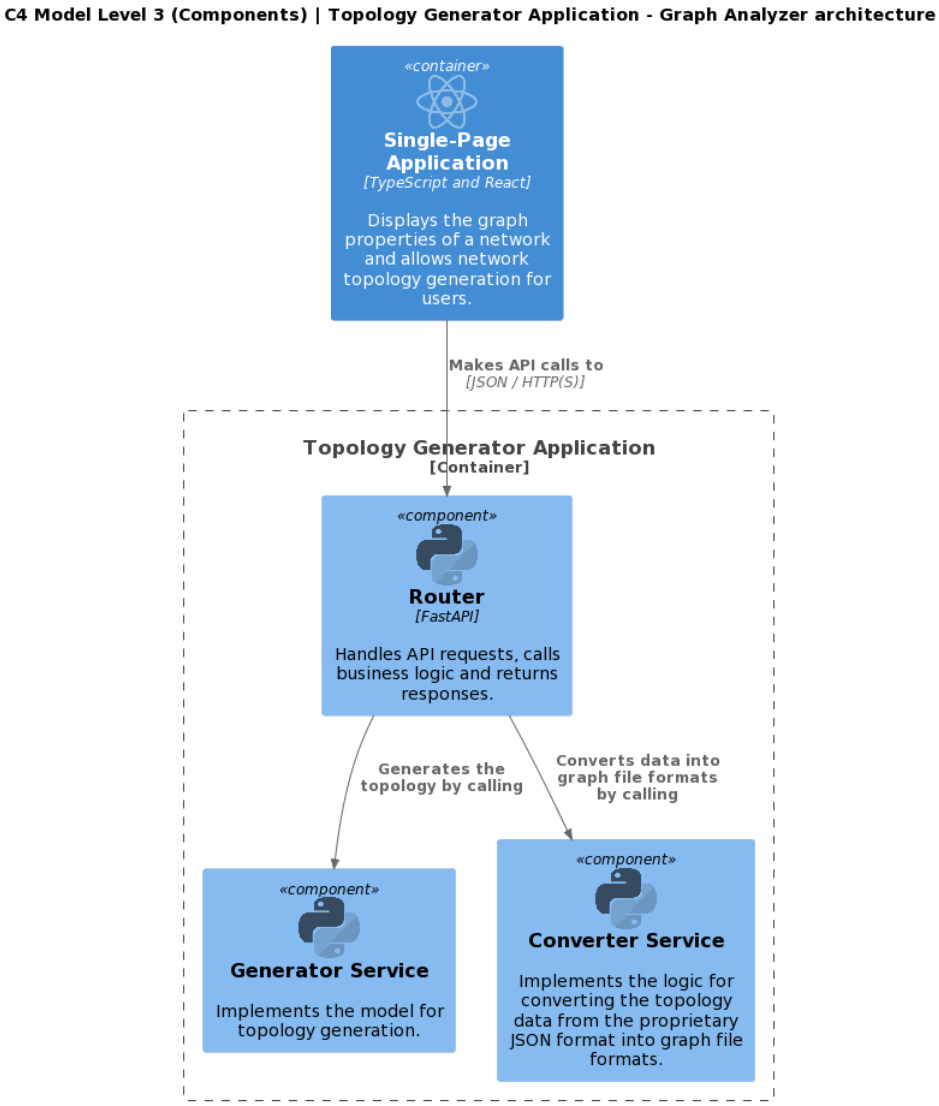


Figure 6.6: C4 Model Level 3 - Component diagram - Topology generator application

6.2 Deployment

This section describes how the Graph Analyzer system can be easily deployed in a Kubernetes cluster using a Helm chart. It is based on previous work done in [6].

