CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Design & build a web interface for a quantum computer

Bachelor's thesis in Computer science and engineering

Lucas Karlsson
Alexander Jyborn
David Andreasson
Karl Gunnarsson
Vlad Dragos
Jamal Aldiwani

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2022

Bachelor's thesis 2022

Design & build a web interface for a quantum computer

Lucas Karlsson
Alexander Jyborn
David Andreasson
Karl Gunnarsson
Vlad Dragos
Jamal Aldiwani

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Design & build a web interface for a quantum computer

Lucas Karlsson Alexander Jyborn David Andreasson Karl Gunnarsson Vlad Dragos Jamal Aldiwani

Supervisor: Sandro Stucki, Computing Science division, Department of Computer Science and Engineering.
Advisor: Miroslav Dobsicek, Department of Microtechnology and Nanoscience, Quantum Technology Laboratory
Examiner: Robin Adams, institution for Data- and IT

Bachelor's Thesis 2022
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2022

Design & build a web interface for a quantum computer

Lucas Karlsson, Alexander Jyborn, David Andreasson, Karl Gunnarsson, Vlad Dragos, Jamal Aldiwani

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The Wallenberg Centre for Quantum Technology (WACQT) is a research project that aims to develop a high-end 100-qubit quantum computer. To get an overview of the quantum computer and to gain insights about the quantum computer configuration's state through visualizations, a Graphical User Interface (GUI) is needed to display the data from the different configurations. This report describes the development steps of designing and building a web-based GUI for accessing information about different quantum computer configurations. Accomplishing this project needed two groups, a frontend and backend group, working in parallel to realize this project. This report covers the frontend groups contributions to the project. This project was developed in cooperation with WACQT.

Sammanfattning

Wallenberg Centre for Quantum Technology (WACQT) är ett forskningsprojekt vilket har som mål att utveckla en högpresterande 100-qubit kvantdator. För att få en överblick över kvantdatorn och för att möjliggöra ytterligare insikter om kvantdatorns tillstånd genom visualiseringar behövs ett grafiskt gränssnitt för att visa upp datan från de olika konfigurationerna på denna maskin. Denna rapporten förklarar utvecklingsstegen för att designa och bygga ett webb baserat grafiskt gränssnitt för en kvantdator. För att genomföra detta projekt krävdes två grupper, en frontend grupp och en backend grupp som jobbar parallellt för att utföra projektet som planerat. Rapporten täcker frontend-gruppens bidrag till projeketet. Detta projektet utvecklades i samarbete med WACQT.

Keywords: Qubit, Connectivity Map, GUI, React, TypeScript, Web Interface Design

Acknowledgments

We want to acknowledge and thank our advisor Miroslav Dobsicek, for providing all the theoretical information introducing the field of quantum computers and clarifying the requirements to accomplish this project. Our supervisor, Sandro Stucki, for mentoring us and for all the intuitive feedback that helped optimize our project's quality.

We would also like to thank Sandoko Kosen for testing our final GUI and providing us with helpful tips for improving visualizations in the GUI. Finally, we want to thank the backend group for the cooperation and all the time and effort put into completing our scheme.

Lucas Karlsson, Alexander Jyborn, David Andreasson, Karl Gunnarsson, Vlad Dragos, Jamal Aldiwani, Gothenburg, June 2022

Contents

List of Figures

1

Introduction

This chapter introduces the project and the purpose of this thesis. It defines the scope of the project and what goals we have set for ourselves. It also introduces some relevant projects that already exist with a similar use case as our project. Then the section about the management workflow for the project discusses how we used SCRUM, cross-group collaboration, and GIT. Finally, the chapter concludes with the contributions that we made to this project.

1.1 Purpose

Wallenberg Centre for Quantum Technology (WACQT) is a large-scale research project at Chalmers University of Technology. Their aim is to develop a Swedish high-end 100-qubit quantum computer based on superconducting circuits [1]. The project is planned to span 12 years. In our Bachelor thesis project, we designed and built a web application displaying and visualizing information from different quantum computer configurations on the behalf of WACQT.

Our general purpose in working on this project was to develop a web-based GUI that displays the information of the quantum computer configurations. The web-based GUI allows the researchers at WACQT to get a better overview of the state of the different quantum computer configurations. The GUI also allows the user to gain insight about the quantum computer configuration's state through different visualizations (as illustrated in Figure. 1.1) described further in Chapters 3 and 4.

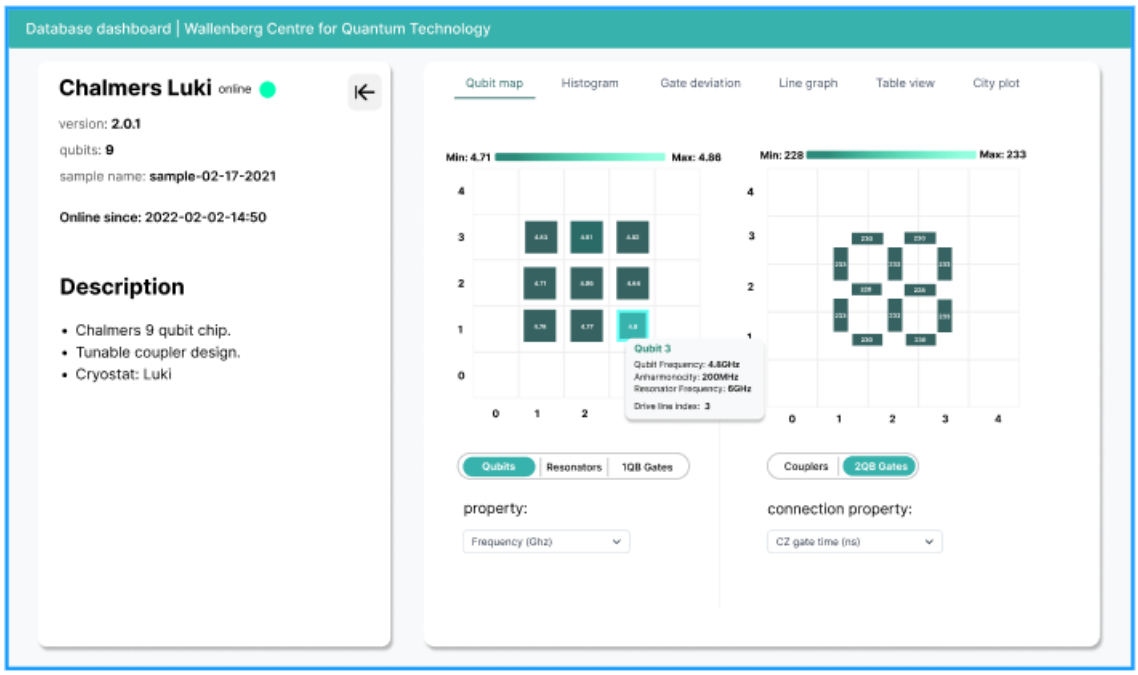

Figure 1.1: Version 2 mock-up of the detailed view, with the qubit and coupler maps.

The project is divided into a frontend group and a backend group. The frontend group focuses on designing and implementing the visual components that the user can see and interact with on the web application. Such components include diagrams, data visualization and simple buttons or text fields. The backend group focuses on implementing the database necessary for storing all the data that the frontend group will fetch to create the visualizations. The backend group also focuses on manipulating the data received from the quantum computer to format it sensibly. The formatted data is served to the frontend group via a REST API. The REST API has endpoints suited for all possible visualizations, further described in Chapters 3 and 4. As the frontend group, this part of the project focuses on designing and implementing the GUI side of the project.

The final product from the group is a repository containing the source code and documentation of the interface. The aim for this repository is that the code should be written with further development in mind. Thus, it should be possible for any other developer to extend or modify the already existing code.

1.2 Scope

The scope of this project is to display and visualize data for the different quantum computer configurations. Therefore, the frontend that we develop in this project does not include an interface for programming quantum chips. This will be discussed in Chapter 6 as future work.

Information about programming in JavaScript/TypeScript and their respective frameworks is well documented and broadly available online. These frameworks will be further discussed in chapter 4. Thus, the report will focus on the design choices made during development.

Furthermore, there will be a brief introduction on how a quantum computer works in chapter 2. But the thesis will not go in-depth about the quantum physics of how a quantum computer works, as the project is primarily about creating a web interface for visualizing data.

The product we develop will rely on the data served through the API developed by the backend group. Therefore, how the data is "created/calculated" will not be discussed in detail in the report as this is not part of our project scope.

1.3 The goals of the project

To measure the success of the project, we have set the following criteria to be applied to the final product:

- Features implemented compared to features not implemented for the final version of the GUI.
- Visualization coverage, how much of the data received from the backend can be visualized.
- The final user experience feedback given by the WACQT research group.
- Test coverage. We aim to achieve 80% test coverage. This should ensure that most use cases are tested while not impeding the development process to achieve 100% test coverage. How we define test coverage will be described in section 2.4.

How well we managed to reach these goals will be discussed in Chapter 6.

1.4 Workflow

This section describes the methods used to organize, develop and control our project.

1.4.1 Scrum

We worked in an agile manner using Scrum [2], with a sprint period of one week and used the following roles: Scrum Master, designer, and coder. Each week we swapped the roles so that everybody could be able to try on a role. We, later on, decided on fixed roles depending on our preferences after everyone had the opportunity to try on the different roles. We decided on fixed roles, as that would let us work on what we thought was most interesting. Twice a week, meetings were held to assess progress and any issues that had come up during the sprint. The scrum board was hosted at Trello and will be discussed later in the tools and technologies section in Chapter 4. The reason for choosing scrum as our productivity framework is because of the great number of benefits it offers when managing the development. Scrum

forces us to divide big, complex solutions into smaller, more manageable tasks that let the team make steady progress over time.

1.4.2 Cross-group collaboration

The GUI developed by the frontend team receives data in JSON [3] formatted by the backend server, which is developed by the backend group, via a REST API [4]. The data describes the state of several quantum computer configurations. For instance, data includes the number of qubits, qubit frequency, qubit anharmonicity, and resonator frequency. We then visualize that data in tables, charts, and other UI components on the frontend side of the project. Therefore, collaboration between the groups is necessary, and weekly meetings with the backend group were held to coordinate the groups and solve eventual problems. We also had a shared way of communication via Slack. We used Slack as a quick way of communication for questions or problems that could easily be solved via text.

1.4.3 Version control

To maintain good code quality and make it easier for multiple people to be working on the code simultaneously, we used GIT [5] as our version control software and had the code uploaded on GitHub [6]. We developed new code on local feature branches. Then we reviewed each other's code before merging it into the main branch to maintain a good code standard.

1.5 Contributions

We made the following contributions to our project:

- We designed mockups for the software, this is discussed in Chapter 3
- We implemented the designs into a functional web app, this is discussed in Chapter 4
- We presented the results of the project and show the final product in Chapter 5
- We implemented tests to maintain code quality. This is discussed in Chapter 5

Chapter 2 gives the background information needed to understand the rest of the thesis. In chapter 6, we discuss whether we achieved our goals discussed previously, and what future work could be done on the application.

2
Background

This chapter introduces a sequence of theories that helped develop this project and made it easier to achieve good results for the product. It starts with an introduction to quantum computers and quantum chips. It will also show the theory behind designing our GUI and the testing method that has been used to validate our product. Finally, the chapter will end with a list of criteria that will evaluate our project.

2.1 Technical background

This section introduces the basics of what a quantum computer is and provides some context for the visualizations we created.

2.1.1 Introduction to quantum computer

A quantum computer is a computer that utilizes quantum effects such as quantum entanglement and superposition to perform computations [7]. These effects allow quantum computers to compute at a much faster rate than classical computers. While classical computers perform computations on bits, quantum computers perform computations on qubits. A single qubit can effectively represent multiple states simultaneously, in contrast to a classical bit representing either 1 or 0. A qubit achieves this by utilizing a quantum effect called superposition. As quantum computers exhibit quantum behaviour, they tend to be better at simulating and making computations than classical computers in areas where quantum mechanics are present. Some of these areas include physics, chemistry, and pharmaceutical science [8].

2.1.2 Visualization of quantum chip properties

The quantum chip's properties are divided into two sections:

1. Static chip configurations.
2. Dynamic runtime properties.

The static properties would present the name and version of the active quantum computer configuration, the number of qubits, and where the qubits are located. The change of static values will be long-term due to the aged chip. Meanwhile, the dynamic properties change values due to external causes, such as an electromagnetic field or cosmic rays.

Static properties and dynamic properties exist for the following listed components of a quantum chip.

1. Qubits, in contrast to a classical bit that can represent either a 1 or 0 a qubit can represent multiple states at once. A qubit achieves this by utilizing a quantum effect called superposition[7].
2. Resonators are pieces of wire connected to the qubits. The qubits state affect the resonators frequency, which in turn is used t0 obtain information about the qubit state.
3. Couplers are what physically connect the qubits together.
4. Gates or quantum logic gates are used to modify quantum states or states of qubits similarly to logic gates in classical computing. The gates can use any number of qubits. In this project, two qubit gates and one qubit gates existed.

Because there are a lot of properties to visualize, our supervisor from WACQT classified them into five types. The keys grouped into a classification are meant to be visualized together.

- Type 1
- Type 2
- Type 3
- Type 4
- Type 5

A benefit of classifying properties that belong into different types is that future visualizations could be added to their respective type. For instance, there are multiple ways of visualizing the data from the different classifications. One could argue that, for example, a heat-map is better than a city plot for type 5, which is further explained in 3.11. Working with classifications this way enables the developer to easily switch or add another visualization for a chosen type without having to modify already written code or making changes to the API, which is explained in 4.1.2.

2.2 State of the art

There are three prominent companies in the context of creating interfaces for quantum computers. The quantum computer with the largest amount of qubits is owned by IBM and has 127-qubit processor [9]. The second largest is owned by Google and has a 72-qubits processor [10]. Microsoft is also developing the Rigetti quantum computer in cooperation with Rigetti Computing [11], which will be provided through the cloud to users of Microsoft's Azure [12].

2.2.1 IBM

IBM's interface are built as a web application that contains the three main parts [13]:

- IBM Quantum Composer: A graphical programming tool that will help build and run quantum circuits without the need to write code to visualize qubit states.

- IBM Quantum Lab: A cloud programming environment that helps build quantum applications combined with Qiskit scripts that are an SDK for quantum computations [14].
- IBM Quantum Services: A visualization of IBM quantum programs, systems, and simulators properties made it the most helpful tool for our project.

2.2.2 Google

Google has also built a development framework that also provides tools for compiling and running quantum algorithms [15]. This framework consists of five main tools to develop quantum algorithms:

- Cirq: An open-source framework for programming quantum computers [16].
- ReCirq: Research using Cirq: An open-source library for researching experiments using Cirq [17].
- OpenFermion: An open-source library for simulating fermionic systems [18].
- TensorFlow Quantum: An open-source library for hybrid quantum machine learning [19].
- Qsim: An open-source simulator for quantum circuit [20].

It also contains a Quantum Computing Service that provides access to Google's quantum computing hardware and quantum simulators.

2.2.3 Azure/Microsoft

Azure is a fast and flexible cloud computing platform operated by Microsoft. It has more than 200 products for cloud services [21]. Microsoft also tried to catch up with Google and IBM by publishing Quantum Services made by Honeywell and IonQ in their Azure platform [22]. This service provides free educational content for students. It supports Microsoft kernel IQ# and Python-3 kernel (ipykernel). This helps build quantum applications combined with Q#, Qiskit, and Cirq scripts. Azure quantum development kit tools provide two main services:

- Quantum computing service: An educational and experimenting tool with different quantum hardware providers.
- Optimization service: A tool for simulating and developing the best solutions to reduce operations cost in several domains, such as energy, transport, finance, etc.

This cloud is unique by combining those three quantum programming languages and running algorithms on multiple quantum systems.

2.2.4 Comparison between current solutions

After comparing these three systems/frameworks during the planning phase of the project, we decided that the one most relevant to our goal of creating a web-based GUI is IBM Quantum because of how well-organized the tabs are and the beneficial features for the visualizations of connectivity maps and their properties. Meanwhile, the visualizations in Google Quantum AI and Azure Quantum will be printed either through the Command-Line Interface (CLI) or by implementing some functions using Python. Although, one of our supervisors provided us with some example plots

from Google Quantum AI that were useful for the design process of the project. These are presented in Figure 2.2 and Figure 2.3.

IBM's and Google's solutions provided us with a starting point for creating our design. Our main focus when comparing and researching IBM's and Google's solutions was the visualizations for the qubit map and the coupler map. The qubit map and coupler map represent the layout of the qubits and couplers on the quantum chip. Figure 2.1 presents IBM's visualization for a combined qubit and coupler map, displaying the qubits and their respective couplers in the same view. Google's alternative, on the other hand, separates the qubit map and the coupler map into two different visualizations, which are presented in Figure 2.2 and Figure 2.3.

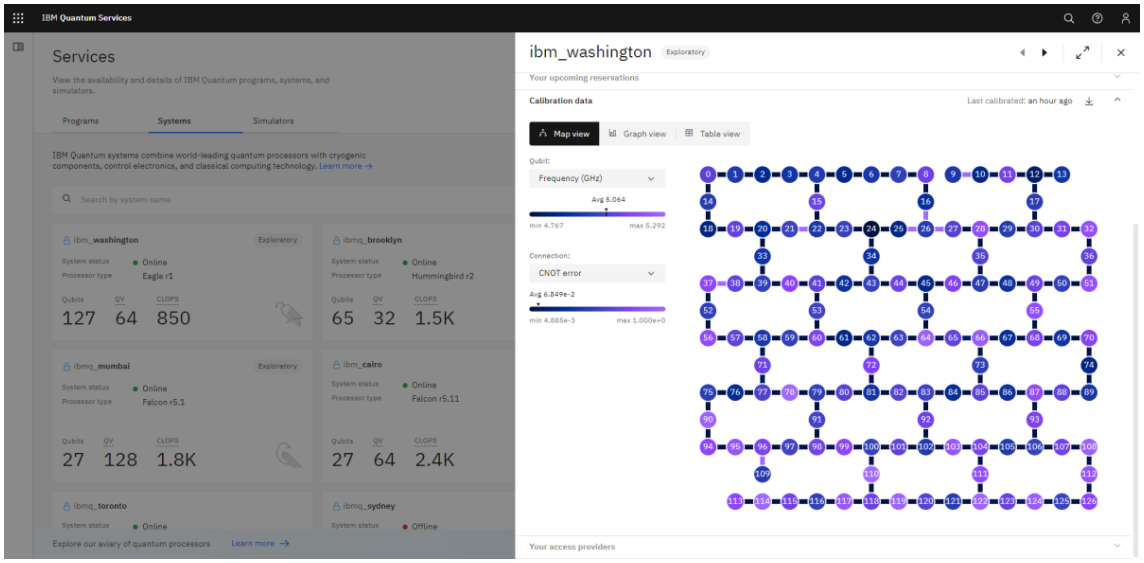

Figure 2.1: Google visualization of the qubit map.

Figure 2.2: Google visualization of the qubit map.

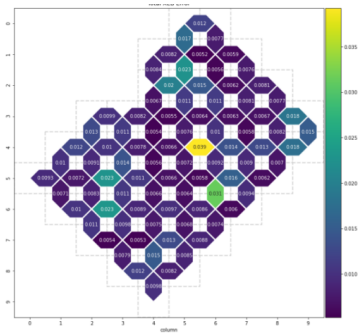

Figure 2.3: Googles visualization of the coupler map.

Another focus during the research process was to study the other UI elements and the navigation of IBM's interface. IBM's interface represents the different quantum computers with a grid of cards, which is presented in Figure 2.4. There exists functionality for searching for a quantum computer, filtering, and sorting. When clicking on one of the cards in the grid, a new detailed view appears. The detailed view contains additional information about the selected quantum computer and some different visualization options. IBM's interface has three different visualization options. The map view which is presented in Figure 2.1, a bar chart presented in Figure 2.4 and a simple table view.

