Are your time servers on time?

Active Internet Measurements to Evaluate Time Servers

Group 15D

Călin-Mihai Olaru	5919673
Horia-Andrei Botezatu	5933951
Șerban Orza	5976790
Mihai-Valentin Nicolae	6010059
George-Matei Andrei	5986443

Date: 26 June 2025
Course: Software Project (CSE2000)
Coach: Jana Dönszelmann
Teaching Assistant: Yigit Çolakoğlu

Department of Computer Science

Preface

This report was written by five second-year Computer Science and Engineering students at the Delft University of Technology, for the purpose of the Software Project (CSE2000) course, in the fourth and final quarter of the second academic year.

We would like to extend our gratitude to the Delft University of Technology for giving us the opportunity to gain practical experience working on a project with future applications. In particular, we would like to thank our teaching assistant, Yigit Çolakoğlu, and our coach, Jana Dönszelmann, for their continous support and guidance throughout this project. We are also very grateful to our client, Giovane Moreira Moura, for his invaluable advice and feedback. We would also like to show our appreciation to our Technical Writing lecturer, Florian Gerritsen, and our Teamwork lecturer, Inge Schouten, for teaching us the necessary skills to write this report and to properly work as a team. Last but not least, we would like to thank the RIPE Atlas community, who donated over 800 million credits for us to be able to perform measurements.

This text was written within the course Software Project (CSE2000) of the Faculty of Electrical Engineering, Mathematics and Computer Science of Delft University of Technology. Total word count: 11997. Statement on the use of generative AI:
In writing this text, we:
0 Did not use generative AI.
0 Used generative AI to generate (parts of) the content of the text. Where applicable, I justified this in a footnote and/or with a literature reference.
0 Used generative AI (for inspiration) to structure the text.
X Used generative AI to improve the grammar, style, and/or spelling of the text.

Summary

Time synchronization represents an essential component of all modern systems, especially in the online medium. Synchronization of all clocks worldwide is achieved through the Network Time Protocol (NTP) servers, also known as time servers. However, even these servers can be inaccurate or even fail at times. Currently, there is no convenient, publicly available way to check whether an NTP is on time and/or accurate, and the solutions that exist offer very little relevant information. This poses a great security risk, as online systems depend on these servers for secure communication.

Our team set out to develop a fast, open source, publicly available solution to reliably measure the accuracy of NTP servers. The development process of such an application is detailed in the report. We began by researching various RFC documents and scientific papers to better understand the field of NTP servers and choose the right tools for the job. Then, we implemented basic versions of both the front-end and the back-end components and ensured their functionality in isolation. After both components worked individually, we developed the Application Programming Interface to actually integrate them. The front-end was then enhanced to support historical data visualization via charts. Once all parts of the application were properly implemented, it was time to also integrate the external RIPE Atlas API. The final stage was refining the application by enhancing the testing, documentation, and design to get the project ready for deployment.

Throughout the project, we encountered multiple technical challenges. Firstly, we had to differentiate between measuring a server using its IP address and measuring it using its domain name, as those could give different results based on the user's location. We solved this by mapping the domain name to a list of possible IP addresses and querying each one to find the best result. Then, we had to deal with receiving incomplete data or even no data from RIPE Atlas measurements. There were very little checks in place for this by default, so we had to make the calls for measurements more robust. Finally, querying a probe near the client using RIPE Atlas without storing their IP proved to be a challenge, so we used a database that is updated daily to map their masked IP to a nearby location.

For future developers working on this solution, we have a few recommendations. Firstly, as it stands now, the application is now fully functional and accessible. Before contributing, we recommend reading the full report to fully understand how the solution was designed, implemented, and tested and to fully understand the choices that were made. We also recommend a thorough reading of the project documentation to better understand the functionality. We invite collaboration and feedback from the open-source community to improve and expand upon the existing product.

Contents

1 Introduction

Clock synchronization underpins modern life. It is essential for financial transactions to work, trains to run, computers to communicate securely, and much more. On the internet, this synchronization is primarily achieved using the Network Time Protocol (NTP), which allows clients to retrieve the correct time from designated time servers. When time data is unreliable, the consequences can be severe, leading to errors in sensitive systems and processes. If checks are not performed regularly, these issues can persist, even though they are often easily resolved. Despite the importance of this matter, there is a surprising lack of accessible public tools that allow users to easily verify the accuracy and reliability of NTP servers.

The aim of this report is to analyze how a public, open-source web application can be developed to measure the accuracy and quality of NTP servers in real time, and describe the development process of such a tool, covering both technical aspects and usability. The research provides insights into the decision-making process and explains why specific tools, technologies, and frameworks were chosen to ensure maintainability, sustainability, and an optimal user experience. The result of this is a web application on which users can enter an NTP server's IP address or domain name to view key measurements such as offset, round-trip time, jitter, and stratum, along with some metadata of the server. This allows them to check the precision and stability of the server in a very convenient manner. The application also offers historical data visualization through intuitive graphs, making it possible to identify consistency issues over time. Historical data of two NTP servers can be compared, so users can see which one is more reliable. Measurement results can be downloaded in various formats, allowing for further analysis.

