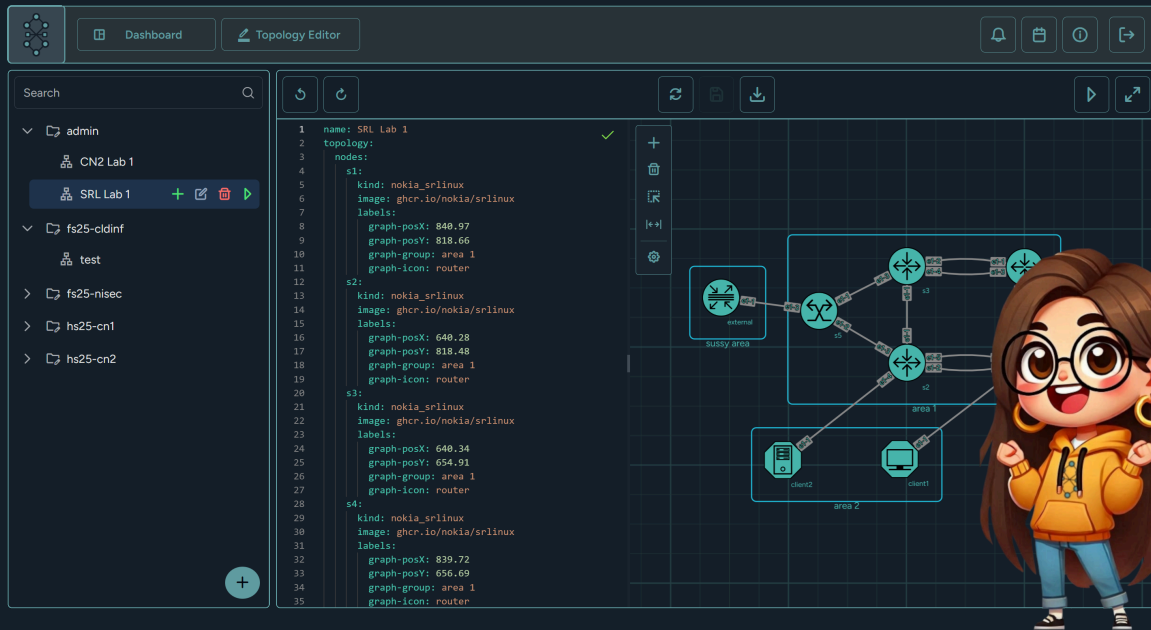


# Antimony

A visual approach to designing and deploying  
Containerlab networks.



Kian Gribi  
Tom Stromer

2025-06-13

Reviewed by Prof. Dr. Olaf Zimmermann  
Supervised by Jan Untersander and Urs Baumann

# 0

## Contents

1. Abstract .....	3
2. Definition of Terms .....	4
3. Introduction .....	6
3.1. Key Objectives .....	6
3.2. Vision .....	6
4. Results .....	8
5. Conclusion .....	9
6. Product Documentation .....	10
7. Project Documentation .....	11
7.1. Preliminary Work .....	11
7.1.1. Improvements in the Interface .....	11
7.2. Functional Requirements .....	12
7.3. Non-Functional Requirements .....	14
7.4. Project Management .....	17
7.5. Version Control Setup .....	18
7.6. Meetings and Reviews .....	18
7.7. Risk Management .....	19
7.7.1. Risk 1 - Complexity of Golang .....	19
7.7.2. Risk 2 - Time Management .....	20
7.7.3. Risk 3 - Container Runtime Variability .....	21
7.7.4. Risk 4 - Migration from Vis.js to Cytoscape .....	22
7.8. Design Decisions .....	24
7.8.1. Switching From the LTB to Containerlab .....	24
7.8.2. GoLang as Main Language .....	24
7.8.3. Object Persistence - GORM .....	25
7.8.4. Web Server - GIN .....	25
7.8.5. Real-time Communication - Socket.IO .....	26
7.8.5.1. Why not gRPC? .....	26
7.8.5.2. Why Socket.IO? .....	26
7.8.5.3. The Socket.IO Library .....	27
7.8.6. Migration from Vis.js to Cytoscape .....	27
7.8.7. Domain-Driven-Hexagon Architecture [1] .....	28
7.8.8. Authentication .....	28
7.8.9. Containerlab Interface .....	29
7.8.10. Handling of Topology Metadata .....	30
7.8.11. Connecting to Containers .....	30
7.9. Difficulties .....	31

7.9.1.	Unfamiliar Environment .....	31
7.9.2.	Talking to Containerlab .....	31
7.9.3.	Local Containerized Keycloak Deployment .....	32
7.9.4.	Clabernetes Integration .....	32
7.9.5.	Working With Windows Subsystem for Linux .....	33
8.	Quality Assurance .....	34
8.1.	Code Linters .....	34
8.2.	Automated Testing .....	34
8.2.1.	Code Tests - Testify .....	35
8.2.1.1.	API Tests .....	35
8.2.1.2.	Unit Tests .....	35
8.3.	Manual Testing .....	35
8.3.1.	Usability Tests (UX) .....	36
9.	Bibliography .....	37
10.	Appendix .....	39
10.1.	Non-Functional Protocols .....	39
10.1.1.	License Compliance .....	43
10.2.	List Of Tools .....	49

# Abstract

Containerlab is a powerful framework for container-based network emulation but lacks a user-friendly graphical interface and fine-grained access management. This makes it less accessible to users unfamiliar with CLI-based workflows and less suitable for usage in large-scale lab environments. This thesis presents the completion of Antimony, a tool that addresses these very problems by providing a server that communicates with the Containerlab tool chain and a user-friendly interface for designing, deploying and maintaining network topologies.

The goal of this thesis is to develop a platform that simplifies the integration of Containerlab into educational lab environments. A user-interface makes it easier for students and teachers to understand networking concepts and design networks through visual topology management.

Building on our previous thesis, whose goal was to develop an initial interface prototype, this work focuses on finalizing that prototype and designing a robust server that acts as the binding between the interface and Containerlab. By developing our own server, we are able to implement features such as log streaming and fine-grained access management.

The resulting product is a user-friendly platform that can be deployed locally for personal testing, as well as in large-scale educational environments. Thanks to our flexible authentication scheme, it is possible to seamlessly integrate with existing university infrastructure such as Azure AD or other OpenID providers.

# Definition of Terms

Term	Definition
Antimony	Antimony is a chemical element. It has the symbol SB and the atomic number 51. A lustrous grey metal or metalloid, it is found in nature mainly as the sulfide mineral stibnite. Antimony compounds have been known since ancient times and were powdered for use as medicine and cosmetics.
Containerlab <sup>1</sup>	Containerlab is a CLI-based toolkit to emulate network topologies by deploying their nodes as containers.
Clabernetes <sup>2</sup>	Clabernetes is another CLI-based tool that deploys Containerlab topologies into Kubernetes clusters.
Topology	We define a topology as the deployable representation of a Containerlab / Clabernetes configuration. Topologies can be created and edited in Antimony's editor.
Lab	A lab is defined as a deployed instance of a topology. A lab can be deployed, rescheduled and stopped in Antimony's dashboard.
GoLang <sup>3</sup>	The programming language used in the Antimony backend
OIDC (OpenID Connect) <sup>4</sup>	An open standard for user authentication, built on OAuth2. Enables Antimony to integrate with institutional identity providers.
OST	The abbreviation for "Fachhochschule OST – Ostschweizer Fachhochschule", a university of applied sciences in Eastern Switzerland.

<sup>1</sup><https://containerlab.dev/>

<sup>2</sup><https://containerlab.dev/manual/clabernetes/>

<sup>3</sup><https://go.dev/>

<sup>4</sup><https://openid.net/developers/how-connect-works/>

LTB	The Lab Topology Builder, the predecessor of Antimony, was used for teaching students about networks and was based on an in-house solution instead of Containerlab.
WSL	Windows Subsystem for Linux (WSL) is a virtualized Linux environment that runs on Windows operating systems, allowing users to run Linux command-line tools and applications natively within Windows.
Keycloak <sup>5</sup>	An open-source identity and access management system used as the OpenID Connect (OIDC) provider in the Antimony project.

*Table 1: Defintion of Terms*

---

<sup>5</sup><https://www.keycloak.org/>

# Introduction

The currently used tool for network emulation at OST is the Lab Topology Builder, an in-house, all-in-one solution for deploying and managing virtual network labs. In line with OST's goal of migrating to a Containerlab-based environment, a major challenge remains; there is no graphical interface available for the Containerlab tool chain. This gap is what Antimony aims to fill.

In our previous thesis from last semester [2], the foundation of Antimony was laid by developing an interface prototype which provided a more user-friendly way to interact with Containerlab. However, that prototype only addressed the interface part of Antimony and was nowhere near being able to be used in production environments.

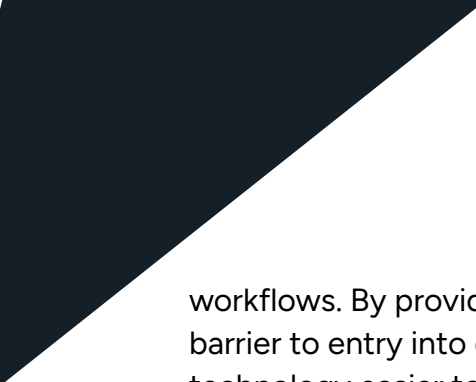
In this bachelors' thesis, we aim to transition from this prototype to a fully-fledged platform that is ready to be used in production. This includes developing a robust server application that integrates with Containerlab and associated technologies as well as improving and refactoring the existing UI, thus transforming Antimony into a comprehensive network emulation platform. The final product is primarily intended for use within OST, but is also designed to serve the broader Containerlab community.