6.2.1 Helm Chart

The existing Helm chart was updated from the term project in order to integrate the changes made during this thesis.
The Helm chart consists of five parts:

1. Frontend
2. Neo4j helm dependency
3. API
4. Graph Generator
5. Data Collector

As mentioned in section 2.3.2, it should be possible to use the existing Graph Analyzer independently from the Graph Generator if desired. In order to achieve this, some adjustments to the Helm Chart were necessary.
The configuration snippet in listing 6.1 shows how the Graph Generator can be excluded entirely from deployment. It also shows how it is possible to run the Data Collector in Jalapeño or GEXF listener mode.

```
graphGenerator:
  enabled: true
config:
  jalapeno:
    enabled: false
    jagwURL: "JAGW IP"
  gexflistener:
    enabled: true
```

Listing 6.1: Helm deployment - values.yaml

6.2.2 Kubernetes

Figure 6.7 gives an overview of which resources are used in Kubernetes. Three endpoints are exposed to the outside via an ingress object:

1. Frontend
2. API
3. Graph Generator

The two services for the Neo4j graph database and the Data Collector are not exposed externally and are only available within the Kubernetes namespace. These two services do not have to be exposed since they are only called within the namespace, increasing the system's security.

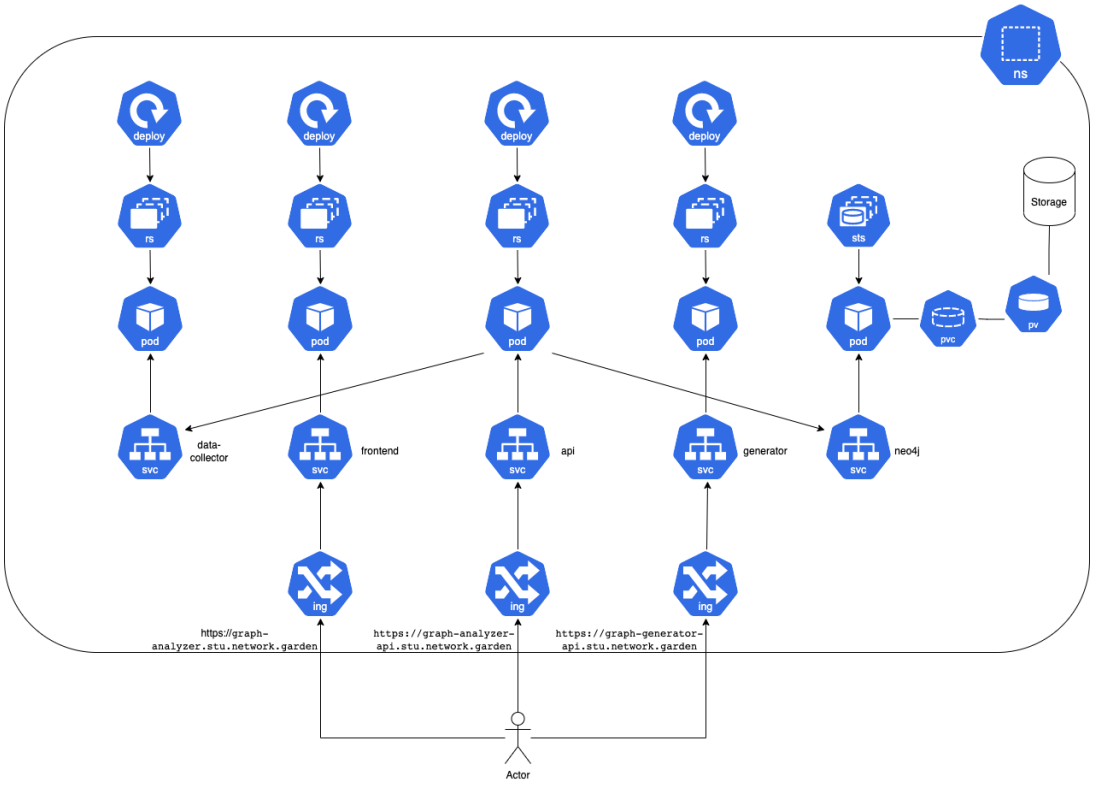


Figure 6.7: Kubernetes deployment

6.3 UI and UX

The UI and UX of the Frontend integration were not required parts of this thesis. Despite that, a basic wireframe was created to provide orientation during the implementation. Figure 6.8 shows the Generator page embedded into the Graph Analyzer Frontend.