The report is structured as follows: Chapter 2 focuses on analyzing the problem, outlining the stakeholders, and identifying the main use cases of the product. Chapter 3 presents the research on the advantages and disadvantages of different frameworks and tools that were considered, along with partial existing solutions to our problem. Chapter 4 outlines both the functional and non-functional requirements of the project, as well as how their prioritization changed from the beginning to the end. Chapter 5 details the development methodology, including the teamwork process, tools and policies adopted. Chapter 6 takes a more detailed look into the technical details and design choices of the two main components of the tool: the back-end and the front-end. Chapter 7 describes the quality control measures and the testing done on the project. Chapter 8 discusses ethical implications, diving into the commitment to maintaining user privacy. Finally, Chapter 9 concludes the report with a final discussion about the state product, key conclusions, and recommendations for possible future improvements to the product.

2 Analysis of NTP servers time synchronization

The aim of this chapter is to provide insight into the challenges associated with time synchronization using NTP servers and to explain the motivation behind building an NTP server measuring tool. In Section 2.1, we give a more detailed description of the problem and explain its relevance. Furthermore, Section 2.2 establishes the stakeholders of this project. Finally, in Section 2.3, we present some possible use cases of our solution to give more context.

2.1 Problem statement

The aim of this report is to outline the development process of an open-source, publicly accessible platform that enables real-time assessment of Network Time Protocol (NTP) server quality and more in-depth statistics. Accurate time synchronization is a critical component in many modern systems, particularly those involved in secure communications, financial transactions, distributed computing, and data logging. Despite its importance, there is currently an absence of tools available to the public that allow for transparent evaluation of how accurately and reliably NTP servers operate.

The absence of such a tool poses risks in a multitude of applications. For example, in secure systems, even a minor deviation in time synchronization can result in expired certificates, failed handshakes, or invalid digital signatures. In financial systems, accurate timestamps are vital for logging transactions. Likewise, cloud services and telecommunications infrastructure depend heavily on consistent and precise timekeeping. Without a method to evaluate the performance of NTP servers, users must rely on blind trust, which may lead to subtle, but significant issues.

The need for this project stems from the increasing dependence on digital infrastructure and the distributed nature of modern networks. As organizations and average users rely more heavily on systems that are being expanded across continents and different time zones, the challenge of maintaining synchronized time across these networks becomes more complex. Nowadays, many commercial and institutional systems use public NTP servers for synchronization, but few actually verify whether those servers are on time. The risk posed by these underperforming NTP servers is therefore not negligible, especially in systems where milliseconds can make the difference, such as high-frequency trading, distributed databases or telecommunications. In some cases, the error can be a lot more significant, as was the case when many Active Directory Domain times were showing the time 12 years incorrectly (Morowczynski, 2020).

The purpose of the proposed solution is to be able to compute and show metrics such as offset, jitter, round-trip time and stratum for a single measurement, as well as store historical performance data and display using graphs. Another feature is presenting aggregate data about NTP servers from multiple vantage points around the globe. Users are able to interact with all this data through an intuitive and accessible interface, without the need for login credentials or the submission of any personal information, therefore ensuring privacy and ease of access.

2.2 Stakeholders

We have identified four main stakeholders for this project, each with different interests. Firstly, the SIDN Labs and the TU Delft Cyber Security group are primarily interested in advancing research in the fields of cyber security and computer networks. In particular, they are focused on time synchronization and measurement systems, which are critical to the security and reliability of digital infrastructure.

Another important stakeholder is the open-source community, which intends to build upon the existing product by extending the functionality or contributing improvements to existing features. The way this project helps them is by providing a publicly available foundation on which to build.

Furthermore, the internet community, made up of other researchers and professionals in the fields of cyber security and computer network, is interested in doing research on different NTP servers. As such, this project supports efforts to maintain the stability and accuracy of the global Internet infrastructure by providing a tool to measure the performance of such servers.

Finally, the developer team, consisting of students, views this project as an opportunity to gain valuable hands-on experience in software development and to deepen their understanding of NTP servers.

The product is aimed mainly at people who already have experience in the field of NTP servers. As such, making the application accessible to the general public is not in the current scope of the project.

2.3 Use Cases

In order to better perceive the practical utility of the system, we have outlined multiple use cases which illustrate how different types of users could interact with the platform with specific purposes. These use cases also make more apparent the system's functional and non-functional requirements, particularly in regard to responsiveness, result accuracy and the visual clarity of the data.