## 3.1. Key Objectives

- Complementing Antimony's UI with a backend to complete the Antimony platform.
- Implementing access and user management, including authentication via OpenID Connect.
- Providing an easy deployment mechanism to simplify local and large-scale deployment as much as possible.
- Ensuring good documentation so that the open source community can use Antimony and contribute to it in the future.

## 3.2. Vision

Designing network topologies for lab environments should be accessible and intuitive for everyone, from students just beginning their journey into the world of networking, to experienced professionals seeking simplicity in their day-to-day



workflows. By providing a graphical interface for Containerlab, Antimony lowers the barrier to entry into container-based network emulation and makes this powerful technology easier to understand and use.

Antimony aims to bridge the gap between usability and capability. It is designed not only as a teaching aid within OST, but also as a practical tool for the broader Containerlab community. The goal is to create a well-structured, reliable, and extensible platform that simplifies the creation, management, and orchestration of network topologies.

Ultimately, Antimony strives to become a valuable resource in the field of network emulation and a go-to tool for anyone working with Containerlab environments.



# 4

## Results

The result of this work is a tool that provides all essential features required to create, build, explore, and understand network topologies. Antimony is designed for flexible deployment, it can be easily run locally by a single user or installed on a server for collaborative use by teams, such as instructors and students. The system supports multiple authentication strategies, including OpenID Connect (OIDC) for integration into institutional environments, as well as a local user management option for standalone usage.

The backend serves as the central middleware that connects the Antimony frontend with the underlying network emulation framework. Antimony integrates with Containerlab, enabling users to deploy container-based network labs in a straightforward manner. Communication with Containerlab is handled through command-line interfaces, allowing Antimony to orchestrate topology deployment, lifecycle management, and status retrieval.

Although the graphical frontend existed as a prototype prior to this thesis, it has been significantly improved and extended. Key enhancements include a more intuitive and powerful Node Editor, support for editing bind files, and functionality for importing pre-existing topologies directly from Git repositories.

# 5

## Conclusion

In conclusion, we were able to deliver a production-ready tool that meets the teaching requirements of OST, which was our primary goal. Additionally, Antimony provides a valuable contribution to the broader Containerlab community by simplifying the management and visualization of topology networks through a graphical interface.

Throughout the development process, we encountered several challenges, such as implementing OpenID Connect (OIDC), building a backend in Go without prior experience, and creating a middleware layer to interface with both Containerlab and Clabernetes. Given our limited experience, we focused on working with well-established frameworks, learning from reputable templates, and regularly seeking guidance from our thesis supervisor and the Containerlab community.

While Antimony is now usable in both academic and broader contexts, there is still room for growth. From the beginning, the project was intended to be open source, not only to ensure long-term maintenance but also to allow external contributors to build on our work and introduce new features. Certain planned features such as a command-line interface for interacting with individual nodes, or full Git integration for both importing and saving topologies, could not be completed due to time constraints.

This project offered a steep learning curve, but it provided valuable experience in full-stack development and practical problem solving. We learned how to structure and deliver a real-world software tool, how to work with unfamiliar technologies and how to communicate effectively within a development team. These skills will continue to benefit us in future technical and collaborative endeavors.

# Product Documentation

While this thesis serves as the primary documentation for the general planning, methodology, and design decisions behind Antimony, the actual product documentation, including the system architecture and implementation details is maintained separately in the project's Git repository.

This separation ensures that end users and contributors can directly access up-to-date technical documentation that is aligned with the current state of the project. Meanwhile, this thesis remains focused on explaining how the project evolved, why certain design decisions were made, and what processes were followed.

The product documentation includes:

- Architecture of Antimony
- Setup and deployment instructions (local and server-based)
- User guides for interacting with the graphical interface
- Configuration and authentication options
- Contribution and development guidelines

For full documentation, please refer to the repository:

<https://antimony-team.github.io/antimony>

# Project Documentation

The purpose of this chapter is to document the key aspects of our development process, design decisions, and how we addressed challenges throughout the project. Furthermore we will outline our progress, the functional and non-functional requirements we defined, and the tools and practices we used to manage and deliver the final product.

## 7.1. Preliminary Work

Prior to the start of this thesis, a functional interface prototype for Antimony had already been developed as part of the semester thesis we wrote last semester [2]. This prototype included a graphical user interface, covering most of the features of what we thought the final product would look like back then. Needless to say, this initial scope of features was expanded greatly as we started with the planning of the bachelor's thesis.

### 7.1.1. Improvements in the Interface

This thesis' focus was primarily on implementing the Antimony server, finalizing the platform as a whole and making it ready for large-scale production use. But to do so, we identified several features in Antimony's interface that needed to either be improved or added entirely.

In addition to this, we also tried to incorporate a lot of the feedback we collected from surveys and user experience tests from last semester's thesis.

The most significant and important improvements in the interface included:

- **Cytoscape Migration:** Migrating from our old visualization framework Vis.JS<sup>6</sup> to Cytoscape.JS<sup>7</sup>. This change allowed us to overcome some limitations of Vis.JS and implement more advanced visualization features such as the grouping of nodes. More information about this change can be found in Section 7.8.6.
- **Integrated Terminal:** We added a new dialog that acts as a terminal emulator for students to access the deployed nodes.
- **Enhanced Lab Dialog:** We redesigned and extended the lab dialog that displays information about a currently running lab by adding the aforementioned

---

<sup>6</sup><https://visjs.org/>

<sup>7</sup><https://cytoscape.org/>

integrated web terminal, a new sub-dialog for log streaming and some additional overlays to display runtime information.

- **Sync Integration:** Suggested by Containerlab's lead developer Roman Dodin<sup>8</sup>, we added a feature that allows users to sync a lab topology with an external resource such as a topology stored within a Git repository.

In addition to these major improvements, we performed general refactoring of the interface's code base. This refactoring included but is not limited to improving the performance of the topology editor and the interface as a whole, completely re-implementing authentication handling and resource management, updating the styling to match the new features and introducing various small usability enhancements.

## 7.2. Functional Requirements

In order to have a clear overview of the scope and the features required in this project, we have defined a list of functional requirements.

Due to our agile approach to project management, we decided to describe all our functional requirements as user stories. The stories will be written in the perspectives of Zoey, a student at OST and Jeff, a lab instructor working at the institute for networking at OST. Requirements prefixed with *[R]* contribute to our Definition of Done and are treated as mandatory, while others are considered optional.

---

<sup>8</sup><https://www.nokia.com/people/roman-dodin/>



### **Zoey the Antimony Girl**

Zoey is a student at OST learning about Cluster Networks in the CN2 (Computer Networks 2) module.



### **Jeff**

Jeff is a lecturer who oversees lab Zoey's and the other students exercises.

- [R] Jeff can easily deploy Antimony in his infrastructure.
- [R] Jeff can connect Antimony with his existing OpenID provider<sup>9</sup>.
- [R] Jeff is able to deploy a topology definition.
- [R] Jeff is able to redeploy a running lab.
- [R] Jeff is able to destroy a running lab.
- [R] Jeff is able to force-deploy a lab that is currently scheduled.
- [R] Jeff can now group the nodes in his topology and create network areas.
- [R] Jeff can manage what topologies and labs Zoey has access to.
- Zoey can look at the deployed lab and see the node's IPs.
- Zoey is able to open an interactive terminal to the lab.
- Jeff and Zoey can look at the deployed lab and each node's logs.
- Jeff can also use Clabernetes<sup>10</sup> as provider and deploy labs on K8S<sup>11</sup>.

---

<sup>9</sup><https://openid.net/>

<sup>10</sup><https://containerlab.dev/manual/clabernetes/>

## 7.3. Non-Functional Requirements

We defined our non-functional requirements (NFRs) based on the ISO 25010[3] standard, which specifies the nine most important types of NFRs to ensure software product quality. We didn't consider all of them due to the scope of this project. All the categories we did consider are marked in bold.

Supporting documents such as testing protocols and license summaries can be found in the appendix under Section 10.1.