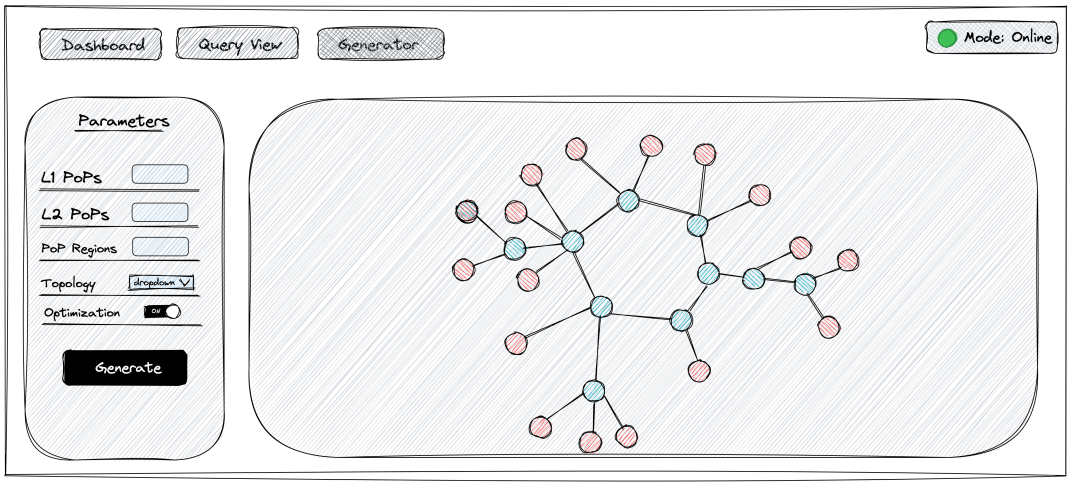


Figure 6.8: Frontend - Generator page wireframe

6.4 Quality Measures

The following sections describe measures taken to achieve a high standard of code quality. Parts have been taken from the previous work done in [6].

6.4.1 Graph Generator Tests

It was decided to only have basic smoke tests covering the API endpoints of the Graph Generator due to time constraints and the questionable advantages of more detailed tests. Due to the nature of the Graph Generator, generated network topologies are random and not predictable. Thus, testing the output is difficult and one could even argue not feasible.

6.4.2 Manual Tests

At the end of the thesis the system is tested manually and the results are documented in a test protocol. The test protocol can be found in table 7.7.

6.4.3 Git Process

It was decided to use a multi-repository approach, which involves splitting the system's source code across multiple repositories.

6.4.4 Workflow

An issue in Jira is present for every task to be done. A corresponding application milestone is attached to the issue. A separate branch corresponding to an issue is created for every task. After the task has been completed, the feature branch is merged into the main branch. The person that is assigned to the issue needs to ensure that the feature branch incorporates the latest changes from the main branch. Only then is the branch considered mergeable. A merge request is then created and a reviewer is assigned. A successful review is indicated by the reviewer's approval message and the feature branch can be merged into the main branch.

6.4.5 Code Review

Merge requests are used in GitLab as mentioned in section 6.4.4. Since the team is only composed of two people, the non-author of the merge request is assigned as the reviewer. The assigned reviewer is then responsible for checking the changes. When all found issues have been resolved, the merge request is approved by the reviewer and merged. GitLab features such as threads and suggestions are used during the review process.

6.4.6 CI/CD

To ensure code quality standards and simplify processes the CI/CD pipeline of GitLab is used. No significant changes were made concerning the existing Graph Analyzer system. Details can be found in the term project [6]. Only the changes made for the Graph Generator are detailed below.

6.4.6.1 Graph Generator

The Graph Generator CI/CD pipeline consists of 4 stages:

- lint
  - black[1] is used to check if the code is formatted correctly.
  - flake8[2] checks the code for errors and styling issues.
  - mypy[3] checks the code for possible static-type errors.
- test
  - Code Quality[4] check from GitLab.

[1] `https://github.com/psf/black`
[2] `https://github.com/PyCQA/flake8`
[3] `https://github.com/python/mypy`
[4] `https://docs.gitlab.com/ee/ci/testing/code_quality.html`

- Static Application Security Testing (SAST)[5] check from GitLab to check for known vulnerabilities.
- Secret Detection[6] from GitLab to check for any leaked credentials in the codebase.
- licensecheck[7] to check if the licenses of the used Python libraries are compatible with the project license.
- coverage[8] to check the test coverage.

- build
  - kaniko[9] is used to build the container image and push it to the GitLab container registry.