Use Case 1: Regional NTP Server Accuracy Verification

User: Network Administrator at a multinational organization
Goal: To verify the accuracy of `time.google.com` from multiple geographical locations
Scenario: A system administrator responsible for configuring network infrastructure wishes to find out if Google's public time server (`time.google.com`) maintains its accuracy across multiple locations in Europe and Asia. By testing these servers on our website, they can find measurement data from both the local probe and from the RIPE Atlas probes found in different regions. The administrator uses the offset and jitter measurements to assess the server's performance. If the results then indicate a notable discrepancy in specific regions, the administrator could then consider switching to region-specific time servers.

Use Case 2: Historical Comparison of Public Time Servers

User: Academic Researcher
Goal: Compare performance of `time.windows.com` and `time.cloudflare.com` over the past week
Scenario: A researcher investigating trends in NTP server stability across major providers and locations wants to compare two of the most commonly used servers in Windows and Cloudfare-based systems. They use the platform to retrieve historical measurements from both servers over the past week. The front-end visualisation allows them to easily compare different data metrics which have been stored in the database. The researcher can then also easily export the data as CSV to include it directly in their academic report.

Use Case 3: Detecting Faulty NTP Server Behavior

User: Open-source developer
Goal: Verify if an NTP server is providing incorrect timestamps
Scenario: A developer who contributes to a distributed timestamping protocol thinks that one of the default time servers their nodes are using is skewing data. Using the measurement tool provided by the website, they can easily and quickly confirm that the server is sending their responses with an offset a lot larger than what was considered acceptable in their system. This behaviour is also seen in the visualisation portion, which also aids the developer in debugging their application.

Use Case 4: Choosing the Best Time Server for a Personal Setup

Actor: Privacy-conscious individual
Goal: Identify the most stable and accurate NTP server in their region
Scenario: A tech-savvy individual who runs a personal VPN and home server setup wants to ensure accurate system time without relying on default OS configurations. They test multiple public NTP servers using the website and examine the offset, delay, and reachability of each. Based on these metrics, they can choose a server that is both highly available and stable in their timezone.

3 Research

This chapter covers the research to ensure that the best technologies and practices are applied throughout the project. In Section 3.1 we talk about similar existing tools that measure network performance, and what parts from them we have adapted into our own product. Section 3.2 covers research about NTP, the data it provides, and what factors can affect the measurements. Solving domain names to IPs and privacy considerations that were taken to protect the user are discussed in Sections 3.3 and 3.4. Finally, in Sections 3.5 and 3.6 we discuss how the program integrated with RIPE Atlas and how the probes were chosen to provide the most accurate information to the user.

3.1 Existing Tools for Evaluating Network Performance

Despite the lack of publicly available tools to measure the performance of NTP servers, there are tools to check the speed and quality of networks and internet connections. We used these as inspiration on how the layout of the UI could look and what to look for when measuring the performance of a server. We were able to identify three of these tools.

Internet.nl provides information on the quality of the internet connection, quite similar to our goal. However, it provides more information than necessary for our product, and the UI is much more cluttered due to this. What we would like to incorporate from them, in addition to adapting everything for NTP servers, is the speedy evaluation they have, the ease of access, as well as the information retrieved and statistics that they offer. We will not have as much information clutter on the screen as Internet.nl, as we do not need to offer as many features.

Speedtest by Ookla is a website that provides information on the speed of internet connections. What we would like to adapt from their design is their intuitiveness on how to use the website, graphical information on how the evaluation is progressing with the aim to inform the user about the length left of the process, and the proper visualizations offered. Something we will do more differently from this website is clearly indicating that we will use the user's IP for their geolocation and that their IP will not be stored or distributed.

Fast.com is a tool to measure the speed of the internet connection, in a similar manner to `speedtest.net`. The difference is that it does not store and distribute the IP, which is something we want to also adapt. One distinct feature is that it is very minimalist and displays only speed, which is not enough compared to our goal.

3.2 Research on NTP

Considering the designs in the previous section, our server performs NTP measurements that would provide relevant details in order to help users assess the accuracy of an NTP server. The most important details are round trip time (network delay), offset (the estimated difference between your local clock and the server's clock), and stratum, but there are also some complex details such as:

- Root Delay – the total round-trip delay between the NTP server and its primary reference source (usually a stratum 0 device such as a GPS or atomic clock).

- Jitter – the variability in delay over multiple samples.
- Poll – the time interval that could safely pass between performing 2 different requests)
- Root Dispersion – maximum error in the clock offset between an NTP server and the primary reference clock.

These fields are key indicators of how accurate and reliable an NTP server is. All of them are already provided by ntplib. In addition, we included a field called "vantage point" as part of our integration with RIPE Atlas, so that we could consider measurements taken from different server locations across the globe. It also helps maintainability in the case that we would host multiple measurement servers across the world. When a domain name resolves to multiple IPs addresses, we measure all of them from our server and from RIPE Atlas probes, and present the results to the user.