- **Functional Suitability**: The range of functionality is covered (Covered in FRs).
- **Performance Efficiency**: The application is performant and runs smoothly.
- **Compatibility**: The application is cross-browser compatible.
- Interaction Capability
- Reliability
- Security
- **Maintainability**: The application is written in a modular and scalable manner.
- Flexibility
- Safety

<b>ID</b>	NFR-1
<b>Category</b>	Functional Suitability
<b>Description</b>	The system must ensure that all required functionalities (i.e. user stories prefixed with [R]) are implemented correctly and perform reliably to support all of our user's workflows.
<b>Measure</b>	All supported functionalities will be verified through manual acceptance testing. Each required feature and workflow will be tested against predefined test cases, and results will be documented in a formal test protocol to confirm functional completeness and correctness.
<b>Importance</b>	<b>High</b>

Table 2: NFR-1

---

<sup>11</sup><https://kubernetes.io/>

<b>ID</b>	NFR-2
<b>Category</b>	Performance Efficiency
<b>Description</b>	The backend should maintain high responsiveness and efficient resource utilization under expected loads. There should be no long loading screens or passing of unnecessary data.
<b>Measure</b>	<p>Performance will be measured manually in various modern browsers.</p> <p>Production builds will be used to test performance, as those are the ones users will encounter.</p> <p>None of the loading times should exceed 1 second in a lab environment where server latency can be neglected.</p>
<b>Importance</b>	<b>High</b>

*Table 3: NFR-2*

<b>ID</b>	NFR-3
<b>Category</b>	Compatibility
<b>Description</b>	Antimony should be available and function seamlessly across all modern web browsers (e.g., Chrome, Firefox, Edge, and Safari).
<b>Measure</b>	<p>The UI will be manually tested on each required browser after completing all functional requirements.</p> <p>If the displayed interface is deemed adequate, the test is passed.</p> <p>Babel is used to ensure compatibility with older browsers.</p>
<b>Importance</b>	<b>Moderate</b>

*Table 4: NFR-3*



<b>ID</b>	NFR-4
<b>Category</b>	Interaction Capability
<b>Description</b>	The web UI must have an intuitive and user-friendly interface to allow users to navigate through lab management tasks without requiring knowledge of the Containerlab or Clabernetes CLI commands. The basic interaction design of the UI was validated in a previous thesis through structured usability tests. In this thesis, we preserved and extended this existing design.
<b>Measure</b>	No new formal usability tests will be conducted in this thesis, as the UI design builds upon a previously tested version.  Instead, we ensured consistency with the validated interaction patterns, and conducted informal tests during development to verify the usability of newly added features.
<b>Importance</b>	<b>High</b>

*Table 5: NFR-4*

<b>ID</b>	NFR-5
<b>Category</b>	Maintainability
<b>Description</b>	The code-base should follow best practices for code quality to ensure future developers can maintain and extend the product efficiency.
<b>Measure</b>	Code quality will be assessed using Google Chrome's Lighthouse tool. Additionally, code reviews will be conducted with Jan Untersander to confirm the maintainability of the backend.
<b>Importance</b>	<b>High</b>

*Table 6: NFR-5*

<b>ID</b>	NFR-6
<b>Category</b>	License Compliance (Non-standard)
<b>Description</b>	The project must use only free, open-source libraries with licenses compatible with the MIT license, under which Antimony will be distributed.
<b>Measure</b>	A license review will be conducted for each third-party dependency to ensure compliance with the project's licensing strategy.
<b>Importance</b>	<b>High</b>

Table 7: NFR-6

## 7.4. Project Management

For this project, we decided to use the GitHub project organization tools. This decision was made mainly due to the fact that our previous experience with tools with the single use of providing project management was that they carry too much overhead for a small team like ours. Our main goal was to distribute the work into multiple milestones to reach after certain time periods to have an incremental working prototype.

As illustrated in Figure 4, we first focused on developing a working server prototype and then gradually added features and improvements to the existing interface prototype until both of them were working together properly. After that, we started focusing on the smaller and optional features as well as QoL and performance improvements.

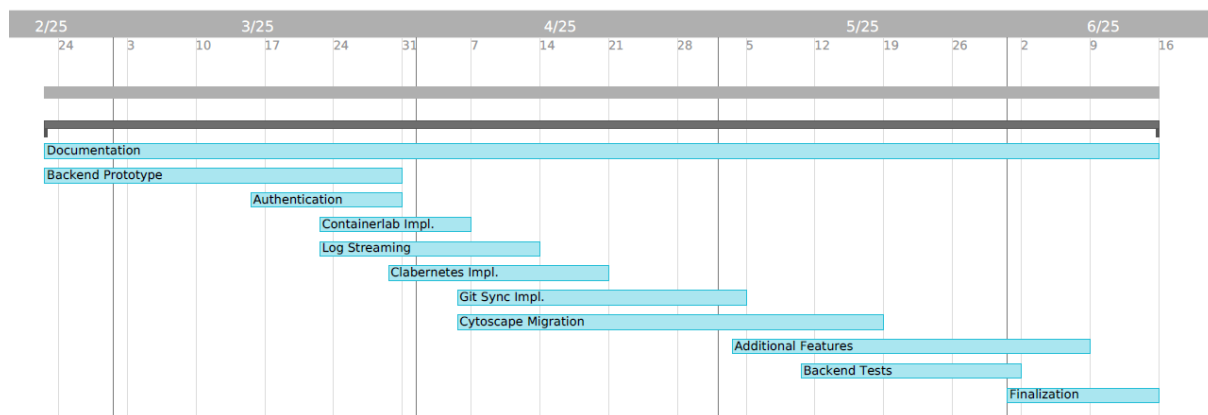


Figure 4: Project plan of Antimony

## 7.5. Version Control Setup

Since Antimony was going to be published as an open source project with the intention that OST and maybe even the Containerlab community continues to maintain it in the future, it was very important to use a clear and scalable repository structure. We wanted to ensure that any technically proficient user or contributor could quickly understand the project layout and start contributing with minimal setup and research effort.

One important decision was whether to use a mono-repo (single repository) or multi-repo (multiple repositories) structure. Given the clear separation between the project's services, we decided to utilize a multi-repository structure within the Antimony Team GitHub organization, as illustrated in Figure 5.

The Antimony repositories are organized as follows.

- **antimony:** The main repository, containing the project documentation, deployment and configuration files. It serves as the entry point for new users who want to start setting up or contributing to Antimony.
- **antimony-backend:** Contains the Antimony server logic, written in Golang.
- **antimony-interface:** Contains the Antimony interface written in TypeScript and React.

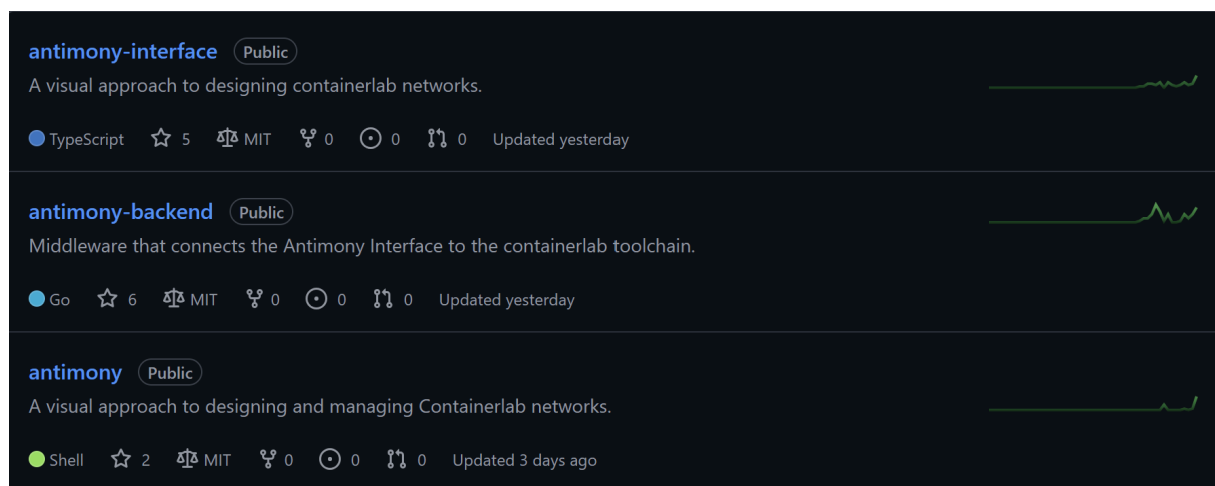


Figure 5: The Antimony Repository

## 7.6. Meetings and Reviews

Throughout the project, we held weekly meetings with our supervisors. These sessions served not only to provide clear visibility into our progress, but also to ensure continuous feedback on the features that we worked on. Additionally, they

offered an opportunity for us to share suggestions on potential enhancements, as well as raising potential concerns.

## 7.7. Risk Management

Our risks are listed below. Each risk definition starts with a brief description of the problem, followed by an estimation of the probability of the risk occurring, as well as the impact that the occurrence would have. And lastly, our plan to mitigate these risks.

Risk severity is evaluated using a scoring system that rates each risk with a score between 1 and 10, where 1 is the lowest possible score and 10 the highest. For readability reasons, we used percentages to display the scores.

### 7.7.1. Risk 1 - Complexity of Golang

Since neither of us have had any experience with the programming language Go, there is a moderately large risk that we can't properly assess the difficulty of certain tasks. From a distant inspection of Go, it didn't seem to be of extreme complexity, considering we have had exposure to programming languages with similar syntax and paradigms. Still, since a major part of our project was going to be written in Go, it was still a large risk factor.

**Probability (40%):** We were expecting to run into at least into sort of problems, since we were starting from scratch.

**Severity (6/10):** From a short inspection we already found close relations to other program languages such as python, thus we don't expect problems that could endanger the entirety of the project.

**Mitigation:** To mitigate this risk, we thoroughly researched common challenges in Go and familiarized ourselves with the best practices early on in the project.