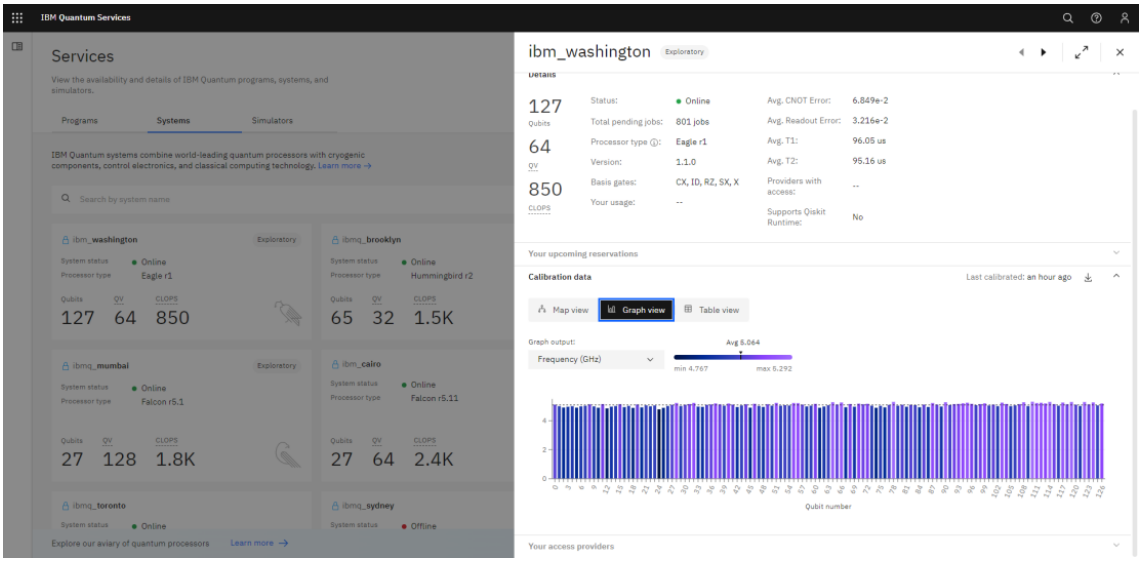

Figure 2.4: Google visualization of the qubit map.

The design process of the project will be further elaborated upon in Chapter 3.

2.3 User Interface (UI) Design

User interface design is the field of designing user interfaces based on our understanding of how humans interact with UI's [23]. When users observe a web page, they have a certain intuition about what the components on the page look like and

what they do. This intuition is built over time by interacting with different kinds of UI. When the design of a UI deviates from the user's expectations, their intuition is broken, leaving them confused. Knowing what a user is trying to achieve is imperative when designing a UI [24, p. 1].

The more that is known about the user that is intended to use a UI, the better it can be tailored for that specific target group. These target groups are in UI design, represented by personas. Personas are simplified models of users. They are often given a name and occupation. However, the most important aspect of a persona is what they are trying to achieve[24, p. 10]. Personas are often split into a primary persona and a secondary persona. The primary persona represents the largest target group, while the secondary persona represents the smaller target group [25].

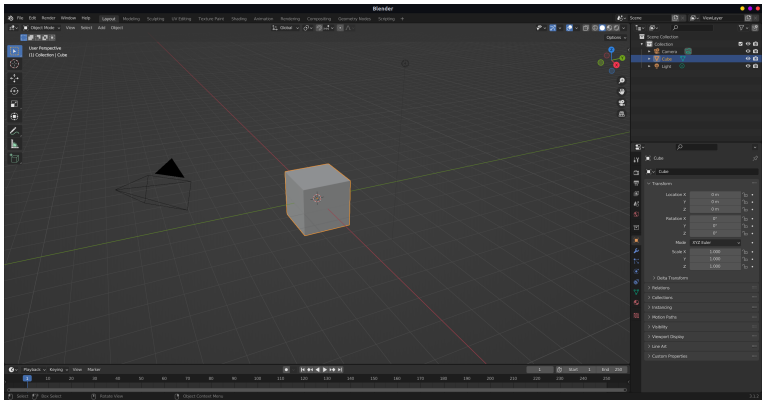

Figure 2.5: A screenshot of the 3d modelling application Blender.

Depending on the user that a persona models, it is also given an experience level of either novice, intermediate, or expert. A novice user is somebody who may be seeing a UI for the first time and does not want to spend time learning it. They typically prefer a simple but functional UI over a more complicated and powerful one. Expert users are persons who likely use an application in a professional setting and have, therefore, more interest in learning a more complex and powerful UI. Intermediate users lay somewhere between novice users and expert users. Depending on the expertise level of the primary and secondary persona, the complexity of a UI can be adapted to better fit their needs [24, p. 4]. The 3d modelling application Blender [26] could be considered to have a UI designed for intermediate to expert users. Blenders UI, as can be seen in figure 2.5, is quite complex, meaning that a user would need to spend a significant amount of time learning the application before they would be able to use it effectively.

In this project, the primary persona was modelled after the experimentalist at WACQT which we also consider to be expert users. The design of the UI in this project was directly influenced by the requests and feedback of the group's supervisors and the experimentalists at WAQCT. For further reading on the group's design decisions, see Chapter 3 and for the discussion on the user-friendliness of the UI, see Chapter 5.

Design patterns in the context of UI design are the building blocks of a UI. They are reusable solutions to common problems that users might have. The patterns range from single components such as date pickers and navigation bars to more complex patterns that involve the whole structure of a page [27].

Visual hierarchy is the concept of structuring a UI in such a way that the importance of elements is visually communicated to the user through their size, color, position, and shape. More important elements are given more prominence in the UI. This can be achieved by increasing the size, changing the color, or positioning it far away from other elements. Additionally, if it is a typographic element, changing the font type and the boldness could also be used to make the element stand out more, as illustrated by figure 2.6. Elements can also be decreased in prominence by doing the opposite. Having a clear visual hierarchy in a UI makes consumption easier for the user because the different elements are not competing for the user's attention. Instead, the user can sequentially consume the information on the page according to its importance [24, p. 209].

Figure 2.6: An example of how visual hierarchy can be used to direct the users' attention.

2.4 Testing

Testing is one of the main steps of validating software to find bugs in the written code [28]. The importance of this validating method is to identify and remove the errors that accrue in the code. Several types of testing can be applied [29]. In this project, an end-2-end test is implemented to perform a set of tests, ensuring that the application flow and user interface (UI) behave as expected. There are several options used to perform framework testing. We will use Cypress.io [30] because of our previous experience, besides the fast debugging methodology this tool provides. Later in Chapter 4 we will go into more detail about the test implementation of the

GUI.

Test coverage is the metric for measuring how much of a codebase is being tested by a test suite. There are multiple ways that test coverage can be measured. In this project, we define test coverage as the number of features, as specified in chapter 3, tested by the test suite used. We deem this necessary because there is no standardized metric for test coverage for end-to-end tests. This is a type of specification-based testing where a specification is written for an application, and tests are made to assert that the application's behaviour matches that specification. Specification-based testing tests how an application behaves rather than how it achieves that behaviour [31]. In the context of web applications, these could be tests that assert that the HTML that a web application produces is correct, rather than testing how it produces that HTML. Specification-based testing is well suited for web applications because they often consist of many parts, a frontend, backend, databases, external services, etc. We cannot test parts other than the frontend, but we rely on the creators of these individual parts making sure they work as intended. However, the application as a whole can be tested using specification-based testing. For further reading on how specification-based testing was applied in the project, see the testing section in chapter 4.4.

3
Design of web interface

This chapter focuses on the design phase of the project. The chapter is separated into two parts, one for each version we created during the project. Version 1 was created based on the initial requirements we received from WACQT through Miroslav [32]. Version 2 was created after receiving feedback from WACQT on version 1 and getting new requirements for version 2 [33]. These versions will present the requirements and how we fulfilled those requirements by creating a design that we would later implement.

3.1 Version 1

For both versions of the design, our primary persona was modeled after the experimentalist working at WACQT. Our persona was considered to be an intermediate to the expert user since this would be a tool used in a professional setting. We did not explicitly choose a secondary persona. However, we made the design so that it could also be easy to use for somebody familiar with quantum computers. Therefore, the design process consisted of trying to streamline features requested by the experimentalist while keeping the UI relatively simple and easy to understand for other users.

3.1.1 Requirements

Down below are the requirements that we initially received from WACQT through Miroslavs slides for the first version of the GUI [32]. We used this list of requirements when designing the GUI mockups for the first version.

- An overview of the quantum computer configurations
  - Have a page containing an overview of the quantum computer configurations.
  - In the overview, display the configuration name, sample name, number of qubits, description of the configuration, and the online date.
- A detailed view for a quantum computer configuration
  - The basic information from the overview.
  - Information about every qubits frequency, anharmonicity, resonator frequency, and driveline index if available.
  - Visualization for the layout of the qubits and their couplers.

Qubit frequency and anharmonicity are frequencies. These frequencies describe the physical properties of a superconducting circuit which implements a qubit. The

superconducting circuit behaves as an anharmonic oscillator that oscillates with a certain frequency and a certain level of anharmonicity. Every resonator resonates at a certain frequency which is the resonator frequency. Quantum chip components like qubits and couplers receive signals to them through drive lines. The drive line index is the index of the drive lines on the chip.

3.1.2 Design

As WACQT asked us to find a sensible way of displaying the following qubit-specific data: Qubit frequency, qubit anharmonicity, resonator frequency, and qubit drive line index, if available, we decided on the following visualizations for version 1.

- A qubit map, visualizing the layout of the qubits and their couplers, where the qubit-specific data mentioned above can be accessed.
- A bar graph with different visualization options for the data mentioned above.
- A line chart where the data mentioned above except the driveline index can be plotted over a time frame.
- A table-based visualization contains the data for the qubits.

When designing the views and components for the GUI, the communication between the group and WACQT was critical, as they make the demands and deem what is user-friendly concerning their specific needs. Figure 3.1 displays the first mockup draft for version 1 of the landing page, containing all available systems. For the overview of the systems, we have created a mockup with the following components:

- A card that contains information about WACQT.
- A card that contains information about the current number of quantum computer configurations online.
- A grid of cards that represents the different quantum computer configurations. The cards will be used to navigate to a detailed view of every quantum computer configuration.
- Cards containing the basic information for a configuration, such as a name, the sample name, number of qubits, and the online date. We decided to omit the description in the overview and move it to the detailed view, as it would not fit on the cards.
- A search field to find a specific quantum computer configuration faster.
- A filter drop-down menu to filter out online or offline configurations.
- A drop-down menu to sort the quantum computer configurations on their name, online date, online or offline status, and amount of qubits.

We chose to use cards to section off information on the page, creating cognitive separation for the user and allowing them to process the information more effectively [34]. We were inspired to use cards from the research on IBM's interface which is described in Chapter 2 section 2.2. Thus, a user coming from IBM's interface should be familiar with the layout of the overview page presented in Figure 3.1. We decided to move the description to a different card in the detailed view of a quantum computer configuration, as this is information of less importance and would be static. In comparison, the version, the sample name, and the online date could change more often and be of greater interest to the user. Figure 3.1 displays the finished mockup

of the overview page for version 1.

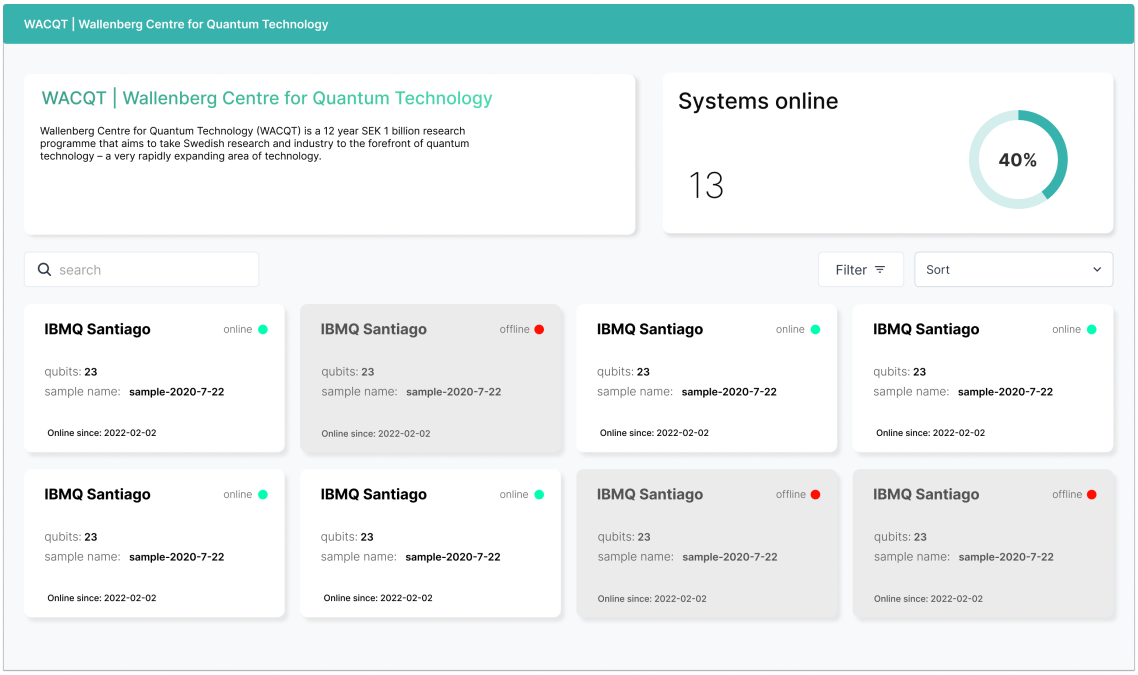

Figure 3.1: Our mockup of the overview of the quantum computer configurations.

We also added a search bar, filter, and sort button. These are common UI elements that the users should be familiar with and, therefore, be easy to use. Throughout the design, the text followed a hierarchy from most to least important, allowing for a more effortless reading experience for the user. Figure 3.2 displays an example of this. The name of the quantum computer configuration has the largest font size and is emphasized using a heavier font weight. The values are also emphasized with a heavier font weight to highlight that this is significant information.

Figure 3.2: Mockup of a card containing information of a quantum computer configuration.

For the detailed view of a quantum computer configuration, we have created a set of mockups for version 1. Figure 3.3 displays the mockup of the detailed view with the map view selected. The map view contains the following components:

- A card containing the same basic information for a quantum computer configuration as in the overview. This eliminates the need to navigate back to the overview to access this information.
- A card containing the description of the quantum computer configuration.
- A card containing the different visualization components, which for version 1 are the map view, the graph view, and the table view.
- A visualization of the qubit map with their respective indices and the qubits couplers.
- A drop-down menu for selecting a visualization option for the qubits. For version 1, this will be qubit frequency, qubit anharmonicity, and resonator frequency.
- A drop-down menu for selecting visualization options for the couplers. This was not a requirement for version 1, but we decided to include it in the mockup for further iterations of the GUI.
- A component for displaying the visualization options minimum, maximum, and average value. The component also includes a color span. The colors of the qubits will correspond to their value.
- A component that is displayed when hovering a qubit. When hovering the component, it will display the individual component's qubit frequency, qubit anharmonicity, resonator frequency, and the qubit drive line index if available.

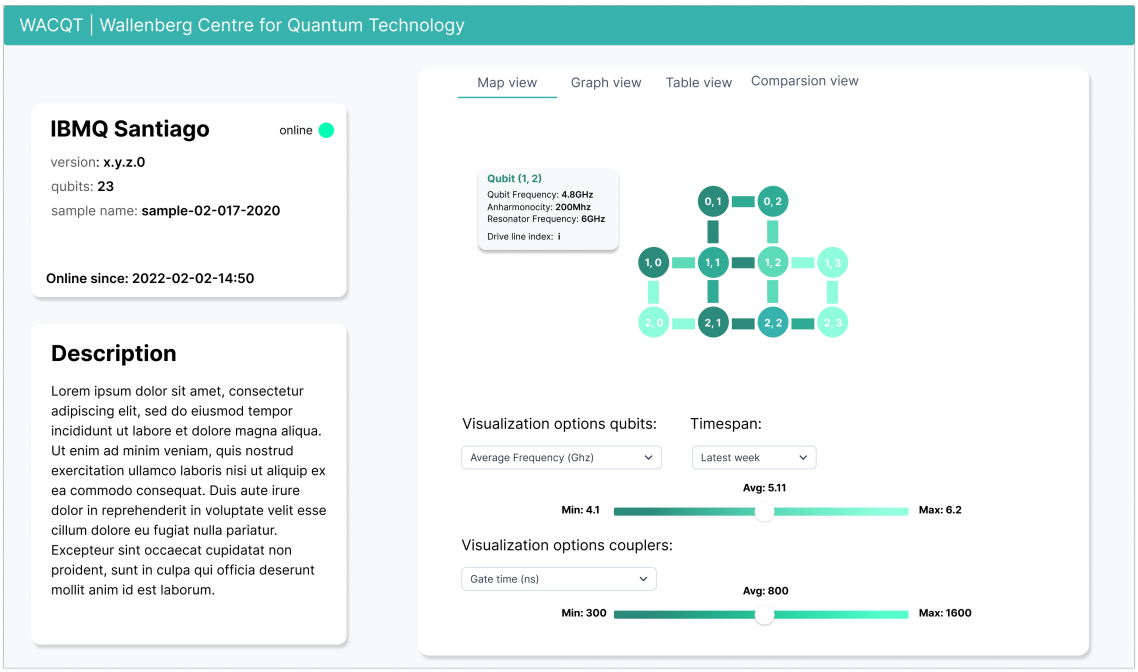

Figure 3.3: Our mockup of the detailed view, with the qubit map component selected.

For the detailed view in general, we decided to separate the information over three cards, once again to create cognitive separation for the user [34]. We decided to move the description to the bottom left card in the detailed view. We also decided to include the card from the landing page shown in 3.2. By including the information

from the landing page card, the user can access this information card in the detailed view. This eliminates unnecessary navigation where the user has to navigate back to the landing page to access this information. We decided to include a navigation bar for seamless navigation between the different visualization modes. For all the visualizations in the visualization card we followed a visual hierarchy from most to least important. The charts and visualizations are emphasized by taking up the majority of the cards space while the controls and navigation bar takes up less space.

The qubit map visualization is displayed in figure 3.3. After the research on IBM's and Google's alternatives for visualizing the qubit and coupler map described in Chapter2 section 2.2, we decided to represent the qubits and their respective couplers in the same view. We designed the combined qubit and coupler map as a combination of the solutions from IBM and Google. The qubits and couplers are presented in the same view as in IBM's visualization but with clear indices like Google's plots. The qubit map consisted of circles that represented the qubits and lines that represented couplers. We also decided to add a tooltip for the qubit map to easily display the qubits properties. Tooltips are user-friendly because they give the user the control to show additional information, allowing them to process the information in chunks. Tooltips are also a common UI pattern. Using tooltips comes with benefits for both our primary persona and our secondary persona. For the experimentalists at WACQT, the tooltip can be used to access a lot of information quickly. On the other hand, hovering the tooltip is an action of the user. Thus, the information is not forced onto a novice user. We added dropdown menus to control the properties of the qubits and the couplers. Drop down menus are common UI elements that the majority of users should be familiar with.

Another method of visualization we chose was a bar graph. We chose this because discrepancies between qubits or couplers can easily be detected visually in a bar graph. Figure 3.4 displays the bar graph component, and it differs from the qubit map component in the following way, it contains:

- A bar graph-based visualization.
- A drop-down menu for choosing a graph visualization option.
- A drop-down menu for selecting qubits or couplers on the x-axis.
- A drop-down menu for selecting the data for the y-axis. For version 1 the data will be qubit frequency, qubit anharmonicity, and resonator frequency.