There are important factors that may affect the measurement of the accuracy and reliability of NTP servers:

- Geographical Distance – the farther the client is from the server, the higher the delay and potential time drift due to unstable routing.
- Packet loss – requesting a measurement may unexpectedly fail.
- Asymmetric routes – if the path to and from the server differs, round-trip assumptions become flawed.
- Latency – high round-trip delay makes offset estimation less accurate.
- Having the same IP type (IPv4 or IPv6) as the NTP server.

3.3 DNS Resolution

NTP servers may be determined by their IP address or domain names. In the second case, a domain name may have multiple IP addresses all over the world to improve performance. As we said earlier, the geographic distance between an NTP server and a client significantly impacts synchronization accuracy. This creates a new problem: obtaining the IP addresses of an NTP server's domain name that are close to the client. In addition, a client's computer will always use the closest NTP server to synchronize its clock. We want to show the most relevant IP addresses for our clients.

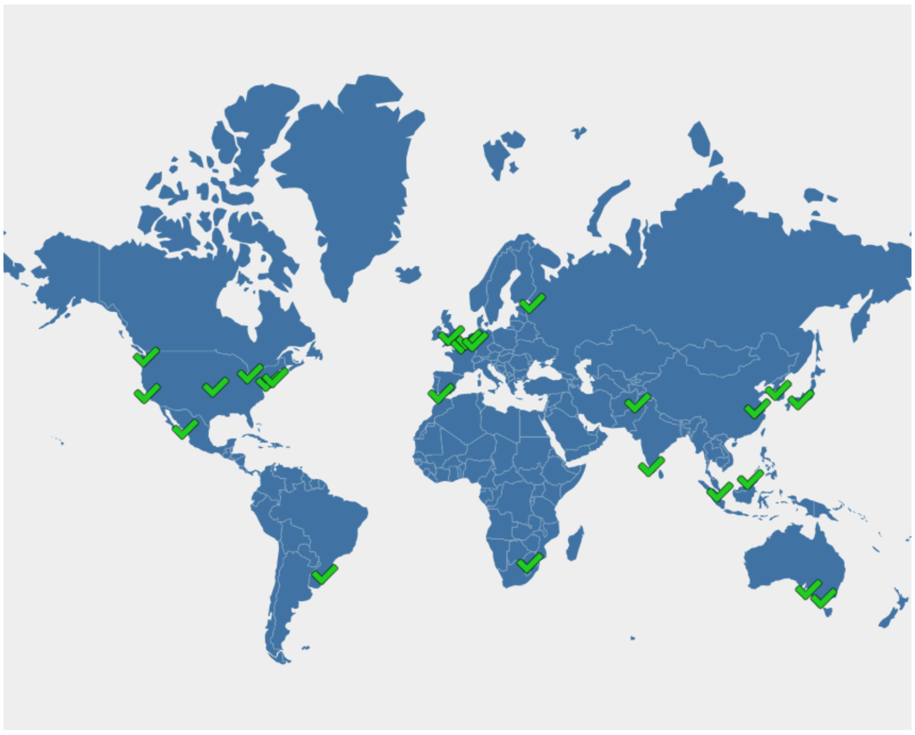


Figure 3.1: Locations of IP addresses for time.apple.com

There are a few strategies to tackle our problem:

1. `Using EDNS Client Subnet (ECS)`
   EDNS stands for Extension Mechanisms for Domain Name System, and according to Contavalli et al. (2016), EDNS Client Subnet is an extension to DNS that allows us to include a partial client IP address in the DNS query. It helps the DNS find IPs that are geographically relevant to the client (Streibelt, 2013). This would definitely solve the problem as it directly provides the closest IP addresses near the client. However, there is a trade-off between location awareness and privacy, and we depend on an external EDNS.

2. `Using a local database of domain names and their IP addresses`
   It offers fast and offline queries, but it introduces high maintenance overhead as the data would need to be updated frequently to remain accurate.

3. `Combining both approaches with caching`
   This involves temporarily caching EDNS responses to avoid repeated queries and improve performance. However, the data may quickly become invalid and managing it introduces added complexity.

These methods have their strengths and weaknesses, and, after analyzing them, we ultimately chose the first one as it best fits the goals and constraints of our application. There are the main advantages of the first strategy over the others:

- Real-time and always up to date, as it returns real IP addresses that the client would use for domain names in reality.

- Better geolocation accuracy and no need to maintain and/or synchronize another database.

- Scalability, as it supports all possible domain names, including those not known in advance. A local database may miss some domain names, as we do not have infinite memory to store everything.
- Maintainability, as the EDNS providers take care of the infrastructure.