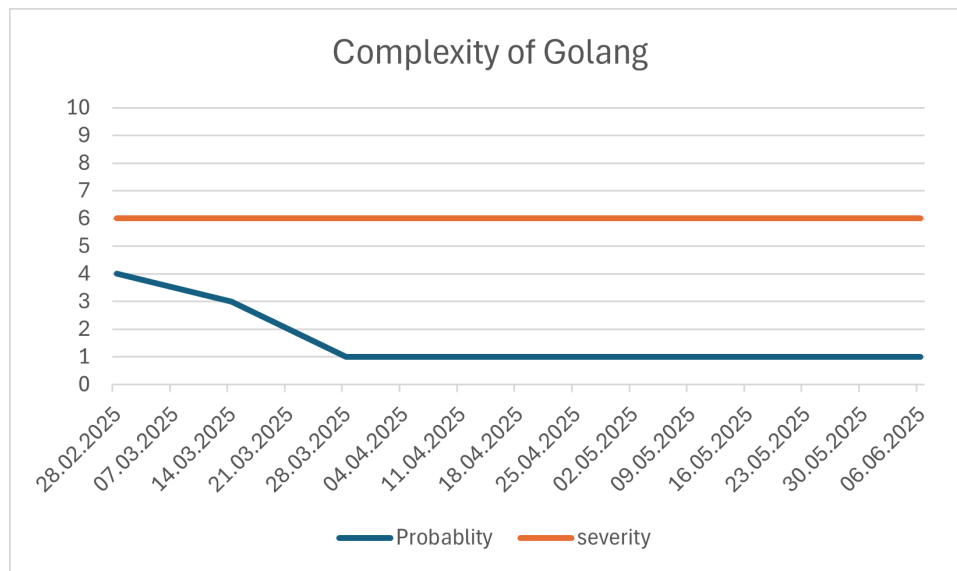


Figure 6: Risk complexity of Golang over the course of the project

**Conclusion:** Even though we were told that Golang has a low entry barrier for beginners, we initially viewed the lack of prior experience with the language as a significant risk. In hindsight, this concern proved to be somewhat overstated. We encountered specific challenges, most notably the absence of a structured concurrency model, which required us to manage concurrency manually. However, the transition into Go development was smoother than anticipated. As shown in Figure 6, the probability of this risk decreased sharply after the first few weeks. Once we had completed the core implementation of the prototype server, the learning curve had flattened and the complexity of Golang no longer posed a major threat to the project's progress.

## 7.7.2. Risk 2 - Time Management

Since this project had a clear deadline and the end goal was a working product that could be used in production, it was important to keep a close eye on the remaining time throughout the project. There was a chance that something unexpected could happen which would set us back on our schedule, especially considering other risks.

**Probability (40%):** We already partially ran into this risk during the development of the Antimony prototype last semester, so it's definitely a risk to keep in mind and regularly reflect on.

**Severity (8/10):** The occurrence of this risk could have been very impactful, considering that we might not have been able to implement some of the required and critical features if enough time was lost with debugging or other non-productive tasks.

**Mitigation:** Our mitigation strategy for this risk was to define the project scope in an agile manner and focus on the most important features first. Once these features were implemented and properly tested, we moved on to the next batch of less important features.

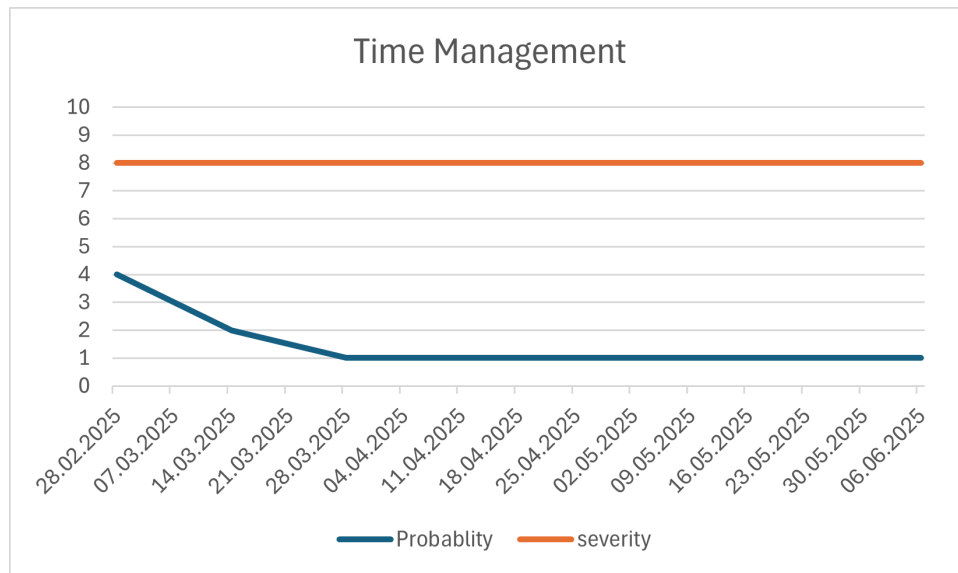


Figure 7: Risk Time Management over the course of the project

**Conclusion:** This risk was well justified and rated accurately, as the overall workload turned out to be higher than initially expected. This was especially true because, again, the finished product was meant to work in real-world production. This added pressure to clean up and refactor significant parts of the code-base to meet production-level standards. In hindsight, our decision to start early and prioritize a fast initial prototype as seen in Figure 7 was crucial to the successful completion of the project. It gave us the necessary buffer to iterate, polish, and stabilize the application in time.

### 7.7.3. Risk 3 - Container Runtime Variability

Differences between the development environment (e.g., Docker Desktop on Windows using WSL2) and the target production environment (typically native Linux containers) can lead to compatibility issues. These discrepancies may result in unexpected behavior, especially when relying on container networking, volume mounts, or system level integrations that behave differently across platforms.

**Probability (20%):** This risk is difficult to quantify precisely, but we estimate a probability of around 20%. While modern container tooling reduces some inconsistencies, edge cases and runtime specific behavior still pose a threat.

**Severity (10/10):** The severity of this risk is considered critical. If the application fails to run or function correctly in a native Linux container environment its intended

production setting the core functionality of the Antimony tool would be compromised, potentially rendering the entire system unusable.

**Mitigation:** To reduce the likelihood and impact of this risk, we followed several mitigation strategies:

- Containerization: Ensuring that the application itself is fully containerized with docker to increase portability and reduce dependency on the host system.
- Automated Testing: Unit tests and integration tests will be implemented to verify the application's core functionality across environments.
- Environment Validation: We will continuously test the application in its target (native Linux) environment during the development process. This allows early detection of runtime specific issues and ensures compatibility before deployment.

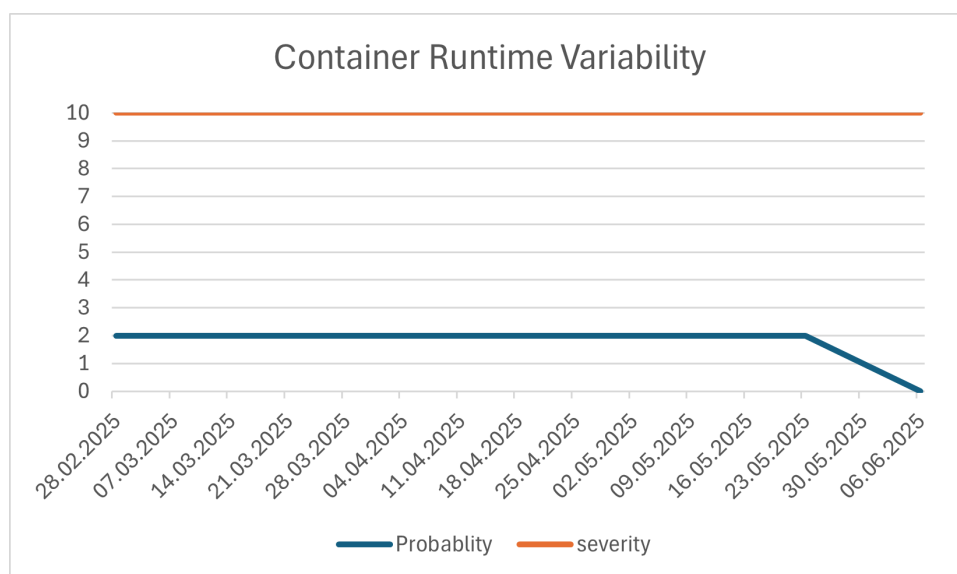


Figure 8: Risk Container Runtime Variability over the course of the project

**Conclusion:** Recognizing this risk early proved to be the right decision. During development, we encountered numerous issues that were dependent on differences between operating systems and on Docker behavior—compounded by the fact that Containerlab itself relies on Docker. To shield Antimony users from these complexities and inconsistencies, we made the deliberate choice to fully containerize the Antimony application. This step was essential to ensure portability, reliability, and ease of use across diverse environments. Since we only Containerized the application at the end of this project we couldn't reduce the probability of this risk until the end as seen in Figure 8

#### 7.7.4. Risk 4 - Migration from Vis.js to Cytoscape

Due to limitations in Vis.js which prevents us from implementing certain required features for network cluster visualization, we have decided to refactor this part of

the frontend and migrate to Cytoscape.js. This transition poses several risks, as the visualization component is a core and deeply integrated part of the Antimony frontend.

**Probability (60%):** There is a significant probability that we will encounter issues during the migration process. These could include compatibility problems, unexpected behavior in the visualization logic, or difficulty in replicating existing features using Cytoscape.js.

**Severity(3/10):** The potential impact of this risk is moderate. The primary concern is the loss of valuable development time, which is limited in the scope of our bachelor thesis. However, the migration is unlikely to cause permanent damage to the system, and no critical data loss or core functionality failure is expected.

**Mitigation:** To minimize this risk, we will maintain a functional backup of the current Vis.js implementation. This allows us to fall back to a stable version if the Cytoscape migration encounters critical issues or cannot be completed in time. Additionally, we will approach the migration incrementally, testing core features step-by-step to ensure that progress is continuously validated.