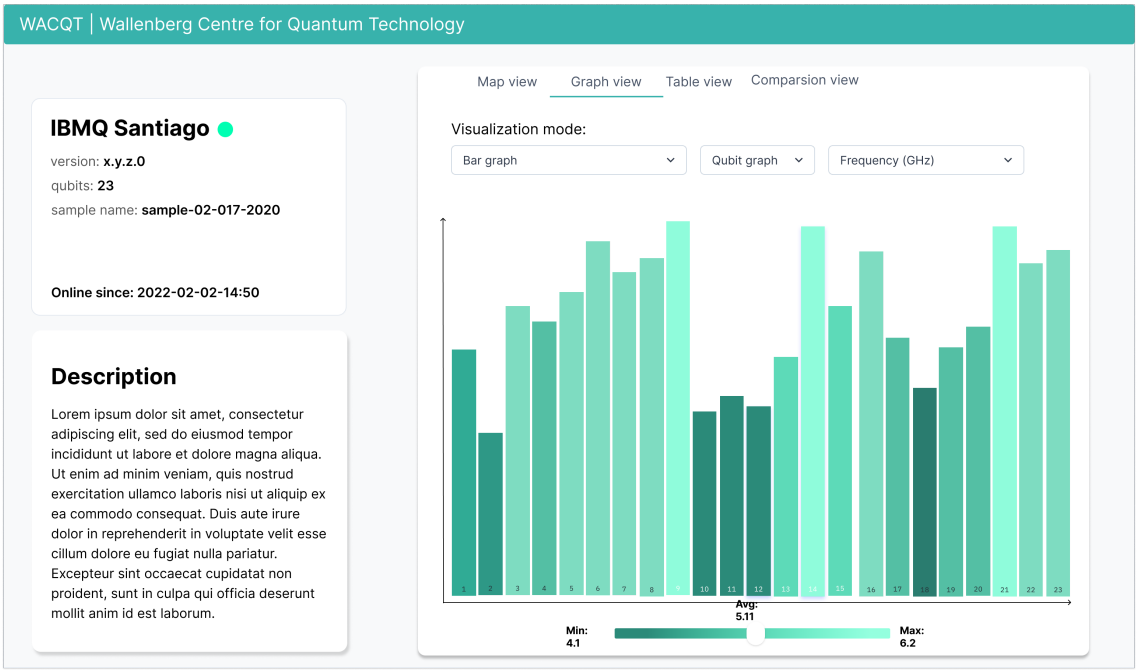

Figure 3.4: Our mockup of the detailed view, with the graph component, and bar graph visualization option selected.

Figure 3.5 displays the line chart visualization component. The main purpose of the line chart is to display data over time, allowing for trends to be observed. At the current time, when we designed the mockup for the line chart, we had not yet decided on the specific approach to how the data should be displayed. However, we envisioned that the properties of multiple qubits should be able to be visualized, each qubit represented by a line. Additionally, a single qubit and its specific data should also be displayed. Our idea was that the line chart was meant to be a very flexible means of visualization capable of displaying a variety of time-indexed data. The line chart contains the following components:

- Contains a line chart-based visualization.
- Contains a drop-down menu for selecting a time span.
- Contains a drop-down menu for selecting the data for the y-axis. The options will be the same as the options for the bar chart.

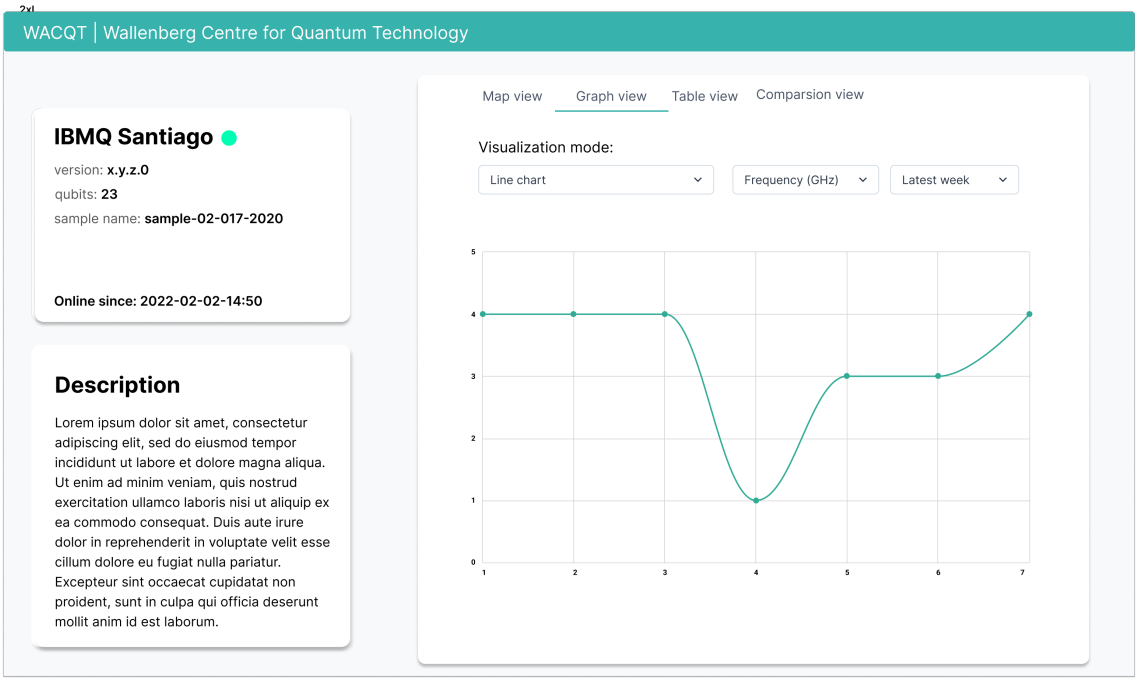

Figure 3.5: Our mockup of the detailed view, with the graph component and line chart visualization option selected.

Figure 3.6 display a table view for the qubits that contains the same information as the qubit map, but the data is presented in a table. We decided to include a table view to make it easier to compare data for several qubits at once and easily get and overview of all the data for the qubits. The table view contains the following components:

- A table containing columns for the qubits, qubit frequency, qubit anharmonicity, resonator frequency, and the driveline index.

- A drop-down menu for sorting the results in the different columns.

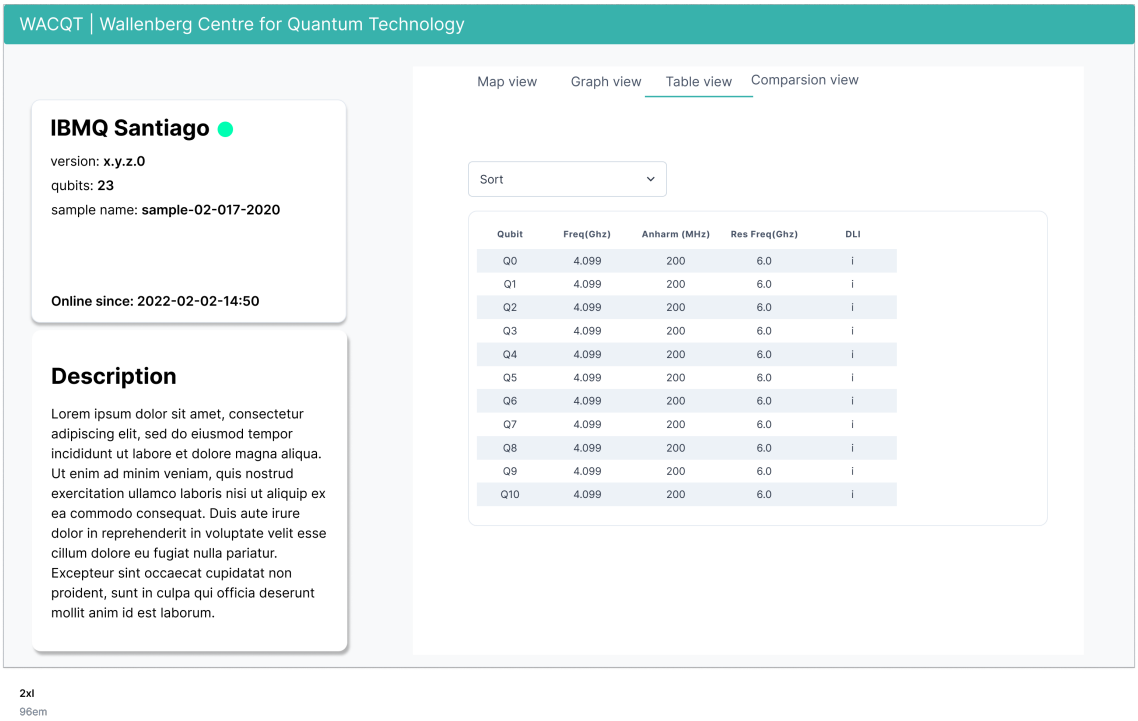

Figure 3.6: Our mockup of the detailed view, with the table view component selected.

For version 1 of the GUI, we chose to provide a variety of visualization methods, some of them overlapping in functionality; however, we believed that they each had a unique advantage and offers different perspectives. Additional visualization methods could be added as the project was developed and feedback from WACQT was received.

3.2 Version 2

In version 1, we designed the GUI from the requirements we had at the time. However, halfway through the project, we got more information from Miroslav [33] about what has needed and more in-depth information about how the GUI is supposed to be used. The new information contained the requested visualization from the experimentalists at WACQT that they deemed useful.

3.2.1 Design

With the new information provided by Miroslav, we redesigned the detail view part of the GUI, where we previously had the visualizations that we initially decided on. Miroslav provided us with five examples on different visualization that could be applied to the classifications introduced in Chapter 2 section 2.1.2 that the experimentalists at WACQT deemed useful. In addition, we also got a table of keys and what classification type the keys belonged to.

The visualizations could now be applied to the 5 different classifications:

- Type 1: A visual representation of the qubits and couplers.
- Type 2: A Histogram.
- Type 3: A box plot.
- Type 4: A Line graph.
- Type 5: A City plot.

The new mock-up for the type 1 visualization is displayed in 3.7. It is similar to the one we had in version 1 of the design. However, instead of combining both the qubits and the couplers into one bit map, we separated them into two different visualizations. When the qubit map and coupler map are separated, it is more intuitive to choose what to display on the different bit maps. Thus creating clear visual separation for the user. In addition, we choose to use radio buttons selecting between qubits, resonators, or one qubit gates for the qubit map visualization. For the coupler map, we choose to use radio buttons for selecting either couplers or two qubit gates. Lastly, we choose to use a drop-down menu for each map to choose what parameter to visualize. Radio buttons and dropdown menus are common UI control elements that the majority of user should be familiar with.

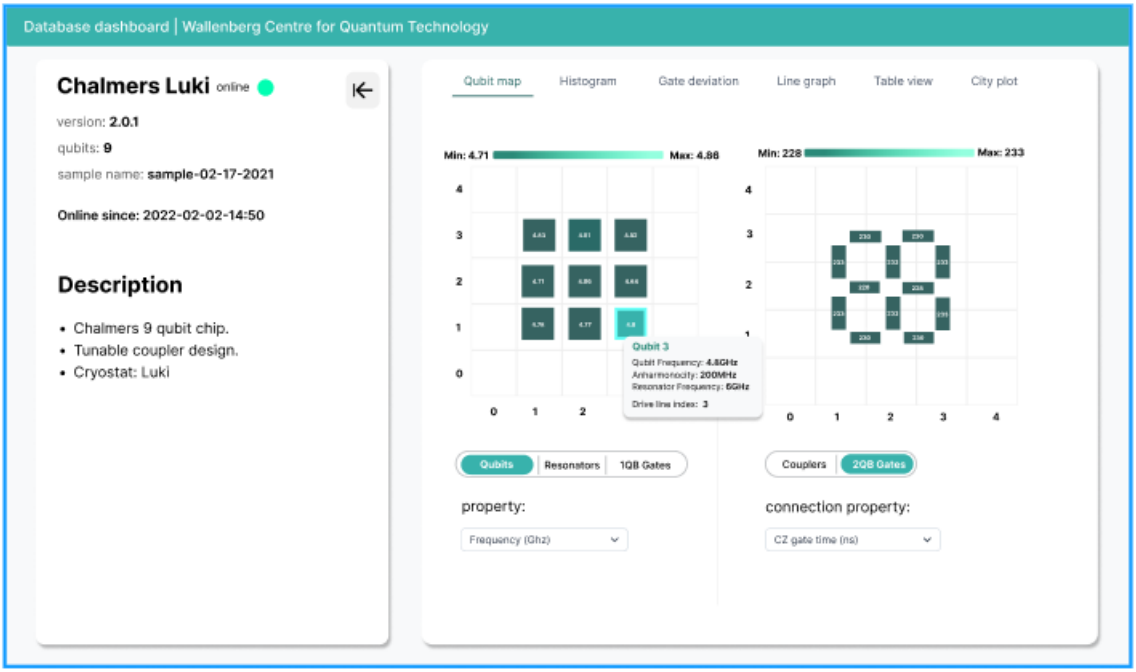

Figure 3.7: Version 2 mock-up of the detailed view, with the qubit and coupler maps.

The new mock-up for the type 2 visualization is displayed in 3.8. The type 2 visualization is a histogram for this version. Previously we had a bar graph that was supposed to do the same thing, but on the advice of the experimentalists at WACQT, we changed the bar graph into a histogram. This visualization also uses radio buttons for choosing one of the following dynamic properties and qubit-specific keys T1, T2, or Tϕ to visualize. T1 is the time of how long it takes for a qubit to degrade from a state to another. T2 is the phase lost time. Tϕ is combination of T1 and T2 times and acts as an overall qubit longevity measure. The mock-up also includes a date picker for choosing the time interval for the data to be displayed.

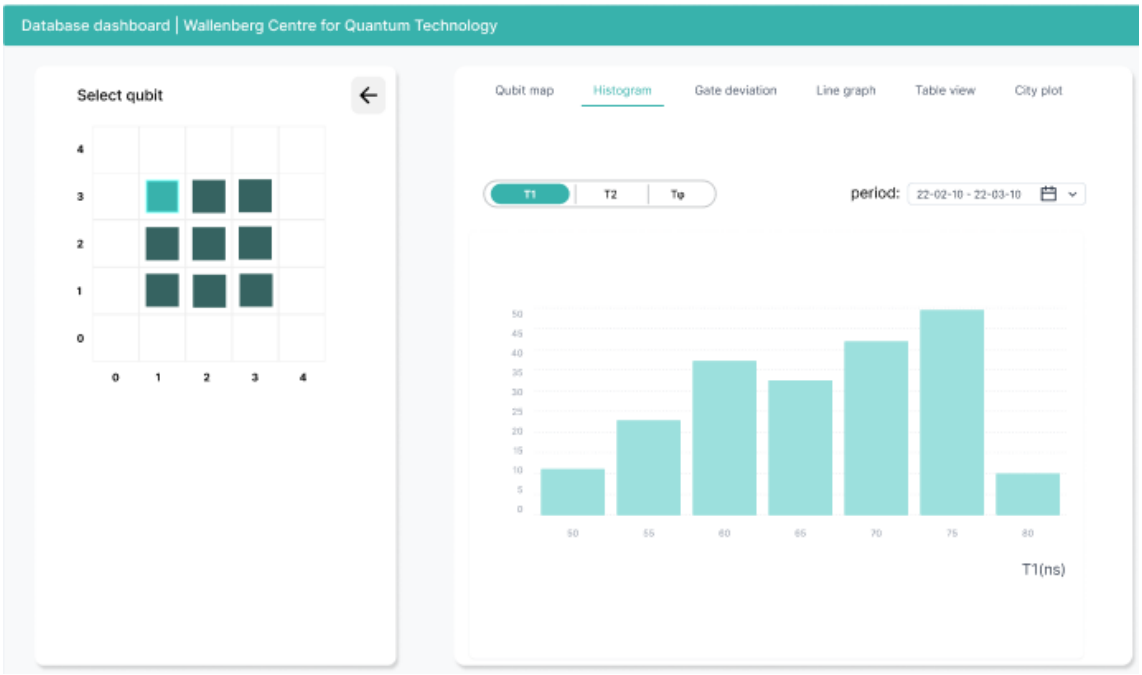

Figure 3.8: Version 2 mockup of the detailed view, with the histogram graph.

The type 3 visualization is a plot to display each qubits deviation/distribution of gate errors. Miroslav provided an example of how the visualization could look, which is displayed in figure 3.9. The type 3 visualization displays the qubit-specific key gate error on the y-axis and the qubit index on the x-axis. The user will also be able to use the date picker component presented in the type 2 visualization to pick a time span for displaying the data. The date picker is also a common UI element that the users should be familiar with. Due to the more simplistic design of the type 3 visualization and the minimal user input, we decided not to create a mockup for this visualization. We instead used the figure Miroslav provided as inspiration for the implementation of the type 3 visualization shown in figure H.6.

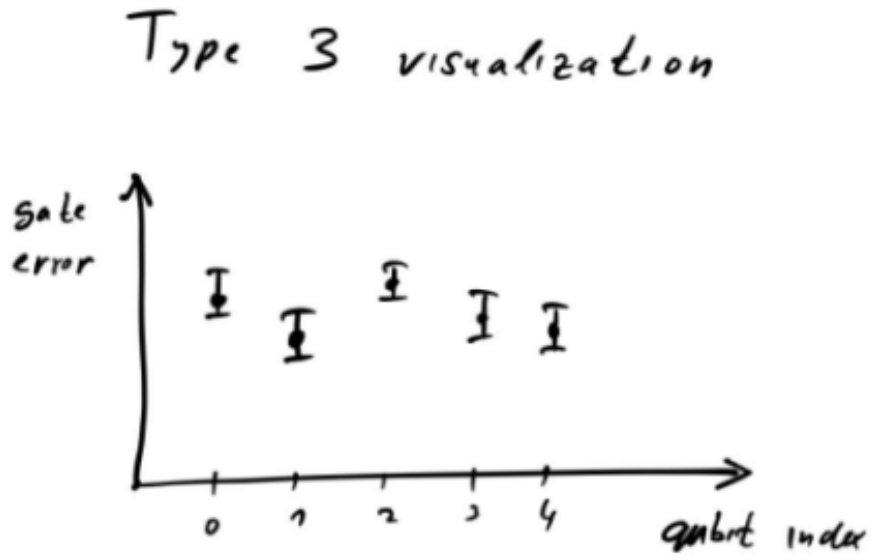

Figure 3.9: Example of type 3 visualization from Miroslavs presentation.

Figure 3.10 displays the new mockup for the type 4 visualization. The type 4 visualization is a line chart which we had already designed in version 1 but at that time we thought the most useful and most common use case of a line chart would be to see values over time. Instead we got feedback from experimentalists and Miroslav that the most important use case is to see correlations between keys. Therefore the new line chart displays values for the majority of the keys as domains and co-domains. So for the updated visualization the needed inputs are domain, co-domain, and what key to use for each. To choose all these inputs, we used radio buttons for the domain and co-domain, where you can choose either qubit, resonator, coupler, or gates. Two drop-down menus are used to select the key to be visualized for the domain and the co-domain. These inputs are displayed in rows with all the inputs in one row for the domain and one for the co-domain making it easy for the user to see the chosen inputs.