While the first one is highly reliable in real time, is always up to date, and it returns real IP addresses that the client would use for domain names in reality, it consists in sending a part of the client's IP address to an external party. Before discussing this trade-off in the next topic, let's analyze the solution we found.

An important factor in the first strategy is choosing the right DNS resolvers. According to Al-Dalky and Rabinovich (2020), Google DNS, Cloudflare DNS and Quad9 are some of the best options. The disadvantages of this strategy are that it leaks partial client information by disclosing the first bits of the client IP, and that our server depends on third-party DNS providers. Despite the disadvantages of the first method, it is still the best approach we found, as it covers all possible valid domain names of NTP servers. Although using MaxMind GeoLite2 to locally store IPs for the most common NTP domain names may be faster for some domain names, we would be limited. There is no way to get a complete list of all NTP domain names or to guarantee full coverage of these servers, as anyone could run an NTP server and point a domain name to it. Even though the last two strategies may be faster for some NTP servers, we would eventually get back to the EDNS approach for those we could not find. Additionally, the third method appeared ideal, but caching results turned out to be inefficient as it introduces considerable overhead without substantial improvements. Users are distributed globally, and the presence of rate limiting on our server already discourages frequent repeated queries from the same client. From a privacy standpoint, the third approach may be problematic as we would partially cache the client IP even more than needed. Cache hit chances are usually low and results are not worth storing as most of the users would only do a few requests per day.

In conclusion, for this problem, DNS resolution via EDNS fits best, but it is important to pay attention to what EDNS we use, as not every DNS has this extension available. It is recommended to use popular and reliable DNS such as Google DNS or Cloudflare DNS (Al-Dalky & Rabinovich, 2020).

3.4 Privacy consideration regarding DNS

In the previous section, we discussed the benefits of using an EDNS, but now we will present the involved trade-off between privacy and accuracy. These EDNS resolvers are widely trusted, highly available, and have ECS support. However, we need to send a part of the client's IP to them and the privacy of our clients depends on how many bits from the client's IP we transmit (see Table 1). The part of the client's IP that is sent is determined by the EDNS Client Subnet Mask. We will now analyze some comparisons between different mask lengths for the IPv4 type of IP addresses (an IPv4 has 32 bits and an IPv6 has 128 bits).

Table 1: Privacy vs Accuracy Trade-off Based on IP Mask

Mask	**IP Range**	**Privacy**	**Accuracy**	**Use Case**
32 bits	1 IP	Very low	Highest	Precise geolocation needed
24 bits	256 IPs	Medium	High	City-level geolocation, network-level
16 bits	65,536 IPs	High	Medium	Country-level geolocation
12 bits	1 million IPs	Very high	Low	Higher privacy, coarse region
8 bits	16 million IPs	Highest	Low	Highly private, poor for geolocation

As you can see in Table 1, the choice of subnet mask length directly impacts the trade-off between location accuracy and client privacy. Our goal is to identify the IP's network and obtain geolocation information with higher precision than the country level. Considering that some countries may be very large and country-level may not be enough, analyzing the advantages and disadvantages from the table above suggests that a mask of 24 bits should be the best choice. Besides, it is also considered a good practice to use 24 bit masks (in the EDNS from Google, 8.8.8.8, the default mask is 24 bits).

For IPv6 type, the situation is similar, as 56 bit masks are the best option there. We need a somewhat high level of accuracy, but we still want to protect our clients from exposing their IP addresses.

3.5 RIPE Atlas Integration

After analyzing the performance of the measurements made from our server, we realized that in some cases both the client and the NTP server we want to test may be geographically or topologically distant from our server. Such distances can introduce significant delay and influence the reliability of the measurements. To address this problem, we integrated RIPE Atlas, a globally distributed measurement platform that has thousands of probes hosted by volunteers from all over the world (Izyumov, 2024). For each client, we determine the closest probes near them and perform measurements from these probes. This would allow us to estimate the reliability and latency more accurately from the perspective of the client's network environment. In Section 3.6, we discuss the criteria for selecting the most relevant probes.

In addition, it is worth noting that, even though RIPE Atlas has high availability, sometimes probes may suddenly disconnect and fail to perform measurements, due to unexpected reasons. We collect data not only from probes that succeed but also from those that fail, and present the results to the user. You can see in Figure 3.2 an example of how diverse these probes can be.