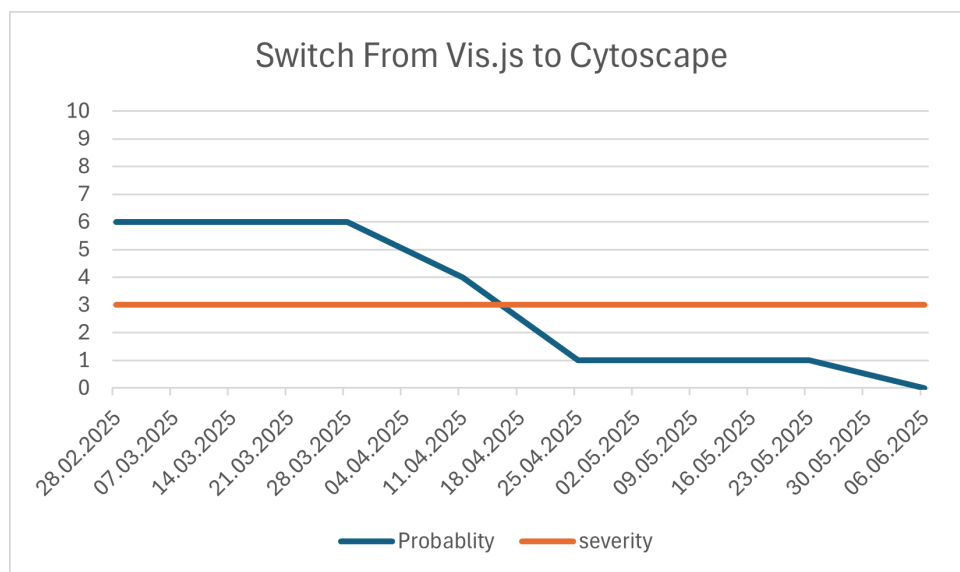


Figure 9: Risk Migration from Vis.js to Cytoscape over the course of the project

**Conclusion:** Including the migration to Cytoscape.js as a risk in our planning proved to be the right decision. While the migration was ultimately successful, it was time intensive and introduced a number of unexpected issues and bugs that required significant attention. Fortunately, our rapid progress during the first seven weeks of the project allowed us to begin the migration early, as seen in Figure 9 we could reduce the risk majorly. This early start gave us the necessary buffer to not only fix the resulting issues but, more importantly, to identify them in the first place through



continued development and testing. This proactive approach likely spared us from significant time pressure later in the project.

## 7.8. Design Decisions

In this chapter, we will take a closer look at the technological and architectural decisions we made during the development of the project. We are going to explain why we chose certain tools and libraries, how they helped us tackle specific challenges, and how they fit both the goals of the project and our own experience as developers.

### 7.8.1. Switching From the LTB to Containerlab

Since there is a predecessor to Antimony, the Lab Topology Builder (LTB), a question naturally arises: why switch to Containerlab as the foundation for network topology emulation in the first place?

There are several important reasons why. While LTB was developed in-house at OST and has served well in past courses, it also lacks maintenance by OST staff. In contrast, Containerlab is an actively maintained, widely adopted open source tool, used in both academia and industry. By building on top of on Containerlab, we leverage a broader ecosystem of tools, documentation, and community support, reducing the long-term maintenance burden on OST.

### 7.8.2. GoLang as Main Language

There was a big decision to make before we even started the project. Do we try ourselves on GoLang, a new programming language, we both had never used before, or do we stay with a more familiar language, like Python for the implementation of the Antimony server? Since this decision had a major impact on our whole project as well as on the complexity of the project, we made an evaluation as seen in Table 8.

Language	Advantages	Disadvantages
GoLang	<ul style="list-style-type: none"><li>• Containerlab is written in GoLang</li><li>• Excellent performance and concurrency (goroutines)</li><li>• Strong static typing and compile-time checks</li><li>• Good support for building CLI tools and backend services</li><li>• Good error handling</li></ul>	<ul style="list-style-type: none"><li>• No prior experience</li><li>• Verbose for some tasks compared to dynamic languages</li><li>• Longer development cycles during learning phase</li></ul>

Python	<ul style="list-style-type: none"> <li>• Extensive prior experience</li> <li>• Simple, expressive language</li> <li>• Huge ecosystem of libraries</li> <li>• Very fast prototyping and iteration</li> <li>• Great for scripting and glue code</li> </ul>	<ul style="list-style-type: none"> <li>• Messy library/environment handling unless carefully managed (e.g., with virtualenv/poetry)</li> <li>• Slower runtime overall performance</li> <li>• Dynamic typing can lead to runtime errors if not carefully tested</li> </ul>
--------	--	---

*Table 8: Our Comparison between Python and Golang*

In the end, we decided to go for Golang. A major point of the outcome of this decision were that GoLang is the main language in the networking community including Containerlab as well as the concerns from our supervisor regarding the library management in python.

### 7.8.3. Object Persistence - GORM

For our project, we needed a way to efficiently interact with our PostgreSQL database without writing extensive SQL queries. Since we were inexperienced with Go and wanted to focus on delivering core functionality rather than writing and optimizing raw SQL, we evaluated several options for object-relational mapping (ORM).

We considered:

- **sqlx:** A lightweight extension of Go's database / SQL package. While flexible, it still required us to write and maintain most queries manually.
- **Ent:** A powerful and type-safe ORM. However, it introduces a more complex code generation step, which we felt would increase the learning curve for our project.
- **GORM:** A mature and widely used ORM with a simple API and broad community support.

We ultimately decided to go with GORM because it provided the best balance between ease of use and advanced features, such as automatic migrations, transactions, and relationship handling. Going with GORM also allowed us to implement a structured and maintainable database layer with minimal boilerplate, while keeping our focus on Antimony's core features.

### 7.8.4. Web Server - GIN

Similarly, when choosing a web framework for our API, we compared several options:

- **Built-in net/http:** fully functional, but very low level. We would have had to implement routing, middleware, and other common patterns manually.
- **Echo:** another popular lightweight framework, but its community and ecosystem appeared slightly smaller than Gin's at the time of our evaluation.
- **Gin:** a widely adopted framework that provides excellent performance, an easy-to-use API, built-in middleware support, and good documentation.

We ended up choosing Gin because it allowed us to get started relatively quickly, while also offering the flexibility and performance we needed for a production-ready API. Its strong support for JSON bindings, request validation, error handling, and fast routing made it the best fit for our use case. In addition, Gin's minimal memory overhead helps us maintain high responsiveness even under load.

## 7.8.5. Real-time Communication - Socket.IO

A key requirement for Antimony was real-time communication between the server and the interface. Uses for real-time communication in Antimony are status updates of running labs, log streaming, the integrated terminal and more. A stateless HTTP API alone are insufficient for these use cases due to their polling overhead and lack of server-push capabilities. Therefore, a bidirectional, event-based communication channel was required.

### 7.8.5.1. Why not gRPC?

The first and most obvious answer would be going with an RPC-based solution<sup>12</sup>. We briefly evaluated using gRPC, one of the largest RPC frameworks that provides excellent performance. However, gRPC is not natively supported in browsers without additional tools like gRPC-Web, which adds additional complexity.

Native web sockets are already natively supported in browser environments and provided a good alternative to RPCs. However, using raw web sockets again comes with a lot of overhead and manual packet crafting and parsing.

### 7.8.5.2. Why Socket.IO?

In order to take advantage of web sockets, we decided to use Socket.IO, a web socket based library that abstracts a lot of the low-level logic to maintain web socket connections.

- Browser-native support and mature client libraries.
- Support for both web sockets and HTTP long polling.
- Provides a lot of abstraction over complicated raw web sockets.
- Has a large ecosystem and good documentation.

---

<sup>12</sup>[https://en.wikipedia.org/wiki/Remote\\_procedure\\_call](https://en.wikipedia.org/wiki/Remote_procedure_call)

- We already used it for the interface prototype.

#### 7.8.5.3. The Socket.IO Library

Instead of using the largest community-maintained Socket.IO library, go-socket.io, which is no longer actively maintained, we selected socket.io<sup>13</sup> a lightweight and actively maintained alternative. To structure our real-time communication more clearly, we implemented a custom abstraction layer consisting of:

- SocketManager: A global manager for the Socket.IO server and user authentication.
- NamespaceManager: A type-safe, generic wrapper for managing individual namespaces, connected clients, and sending messages.

This abstraction provides several advantages: it integrates authentication via SocketAuthenticatorMiddleware for secure namespaces, tracks connected users and their roles (e.g., admins), and supports message backlogs, so newly connected clients can immediately receive relevant recent state. Furthermore, it offers a simple API for sending messages to all clients, to selected users, or to administrators only. This design makes the system both flexible and maintainable.

We did not implement a full formal message protocol on top of Socket.IO, but we introduced consistent conventions to keep the message flow structured. All messages are transmitted using the 'data' event and are strictly JSON encoded. The NamespaceManager ensures that all messages pass through a well-defined onData callback, which improves predictability and simplifies testing. While this is not a complete formal protocol (e.g., versioned schema), it provides sufficient structure for the current needs of Antimony and can be extended in the future if a richer protocol is required.