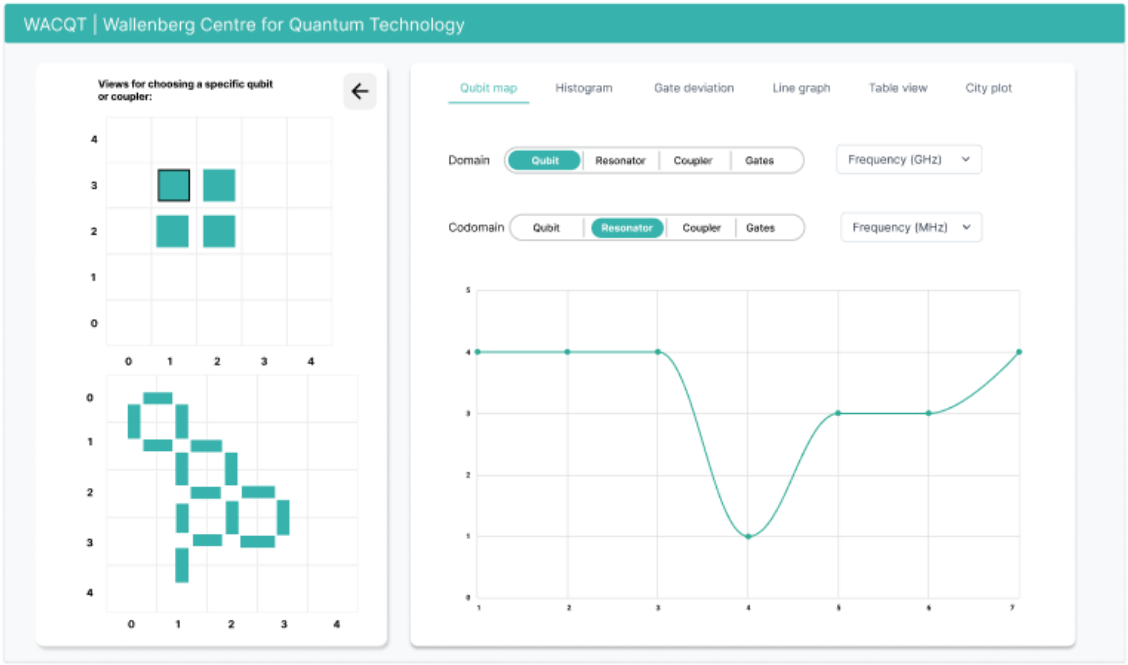

Figure 3.10: Version 2 mockup of the detailed view, with the line graph.

The Type 5 visualization is a city plot. During the design phase, we decided to focus on the first four types of visualizations due to the complexity of creating a 3D-based visualization. If there would be time over at the end of the implementation phase, we would implement the city plot. In the end, we did implement a city plot, which is further elaborated upon in Chapter 4 and Chapter 5. Miroslav also provided us with an example of what the visualization could look like, shown in figure 3.11. The idea for the city plot is to visualize the static property and coupler specific key, $xtalk\{i, j\}$, which is a value that represents unintended consequences or influence between $coupler_i$ and $coupler_j$. At the design phase, we were unsure if a city plot would be the best way to represent this. During the user test described in Chapter 5 and discussed in Chapter 6 a physics researcher from WACQT explained both advantages and disadvantages of using a city plot instead of something like a heat map.

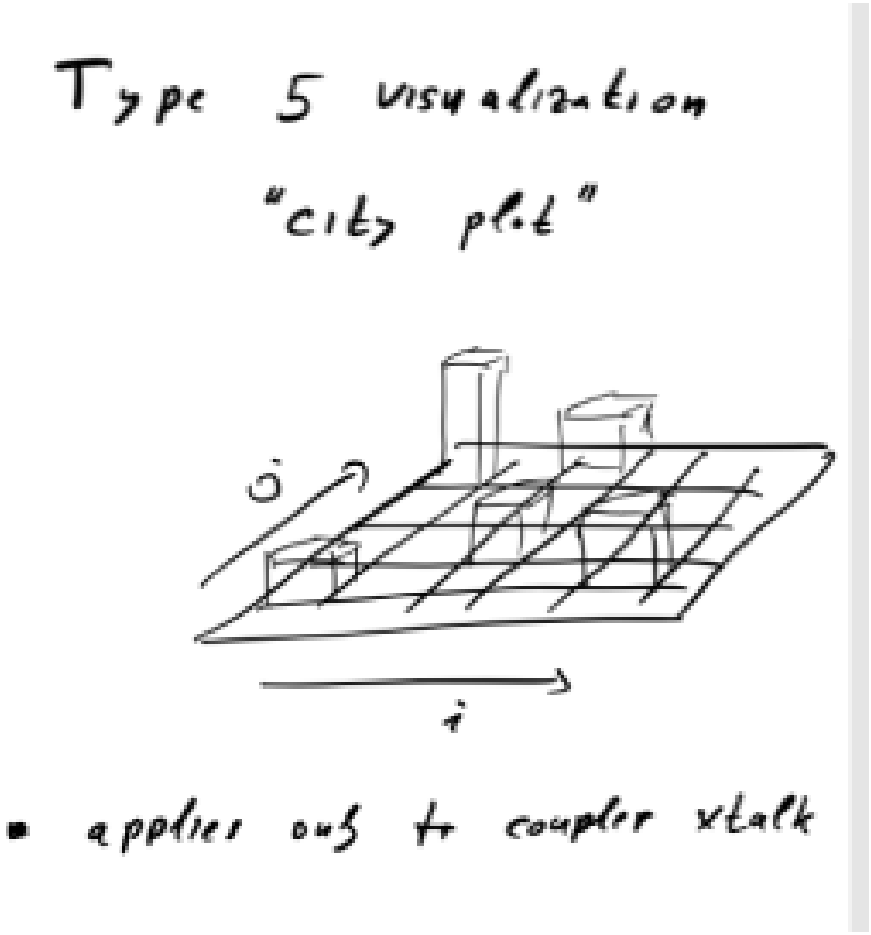

Figure 3.11: Example of type 5 visualization from Miroslavs presentation.

There will also be a table view available in this version of the GUI that will be the same as in the previous version, which will display all kinds of data. During the design process for version 2, we also came up with the navigational view that can be seen on the left-hand side of all mock-ups above. This navigational view allows the user to navigate between different qubits and couplers for a specific backend without going back to the qubit map and choosing one every time, making the navigation much smoother.
We also decided on displaying skeletons when rendering in components. Skeletons provide the user with clear visual feedback that something is loading. The primary reason for using skeletons in this project is for when large amounts of data is requested from the API. It could take several seconds before the frontend receives a response. The implemented skeleton is presented in figure H.9 in Chapter 5.

4

Implementation and development of web interface

This chapter is similar to the previous one, but instead of the design stage, this chapter will first introduce all the development tools and technologies used for implementing the design and then present the two iterations, versions 1 and 2.

4.1 The REST API

In this section, the importance of server state and the endpoints used for fetching data is discussed in detail.

4.1.1 Fetch, store and update data

When working with an API, one have to fetch data, store it and update it in the client, which in our case is in the frontend. With NextJS [35] as the chosen framework for the frontend, we have a lot of choices on how to fetch the data and represent the server state of the quantum computer configurations. However, instead of using any NextJS predefined mechanisms or fetching the data and using the React hook useState to store the data in a client side local state, we choose the React library called React Query [36]. React query is a data-fetching library that makes fetching, caching, synchronizing and updating server state easier compared to the use of the standard react hooks as the useState hook and fetching. Instead of relying on a useEffect that is triggered on the change of a state and then re-renders the component, react query does this automatically and therefore is a reasonable to use to always represent the server state.

The more traditional way of state managing libraries are often very good at working with client state, but not as good with async or server states. Because server states are completely different from client states and the challenge's server state introduces are often very hard to solve without the correct tools. Problems like caching, updating out of date data, knowing when data is out of date and managing garbage collection are just a few on the list.

The usage of React Query in this project can be seen in listing 4.1.2, where data is fetched from an endpoint in the backend. React Query provides us with the useQuery hook. The result object of the useQuery hook contains properties with

information about the queries state. We choose to destructure the result object into three properties. The data property which contains the data from the response, the isLoading property to check if the query has no data and is still fetching and the error property to check if there was an error in the query. There are more properties that can be destructured from the result object, although for most cases the data, isLoading and error properties are sufficient. Every time new data is refreshed, the React Query hook provides a new result object. The properties of the result object can then be accessed in the JSX of a component.

4.1.2 Endpoints

Before starting the development of version 2 the frontend group made a very specific request of endpoints that would be needed for further development. These endpoints and their description can be found at [37]. Here are all the endpoints we used and in what context:

- **/devices**, this is the endpoint we use for visualizing the homepage device cards, see figure 4.3.
- **/devices/<device>**, this is the most detailed information we can get from a specific device. From here we get everything that is connected to a single device. For example, the endpoint is used for getting the data needed with our qubit map tooltip, see figure H.3.
- **/devices/<device>/<type>**, This is the endpoint that gives us the exact information we need for a specific visualization, which we here denote as type. For example type 5, the city plot only needs some values connected to a device and not all, fetching data from **/devices/device/type5** gives us the values we need.
- **/devices/<device>/<type>/period?from=<date>&to=<date>**, this is the endpoint that we use for time sensitive data, if we want to look at the history of a device and thus limit the timeframe we modify the URL parameters this can be seen in action in the top right of figure H.6 and the following example demonstrating the endpoint call:
  /device/<device>/<type>/period?from=20200112&to=20200116.

Here is a code snippet of how we use React Query to fetch the data for the histogram for a set period of time. The data from the response is accessible in the data property of the destructed object.

,

```
1  const { isLoading, data, error, refetch, isFetching } = useQuery(
     'histogramData', () =>
2    fetch( 'http://<URL_BACKEND>:8080/devices/' +
3           backend +
4         '/type2/period?from=' +
5         state.timeFrom.toISOString() +
6         '&to=' +
7         state.timeTo.toISOString()
8    ).then((res) => res.json())
9  );
```

Listing 4.1: The code for fetching data from an endpoint

As explained in 4.1.1 the different properties in the destructured result object at row 1 contains the data and information about the query which in our case is the fetch function that is called inside the `useQuery` hook. `isLoading` is a boolean representing if we are still waiting for a response, `data` is the data we retrieved from the query, and `error` is a boolean that is true if an error occurred when retrieving data. In addition to the most common properties introduces in 4.1.1 we also decided to destructure the isFetching property and the refetch function. As the histogram is used for visualizing data over a period of time, the data will be refetched often. The isFetching property is a boolean and is used to render the loading skeleton introduced in Chapter 3 and presented in Figure H.9 on every refetch. The refetch function is passed down as prop to the date picker component to manually refetch the data when the date is changed.
Looking at row 3 and more specifically `http://<URL_BACKEND>:8080` this is the URL where a user can make requests to the API as seen in the above snippet. In this example, we have the following API call being sent: `/devices/backend/type2/period?from=2011-10-05T14:48:00.000Z&to=2021-10-05T14:48:00.000Z`

4.2 Development tools and technologies

Several tools were used to carry out the development steps of this project. These tools were:

- Box: A collaboration and workflow platform for managing documents. It provides cloud storage services for different file formats. This tool is used because it was supported by our university, and it made it easier to upload and access the files from the tutorials we got from teaching assistants, lecturers, and all the information needed to accomplish this project. All the files were shared with collaborators from the two groups [38]. The main benefit of using Box is to separate the information we receive from Miroslav from the groups' internal documents. Box also lets Miroslav upload information to a single location that both the frontend and the backend group can access.

- Google Drive and Google Calendar: Drive is a file storage and documents editor designed by Google. We choose google drive to organize our internal documents in the frontend group. We used it for our own documentation of meeting protocols, help session, feedback, and the diary of the project development steps. Also, we used Google Calendar to keep everyone up to date regarding all the important meetings and keep our project flow as planned.

- Overleaf: An online collaborative Latex editor that has been used for writing the planning report and this thesis, using Latex makes it easier to make sure the layout of the thesis is like we want it to be.

- Discord, Zoom, and Slack: These are all communication tools that we used to keep us in contact with each other in the frontend group, with our supervisor as well as the backend group. We used Discord for our internal communication in the frontend group. We used Slack for text-based communication with the backend group and our supervisors. Zoom was used for our joint meetings with the backend group and also as a tool for attending our supervisor meetings if you could not attend in person.

- Trello: A collaborative web-based application for organizing tasks of a project and keeping track of other collaborators progress. This is where we host our scrum board [39]. We chose Trello because it is a simple application that most group members were already familiar with.

- Figma: A web-based graphics editor and prototyping tool with additional offline features enabled by desktop applications. Designers and developers use it for building digital products and allowing them to edit comments, review designs, and code together. The frontend team used Figma because of previous designing skills and the smoothness of drawing custom shapes and leveraging a detailed grid system. Nevertheless, the frontend members used different operating systems. This made Figma the best choice that fits all the different platforms the group uses [40].

4.2.1 Tech Stack

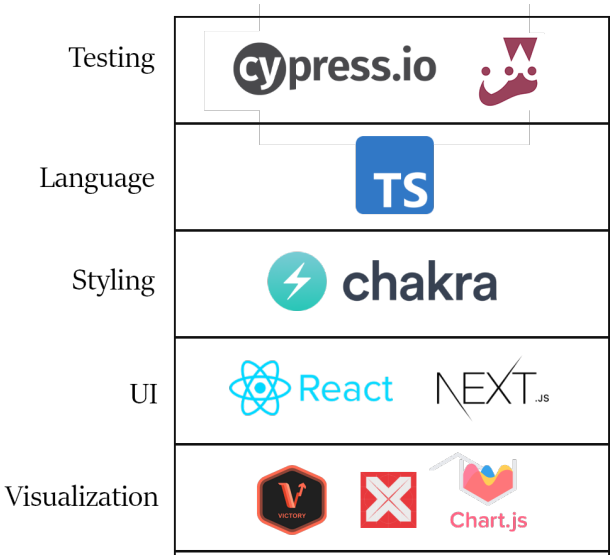

Figure 4.1: The stack of the technologies that were used in the project.

The application is built using the web component library React [41]. React was chosen as most of the group members were familiar with it. Additionally, it is supported by a large company, namely Meta. Moreover, React is a popular choice for web development and is relatively easy to learn, which means that for further development, this is a good choice as most web developers already know how to work with it.

React can either be programmed in regular JavaScript or TypeScript [42], a statically type-checked superset of JavaScript. We chose the latter option, as static type checking reduces the number of bugs and improves the development experience by improving the readability of the code and by allowing code editors to show correct code hints. Furthermore, we believe that using TypeScript makes the finished product easier to develop in the open-source domain and results in a higher quality codebase.

We also decided to use the React framework Next.js [35]. Next.js comes with TypeScript support, a solution for routing where every page is a route, support for dynamic routes, and an included bundler to minimize bundle size. The dynamic routes helped with rendering the different detailed views with the different visualization for quantum computer configurations.

Testing is an integral part of modern software development. It improves code quality and ensures the correctness of an application. For this purpose, we looked at several testing options. Firstly, we looked into some potential testing frameworks that are suggested on the Next.js website [35]. The first one is called Cypress, which is a framework for writing end-to-end tests [43]. The second suggestion is the Jest testing framework in combination with React Testing Library for writing unit tests

[44, 45]. We later decided to focus on only Cypress and use that as our main testing library and primarily end-to-end testing. End-to-end testing allowed us to test the application as a whole rather than individual components using unit testing. Ideally, we would have done both end-to-end testing and unit testing. However, due to time limitations, we chose to implement only end-to-end tests.

To ensure the website's design would maintain the same style for all different screens and components, we decided to use a component library to take care of the base CSS needed for different components. The library we chose is called Chakra UI [46]. We found Chakra to be easy to use and intuitive. It helped with speeding up development time. Chakra is also relatively popular and has good documentation, so later on, when we have given this project to new developers, further development will be more accessible. Chakra also follows the WAI-ARIA standards [47], which helped with the user-friendliness of the application.

For the visualizations, we looked into two potential libraries. The first one is the high-level charting library Charts.js [48]. While we first looked into Charts.js, we ended up choosing another charting library; namely, Victory [49]. We chose Victory instead because of its support of histograms, which was a requirement for version 2 of the GUI. Victory is also based on React, allowing for easy integration into the tech stack.

However, Victory did not meet our needs for the qubit map visualization. For the qubit map visualization, we also decided to add visx [50]. Visx is a low-level library for creating visualizations developed by Airbnb. Since the qubit map was not a standard way of visualization, we had to build it ourselves. Visx offered useful visualization primitives for this purpose. Our implementation of the qubit map, as can be seen in figure 4.2, was based on visx's network graph example [51].

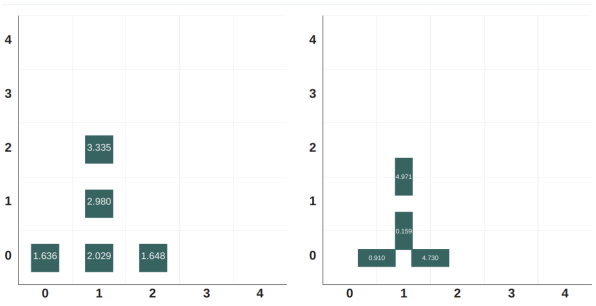

Figure 4.2: The qubit map visualization.

4.3 Iterative development

The implementation happened in two major iterations in the same way as the design, with a first version and then a second one when we got further information.

4.3.1 Version 1

For the first version, development started a little slow as we first spent some time getting familiar with the tools we chose, and development time was spent on research. So the first thing we did was set up the project with all the dependencies we knew we needed to get started, such as React, Next.js, TypeScript, and Chakra UI.

We started by developing the overview page, as this was the entry point to our application. Additionally, this was also the part of the application that we thought would change the least during future iterations. The overview page consisted of a navbar, a card with the description of WACQT, a card displaying the number of quantum computer configurations online, and a grid of cards containing the information on the quantum computer configurations. This can be seen in the first design in the previous chapter 3. All of these components were relatively easy to implement, as we first just created them with dummy data.

Chakra facilitated rapid and consistent UI development. The code snippet below shows an example of how Chakra was used for the implementation of the navbar, which can be seen in figure 4.3.

```
1  const Navbar = () => {
2    return (
3      <Container bg='teal.400' maxW='full' centerContent id='main-
     navbar' data-cy-main-navbar>
4        <Container py='2' maxW='8xl'>
5          <Box>
6            <Text
7              as='h1'
8              fontSize='2xl'
9              color='white'
10             fontWeight='bold'
11             textAlign='left'
12             w='fit-content'
13           >
14             <NextLink href='/' passHref>
15               WAQCT | Wallenberg Centre for Quantum Technology
16             </NextLink>
17           </Text>
18         </Box>
19       </Container>
20     </Container>
21   );
22 };
```

Listing 4.2: The code for the navbar.

All the elements used in the code snippet, except for NextLink, are from Chakra. Chakra uses properties instead of CSS classes to style elements. For example, on line 1, bg=’teal.400’ sets the navbar’s background to a teal colour.

After implementing the overview page with dummy data, the backend group just finished the first endpoint so that we could fetch the actual data for the cards. We

implemented this but kept dummy cards as the endpoint only returned a single card, and we wanted to show off the sort, search, and filter functionality in the midterm presentation.

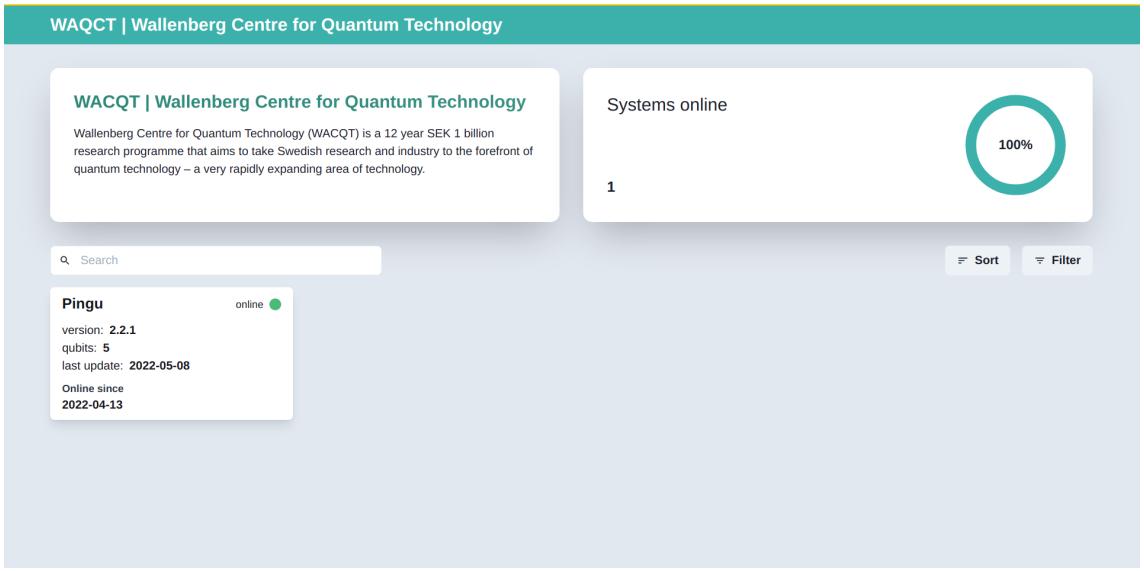

Figure 4.3: The implementation of the overview page.