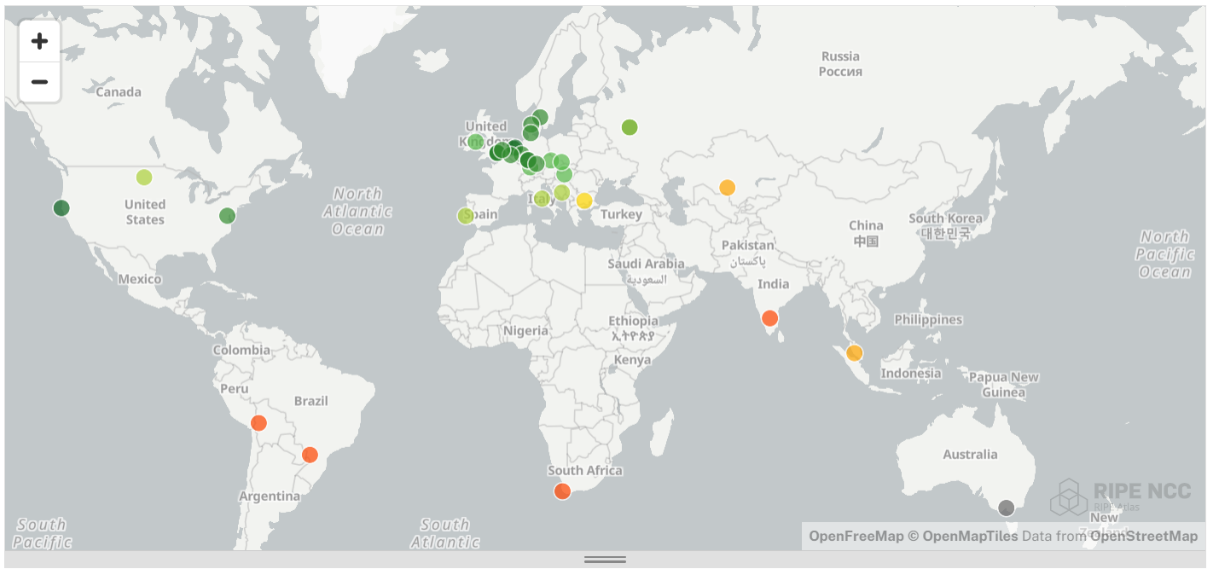


Figure 3.2: An example of measurements for offset from probes all over the world for `time.google.com`. Green means high accuracy, while red means poor accuracy.

3.6 Probe Selection and Fallback Mechanism

As we presented in Section 3.2, there are several factors that can affect the measurement of an NTP server. There are multiple meanings of the term ”close to the client”. It may be by distance, by network, or by routing. It is obvious that we need to keep all of these aspects in mind when we search for the best probe(s) available. This is why we need to get useful information from the client IP, such as the ASN (network), prefix (family of the IP), country, and continent.

We encountered a challenge in trying to retrieve these metadata for clients that connect to our server with an IPv4 address, but want to measure an IPv6 server. These two IP types are different protocols and cannot communicate directly. We explored a solution involving reverse DNS to convert an IPv4 address to its respective IPv6 address. By resolving the PTR record (Pointer) of the IPv4, we can get the domain name of the client's IP. We then query this domain name for AAAA records to obtain the IPv6 address. However, this process only works under some specific conditions, for example, when the client's network has been properly configured and the domain name was mapped to an IPv6 address. In cases where this does not work, all we can do is simply get the country and the ASN from the IPv4, which are highly likely to be the same (Strowes, 2017).

RIPE Atlas provides probes by ASN (network provider), by prefix, by country, or area. After careful considerations and analyzing all aspects, we concluded that the following order of priorities is the best:

1. Same ASN and same prefix
2. Same ASN and same conutry
3. Same ASN
4. Same prefix

5. Same country
6. Same area

This hierarchy ensures that measurements are always taken from the most relevant available vantage points. For example, if there are no probes available on the same ASN and with the same country as the client's IP, we would search for probes with only the same ASN and so on. The ultimate fallback is on the same area (where enough probes are guaranteed to exist).

Additionally, we incorporated system tags into our search (”system-works-ipv4” or ”system-works-ipv6”) to ensure that the probes would be able to communicate with the NTP server on the appropriate IP version (IPv4 or IPv6). As highlighted in Bajpai (2017), the selection of the best probes is strongly influenced by both the IP version and the topological closeness between the client and the NTP server (especially in dual-stack environments), and system tags play a crucial role.

4 Requirements

The chapter presents the requirements established in collaboration with the client. In Section 4.1, it can be observed that the functional requirements are categorized based on the MoSCoW prioritization methodology: Must, Should, Could, and Won't. Furthermore, Section 4.2 highlights the non-functional requirements, offering a comprehensive view of the system's expected performance, usability, and other essential qualities. A complete list of the requirements can be found in Appendix A: Requirements Overview

4.1 Functional Requirements (MoSCoW)

The requirements are derived based on the client's requests both in meetings and the original project posting, taking into account the previous information learned during CSE2115 Software Engineering Methods, and aligning with the goals of the CSE2000 Software Project course. The entire requirements engineering process was conducted during the first week of the project, with slight adjustments throughout the duration of the project.