### 7.8.6. Migration from Vis.js to Cytoscape

During our Work on Antimony we realized some major shortcomings of vis.js that are essential for our End product, Thus we decided to Search for an alternative to avoid these problems and improve Antimony. As Already seen in our previous analysis [2] there were multiple contenders to be used but after new evaluation we decided that Cytoscape is more suitable for our project. This is due to that Cytoscape has grouping, which was an important feature for us, as well as that Cytoscape is more suitable to work with in connection with git commands. We also realized that Cytoscape is just generally more advanced, and since we almost fully removed the vis.js specific features already there was no benefit in staying with vis.js.

During our work on Antimony, we encountered several limitations in Vis.js that no longer matched the needs of our evolving product. While Vis.js is well suited for

---

<sup>13</sup><https://github.com/zishang520/socket.io>

force-directed (physics-based) graphs, our project had moved towards requiring a more structured and predictable graph layout — particularly to support features such as version control of topology files and consistent rendering in Git-based workflows.

As described in our previous analysis [2], we evaluated several alternatives and ultimately selected Cytoscape.js. Cytoscape.js provides a more suitable architecture for our use case: its flexible layout engine enables static, reproducible graph layouts, which greatly simplifies version control and topology diffing. Additionally, features such as grouping (while not essential) further improved the usability of the editor.

Since we had already phased out Vis.js-specific features (such as the physics engine), there was no significant downside to migrating. On the contrary, adopting Cytoscape.js improved both the technical maintainability of the code base and the user experience of the topology editor.

### 7.8.7. Domain-Driven-Hexagon Architecture [1]

As we were both new to Go, we opted for a well, established and maintainable architectural pattern that we had previously worked with for the server: the Domain-Driven Hexagonal Architecture. This approach enabled us to clearly separate the domain logic from the surrounding infrastructure, including CLI execution, socket communication, authentication and configuration.

The architecture helped us manage complexity by enforcing clear boundaries between the core application logic and the surrounding systems, such as authentication, deployment, and socket communication. It enabled us to divide responsibilities between the domain layer, which handles business logic and data structures, and various infrastructure components, such as the authentication manager, deployment handlers, and socket interfaces. This separation made the server's code easier to read and improved its testability, modularity and future maintainability.

Furthermore, this design makes it easier for new contributors to navigate the code-base, since responsibilities are organized into logical layers, for example, `src/domain` for core entities and use cases, and `src/auth`, `src/socket`, and `src/deployment` for infrastructure concerns.

### 7.8.8. Authentication

In designing Antimony's authentication system, we aimed to support both institutional deployments at OST and flexible use cases outside OST.

To meet this need, we implemented a dual authentication strategy:

- **Authentication via OpenID Connect**

This is the preferred mode for production deployments at OST. By integrating with OSTs institutional identity provider (Keycloak), Antimony can leverage existing user accounts and group memberships. This enables us to support advanced features such as role-based permissions (e.g. separating instructors from students) and preventing unauthorized actions (e.g. students deleting labs created by others).

- **Native Login**

In addition to authentication via OpenID, we implemented a simple native username / password login method. This authentication method enabled the local use in small environments where large infrastructure services like an OpenID provider were either not necessary or required.

The AuthManager component encapsulates this logic and exposes a unified API to the rest of the system. Both authentication modes issue JWT-based tokens for session management and integrate seamlessly with Antimony's real-time socket layer via authenticated namespaces.

This flexible approach allows Antimony to meet the specific needs of OST while remaining broadly usable in other contexts.

## 7.8.9. Containerlab Interface

Since Antimony is built around Containerlab as its core network emulation tool, the integration with Containerlab was a critical part of the system design. However, Containerlab currently provides only a CLI interface, while there was an experimental effort to implement an API part <sup>14</sup> part <sup>15</sup>, this project has been inactive for over a year and is not considered stable nor is a completion date in sight.

This presented us with two possible options:

1. Implement our own API wrapper for Containerlab, acting as an intermediate layer.
2. Interact directly with Containerlab via its CLI.

We chose the second option — to interact directly with the Containerlab CLI. Implementing an additional API layer on top of the CLI would have introduced unnecessary complexity and maintenance overhead, without clear benefits. Furthermore, the Go programming language offers excellent support for working with CLI processes in a controlled and reliable way, making this approach straightforward to implement.

To ensure future flexibility and extensibility, we abstracted the Containerlab integration behind a generic interface, called DeploymentProvider. This interface

---

<sup>14</sup><https://github.com/srl-labs/containerlab/pull/1906>

<sup>15</sup><https://github.com/srl-labs/containerlab/pull/1944#issuecomment-2419665119>

allows us to easily support additional backend implementations in the future, such as an experimental Clabernetes deployment engine, without requiring changes to the rest of the system.

### 7.8.10. Handling of Topology Metadata

Our initial approach for managing metadata was to store it in a separate file. This had clear advantages: it kept the topology file clean and maintained a strict separation between the actual network topology and editor specific parameters. However, this design also introduced additional complexity, as it required managing and importing a separate file.

During the course of our thesis work, we discovered that the official VS Code extension from Containerlab<sup>16</sup> handles metadata inline, embedding it directly within the topology file itself. Although we had already implemented the logic to support separate metadata files, we decided to align our approach with the standard used by the Containerlab VS Code extension.

Switching to inline metadata offered several practical benefits. Most notably, it allowed us to consolidate all relevant information into a single file, simplifying the import process. More importantly, it ensured compatibility with the official Containerlab tooling a significant advantage for future maintainability and usability.

This change did come with some trade-offs. For example, we had to abandon features such as compounding groups, which allowed groups to contain both, other groups and individual nodes, relying on the previous separate metadata structure. Nonetheless, we believe the improved integration and consistency with Containerlab's ecosystem outweighed these limitations.

### 7.8.11. Connecting to Containers

Since our tool is deploying containers for labs in educational environments, it will eventually be required for students to somehow be able to access the deployed instances. So far, this was solved by using a Web SSH<sup>17</sup> library that was hosted independently. This was also the first possibility that we considered when thinking about how to implement this feature into Antimony.

The more difficult alternative to this was implementing our own web terminal, either via Containerlab's own exec command<sup>18</sup>, or via Docker Exec<sup>19</sup>. The main advantage of this solution was that it didn't require any networking setup and eliminated the need for additional authentication or the setting up of SSH keys.

---

<sup>16</sup><https://containerlab.dev/manual/vsc-extension/>

<sup>17</sup><https://github.com/huashengdun/webssh>

<sup>18</sup><https://containerlab.dev/cmd/exec/>

<sup>19</sup><https://docs.docker.com/reference/cli/docker/container/exec/>



Even though the second option was more difficult to implement, we decided to go with that solution, as having our own terminal implementation can save a lot of debugging and setup time later on when it comes to deploying Antimony in a large-scale environment.

## 7.9. Difficulties

Every software project comes with its own set of challenges, and this thesis was no exception. In this chapter, we document the key difficulties we encountered during the development of Antimony, and how we addressed them. These challenges ranged from working with unfamiliar technologies, to dealing with limitations of third-party tools, to managing architectural complexity. Reflecting on these difficulties not only helps explain certain design decisions, but also illustrates the learning process we underwent throughout the project.

### 7.9.1. Unfamiliar Environment

One of the key challenges we faced during the project was working in an unfamiliar technical environment. Both the language we used to implement the server, GoLang, as well as the core network orchestration tool Containerlab were new to us at the start of this thesis.

In particular, Go's concurrency model presented some initial difficulties. While Go provides powerful primitives such as goroutines and channels, it does not include a built-in structured concurrency model or task scheduler. As a result, we encountered situations where asynchronous processes could be lost or mismanaged if not carefully handled. To address this, we had to develop our own mechanisms to ensure reliable process management and state tracking in the backend.

Additionally, although we had worked on the Antimony interface in a previous thesis, we had not used Containerlab directly. Our prior work focused primarily on building the user interface, not on interacting with the underlying Containerlab system. As such, we had to invest time in learning how Containerlab operates, understanding its CLI options and topology model, and designing our integration accordingly.

Despite these initial learning curves, both areas became more manageable over time as we gained experience. This process of overcoming technical unfamiliarity was a valuable part of the project and contributed significantly to our personal growth as developers.

### 7.9.2. Talking to Containerlab

Since our project fully relies on Containerlab, designing a robust interface for communicating with it was a critical part of our architecture. As described in the



Design Decisions section (Section 7.8.9), we chose to interact with Containerlab via CLI commands, as this is currently the only stable interface available.

However, this choice introduced several challenges. First, managing asynchronous CLI processes in Go proved to be non-trivial. The language does not provide built-in lifecycle management or structured concurrency for such cases, so we had to implement our own mechanisms to handle process execution, cancellation, and cleanup reliably.

Another significant concern is future compatibility. Since there is no official Containerlab API, our implementation is tightly coupled to the current CLI behavior. Any change to the CLI output format can impact Antimony and potentially break functionality. We encountered this issue first-hand during the development process: when upgrading to Containerlab version 0.68<sup>20</sup>, a minor change to the inspect command's JSON output caused a parsing failure in our backend logic.

While our system currently works reliably, this tight coupling to the CLI remains a known risk area for future maintenance.

### 7.9.3. Local Containerized Keycloak Deployment

Another challenge we encountered was during the integration of Keycloak for authentication, particularly in the context of local development and deployment. While integrating Keycloak itself was relatively straightforward, we ran into significant issues when attempting to run Keycloak as part of the same Docker Compose stack as Antimony.