- release
  - Depending on whether it is a commit in the main branch or a tag, the correct image tag is set during the release stage.

To keep the code quality as high as possible, the whole pipeline fails when a single job fails. For example, this can be the case if the code is not formatted correctly or a test fails. The stages and the jobs of a successful run can be seen in figure 6.9.

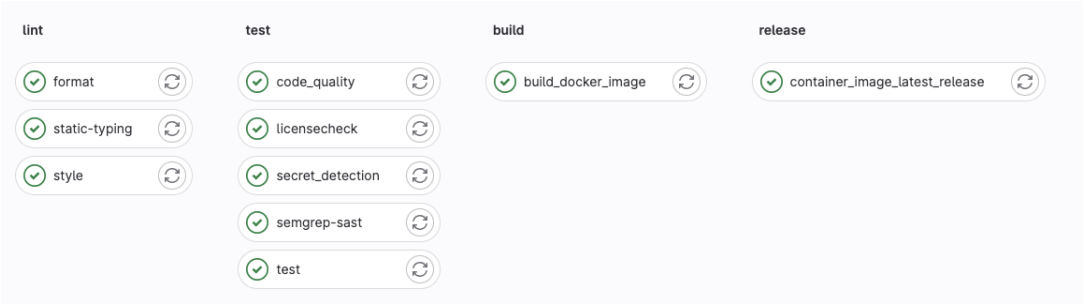


Figure 6.9: GitLab CI/CD

[5] https://docs.gitlab.com/ee/user/application_security/sast/
[6] https://docs.gitlab.com/ee/user/application_security/secret_detection/
[7] https://github.com/FHPythonUtils/LicenseCheck
[8] https://github.com/nedbat/coveragepy
[9] https://github.com/GoogleContainerTools/kaniko

6.5 Metric Tools

The pipeline has been designed to show some code metrics and help the reviewer assess the quality of the code during a review. Figure 6.10 shows an example of a merge request with displayed metrics.

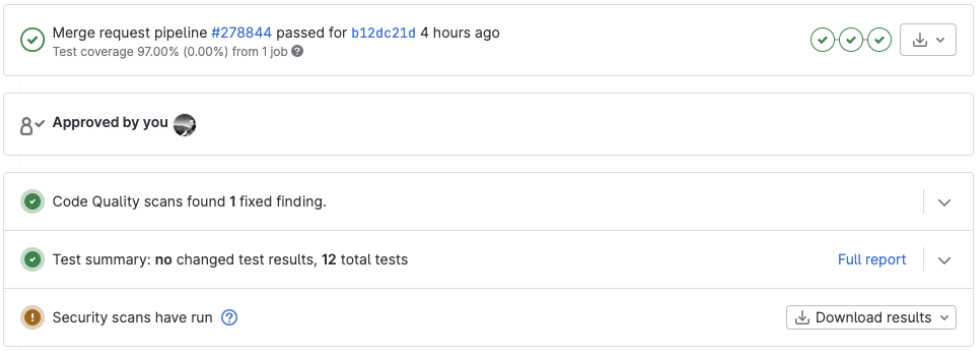


Figure 6.10: GitLab merge request metrics

6.5.1 Test Coverage

As seen in section 6.5, the test coverage is displayed in a merge request. To show whether test cases cover new lines of code the test-coverage visualisation feature from GitLab is used.

6.5.2 SonarQube

Since the SonarQube community edition has limited functionality, no continuous checks are done. The checks are done on a manual basis. An A grade was received for the Graph Generator, as seen in figure 6.11.

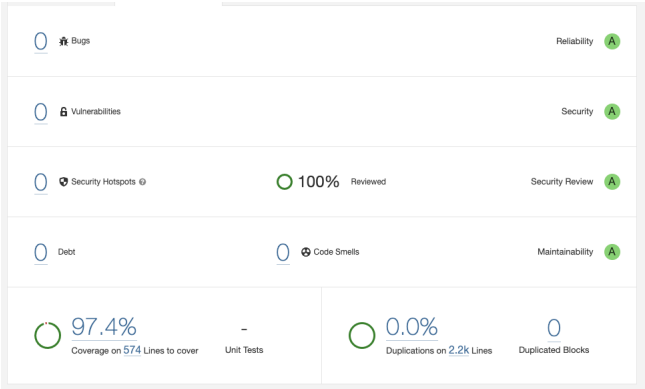


Figure 6.11: SonarQube - Quality Gate status

Chapter 7

Project Management

This chapter shows how the thesis is organised and planned.