The MoSCoW method is used to clearly divide the requirements based on their priority. This widely-used technique helps us distinguish between varying levels of necessity and urgency in system requirements. It categorizes requirements into four groups: Must have, Should have, Could have, and Won't have. Must haves represent the requirements that are critical for achieving a Minimal Viable Product (MVP). Should haves are important but not essential for the initial delivery. Could haves are desirable but less critical features that can be implemented if time and resources permit, often serving as enhancements rather than necessities. Last but not least the Won't haves refer to requirements that are excluded on purpose due to being not feasible within the project timeline, or are considered out of the scope based on client's needs. Moreover, the set of non-functional requirements is comprised of requirements that specify the quality of the system, such as security, technologies, rate-limiting, usability.

4.2 Non-Functional Requirements

The project shall be made open-source on GitHub with full documentation and shall contain a rate-limiting mechanism to ensure servers don't get overloaded. The codebase should be concise and modular, and should use only actively maintained libraries and dependencies. The back-end will be tested with pytest by means of unit and integration, while the front-end will use React Testing Library for UI functionality, targeting a minimum of 80% branch coverage. Users should see results within 6 seconds, with one retry allowed on timeout. The system shall use ntplib for NTP queries, PostgreSQL for data storage, React.js for the front-end, and FastAPI for back-end request handling. SIDN Labs' atomic clocks will be considered as the accurate reference for all measurements.

5 Development Methodology

This chapter provides a detailed overview of the development methodology adopted for this project, highlighting the structured approaches and processes that facilitated a systematic and adaptable workflow. Section 5.1 introduces the project management framework selected by our team, outlining its key principles and rationale. Section 5.2 details our teamwork process, including role distribution, communication protocols, and conflict resolution strategies. Section 5.3 focuses on the tools and technologies utilized to enhance collaboration, streamline development, and maintain project documentation. Finally, Section 5.4 defines the quality assurance policies and standards implemented to ensure transparency, accountability, and the overall reliability of the project deliverables.

5.1 Project Management Framework

We have decided to use the KanBan framework. The way KanBan works is that the team creates a "board", a visual representation of work items, consisting of the following columns: To do, In progress, In review, Done. Each work item is represented by a card that moved through these columns throughout the project. This allowed us to be very flexible in our approach of solving the problem at hand, as well as limiting the number of features we worked on at a given time. The main reason for this decision is that we wanted to be able to modify our priorities and make decisions as we researched and learned more about the subject of NTP servers and how to measure their quality. Due to our limited knowledge in this field at the start of the project, we did not consider a more strict framework like Scrum to be appropriate, as mapping out the project and splitting into "sprints" did not seem realistic at that point.

5.2 Teamwork Process

To facilitate communication with the TA and coach, as well as the client, weekly meetings were arranged with both parties. For the meeting with the TA and coach, a chair was decided each week, who lead the meeting and ensured that all points on the agenda were discussed. These meetings also had a minute taker who wrote down the important information presented and made it available to the rest of the group after the meeting concluded. The agenda for the meeting was written by the whole team at least one day before the meeting and published to the GitLab repository for the TA to review prior to the meeting. There were two types of internal meetings throughout the project:

- Weekly meetings with the TA and the coach that took place on Tuesdays. The purpose of these meetings was to present our current progress, give insight on how we planned to continue, and possibly receive help or guidance with problems we have encountered, according to the agenda for the meeting.
- Weekly meetings with the client occurred on Wednesdays. During these meetings, similar to those with the TA and the coach, we presented the progress since the week before to the client. From there, we presented our plans for the coming week and asked for clarifications in areas where we had uncertainties. In addition to this, we presented details for the implementation so that the client could offer any additional restrictions and information we could use to improve the product.

In-person meetings were the norm with the exception of the case where the client or TA were not available for meeting in person, as they allowed for better communication, smoother interactions, and a more coherent presentation of ideas.

For changes or implementations of a feature to be considered done and merged, the following requirements needed to be fulfilled:

- Be fully implemented
- Be reviewed and approved by two people
- Have sufficient testing done where applicable
- Respect policies related to the CI pipeline implemented (see subsection 5.4: Policies)

5.3 Tools

For version control and project management we use GitLab. Our workflow was organized through milestones, each representing a significant part of the project. These milestones were further divided into multiple issues, which were displayed on our Kanban board for clear tracking and management (see Figure 5.1). Our Kanban board included four main lists:

- *Open*: Issues that had not been started yet. These are tasks that were either newly created or planned for the future.
- *In Progress*: Issues being worked on at the time by team members. Team members were assigned tasks based on their expertise and availability, and they were responsible for updating the status of the task.
- *For Review*: Issues that had been implemented but awaiting review and approval. Peer review is essential in maintaining code quality and ensuring that best practices are followed.
- *Closed*: Completed issues that had been reviewed and finalized. Once an issue was closed, it was considered part of the final product and was documented accordingly.