The core issue arises from the fact that OAuth2/OIDC token flows cannot reliably operate if the authentication token is requested from a service running inside the same internal Docker network without proper configuration. This can lead to problems with hostname resolution, TLS(Transport Layer Security) verification, and potential feedback loops through the reverse proxy (Traefik). On some Linux setups, there are workarounds, such as using `network_mode: host` or custom DNS(Domain Name System), but these solutions are not portable and do not work consistently on platforms like Windows Subsystem for Linux (WSL2) or macOS.

To ensure a consistent and stable development and deployment experience across all platforms, we decided to adopt a simple and robust solution: running Keycloak outside the main Docker Compose stack, as an independent service. This allowed us to avoid the internal network constraints and ensured that all authentication flows work correctly, regardless of the host environment.

### 7.9.4. Clabernetes Integration

---

<sup>20</sup><https://containerlab.dev/rn/0.68/>

It was always a goal to ensure that Antimony could be extended to support Clabernetes. For this reason, we invested significant time in working with Clabernetes during the project.

The main challenge was the complexity of the required knowledge. While learning to work with Containerlab was relatively straightforward, Clabernetes required a much deeper understanding — not only of Clabernetes itself, but also of Kubernetes and its ecosystem. This learning curve made the integration more time-consuming and demanding than initially anticipated.

Despite these challenges, we succeeded in developing a functional prototype and designed the overall infrastructure to support Clabernetes-based topologies. However, due to time constraints and the prioritization of more critical features for Antimony, the Clabernetes integration remains in an experimental state. The core structure is in place, and the remaining work consists primarily of proper testing and fine-tuning of minor bugs. It is our hope that this work can be completed in the future, either by OST students or by open-source contributors.

## 7.9.5. Working With Windows Subsystem for Linux

We also encountered some additional challenges related to differences in development environments. Part of the team worked with Windows Subsystem for Linux (WSL), while the other part worked on native Linux systems.

Since WSL was new to some team members, this increased the initial learning curve when working with Containerlab and Clabernetes, particularly in combination with Docker. We experienced instability in certain parts of the WSL Docker integration, which led to various issues and required setting up a separate native Docker installation within WSL to achieve a stable setup.

Another source of friction was the difference in how networking behaves across WSL and native Linux. Certain solutions, such as the `network_mode: host` configuration which could have been used to address the Keycloak deployment issue Section 7.9.3, only work on native Linux systems. This led to occasional mismatches and additional debugging time during development.

While these issues were not central to the core functionality of Antimony, they did consume time and added complexity to the development process.

# Quality Assurance

To ensure that Antimony meets both its functional and non-functional requirements, we placed a strong emphasis on backend test coverage and maintainability. Our goal was not only to verify correctness but also to establish a robust and extensible testing setup that would enable future contributions with confidence.

## 8.1. Code Linters

Since the frontend was already developed in a previous thesis, we did not adjust the code linters but instead used the existing infrastructure. Detailed information regarding this can be found in the antimony semester thesis documentation[2], at section 7.4.

For the backend, we opted to use a well-maintained community configuration[4] that we found on GitHub, which is based on `golangci-lint`<sup>21</sup>, a widely used and popular Go multi-linter tool.

This setup allowed us to run a comprehensive set of static analysis tools to catch potential bugs, enforce coding style, and improve code quality. Our configuration included a broad range of linters, such as:

- `govet` and `staticcheck` for detecting bugs and suspicious constructs
- `errcheck` to ensure proper error handling
- `revive` and `golines` to enforce consistent code style and formatting
- `gosec` for identifying potential security issues
- Many additional checks targeting code complexity, function length, and best practices

Adopting this tooling early in development helped us improve the maintainability, readability, and robustness of the backend codebase.

## 8.2. Automated Testing

Due to the size of the project and the complexity of its backend components, we focused our automated testing efforts on two main areas:

- API tests: Covering all routes and expected error scenarios.
- Unit tests: Targeting the business logic within the `lab.service` and socket-triggered functions.

---

<sup>21</sup><https://github.com/golangci/golangci-lint>

Our Result of our testing is a 66% coverage which includes all the Routes and respective Error paths, as well as a proper coverage regarding the lab service and the seperate functions to handle labs.

### 8.2.1. Code Tests - Testify

We chose the Testify library for our testing needs. As newcomers to Go, we found Testify particularly helpful due to its:

- Readable and expressive assertions
- Built-in support for mocks
- Widespread community adoption and good documentation

Testify allowed us to write clean and maintainable tests with minimal boilerplate, making it a natural fit for our needs.

#### 8.2.1.1. API Tests

To ensure reliable backend behavior, we implemented test cases for all major API routes and their respective edge cases. A custom test server setup function was created, which initializes the environment with mock data and dependencies for each test run. Each test is executed in an isolated environment to avoid side effects and preserve reproducibility.

This setup allowed us to simulate real usage conditions and validate both successful calls and potential failure scenarios.

#### 8.2.1.2. Unit Tests

In addition to the API layer, we focused our unit testing efforts on the core business logic encapsulated in the `lab.service`. This logic is responsible for handling lab manipulation, socket interactions, and key features such as starting or stopping labs.

To isolate these tests from external dependencies, we relied on mocking techniques provided by Testify. This ensured our tests remained fast, deterministic, and focused on the actual Lab service logic.

## 8.3. Manual Testing

Since Antimony will be used in production, we also conducted manual tests to ensure that all essential functions work as expected. To provide additional assurance, we designed a formal test procedure, covering both individual key functions and a typical end-to-end workflow.

The primary goal of this testing was to verify that students will be able to use the tool effectively in a teaching environment. The complete test checklist, including the tested workflows and functions, is included as a form in the appendix at Table 11.

### 8.3.1. Usability Tests (UX)

Much of the frontend UI was developed during a previous thesis, which included structured usability tests with student participants. These tests validated key user interactions such as topology creation, node manipulation, and metadata editing.

While we did not conduct a full round of UX testing during this thesis, we retained the previously validated design patterns and extended the UI in a consistent and user-friendly manner. This ensured that Antimony continues to offer an intuitive experience, aligned with its educational purpose.

# Bibliography

- [1] "Domain-Driven Hexagon." Accessed: Jun. 09, 2025. [Online]. Available: <https://github.com/Sairyss/domain-driven-hexagon>
- [2] T. Stromer and K. Gribi, "Antimony: Networktopology-Editor for Containerlab." Accessed: Jun. 04, 2025. [Online]. Available: <https://eprints.ost.ch/id/eprint/1258/>
- [3] I. J. T. C. 1/Subcommittee 7 (JTC1/SC7), "ISO/IEC 25010 - System and Software Quality Models." Accessed: Jun. 09, 2025. [Online]. Available: <https://www.iso25000.com/index.php/en/iso-25000-standards/iso-25010>
- [4] maratori, "Golden config for golangci-lint v2.1.6." Accessed: Jun. 10, 2025. [Online]. Available: <https://gist.github.com/maratori/47a4d00457a92aa426dbd48a18776322>

# List of Figures

Table 1	Defintion of Terms .....	4
Figure 1	.....	12
Figure 2	.....	13
Figure 3	.....	13
Table 2	NFR-1 .....	14
Table 3	NFR-2 .....	15
Table 4	NFR-3 .....	15
Table 5	NFR-4 .....	16
Table 6	NFR-5 .....	16
Table 7	NFR-6 .....	17
Figure 4	Project plan of Antimony .....	17
Figure 5	The Antimony Repository .....	18
Figure 6	Risk complexity of Golang over the course of the project .....	19
Figure 7	Risk Time Management over the course of the project .....	21
Figure 8	Risk Container Runtime Variability over the course of the project .....	22
Figure 9	Risk Migration from Vis.js to Cytoscape over the course of the project .	23
Table 8	Our Comparison between Python and Golang .....	24
Table 9	NFR Protocols .....	39
Figure 10	Lighthouse tests from Login-, Dashboard-, Editor-page .....	39
Figure 11	.....	39
Figure 12	.....	39
Figure 13	.....	39
Table 10	performance Test .....	39
Table 11	Manual Tests .....	40
Table 12	Backend licenses .....	43
Table 13	Frontend licenses .....	44
Table 14	Tools we used as aid .....	49

# 10

## Appendix

Here are documents regarding our Meetings, documentation of Non-Functional-Requirements and the Personal Reports.