After finishing the overview page, we continued implementing the first view of the detail view, where we had a combined qubit and coupler map. We started implementing this visualization but did not quite finish it as we got new information about how the qubits and couplers should be displayed which were later implemented in version 2. As we were yet not that comfortable with visx or chartjs, we spent more time researching these to make sure we knew how to implement the rest of the visualizations.

4.3.2 Version 2

When we got the specifications for the second version, we went through all the information and started another iteration of the mock-ups. Because the new mock-ups for version 2 would differ a lot from version 1 mock-ups, we made completely new ones from the ground up. However, when it came down to the programming, we could continue with the code already written because of how extensible we had written it. To continue the iterative development in version 2, there had to be much focus on breaking the problems down into smaller components that we iterate on. Where the version 2 phase is one of the iterations. Similar to version 1 development, we had to keep researching the chosen frameworks and tools to find out how one applies them to the new visualizations.

When we decided on frameworks and chose what suited best for data visualization, we again started coding, creating small components. The most coding done in version 2 were the components that visualize data from the quantum computers. One of the most important changes in this version was the backend rework; we did not

do this, but the other group did. We gave them an API specification that specified what data we wanted, how it should be structured, and where we could access it. This change was massive because the previous calls to the endpoints needed to be changed, but we knew we had to make future calls would be a lot simpler.

The visualizations made in this version are qubit map, histogram, box plot, line graph, table view, and city plot. We knew about most of these visualizations when we started version 1. However, the city plot and the box plot were new.

The radiobutton component, see appendix A, was used for almost every visualization to choose which keys or property to be displayed. The radiobutton takes four props:

- tabs: A list of strings representing the different tabs.
- setTab: A set state function that is passed from the parent element to set the current tab state. An example of this is given in the code below.
- defaultTab: An optional prop to set the default tab on render, else the first tab is the default.

The code for the date picker can be found in appendix B. The two most interesting parts are:

- The handle change function is called when a new timespan is picked and in turn updates the context. This way, the user can seamlessly navigate between the histogram and boxplot.
- The refetch function, which is a prop passed down from the parent element. When the date is changed in the date-picker, the refetch function is called, which in turn triggers React Query to refetch the data with the updated timespan from the context.

The Qubit map was described in section 3.2.1 and implemented during version 2. The components that made the qubit map are the actual map, qubits, and tooltips. As we were unfamiliar with visx, development was quite tricky. Initially, there was some difficulty with positioning the 'qubits' on the qubit map. We also decided to change the data format we wanted to receive from the backend halfway through development. This mainly impacted the development of the qubit map, but improved the development time for the other parts of the application. The qubit map component was used in several visualizations. In the qubit map visualization, the qubit map component was used to visualize the type 1 data, while in the other visualizations, it was used for qubit selection.

The qubit map component uses the **/device/<device>/** and **/device/<device>/type1** API endpoint. The first endpoint is used to get the layout of the components, and the second is used to get the property data about the specific components. There are several modifications needed to use the data from the endpoints. The first one is that the one qubit gates and the two qubit gates are in the same field called "gates", so we split these up into two separate fields. The two qubit gates also need "from" and "to" fields for the two qubits that the gate uses. The components are then grouped into two objects, "links" and "nodes". The nodes object contains the one qubit gates, qubits, and resonator data. The links objects contain the data of the two qubit gates and couplers. All of this processing

is abstracted away by a facade function that takes in the API response and returns the expected format used by the application. The data can then be loaded into the context and used wherever it is needed.

We decided to defer much of the data management to a react context[52], which managed what data was displayed in the qubit map. For example, when the user selects a property from the drop-down below the qubit map, the context registers this and updates the data that is sent to the qubit map. Thus, the map can focus on displaying data rather than managing the external state. This was necessary because keeping the state local to the qubit map component would mean that the state would not persist because of how react state works and, as a result, degrade the user experience. Another option would have been to lift the state to a component higher up in the component hierarchy and pass it down as props. However, this would have resulted in what is called prop drilling. Prop drilling is the process in a react application where props are passed from one component to another by going through other components that have no use for the data but only help in passing it through the component tree.

Once all the data is in the correct format, the qubit map component can be assembled using visx components and functions. The qubit map needs to map from the local chip coordinates to its coordinate space, which depends on the screen size of the monitor that the application is viewed on. Visx has a component called "ParentSize" that can be used to get the height and width of the parent component. The "scaleLinear" is used to map from the local chip coordinate space to the qubit map space. The "Grid" component is used to draw a grid, and the "AxisBottom" and "AxisLeft" are used to draw the axis. The "Graph" component is used with custom link and node components that we made to get the styling and behavior that we wanted to draw the rectangles and the grid.

Below is a list of the data we decided to store in the context.

- The selected qubit ID. We stored the selected qubit ID in the context so that the state could persist component unmounts.
- The type 1 data, we stored this in the context to avoid prop drilling and make state sharing easier.
- The time period Some visualizations used time periods for selecting data, specifically the histogram and the box plot visualizations. By storing the time period in the context, it meant that the user only had to select the period for a single visualization, and the other visualizations would use the same period by default. Thus, improving the user experience.

```
1  const { layout } = useSelectedComponentLayout();
2  const { selectedComponentData, selectedComponentPropertyData } =
    useMapData();
```

Listing 4.3: The functions that give the qubit map the data from the context.

The function on line 1 gives the qubit map the layout of the qubits, and the second's function on line 2 gives it the data of the qubits.

When a user looks at data connected to a specific qubit, they have the option to select the histogram-based visualization. The histogram component displays values in relation to time, and the user can decide on a time span to choose what values to use for the visualization. We implemented the histogram component with Victory, another library for visualizing data. The fetched data is passed into the histogram component depending on the user's chosen key and time span.

Appendix C displays the implementation of the HistogramVisualization component. This is a parent component responsible for fetching and passing down data to a conditionally rendered histogram component. The component accepts a backend as a prop, which is the name of the quantum computer configuration. The React useContext hook is used to access the context, which in this case is used to access the selected qubit that a user might have selected earlier in the qubit map. The context is also used to access the selected timespan, as the Box plot component also uses a timespan. This enables seamless navigation between the two components and eliminates the need to reselect the timespan when switching between the two visualizations.

React Query is used to fetch the type 2 data for the visualization which are the T1 times, T2times, and T*phi* times. A local state for the component is used to track which key to visualize. This is done with the useState hook, with the default state of T1. The setDataToVisualize function is passed down to the radio button component that is used for switching what key to visualize to update the local state to match the selected key in the radio buttons. The useLayouteffect hook is used to render the qubit map in the side panel. The qubit map in the side panel is used to select a new qubit to visualize in the histogram, which is displayed in Figure H.4 in Chapter 5. The qubit map is rendered with the custom hook setSelectionMap which accepts two boolean arguments for rendering the qubit map or/and the coupler map. If a user did not select a qubit in the qubit map visualization, we use the dispatch function from the useContext hook to set the selected qubit to the first one in the response after the useQuery hook executes the fetch.

The histogram component, see appendix D, is conditionally rendered depending on the selected key in the radio buttons, and the component re-renders when a new key is selected and the local state is updated. The refetch function from React Query is passed down to the date picker component, which will re-render the histogram component when the date is changed. The histogram component requires the data to be on the following shape:

```
1 [{x:<THEACTUALVALUE>}].
```

Listing 4.4: The required shape of the data for the histogram component

To transform the data into the correct shape, the map() function is called to map the values of the selected key from the fetched data to an array of objects with the key x and the value of the fetched value. Every value is also multiplied by 1000000 to convert the unit into microseconds. The correctly shaped data is then passed into the histogram component.

Appendix D displays the implementation of the histogram component. The histogram component accepts two props. The data prop on the correct shape, and a label for the histogram. The histogram is created with the components provided by Victory, and the correctly shaped data is passed into the VictoryHistogram component. The different Victory components are styled to be consistent with the mockups presented in Chapter 3.

Appendix E presents the implementation of the BoxPlot component. The BoxPlot component accepts the data prop that is on the correct shape for the visualization. The correctly shaped data is passed into the VictoryBoxPlot component. The Victory components are styled to be consistent with the general theme of the GUI. To get the correct amount of tick counts for the BoxPlot the tick count is set to the length of the data prop.

The implementation for the GateErrorVisualization component is presented in appendix F and is a parent component to the BoxPlot component. This component is responsible for fetching and reshaping the fetched data into the correct shape for the BoxPlot component. React Query is used to fetch the type 3 data which is the gate error of a qubit. The useContext hook is used to access the selected timespan. The Boxplot component requires the data to be on the following shape:

```
[{ x: <THECATEGORY>, y: [<THE ACTUAL VALUES>] }]
```

Listing 4.5: The required shape of the data for the BoxPlot component.

The x value in our case is the qubit ID, and the y value is an array containing all the corresponding gate error values for the specific qubit ID over the selected timespan. To reshape the fetched data, we call the map() function to return a new array of objects, where the x key contains the qubit ID value and the y key contains an array that contains every gate error value for the respective qubit.

The line graph is a simpler component that just draws lines in between data points. This component was implemented using visx. Most of the work was spent retrieving data from a chosen backend and letting the user choose a key that manipulates the displayed data. We have another view that is the same as the line graph but displays the data in another format, the Table view. Made in the same way, but just formatting the data differently.

The city plot is another visualization we made. It is a three-dimension box plot with i j and z axes. i and j represent the index of a coupler in a matrix, and the z value is the $xtalk\_\{i,j\}$ which is a value that represents unintended consequences or influence between $coupler_i$ and $coupler_j$. This component, unlike the rest, is made with a very friendly library called react-graph3d-vis. Not a lot of work for us, since the library generates the 3d box plot by giving it the correct data in the correct format. Below is a code snippet from how we generate the city plot using react-graph3d-vis.

```
    return (
        <Flex flexDir='column' alignItems='center'>
            <Graph3D
```

```
4                   data={data1}
5                   options={{
6                       width: "600px",
7                       height: "500px",
8                       style: "bar",
9                       tooltip: true,
10                      keepAspectRatio: true,
11                      verticalRatio: 1,
12                      animationInterval: 1,
13                      animationPreload: true,
14                      xLabel: "i",
15                      yLabel: "j",
16                      zLabel: "xtalk(i,j)",
17                      cameraPosition: {
18                          distance: 3
19                      }
20                  }}
21              />
22          </Flex>
23      );
```

Listing 4.6: Generating the cityplot.

Flex is just a regular chakraUI flexbox and Graph3D is the component you import from the react-graph3d-vis library that automatically generates a 3d city plot given the above options and most importantly using data which is a list with (x, y, z) values.

4.4 Testing

In the end-to-end tests, the general flow of the application that a user might have was tested, as well as the structure and the contents of the page. Most of the tests used CSS data attributes as selectors, which was recommended in the cypress documentation [53]. Fixtures were used extensively throughout the tests to mock data and API's. The data in the fixtures were copied from the API provided by the backend group.

The API mocking was done by using the built-in 'cy.intercept' function from cypress. An example of how this was done can be seen in figure **??**.

The beforeEach is a cypress function that runs before each test and is usually used for mocking data. We use the beforeEach function to mock the backend API using 'cy.intercept'. The first function call, on line 2, mocks requests to '<apiUrl>/devices' and responds with the data in the device's fixture. Where '<apiUrl>' is the base URL to the API made by the backend group. The ApiRoutes is a javascript object that maps to the backend API routes. We made this object so that all the URLs would be in a single place and easier to change. By mocking the API, the tests become consistent and reproducible.

Due to how we defined testing coverage for this project, we first needed to define the features we wanted for each page. These features are defined after the specifications that we were given.

The following elements were specified for the overview page.

- Navbar.
- WACQT description card.
- Card with the number of systems online.
- Search field.
- Sort button.
- Filter button.
- Grid with cards of the quantum computer configurations.

All of these elements should exist on the overview page. The only exception is the grid with the cards of the quantum computer configurations, which could be empty depending on the state of the search bar and filter button. All the elements should also display the correct content.

The following behaviours were defined for the overview page.

- When the text in the navbar is clicked, the browser should be routed to the overview page.
- When the search field has been typed in, the quantum computer configurations should be filtered based on the content of the search field. The quantum computer configurations should be filtered by their name.
- On the sort button, when order and option have been selected, the quantum computer configurations should be sorted in the order depending on the order selected and by the property selected as an option.
- On the filter button, when the online option is selected, quantum computer configurations that are online should be shown. If the offline option is selected, offline quantum computer configurations should be shown.

We wrote the following test, which can be seen in figure G.2, to assert that the cards with the quantum computer configurations are rendered correctly.

The test is one of several tests that run on the overview page. It first loads a fixture, 'devices.json.'. It then attempts to select the grid with the quantum computer configurations, which has the selector 'data-cy-devices'. It then selects the first card of the grid and checks that the name, status, version, number of qubits, and last update date of the specific quantum computer configuration matched the data in the fixture.

Similarly, we wrote tests for the rest of the specifications of the overview page. Tests were made to assert that all elements existed on the page and that their content was correct. Additionally, tests were made to assert that the elements had the specified behaviour.

Using the requirements stated in chapter 3, we created specifications for each view. We then extrapolated tests from the specifications similarly as exemplified in the paragraphs above.

The qubit map visualization had the most tests because it was the most complex of the visualization. Mainly due to it being a custom type of visualization. We therefore had to make a plethora of tests to ensure that it behaved as intended.

```
it('renders qubits in the correct positions', () => {
    cy.viewport(1440,900);
  cy.fixture('qubitPositions.json').then((positions) => {
    positions.forEach((position, index) => {
      cy.get(`[data-cy-qubitmap-node-id=${index}]`).within(() =>
 {
        cy.get('rect')
          .should('have.attr', 'x', position.x)
          .and('have.attr', 'y', position.y);
      });
    });
  });
});
```

Listing 4.7: A test asserting that the qubits on the qubitmap have the correct positions

Figure 4.7 illustrates one of those tests. It first loads a fixture containing the positions that the qubits should have. Since these positions are constant, the test also requires a fixed viewport of 1440 times 900 pixels. We achieve this by using the c̈y.viewportf̈unction from cypress. It then loops over the positions and searches for a qubit with a specific ID. It then checks the position of the said qubit. These kinds of extensive tests were necessary due to the nature of the component. The tests could have been improved by testing at multiple screens, which would, in turn, require more fixtures.
For the other visualizations, we assumed that the charting libraries used where actually correct and simply tested that the different inputs on the page behaved correctly.

```
it('displays histogram t2', () => {
  cy.visit('/Pingu?type=Histogram');
  cy.wait(3000).then(() => {
    cy.get('[data-index=1]').click();
    cy.get("[data-cy-histogram='T2(us)']").should('exist');
  });
});
```

Listing 4.8: A test asserting that the correct chart is displayed when the T2 radio button is clicked

For example, in figure 4.8, a test can be seen that asserts that after clicking the T2 button in the histogram visualization, the T2 chart is displayed on the screen.

5
Results

This chapter will present the final product we implemented and how it can be deployed, as well as the results from letting a physics researcher from WACQT use the application.

5.1 Final product

In figure H.1 the homepage is displayed, and all quantum computer configurations are listed in a grid view. On the top left there is an info card about WACQT and on the top right an info display on number of online versus offline systems.

In figure H.2 there is an info component that displays information such as name, version, sample date and more about the chosen configuration. On the right side, a map over the chosen key is displayed. The user can also choose which property to display. When hovering over a displayed key in the grid a tooltip is displayed, this can be seen in figure H.3

Figure H.4 displays the histogram component which visualizes T1, T2, Tϕ values over a specific timestamp and qubit. Which qubit is chosen is displayed in the info card on the left side.

Displayed below in figure H.5 is the line graph, which can display any several combinations of keys for different components.

In the figure H.6 a box plot of the gate errors for each qubit over a specific time span is visualized.

Displayed in figure H.7 is the table component which displays a multiple data points for qubits, gates, couplers and resonators.

In the figure H.8 is the city plot component.

In the figure H.9 is the skeleton, mentioned 3. Not seen in the figure is the breathing animation, where the skeleton goes from darker to brighter shades of grey.

5.2 Testing

We achieved 100% test coverage as all the application features were tested. The features specified are based on the requirements that we were given, and we, therefore, believe that they are sound and fully cover the application. The results of the tests can be seen in the figure below, figure H.10.

5.3 User Tests

A user test was performed by a physics researcher that designed the quantum chip currently used at the Chalmers quantum computer. The test was conducted to get general feedback about the application, but mainly focused on the user-friendliness of the design. The test process started on the first page, and we let him see what was there. Then we explained everything that you can do on that page and what can be done if we had more than one backend. We then went through all the different visualizations and got feedback on them. The feedback we received from the researcher will be displayed here in the same order we went through the test.

The homepage with the quantum computer backends was good and easy to understand how to use and could not be improved in any major way. He thought that the ability to sort, filter, and search the backends was good and future proof-functionality for when more backends will be available.
We then navigated to the qubit map, where we got the following feedback:

- ID in the tooltip for each qubit might be redundant and not tell the user anything important.
- Making the tooltip customizable so that the data that that being displayed is useful for the current user, as all users do not need all the information.
- Related to the last point is that a login system would also be a solution because of different types of users for different kinds of information.
- Overall good and easy to use design, and the ability to choose what data is displayed on the qubit is advantageous.

Then we navigated to the histogram visualization, where we got the following feedback:

- Useful visualization for getting an overview over time and easy-to-use layout.
- Bug found if unavailable time is chosen, it crashed.
- Would be nice to be able to export the visualization to a PDF or similar format.
- The qubit navigator used to display what qubit is selected or to select another one was very nice because it gave an easy way to see the selected qubit and switch qubits.

Then we navigated to the box plot visualization, where we got the following feedback:

- Would be nice to see the qubit grid used on the histogram visualization on the left side with the IDs of the qubits.

- Make it scrollable if more qubits are added instead of compressed.
- Functionality to export the visualization same as the previous one.

Then we navigated to the line chart visualization, where we got the following feedback:

- Domain and Codomain is a little confusing and should probably use x and y instead to make it more readable.
- Otherwise, useful to see correlations between different types of data.

Then we would have navigated to the table view visualization, but we had forgotten to merge this to the main branch of the code, so we instead explained it. Fortunately, it is the most straightforward visualization as it is just a table of all values, but we got the following feedback:

- Good to have and useful for seeing all kinds of values easily but does not add that much value as you can see all the values in other visualizations.
- Export functionality would be good.

Then we navigated to the city plot visualization, where we got the following feedback:

- Useful for understanding a certain state and seeing outliers, as it is easy to see height differences.
- A heatmap would probably be more user-friendly for experienced users, but the city plot is good for an average user.
- Hard to export a city plot to a PDF or image, so that is another reason for a heat map.

These were all the different pages and functionality we showed and got feedback on. Some general overall feedback we also got was:

- Excellent and exactly what he wants; with some small changes, it can work perfectly for what he needs from it at the moment.
- Displaying the units on the plots and qubits is confusing with just a huge number and not having the unit.
- Showing the bound of the values in a certain plot to see if it is very off or not.

The conclusions of this user test will be discussed in chapter 6 under subsections User-friendliness and Future work.

6
Discussion

In this chapter, we discuss the outcome of the project and how well we managed to reach our goals defined in the section 1.3. We also discuss the ethical, economic, and social aspects of this project and quantum technology overall, and how this project can be further developed in the future.

6.1 Features implemented

We managed to implement all planned features and fulfilled the requirements we had set up for each version of the project. We also managed to create visualizations for all key-value pairs we got from the backend.

6.2 Maintainability

As the code developed and the libraries used during development were chosen with maintainability in mind, the codebase is easy to keep building on and developing. React and Next.js are both very popular and well documented and used as our primary frameworks used for developing the frontend in the project means that most developers will already have some knowledge about these frameworks or easily be able to get into them and start maintaining the codebase. The libraries used for all our visualizations are also well documented and not too complex to get into if something should be added or changed later.