7.1 Project Plan

This section provides an overview of how the thesis project was structured and approached.

7.1.1 Development Process

For the project, the Rational Unified Process (RUP) is used for planning in the long term. For short-term planning within iterations, Scrum will be used. RUP splits the project into the following 4 phases:

7.1.2 Phases

1. Inception: Vision, Initial risk assessment, Project description
2. Elaboration: Use-case model, Description of software architecture
3. Construction: Building the software
4. Transition: Rollout, Quality checks

7.1.3 Project Milestones

Progress is tracked via two milestone types, project and application. Project milestones are based on the project phases and show the project's progress. Application milestones are more geared toward the actual development of the system. They track specific parts or features of the to-be-developed system. Seven project milestones are defined for the project. These are present as epics in the issue tracker. Epics are not exclusive to these milestones and can contain issues spanning multiple project phases. Project milestones are present as epics in the issue tracker, but no issues will be assigned to them. They are mainly there to give a better context to the roadmap. Table 7.1 lists all project milestones.

Milestone	**Planned**
M1 - Project Plan	Semester week 3 (09.03.2023)
M2 - Requirements & Research	Semester week 6 (30.03.2023)
M3 - End of Elaboration Phase	Semester week 7 (06.04.2023)
M4 - Architecture	Semester week 8 (13.04.2023)
M5 - Alpha	Semester week 13 (18.05.2023)
M6 - Beta	Semester week 15 (01.06.2023)
M7 - Final Submission	End of semester week 17 (16.06.2023)

Table 7.1: Project milestoness

7.1.4 Application Milestones

There are six application milestones defined. These application milestones also act as epics in the issue tracker and can be mapped to system parts. Table 7.2 lists all application milestones.

Milestone	**Planned**
AM1 - Generator Proof of Concept	23.03.2023
AM2 - Generator Prototype	06.04.2023
AM3 - Generator	18.05.2023
AM4 - Directed and Weighted Support	04.05.2023
AM5 - Generator Integration	18.05.2023
AM6 - Robustness Property	18.05.2023

Table 7.2: Application milestones

7.1.5 Roadmap

The roadmap and schedule, which includes planned sprints and milestones, are presented in figures 7.1 and 7.2.