### 10.1. Non-Functional Protocols

ID	OK/NOK	Comments
NFR-1	OK	See Manual Tests at Table 11
NFR-2	OK	See Performance Tests at Table 10
NFR-3	OK	Tested browsers <ul style="list-style-type: none"> <li>• Google Chrome: OK</li> <li>• Edge: OK</li> <li>• Firefox: OK</li> <li>• Safari: OK</li> </ul>
NFR-4	OK	While Functionalitys got added we stuck to the layout of the original Frontend, placing buttons and features in the same locations, consistent with the established layout rules of the Antimony frontend and the results of its user experience tests [2].
NFR-5	OK	<ul style="list-style-type: none"> <li>• Login Page: Best Practice score: 96</li> <li>• Editor Page: Best Practice score: 100</li> <li>• Dashboard Page: Best Practice score: 100</li> </ul> also seen in Figure 10
NFR-6	OK	See license compliance at Section 10.1.1

Table 9: NFR Protocols

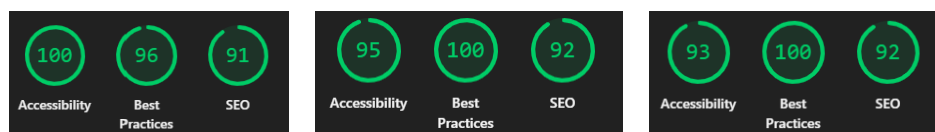


Figure 10: Lighthouse tests from Login-, Dashboard-, Editor-page

Criteria	OK/NOK	Commnet
----------	--------	---------



Login page loads in < 3 seconds	OK	1.48 seconds
Dashboard page loads in < 3 seconds	OK	1.37 seconds
Editor page loads in < 3 seconds	OK	1.31 seconds

*Table 10: performance Test*

Test-case	Expected	T/F
Login via OIDC	Successful login	T
Logout	Return to login page	T
Login native as admin	Successful login	T
Click on editor page	Redirect to editor page	T
Create group "Test_Grp"	New folder appears	T
Create topology "Test_Delete"	Creates new topology file	T
Click on git sync button	Git sync dialog opens	T
Enter "https://raw.githubusercontent.com/holo-routing/containerlab-topologies/refs/heads/master/ospfv3-sr/topology.yml" and click sync now	Topology gets added into text editor	T
Click on settings gear icon in visual editor	Opens physics pop up	T
Click on stabilize graph	Visual graph stabilizes / gets displayed in proper manner	T
Click on garbage-bin icon next	Delete dialog opens	T
Click on OK	Topology gets deleted	T
Create Topology "Test_topo"	New file appears	T
Add node in YAML	Node appears on visual editor	T
Click on + symbol in visual editor	Node editor menu opens	T
Set parameters click save	Node appears in visual editor	T
Right click on node	Context menu opens	T

Left click connect	Connection follows cursor	T
Click on second node	Connection gets added in visual and text editor	T
Delete Connection	Connection disappears in visual and text editor	T
Create connection in text editor	Connection appears in visual editor	T
In visual editor click on group nodes and include both nodes while drawing	Group name dialog should appear	T
Enter test into the dialog and click the button Ok	Group outline and label should appear	T
Rename "graph-group" of node 1 to test2	Each node has separate Group	T
Double click node group	Rename dialog appears	T
Save topology	Notification appears topology saved, topology saves	T
Refresh page	Topology didn't change still 2 nodes with separate groups	T
Deploy topology with name "testDeploy" set deployment start in 1 hour	Notification scheduled lab testDeploy	T
Click on dashboard	Redirect to dashboard page lab testDeploy appears with state scheduled	T
Reschedule lab 1 minute in the past click submit	Notification saved changes	T
Wait 10 seconds or less	Lab starts deploying	T
Click on Lab	Lab dialog opens	T
Click on logs	Terminal with logs regarding Containerlab deployment opens	T

Wait for deployment to finish	Success notification, nodes status point turns green	T
Close log terminal	Returns to lab dialog	T
Right click on node	Context menu appears	T
click stop node	Node status point turns red, notification node stopped	T
Open context menu	Stop node, open terminal, restart node functions not available	T
Click on start node	Notification node started, node status point turns green	T
Click restart node	Notification received node is restarting	T
Open context menu click open terminal	Terminal opens	T
Type in echo test	test should be printed in the terminal	T
stop same node with open terminal	Terminal expires	T
Open context menu try open a second terminal	Nothing happens terminal still expired	T
Press x button	Terminal closes	T
Re-open Terminal of same node	Commands from previous test still present, terminal active	T
Press x button	Terminal closes	T
Try adjusting topology in lab dialog	Nodes cant be moved	T
Press button redeploy lab	Notification appears Lab scheduled, lab deploying, lab goes back into state deploying	T
Wait	Notification appears deployment successful, lab back in state running	T

Click destroy lab	Destroy lab dialog pops up	T
Click OK	Notification appears lab is getting destroyed, lab state changed to stopping	T
Wait	Notification appears lab is successfully destroyed, lab state changed to inactive	T

Table 11: Manual Tests

### 10.1.1. License Compliance

We declare here all licenses of directly used dependencies in this project. All the Licenses fall under free-use for open source project.

Library	License
github.com/charmbracelet/log	MIT
github.com/coreos/go-oidc	Apache-2.0
github.com/docker/docker	Apache-2.0
github.com/gin-gonic/gin	MIT
github.com/glebarez/sqlite	MIT
github.com/golang-jwt/jwt/v5	MIT
github.com/google/uuid	BSD-3-Clause
github.com/joho/godotenv	MIT
github.com/otiai10/copy	MIT
github.com/samber/lo	MIT
github.com/santhosh-tekuri/jsonschema/v5	MIT
github.com/stretchr/testify	MIT
github.com/swaggo/files	MIT
github.com/swaggo/gin-swagger	MIT
github.com/swaggo/swag	MIT

github.com/zishang520/socket.io	MIT
golang.org/x/oauth2	BSD-3-Clause
gopkg.in/yaml.v3	MIT
gorm.io/driver/postgres	MIT
gorm.io/gorm	MIT

Table 12: Backend licenses

Library	License
"monaco-editor/react"	"MIT"
"react-hook/resize-observer"	"MIT"
"tsparticles/engine"	"MIT"
"tsparticles/preset-links"	"MIT"
"tsparticles/react"	"MIT"
"xterm/xterm"	"MIT"
"classnames"	"MIT"
"cytoscape"	"MIT"
"cytoscape-cose-bilkent"	"MIT"
"dayjs"	"MIT"
"file-saver"	"MIT"
"highlight.js"	"BSD-3-Clause"
"highlightjs-line-numbers.js"	"MIT"
"iconoir-react"	"MIT"
"install"	"MIT"
"js-cookie"	"MIT"
"jsonschema"	"MIT"
"lodash-es"	"MIT"
"material-symbols"	"Apache License 2.0"

"mobx"	"MIT"
"mobx-react-lite"	"MIT"
"monaco-editor"	"MIT"
"monaco-yaml"	"MIT"
"object-path"	"MIT"
"primeflex"	"MIT"
"primeicons"	"MIT"
"primereact"	"MIT"
"react"	"MIT"
"react-big-calendar"	"MIT"
"react-cookie"	"MIT"
"react-cytoscapejs"	"MIT"
"react-dom"	"MIT"
"react-graph-vis"	"MIT"
"react-highlight"	"MIT"
"react-loader-spinner"	"MIT"
"react-monaco-editor"	"MIT"
"react-router"	"MIT"
"react-tooltip"	"MIT"
"socket.io-client"	"MIT"
"url"	"Apache License 2.0"
"uuid"	"MIT"
"vis"	"MIT"
"vis-data"	"MIT"
"vis-network"	"MIT"
"xterm-for-react"	"MIT"

"yaml"	"ISC"
"babel/core"	"MIT"
"babel/plugin-proposal-decorators"	"MIT"
"babel/preset-env"	"MIT"
"babel/preset-react"	"MIT"
"babel/preset-typescript"	"MIT"
"eslint/compat"	"Apache-2.0"
"stylistic/eslint-plugin"	"MIT"
"stylistic/eslint-plugin-ts"	"MIT"
"types/file-saver"	"MIT"
"types/js-cookie"	"MIT"
"types/lodash-es"	"MIT"
"types/node"	"MIT"
"types/object-path"	"MIT"
"types/react"	"MIT"
"types/react-big-calendar"	"MIT"
"types/react-cytoscapejs"	"MIT"
"types/react-dom"	"MIT"
"types/react-highlight"	"MIT"
"types/react-tooltip"	"MIT"
"types/react-transition-group"	"MIT"
"types/vis"	"MIT"
"babel-loader"	"MIT"
"babel-plugin-jsx-control-statements"	"MIT"
"compression-webpack-plugin"	"MIT"
"copy-webpack-plugin"	"MIT"

"css-loader"	"MIT"
"css-minimizer-webpack-plugin"	"MIT"
"dotenv"	"BSD 2-Clause"
"eslint"	"MIT"
"eslint-import-resolver-typescript"	"ISC"
"eslint-plugin-import"	"MIT"
"eslint-plugin-jsx-control-statements"	"MIT"
"eslint-plugin-n"	"MIT"
"eslint-plugin-prettier"	"MIT"
"eslint-plugin-unused-imports"	"MIT"
"file-loader"	"MIT"
"gts"	"Apache-2.0"
"html-webpack-plugin"	"MIT"
"ifdef-loader"	"MIT"
"mini-css-extract-plugin"	"MIT"
"monaco-editor-webpack-plugin"	"MIT"
"postcss"	"MIT"
"postcss-loader"	"MIT"
"sass"	"MIT"
"sass-loader"	"MIT"
"style-loader"	"MIT"
"ts-loader"	"MIT"
"typescript"	"Apache License 2.0"
"typescript-eslint"	"MIT"
"url-loader"	"MIT"
"webpack"	"MIT"



"webpack-cli"	"MIT"
"webpack-dev-server"	"MIT"
"webpack-merge"	"MIT"
"workbox-cli"	"MIT"
"workbox-webpack-plugin"	"MIT"

*Table 13: Frontend licenses*

## 10.2. List Of Tools

These are the tools we used while working on Antimony and its documentation.

Task Area	Tools
Research	Google, ChatGPT
Idea generation	Plant UML, teamgant
Coding	Google, Stack overflow, ChatGPT
Teamwork and Organisation	Outlook, Teams, GitHub, Notion
Text creation, text optimization, spelling and grammar checking	Language Tool, ChatGPT

*Table 14: Tools we used as aid*