The backend endpoints that we use and helped come up with are constructed to be used for a completely different frontend without having to do any significant data manipulations on the frontend side.

6.3 User friendliness

When designing the application, the group followed UI conventions and best practices. Much of this was already done by the UI framework that was used. The relatively small scope of the application facilitated a user-friendly and straightforward design.

After the designs were done in Figma, they were sent to the group's supervisors. Any feedback was used to change the design before it was implemented. Although

these were not proper user tests, the feedback given was valuable to the project's development. It improved the overall user-friendliness of the application.

The feedback we got during the user-test interview presented in chapter 5 was mainly positive. The design was easy to interpret, and the researchers are getting used to these interfaces, so it will not be hard to get used to. Most of the feedback we got about what could be changed was not so much user-friendliness related in a design sense, but more quality of life functionality that would be nice to have. For example, being able to save and export the visualizations to PDF, image or similar formats would be nice because a lot of these visualizations will be used to show to investors or put in research papers. Some of this feedback will be discussed later under the section future work, as it won't be implemented during this project. We learned from this user test that we should probably have performed more of these earlier because then we could have implemented some easier functionality and made the application more useful already. As well that user tests are a great way of finding bugs. We only found one bug during the user test, but if we had not done the test we would most likely not had found out that picking a time that had no values on the histogram crashed it, this was a good indication that the test suite might not cover all the edge-cases we thought. This was something we took in consideration when further expanding the test suite after the user test.

6.4 Ethical, economic and social aspects

The decision to develop this project as an open-source web application with a permissive license will increase trust and transparency between developers involved in the further development of quantum computers. The web-based graphical user interface enables users to assess the capabilities of quantum computer technology better, as it enables the data to be accessed through the web instead of only being accessible through the lab. To develop the web-based graphical user interface as open-source with the addition of a permissive license is both fair and non-discriminatory for the developers that want to contribute to the project. On the other hand, as the Apache 2.0 license [54] allows the licensed software to be used in commercial products free of charge, one negative aspect tied to the decision to open-source the project is the question of fair monetary compensation. For instance, larger corporations can use the licensed web application to generate revenue without ensuring that the open-source developers behind the project will be monetarily compensated for their effort to develop the open-source software.

Another relevant aspect is the implications of the development of quantum computers in the context of post-quantum cryptography. Post-quantum cryptography is defined as cryptography where a potential attacker has access to a sizeable quantum computer [55]. Furthermore, conventional encryption schemes such as RSA and ECC will be broken if quantum computing scales as expected. On the other hand, the authors emphasize the need to build cryptography systems where quantum algorithms do not significantly improve speed. Even though our web GUI does not directly relate to the development of post-quantum cryptography, our project

contributes to the development of quantum computers. Therefore, we do find the implications of post-quantum cryptography relevant to our project.

Quantum computer research has shown that quantum computers are more effective for simulating certain areas of physics and chemistry, as these processes are often best described by quantum mechanics [8]. This property of quantum computers has applications within material science, chemistry, pharmaceutical science, physics, and more and could lead to significant progress within these fields [8]. An example of this efficacy of simulating these quantum mechanics is that the computation of a complete electronic wave function of an average drug molecule is calculated to take longer than the age of the universe on our current supercomputers. However, it is theorized to take days on modest quantum computers. Thus, due to our project contributing to the development of quantum computers, we believe our project to be relevant to the aspects discussed above.

6.5 Future work

After this project is concluded, there are still many improvements that can be made and further functionality that can be added. One of the first improvements that could be added is more visualizations that the experimentalists and researchers find helpful. Another improvement is to make the website more mobile-friendly, so users can see the data on the phone easier by having a design optimized for the phone, as the current visualizations are not that clear on the phone. Viewing the latest data offline by caching what the user previously looked at is a further improvement that could be added and might be beneficial if the user somehow loses internet connection.

A major update to the software would be the ability to input Qiskit files/commands to the quantum computer and run it on that machine via the software. Essentially turning the software from a view-only platform to a whole read/write interface for multiple quantum computers.

After getting feedback from the user test mentioned earlier in this chapter, the primary and somewhat easily addable functionality that the researcher would want is to save and export the plots and all the necessary data to use in research papers or show them to investors. Another feature that he thinks would improve the system and is necessary when making it open to the public is having some login/authentication system and different types of users. This is because some information displayed is sensitive and can be used to figure out how the actual chip is constructed and can then be hacked. This information should only be visible for superusers and not an average external user using the system.

Something very important to both us, the developers and the future engineers working on this project is the extensibility of the project. How hard is it really to add a new visualization, and what changes to the UI and backend would be needed. From the API, adding a new endpoint is trivial and forces no major changes to endpoints already existing other than naming conventions. In the frontend, however, adding

another tab in the detail view could render rather difficult depending on a lot of factors. The functionality is very simple but other than being able to function it should also look good and be visually appealing for a user which is the harder part.

6.6 Deployment

As the application is based on the Next.js framework, it is easily deployable without any specific configurations. The easiest way to deploy it is by using Vercel [56], which is a hosting service that takes care of all the configurations needed to deploy the Next-application. It can easily be deployed to any hosting service as long as they support Node.js which all primary hosting services do. Additionally, we also containerized the application using docker, which should make it easier to deploy. We will, however, not deploy it anywhere, but as the code is meant to be used by WACQT, they can quickly deploy it if they wish.

As the application is containerized, it can relatively easily be deployed on AWS [57] using ECS [58] and ECR [59]. We are not experts on AWS, and the following example is meant to illustrate how one could deploy to AWS, but it may not be the best way of doing so. We strongly recommend reading Next's deployment documentation and AWS ECS [60] and ECR documentation [61]. To deploy the application, we must first build an image and upload it to ECR. We start by creating an ECR repository. After that, we can see the commands needed to build and upload the docker image by clicking "View push commands". Note that to build the docker image, you must clone the application's GitHub repository and run the build command inside the resulting folder.

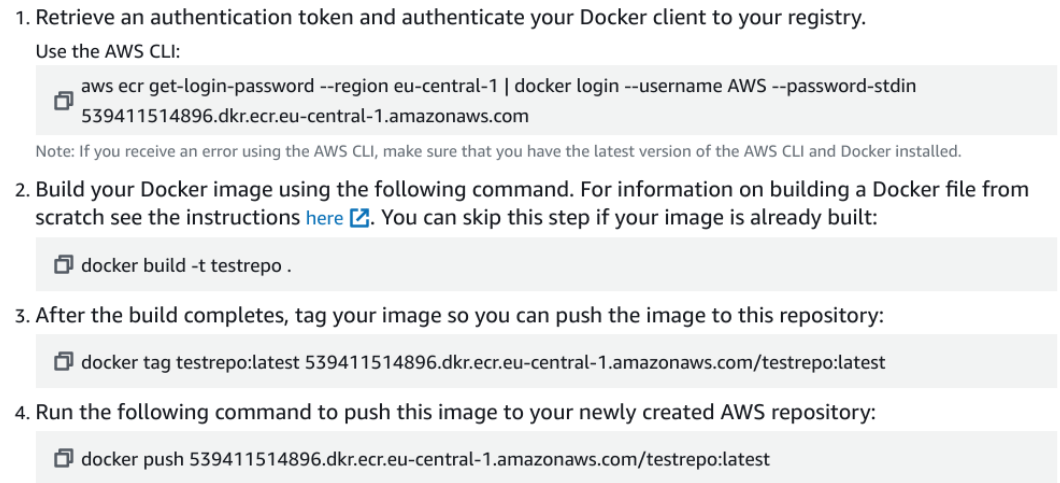

Figure 6.1: ECR upload commands.

We can then go to ECS and create a cluster, select ËC2 Linux + Networking äs the cluster template. The rest of the cluster settings can be left as default. After the cluster has been created, we need to create a task definition and select EC2 as the l̈aunch compatibility.̈ Select bridge as the network mode. Under container

definitions, click add container and write the image URI found in the ECR repository that we created. Under port mappings, input 80 as the host port and 3000 as the container port.

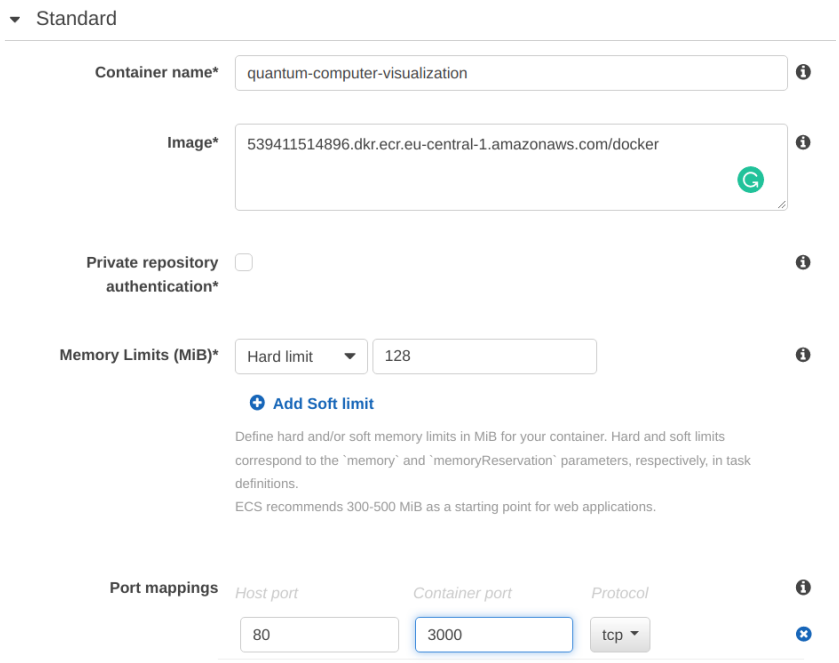

Figure 6.2: Task definition container settings.

Finally, we can go to the cluster we created, and under the tasks tab, we can click run new task. Select ec2 as the launch type and select the task definition we just created. We can now click on our task and view our container instance under containers. Click on the container instance, and under network binding, we can view the IP of the container hosting the application.

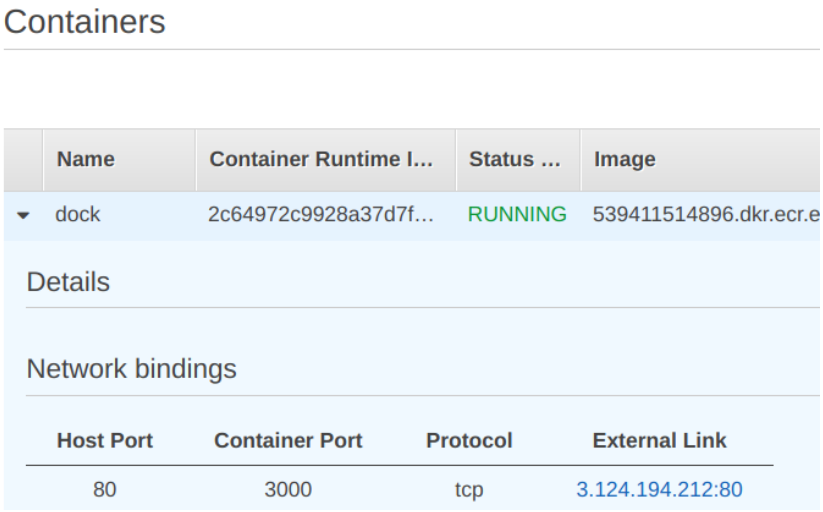

Figure 6.3: ECS cluster container.

7
Conclusion

The main purpose of the project was to create a web interface for a quantum computer. And use this web interface to display data and visualize it in meaningful ways. This was achieved, and the web interface supports five different visualizations to achieve this, all of this presented in the results chapter 5.

A big part of the project was also to make sure that we could give this codebase to other developers as an open-source project and make sure it's maintainable. This was also achieved by using the libraries and frameworks we chose for developing the web application.

To conclude this project, we think it would qualify as a success and hope that it will be further developed to one day be used for real as the main web interface for WACQT quantum computers.

Bibliography

[1] CHALMERS UNIVERSITY OF TECHNOLOGY. *WACQT | Wallenberg Centre for Quantum Technology*. Available: https://www.chalmers.se/en/centres/wacqt. 2020. (Published: 2022-02-10).

[2] Scrum.org. *What is Scrum?* Available: https://www.scrum.org/resources/what-is-scrum. 2022.

[3] Ecma International. *ECMA-404 The JSON data interchange syntax*. Available: https://www.ecma-international.org/publications-and-standards/standards/ecma-404/. 2022. (Published: 2022-02-10).

[4] Lokesh Gupta. *What is REST*. Available: https://restfulapi.net. 2022.

[5] git-scm.com. *GIT*. Available: https://git-scm.com/. 2022.

[6] github.com. *GitHub*. Available: https://github.com/. 2022.

[7] Noson Yanofsky. *An Introduction to Quantum Computing*. 2007, p. 33. DOI: 10.1007/978-94-007-0080-2_10.

[8] Bela Bauer et al. "Quantum Algorithms for Quantum Chemistry and Quantum Materials Science". In: *Chemical Reviews* 120.22 (2020). PMID: 33090772, pp. 12685–12717. DOI: 10.1021/acs.chemrev.9b00829. eprint: https://doi.org/10.1021/acs.chemrev.9b00829. URL: https://doi.org/10.1021/acs.chemrev.9b00829.

[9] Jay Gambetta Jerry Chow Oliver Dial. *IBM Quantum breaks the 100-qubit processor barrier*. Ed. by research.ibm.com. Research.ibm.com [Online; posted 16-November-2021]. Nov. 2021.

[10] Julian Kelly. *A Preview of Bristlecone, Google's New Quantum Processor*. Ed. by Quantum AI Lab. Ai.googleblog.com [Online; posted 05-March-2018]. Mar. 2018.

[11] Rigetti co. *Rigetti Computing*. Available: https://www.rigetti.com/. 2022.

[12] Calif Berkeley. "Microsoft to Bring Rigetti Superconducting Quantum Computers to Azure Quantum". In: *Sovetskoye Radio, Moscow* (2021). Available: https://www.globenewswire.com/news-release/2021/12/06/2346638/0/en/Microsoft-to-Bring-Rigetti-Superconducting-Quantum-Computers-to-Azure-Quantum.html.

[13] IBM. *IBM Quantum Computing*. Available: https://quantum-computing.ibm.com/. 2022.

[14] Gadi Aleksandrowicz et al. *Qiskit: An Open-source Framework for Quantum Computing*. Version 0.7.2. Jan. 2019. DOI: 10.5281/zenodo.2562111. URL: https://doi.org/10.5281/zenodo.2562111.

[15] Google. *Google Quantum Ai*. Available: https://quantumai.google/. 2022.

[16] Cirq Developers. *Cirq*. Version v0.12.0. See full list of authors on Github: https://github .com/quantumlib/Cirq/graphs/contributors. Aug. 2021. DOI: 10.5281/zenodo.5182845. URL: https://doi.org/10.5281/zenodo.5182845.

[17] Quantum AI team and collaborators. *ReCirq*. Oct. 2020. DOI: 10.5281/zenodo.4091470. URL: https://doi.org/10.5281/zenodo.4091470.

[18] Jarrod R McClean et al. "OpenFermion: the electronic structure package for quantum computers". In: *Quantum Science and Technology* 5.3 (June 2020), p. 034014. DOI: 10.1088/2058-9565/ab8ebc. URL: https://doi.org/10.1088/2058-9565/ab8ebc.

[19] Michael Broughton et al. *TensorFlow Quantum: A Software Framework for Quantum Machine Learning*. 2020. DOI: 10.48550/ARXIV.2003.02989. URL: https://arxiv.org/abs/2003.02989.

[20] Quantum AI team and collaborators. *qsim*. Sept. 2020. DOI: 10.5281/zenodo.4023103. URL: https://doi.org/10.5281/zenodo.4023103.

[21] Microsoft Corporation. *Azure*. Available: https://azure.microsoft.com/. 2022.

[22] Microsoft Corporation. *Azure Quantum Services*. Available: https://azure.microsoft.com/en-us/services/quantum. 2022.

[23] U.S General Services Administration. *User Interface Design Basics*. Available: https://www.usability.gov/what-and-why/user-interface-design.html. 2022.

[24] Jenifer Tidwell, Charles Brewer, Aynne Valencia. *Designing Interfaces: Patterns for Effective Interaction Design, Third Edition*. O'Reilly Media, 2019. ISBN: 978-1492051961.

[25] Patrick Faller. *Putting Personas to Work in UX Design: What They Are and Why They're Important*. Available: https://xd.adobe.com/ideas/process/user-research/putting-personas-to-work-in-ux-design/. 2019.

[26] The Blender Foundation. *Blender*. Available: https://www.blender.org/. 2022.

[27] U.S General Services Administration. *User Interface (UI) Design Patterns*. Available: https://www.interaction-design.org/literature/topics/ui-design-patterns. 2022.

[28] Deepak Parmar. *Exploratory testing*. Available: https://www.atlassian.com/continuous-delivery/software-testing/exploratory-testing#link. 2022.

[29] Sten Pittet. *The different types of software testing*. Available: https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing. 2022.

[30] Cypress.io. *Cypress*. Available: https://www.cypress.io/. 2022.

[31] Cindy Tittle Debra Jane Richardson Owen Oalley. "Approaches to specification-based testing". In: *ACM SIGSOFT Software Engineering Notes* 14.8 (1989), pp. 86–96.

[32] Miroslav Dobsicek, Sandro Stucki. *WebGUI for quantum computer: A guide for version 1*. Private correspondence. 2022. (Published: 2022-02-03).

[33] Miroslav Dobsicek, Sandro Stucki. *bachelor-webgui-project-2022-04-17-keys-v2-guide.pptx*. Private correspondence. 2022.
[34] Page Laubheimer. *Cards: UI-Component Definition*. Available: https://www.nngroup.com/articles/cards-component/. 2022. (Published: 2016-11-06).
[35] Vercel. *Next.js*. Available: https://nextjs.org/. 2022.
[36] Tanner Linsley. *React-Query*. Available: https://react-query.tanstack.com. 2022.
[37] Vlad Dragos. *API Proposal*. Available: https://wax-rhodium-260.notion.site/API-Proposal-30d083d9ef6e405995916adc1ad25f87. 2022.
[38] Inc. Box. *BOX*. Available: https://www.box.com/. 2022.
[39] Atlassian. *Trello*. Available: https://trello.com/. 2022.
[40] Inc. Figma. *Figma*. Available: https://www.figma.com/. 2022.
[41] Meta Platforms, Inc. *A JavaScript library for building user interfaces*. Available: https://reactjs.org. 2022. (Published: 2022-02-05).
[42] Microsoft. *A JavaScript with syntax for types*. Available: https://www.typescriptlang.org. 2022. (Published: 2022-02-05).
[43] cypress.io. *A JavaScript testing framework*. Available: https://www.cypress.io/. 2022. (Published: 2022-02-08).
[44] Inc Meta Platforms. *Jest*. Available: https://jestjs.io/. 2022.
[45] Kent C. Dodds. *React Testing Library*. Available: https://testing-library.com/docs/react-testing-library/intro/. 2022.
[46] chakra-ui.com. *Chakra*. Available: https://chakra-ui.com/. 2022. (Published: 2022-02-11).
[47] W3C. *Accessible Rich Internet Applications (WAI-ARIA) 1.1*. Available: https://www.w3.org/TR/wai-aria-1.1/. 2022. (Published: 2022-02-11).
[48] Charts.js. *Simple yet flexible JavaScript charting for designers developers*. Available: https://www.chartjs.org/. 2022. (Published: 2022-02-24).
[49] Formidable. *React.js components for modular charting and data visualization.* Available: https://formidable.com/open-source/victory/. 2022. (Published: 2022-04-28).
[50] Airbnb. *visx a collection of expressive, low-level visualization primitives for React*. Available: https://airbnb.io/visx/. 2022. (Published: 2022-02-24).
[51] Airbnb. *@visx/network*. Available: https://airbnb.io/visx/docs/network. 2022.
[52] Meta Platforms, Inc. *Context - React*. Available: https://reactjs.org/docs/context.html. 2022. (Published: 2022-04-28).
[53] Cypress.io. *Best Practices*. Available: https://docs.cypress.io/guides/references/best-practices#Selecting-Elements. 2022.
[54] apache.org. *APACHE LICENSE, VERSION 2.0*. Available: https://www.apache.org/licenses/LICENSE-2.0. 2022. (Published: January 2004).
[55] Daniel J. Bernstein Tanja Lange. *Post-quantum cryptography*. Available: https://www.nature.com/articles/nature23461. 2022. (Published: 2017-10-17).
[56] Vercel Inc. *Vercel*. Available: https://vercel.com/. 2022.
[57] Inc. Amazon Web Services. *Cloud Computing Services - Amazon Web Services (AWS)*. Available: https://aws.amazon.com/. 2022.