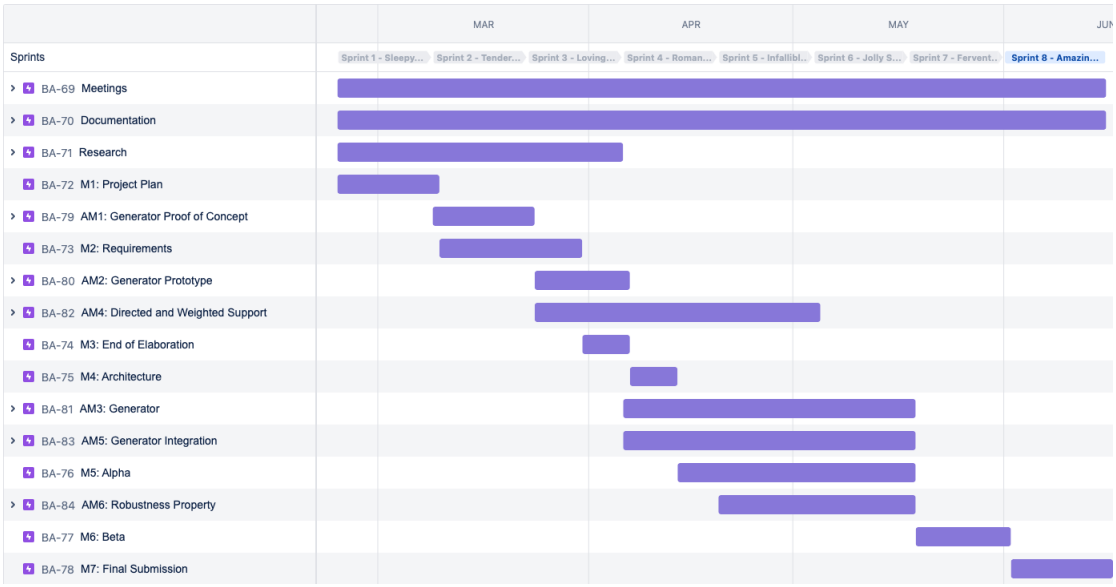


Figure 7.1: Roadmap

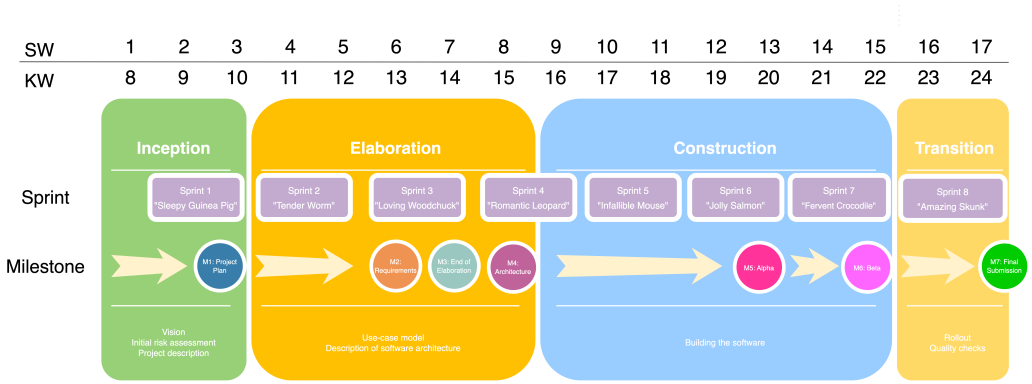


Figure 7.2: RUP phases with project milestones and sprints

7.1.6 Key Dates and Numbers

- Project start: 20.02.2023
- Project end: 16.06.2023 17:00
- Time budget: 720 hours (1 ECTS = 30 hours)
- Working days: Wednesday, Thursday, Saturday, Sunday

7.2 Meetings

7.2.1 Status Meetings

It is planned that there will be weekly meetings with the advisors. They are scheduled for each Thursday afternoon but can be skipped if there is no need for one (for example, in the Construction phase). Physical presence at the meetings is preferred, but remote participation is possible.

7.2.2 Scrum Meetings

A weekly scrum is done instead of a daily one. The reason for this is the different schedules of the two part-time students involved in this project.
In addition, there are previously documented status meetings with the advisors. Table 7.3 lists the planned Scrum meetings.

Meeting	**Frequency**	**When**	**Where**
Sprint Planning	Every two weeks (before every sprint)	Thursday, 13:00 - 17:00	Rapperswil
Sprint Review	Every two weeks (after every sprint)	Thursday, 13:00 - 17:00	Rapperswil
Sprint Retrospective	Every two weeks (after every sprint)	Thursday, 13:00 - 17:00	Rapperswil
Weekly Scrum	Every week	Saturday, 15:00 - 16:00 (Backup: Sunday: 15:00 - 16:00)	Online
Backlog Refinement	Every other week (with Weekly Scrum)	Saturday, 15:00 - 16:00 (Backup: Sunday: 15:00 - 16:00)	Online

Table 7.3: Scrum meetings

7.3 Roles

The role assignment can be seen in table 7.4.

Role	**Member(s)**
Minute Keeper	Pascal Christen
Scrum Master	Lukas Ribi
Product Owner	Pascal Christen
Developer	Lukas Ribi, Pascal Christen
Advisors	Laurent Metzger, Severin Dellsperger

Table 7.4: Scrum roles

7.3.1 Details About The Assigned Roles

- Scrum roles (Scrum Master, Product Owner, Developer) as defined in Scrum
  - Managing the product backlog is the primary responsibility of the product owner. However, the actual work is done collaboratively.
- Minute keeper: Takes notes during the status and team meetings.

7.4 Risk Management

Risks exist and must be recognised and, if possible, mitigated. This section covers both risk detection and mitigation.