[58] Inc. Amazon Web Services. *Fully Managed Container Solution – Amazon Elastic Container Service (Amazon ECS) - Amazon Web Services*. Available: `https://aws.amazon.com/ecs/`. 2022.

[59] Inc. Amazon Web Services. *Fully Managed Container Registry – Amazon Elastic Container Registry – Amazon Web Services*. Available: `https://aws.amazon.com/ecr/`. 2022.

[60] Inc. Amazon Web Services. *Amazon Elastic Container Service Documentation*. Available: `https://docs.aws.amazon.com/ecs/index.html`. 2022.

[61] Inc. Amazon Web Services. *Amazon Elastic Container Registry Documentation*. Available: `https://docs.aws.amazon.com/ecr/`. 2022.

A

Appendix 1 - Radio button implementation

```
/*
To get current tab in parent element pass a setState function as a prop
Example:
  const [tab, setTab] = useState('1')
  <RadioButtons setTab={setTab} tabs={['1', '2', '3']} />
*/
interface RadioButtonsProps {
  tabs: string[];
  setTab: (value: string) => void;
  defaultTab?: string;
}
const RadioButtons = ({ setTab, tabs, defaultTab}: RadioButtonsProps) => {
  return (
    <Box
      borderRadius='full'
      border='1px'
      borderColor='grey'
      p='1'
      m='2px'
      w='fit-content'
      data-cy-radiobutton
    >
      <Tabs
        variant='soft-rounded'
        onChange={(index) => setTab(tabs[index])}
        defaultIndex={defaultTab ? tabs.indexOf(defaultTab) : 0}
      >
        <TabList>
          {tabs.map((item, index) => (
            <Tab
              key={index}
              _selected={{ color: 'white', bg: '#38B2AC', boxShadow: 'none' }}
              >
              {item}
            </Tab>
          ))}
        </TabList>
      </Tabs>
```

```
39      </Box>
40    );
41  };
```

Listing A.1: Radiobutton implementation

B

Appendix 2 - Date picker implementation

```
interface DatePickerProps {
  refetchFunction: () => void;
}

const DatePicker: React.FC<DatePickerProps> = ({ refetchFunction }) => {
  const [state, dispatch] = useContext(BackendContext);

  const [range, setRange] = useState([
    {
      startDate: state.timeFrom,
      endDate: state.timeTo,
      key: 'selection',
      color: '#38B2AC'
    }
  ]);

  const handleChange = (item) => {
    setRange([item.selection]);
    dispatch({ type: DateActions.SET_TIME_FROM, payload: item.selection.startDate });
    dispatch({ type: DateActions.SET_TIME_TO, payload: item.selection.endDate });
  };

  const parseDates = () => {
    return (
      state.timeFrom.toDateString().slice(4, 10) +
      ' - ' +
      state.timeTo.toDateString().slice(4, 10)
    );
  };

  return (
    <Box data-cy-date-picker>
      <Popover onClose={() => refetchFunction()} data-cy-date-picker-test>
        <PopoverTrigger>
          <Button
            boxShadow='4px 4px 2px 1px rgba(0, 0, 0, .1)'
            _focus={{ outline: 'none' }}
```

```
38            >
39              Period: {parseDates()}
40            </Button>
41          </PopoverTrigger>
42          <PopoverContent _focus={{ outline: 'none' }}>
43            <PopoverArrow />
44            <Box>
45              <DateRange
46                editableDateInputs={true}
47                onChange={(item) => handleChange(item)}
48                moveRangeOnFirstSelection={false}
49                ranges={range}
50                locale={sv}
51                maxDate={new Date()}
52                weekStartsOn={1}
53              />
54            </Box>
55          </PopoverContent>
56        </Popover>
57      </Box>
58    );
59  };
```

Listing B.1: Date picker implementation.

C

Appendix 3 - HistogramVisualization implementation

```
interface HistogramVisualizationProps {
  backend: string | string[];
}

export const HistogramVisualization: React.FC<
    HistogramVisualizationProps> = ({ backend }) => {
  const [state, dispatch] = useContext(BackendContext);
  const { setSelectionMap } = useSelectionMaps();

  const { isLoading, data, error, refetch, isFetching } = useQuery(
    'histogramData', () =>
    fetch(
      'http://qtl-webgui-2.mc2.chalmers.se:8080/devices/' +
        backend +
        '/type2/period?from=' +
        state.timeFrom.toISOString() +
        '&to=' +
        state.timeTo.toISOString()
    ).then((res) => res.json())
  );

  const [dataToVisualize, setDataToVisualize] = useState<string>('
    T1');

    useLayoutEffect(() => {
        setSelectionMap(false, true);
        if (state.selectedNode === -1) {
          dispatch({ type: MapActions.SELECT_NODE, payload: 0 });
        }
      }, []);

  if (error) return <span>Error</span>;

  if (isLoading || isFetching) return <VisualizationSkeleton />;

  return (
    <Box>
      <Flex flexDir={'row'} align={'center'} p={3}>
        <Box ml={'3em'}>
```

```
37          <RadioButtons
38            setTab={setDataToVisualize}
39            tabs={['T1', 'T2', 'T' + '\u03C6']}
40          ></RadioButtons>
41        </Box>
42        <Spacer />
43        <Box mr='3em'>
44          <DatePicker refetchFunction={refetch}></DatePicker>
45        </Box>
46      </Flex>
47      {dataToVisualize === 'T1' && (
48        <Histogram
49          data={data.qubits[state.selectedNode].qubit_T1.map((
    t1data) => ({
50            x: t1data.value * 1000000
51          }))}
52          label='T1(us)'
53          data-cy-histogram-t1-clicked
54        ></Histogram>
55      )}
56      {dataToVisualize === 'T2' && (
57        <Histogram
58          data={data.qubits[state.selectedNode].qubit_T2_star.map((
    t2data) => ({
59            x: t2data.value * 1000000
60          }))}
61          label='T2(us)'
62          data-cy-histogram-t2-clicked
63        ></Histogram>
64      )}
65      {dataToVisualize === 'T' + '\u03C6' && (
66        <Histogram
67          data={data.qubits[state.selectedNode].qubit_T_phi.map((
    t2data) => ({
68            x: t2data.value * 1000000
69          }))}
70          label='TPhi(us)'
71          data-cy-histogram-tphi-clicked
72        ></Histogram>
73      )}
74     </Box>
75   );
76 };
```

Listing C.1: The implementation of the HistogramVisualization component.

D

Appendix 4 - Histogram implementation

,

```
interface elementShape {
  x: number;
}

interface HistogramProps {
  data: elementShape[];
  label: string;
}

const Histogram: React.FC<HistogramProps> = ({ data, label }) => {
  return (
    <VictoryChart data-cy-histogram={label}>
      <VictoryLabel
        x={400}
        y={280}
        textAnchor={'middle'}
        text={label}
        style={[{ fill: '#374151', fontSize: 12 }]}
      />
      <VictoryAxis
        tickCount={7}
        style={{
          axis: { stroke: '#9CA3AF' },
          tickLabels: { fontSize: 12, padding: 5, fill: '#9CA3AF' }
        }}
      />
      <VictoryAxis
        dependentAxis
        style={{
          axis: { stroke: 0 },
          tickLabels: { fontSize: 12, padding: 5, fill: '#9CA3AF'
  },
          grid: { stroke: '#9CA3AF', strokeWidth: 1,
  strokeDasharray: '8' }
        }}
      />

      <VictoryHistogram
        cornerRadius={0}
        binSpacing={3}
        style={{
```

```
40            data: {
41              fill: '#9CE0DD',
42              stroke: 0
43            }
44          }}
45          data={data}
46        />
47      </VictoryChart>
48    );
49  };
50
51  export default Histogram;
```

Listing D.1: The implementation of the histogram component

E

Appendix 5 - BoxPlot implementation

```
,
interface boxPlotElementShape {
  x: number;
  y: number[];
}

interface BoxPlotProps {
  data: boxPlotElementShape[];
}

const BoxPlot: React.FC<BoxPlotProps> = ({ data }) => {
  return (
    <VictoryChart data-cy-box-plot>
      <VictoryLabel
        x={40}
        y={30}
        textAnchor={'middle'}
        text={'Gate error'}
        style={[{ fill: '#374151', fontSize: 12 }]}
      />

      <VictoryLabel
        x={400}
        y={280}
        textAnchor={'middle'}
        text={'Qubit index'}
        style={[{ fill: '#374151', fontSize: 12 }]}
      />
      <VictoryAxis
        tickCount={data.length}
        crossAxis={false}
        style={{
          axis: { stroke: '#9CA3AF', strokeDasharray: '8' },
          tickLabels: { fontSize: 12, padding: 5, fill: '#9CA3AF'
  },
          grid: { stroke: '#9CA3AF', strokeWidth: 1,
  strokeDasharray: '8' }
        }}
      />
      <VictoryAxis
        dependentAxis
        style={{
```

```
40            axis: { stroke: 0 },
41            tickLabels: { fontSize: 12, padding: 5, fill: '#9CA3AF'
    },
42            grid: { stroke: '#9CA3AF', strokeWidth: 1,
    strokeDasharray: '8' }
43          }}
44        />
45
46        <VictoryBoxPlot
47          boxWidth={8}
48          data={data}
49          style={{
50            min: { stroke: '#366361' },
51            max: { stroke: '#38B2AC' },
52            q1: { fill: '#366361' },
53            q3: { fill: '#38B2AC' },
54            median: { stroke: 'white', strokeWidth: 2 }
55          }}
56        ></VictoryBoxPlot>
57      </VictoryChart>
58    );
59  };
60  export default BoxPlot;
```

Listing E.1: The implementation of the BoxPlot component

F

Appendix 6 - GateErrorVisualization implementation

,

```
interface GateErrorVisualizationProps {
  backend: string | string[];
}

export const GateErrorVisualization: React.FC<
    GateErrorVisualizationProps> = ({ backend }) => {
  const [state, dispatch] = useContext(BackendContext);
  const { setSelectionMap } = useSelectionMaps();

  const { isLoading, data, error, refetch, isFetching } = useQuery(
    'gateError', () =>
    fetch(
      'http://qtl-webgui-2.mc2.chalmers.se:8080/devices/' +
        backend +
        '/type3/period?from=' +
        state.timeFrom.toISOString() +
        '&to=' +
        state.timeTo.toISOString()
    ).then((res) => res.json())
  );
  useLayoutEffect(() => {
    setSelectionMap(false, false);
  }, []);

  if (error) return <span>Error</span>;
  if (isLoading || isFetching) return <VisualizationSkeleton />;

  return (
    <Box>
      <Flex flexDir={'row'} align={'center'} p={3}>
        <Box ml={'auto'} mr={'3em'}>
          <DatePicker refetchFunction={refetch}></DatePicker>
        </Box>
      </Flex>
      <BoxPlot
        data={data.gates.map((gate) => ({
          x: gate.id,
          y: gate.gate_err.map((e) => e.value)
```

```
37          }))}
38        ></BoxPlot>
39      </Box>
40    );
41  };
```

Listing F.1: The implementation of the GateErrorVisualization component.

G

Appendix 7 - Tests

```
beforeEach(() => {
    cy.intercept('GET', ApiRoutes.devices, {
      fixture: 'devices.json'
    });
    cy.intercept('GET', ApiRoutes.statuses, {
      fixture: 'statuses.json'
    });
  });

it('renders navbar', () => {
  cy.get('[data-cy-main-navbar]')
    .find('h1')
    .should('contain', 'WAQCT | Wallenberg Centre for Quantum
   Technology');
});
```

Listing G.1: API mocking for the tests of the overview page.

```
  it('renders device', () => {
    cy.fixture('devices.json').then((devices) => {
      const device = devices[0];
      cy.get('[data-cy-devices]').within(() => {
        cy.get('[data-cy-device-name]').should('contain', device['
   backend_name']);
        cy.get('[data-cy-device-status]').should(
          'contain',
          device['is_online'] ? 'online' : 'offline'
        );
        cy.get('[data-cy-device-version]').should('contain', device
   ['backend_version']);
        cy.get('[data-cy-device-n-qubits]').should('contain',
   device['n_qubits']);
        cy.get('[data-cy-device-last-update]').should(
          'contain',
          device['last_update_date'].split('T')[0]
        );
      });
    });
  });
```

Listing G.2: A test asserting that the cards with the quantum computer configurations are rendered correctly

H

Appendix 8 - Result

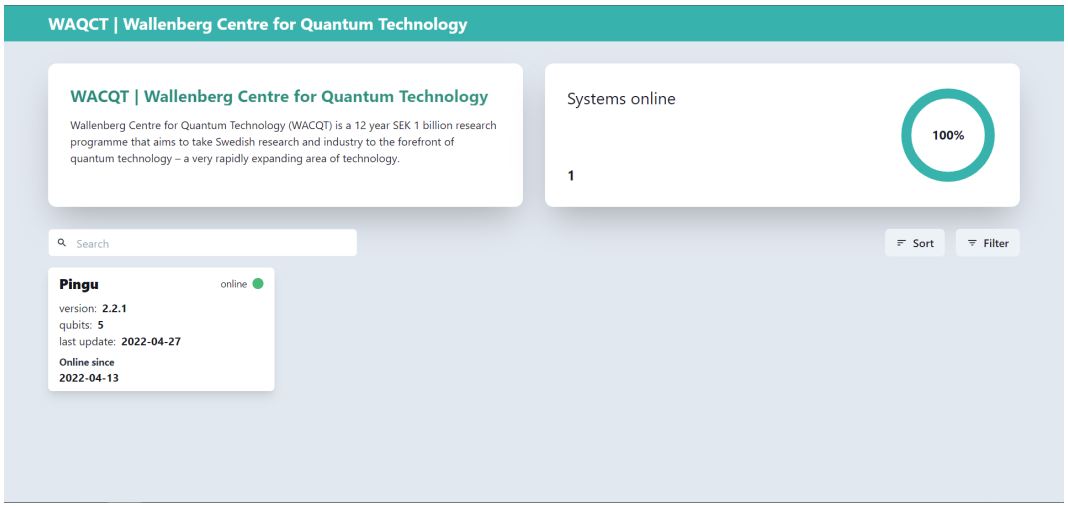

Figure H.1: Homepage where all quantum configurations are available.

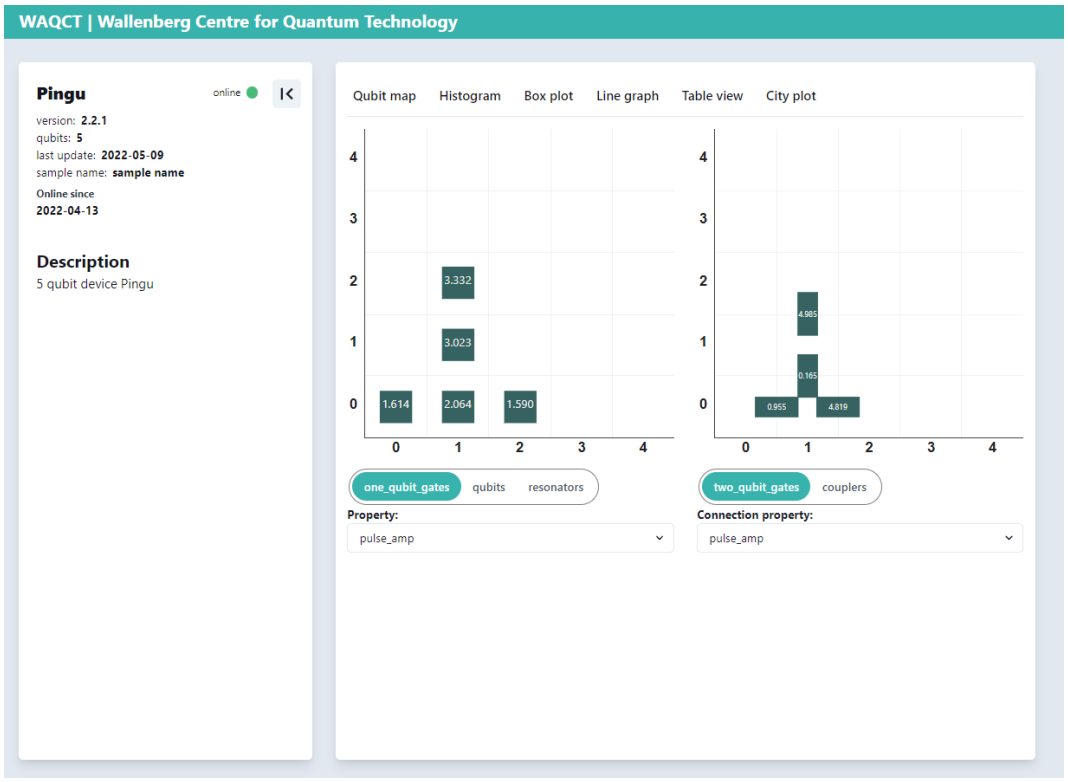

Figure H.2: First page presented when choosing a configuration.

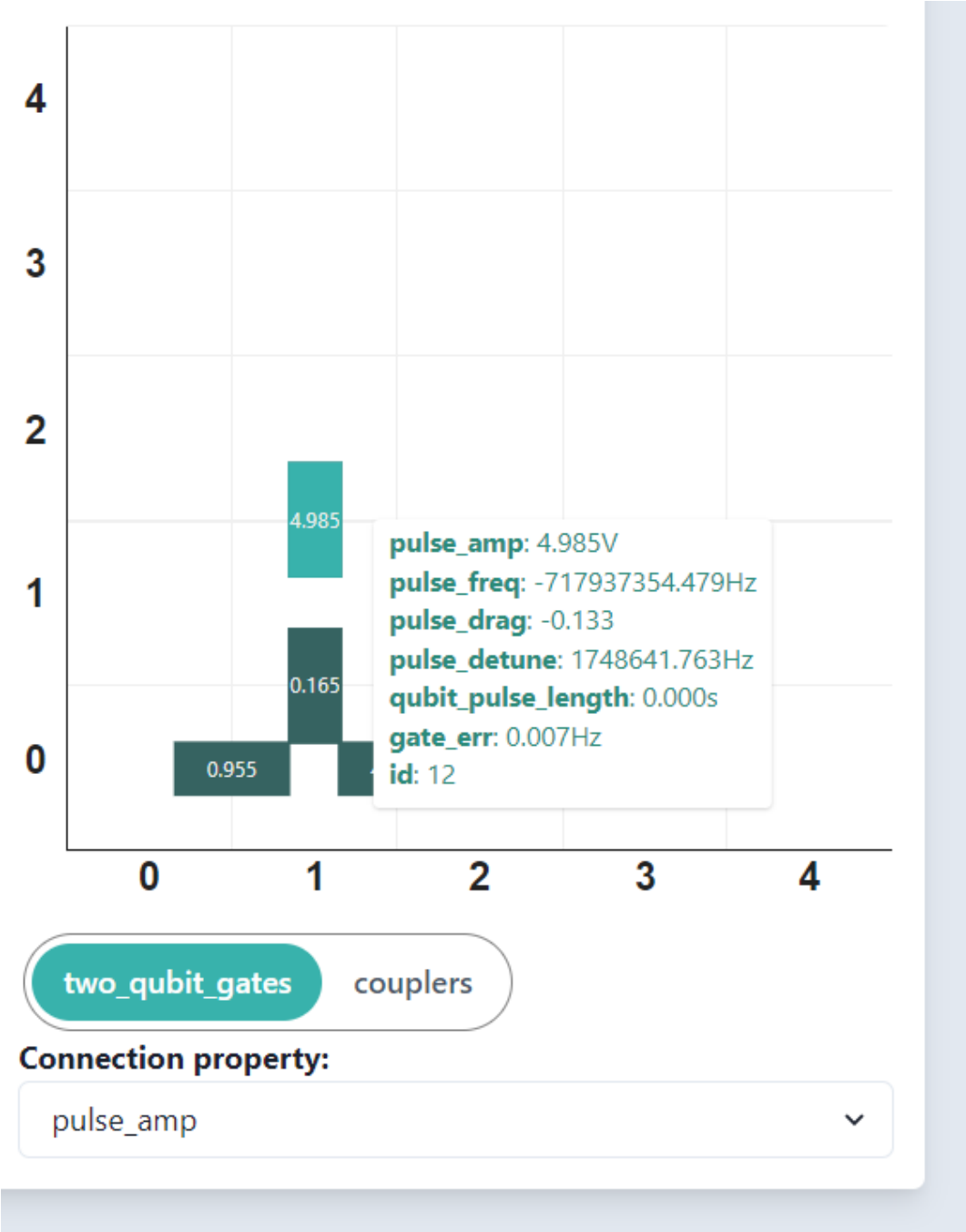

Figure H.3: Tooltip displayed on hover in the gate map.