7.4.1 Risks

Legend (from Low to High):

- Likelihood: Rare, Unlikely, Possible, Likely, Certain
- Severity: Negligible, Marginal, Critical, Significant, Catastrophic
- Impact: Low, Medium, High, Very High

ID	Topic	Description	Likelihood	Severity	Impact
R01	Knowledge	The team members lack knowledge concerning the technical or theoretical aspects of the project. This grievance can delay phases of the project and lead to missed milestones. Missing theoretical knowledge influences the project, as understanding is crucial for generating service provider networks correctly.	Unlikely	Critical	Medium
R02	Health and Safety	A team member gets sick or is involved in an accident. The member cannot work for a certain period, and the pressure on the other team member increases significantly. This fact can lead to project delays and not implemented features.	Likely	Critical	High
R03	Communication	There is no or insufficient communication inside the team or with the advisors. A misconception could be the result of the lack of communication. Therefore, the final result may not have the desired outcome.	Possible	Critical	Medium

R04	Scope	There are too many mandatory requirements that can not be fulfilled, or the existing ones are too big for the scope of this thesis. This circumstance can lead to not implemented features.	Rare	Marginal	Low
R05	Technology	In the middle of the construction phase, the used technology stack needs to be changed because of technical or personal limitations. The final result is at high risk because of this substantial conceptional change.	Possible	Critical	Medium
R06	External Hazards	There is a non-influenceable outside event happening that prevents work on the project as is. For example, this could be a snowstorm, energy blackout, or even war. Work on the project is impossible, as previously, and the project's final form will vary significantly from the planned goals.	Likely	Critical	High
R07	Technology	The provided ruleset has too few details or is incorrect in generating a telecommunications network.	Possible	Significant	High

Table 7.5: Risk management

7.4.2 Risk Management and Mitigation

Besides the external hazard risk, every mentioned risk in table 7.5, especially those with "medium" and "high" impact according to the risks in table 7.5, can be mitigated to a certain degree by engaging via direct communication, frequent meetings and status updates. The mitigations are listed in table 7.6.

ID	Mitigation / Action
R01	Much time was invested in research during the inception and elaboration phases. We were able to produce a prototype that showed that our goal was possible and that we understood the theory. Therefore, the severity is decreased to marginal.
R02	There is nothing one can do about external hazard risks (besides good planning).
R03	After six weeks of working together, no apparent lack of communication or misunderstandings occurred. Therefore, the likelihood is decreased to rare.
R04	We were able to produce a prototype that showed that our goal was possible and that the set (required) goals were manageable. In this case, the severity decreased to negligible.
R05	Technologies were evaluated with appropriate detail, and it is not expected that anything breaks or does not fulfil defined requirements. Considering that, the severity decreases to marginal and the likelihood to unlikely.
R06	As mentioned, there is nothing one can do about external hazard risks (besides good planning).
R07	The ruleset had to be discussed several times since there were some ambiguities, but these could be eliminated. We must tell our Prof. Laurent Metzger and supervisor Severin about the correctness since we were not provided with real network topologies. However, some references suggest the information is correct. Considering that, the severity decreases to critical.

Table 7.6: Risk mitigation

7.4.3 Risk Matrix

The risks described in table 7.5 and the mitigations from table 7.6 are visualised in figure 7.3.