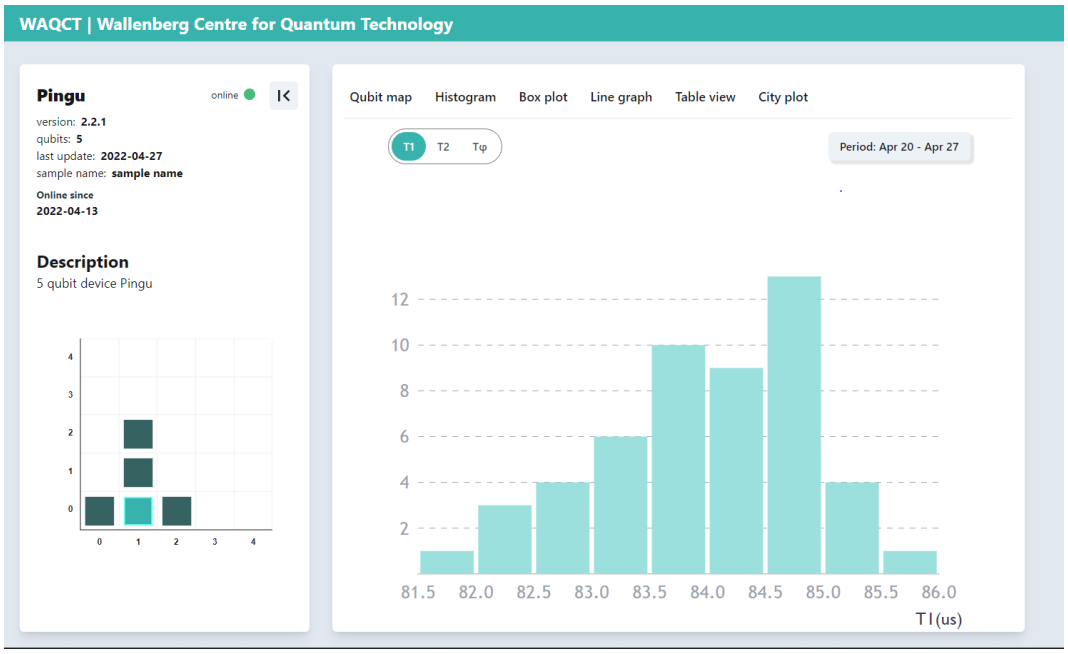

Figure H.4: The histogram component.

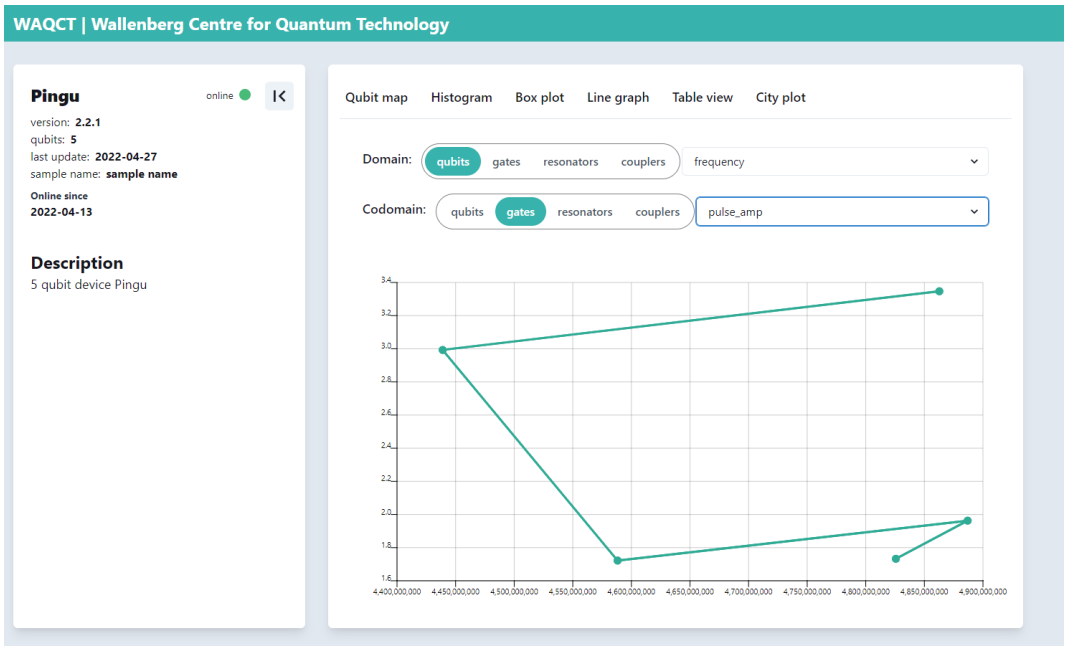

Figure H.5: The line graph component.

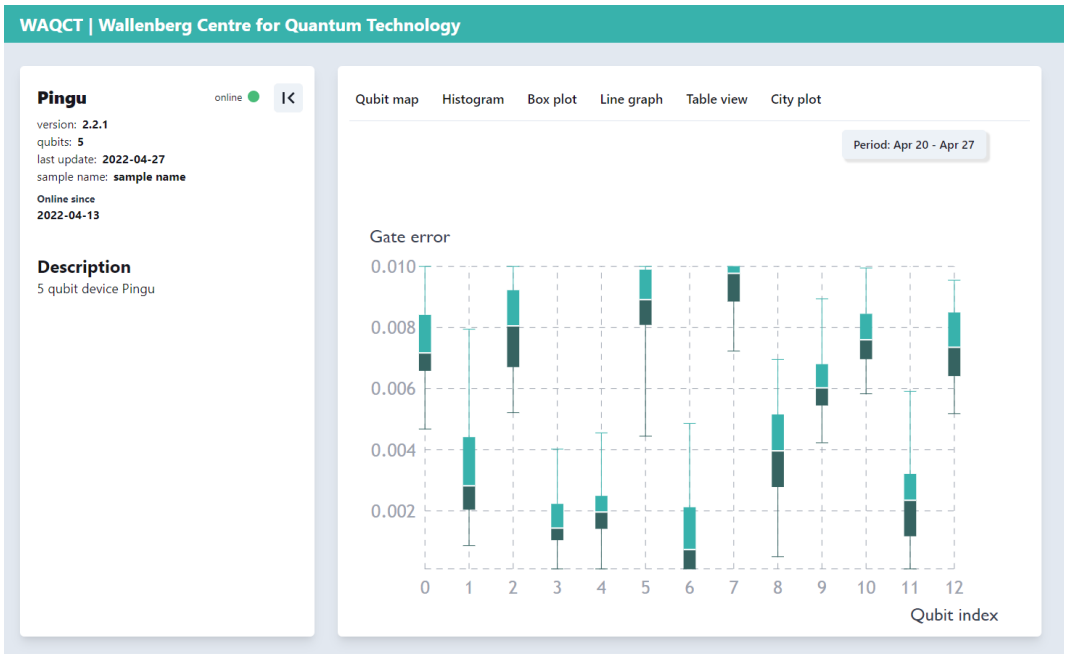

Figure H.6: The boxplot component.

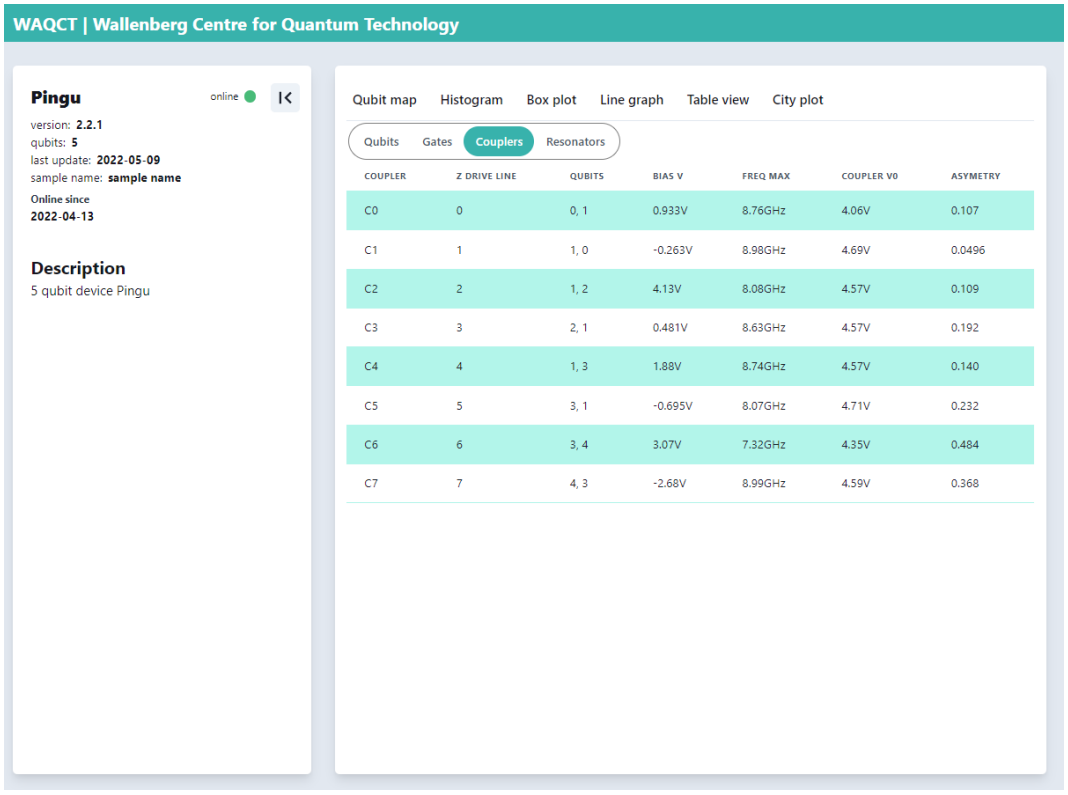

Figure H.7: The table component.

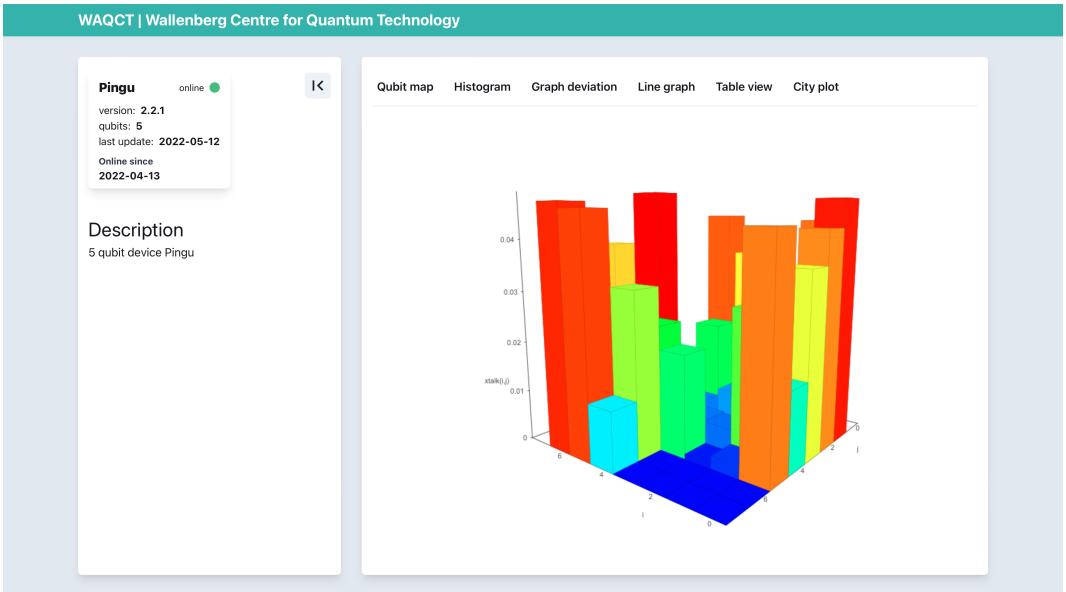

Figure H.8: The city plot component.

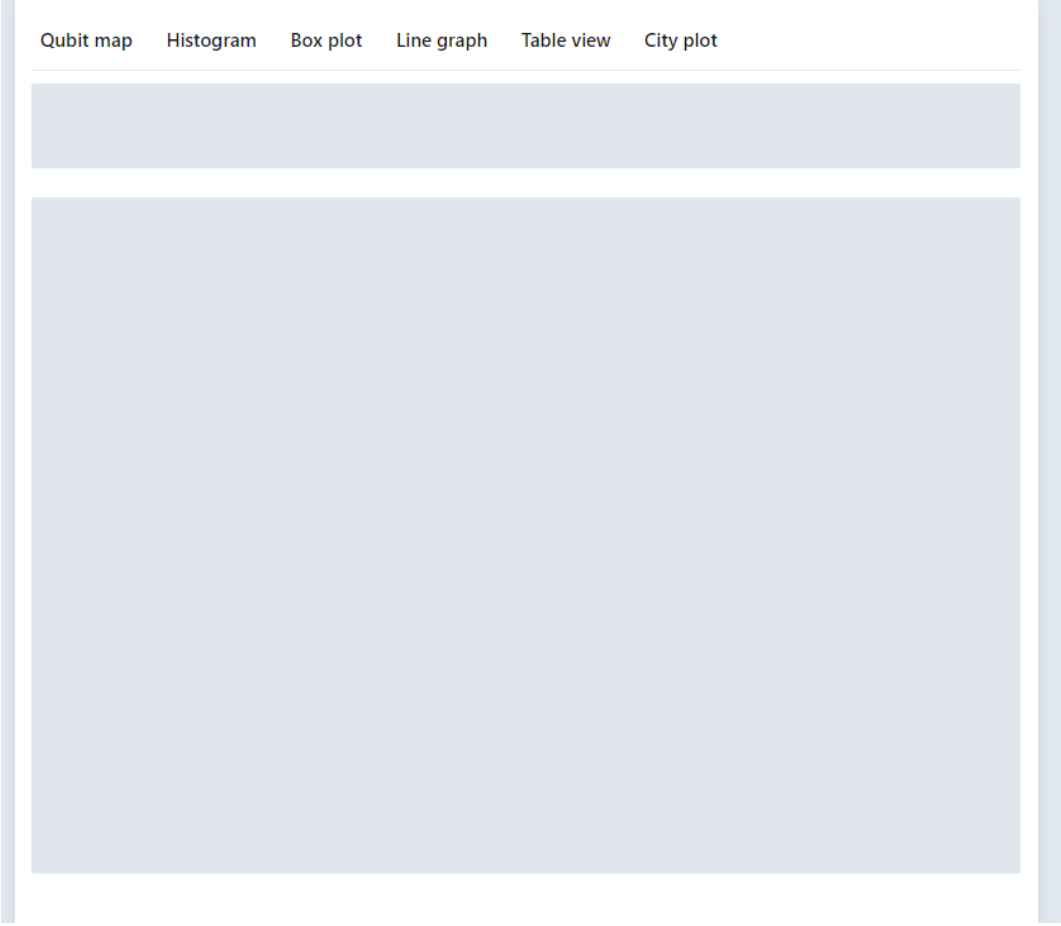

Figure H.9: Example of skeleton while loading histogram.

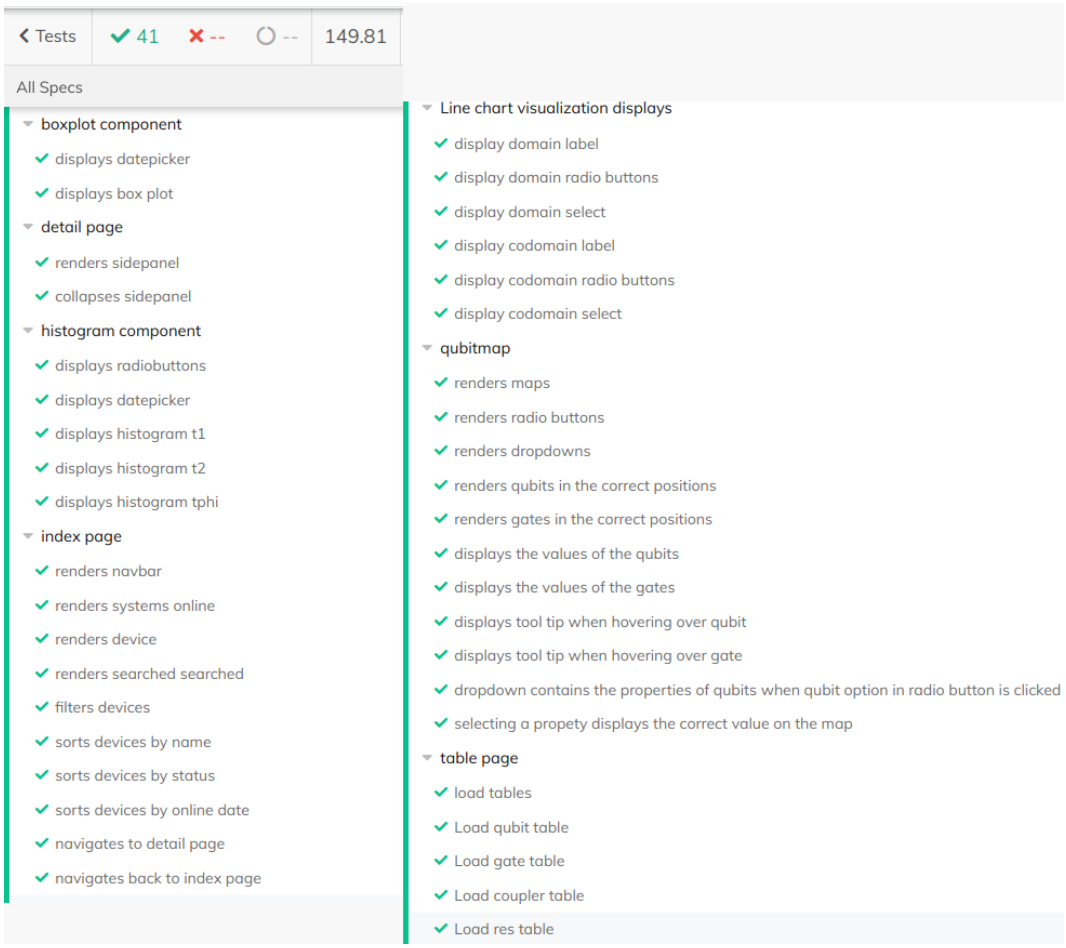

Figure H.10: The test results.