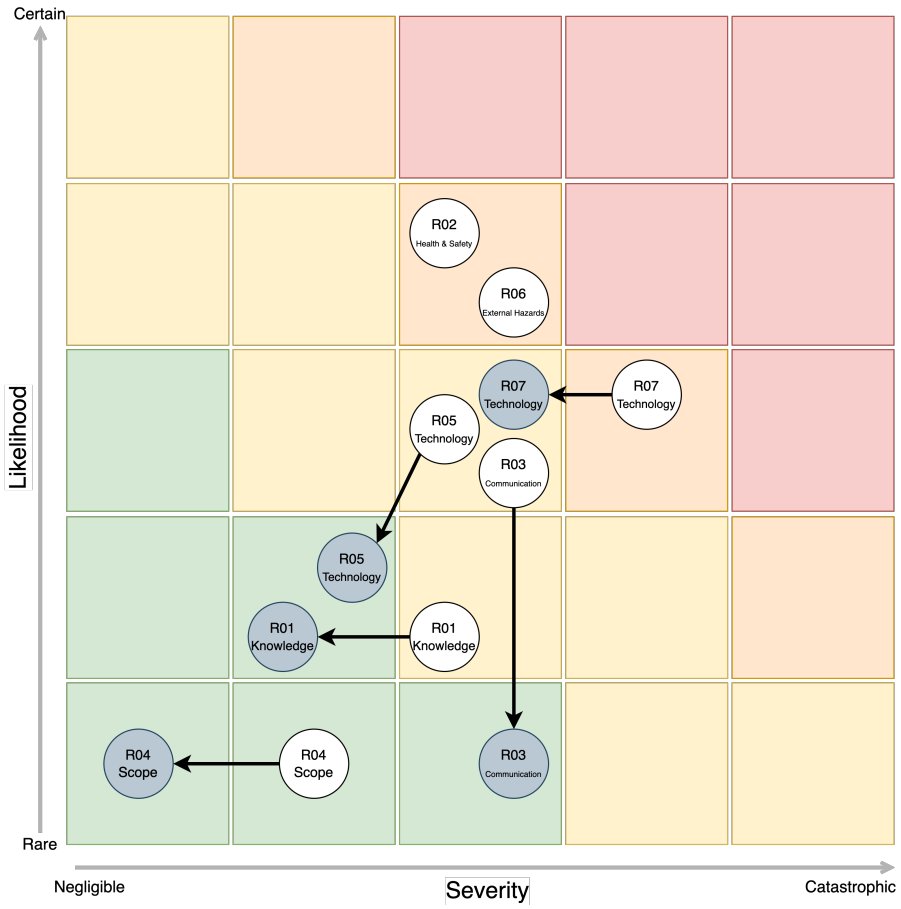


Figure 7.3: Risk matrix

7.5 Planning Tools

7.5.1 Issue Tracker

Jira is used to track issues and handle project management.

7.5.2 Time Tracker

Time tracking is done directly in Jira on tickets. Every team member is responsible for tracking his time on the correct ticket.

Part III

Appendix

Test Protocol

ID	Description	Expected result	Actual result	Status
Frontend - Dashboard				
1	Data Collector in GEXF mode: Test if "Upload GEXF" button is visible.	Visible	Visible	Pass
2	Data Collector in JAGW mode: Test if "Upload GEXF" button is disabled.	Disabled	Disabled	Pass
3	Data Collector in GEXF mode: Test if the upload of a valid GEXF is successful by using a generated topology.	Green banner: File uploaded successfully!	Green banner: File uploaded successfully!	Pass
4	Data Collector in GEXF mode: Test if the upload fails when uploading an invalid GEXF topology.	Red banner: Error uploading file!	Red banner: Error uploading file!	Pass
5	Graph Generator deployed: Test if the "Generator" button is visible.	Visible	Visible	Pass
6	Graph Generator not deployed: Test if the "Generator" button is not visible.	Not visible	Not visible	Pass
7	Diameter propertry is shown.	Two values are visible	Two values are visible	Pass
8	Diameter propertry description is shown after a click on "learn more".	Description is shown	Description is shown	Pass
9	Robustness property is shown.	Two values are visible	Two values are visible	Pass
10	Robustness property description is shown after a click on "learn more".	Description is shown	Description is shown	Pass
Frontend - Generator				
11	Fields are validated when inserting characters in a numeric field.	Error: [Field][1] must be a number	Error: [Field] must be a number	Pass
12	Error is thrown when more PoP regions than the sum of L1 and L2 are chosen.	Error: Amount of PoP regions can not be higher than total amount of PoPs.	Error: Amount of PoP regions can not be higher than total amount of PoPs.	Pass

[1]Multiple fields included in this test

13	Error is thrown when more random edges are chosen than a full graph can have.	Error: Number of random_core_node_connections is too high, would exceed a full mesh with XY edges: XXX YY (current core node edges) XY.	Error: Number of random_core_node_connections is too high, would exceed a full mesh with 36 edges: 500 13 (current core node edges) 36.	Pass
14	Topology gets generated successfully when the input is correct.	Preview and green banner are shown: If you like the generated network, download it here	Preview and green banner are shown: If you like the generated network, download it here	Pass
15	Generated topology can be downloaded.	export.gexf file is saved to Downloads folder	export.gexf file is saved to Downloads folder	Pass

Table 7.7: Test protocol