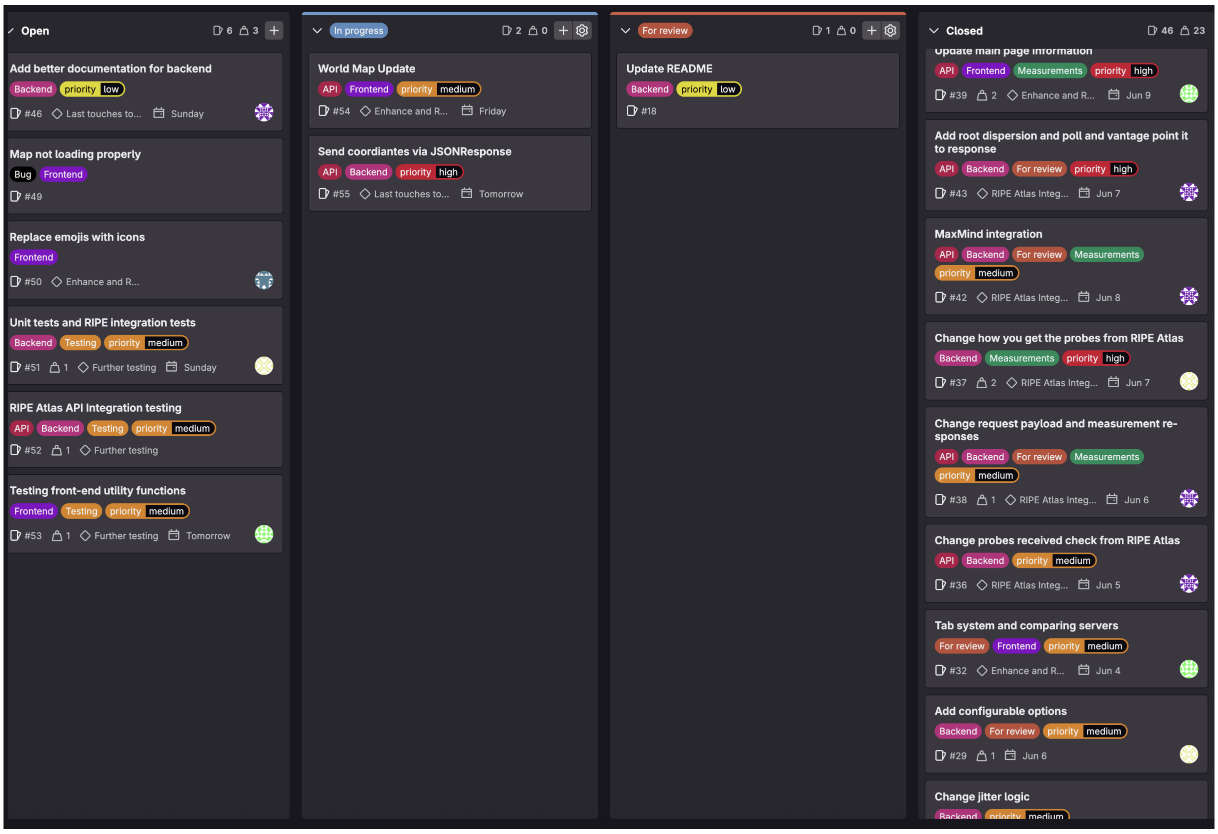
Each issue has been labeled to indicate its associated project component (e.g., Front-end, Back-end, API) and its priority level (low, medium, high). This labeling system provided a quick overview of the status of the project and helped team members focus on high-priority tasks.

Internal communication was primarily conducted through a team Discord server, providing a flexible space for discussions, brainstorming, and quick updates. We maintained separate channels for different aspects of the project, such as general discussions, technical support, and task coordination. This organization ensured that information remained accessible and easy to locate.

A shared Notion page was used for organizing ideas, notes, and materials. This includes documentation on the project requirements, design decisions, and research findings. Notion served as a collaborative space where team members could draft and review documents, share resources, and maintain a structured knowledge base.

For remote meetings with the teaching assistant, we used Microsoft Teams when in-person sessions were not possible. This allowed for efficient communication, screen sharing, and live demonstrations of the progress of our project. Meeting minutes were recorded and shared with all team members to ensure transparency.

Communication with the client was managed through a separate GitLab repository, where we exchanged information using GitLab issues. This approach ensured that questions are answered quickly. We also maintained an official communication channel via Outlook email for any formal correspondence with the client.


Open 6 3

Add better documentation for backend
Backend priority low
#46 Last touches to... Sunday

Map not loading properly
Bug Frontend
#49

Replace emojis with icons
Frontend
#50 Enhance and R...

Unit tests and RIPE integration tests
Backend Testing priority medium
#51 1 Further testing Sunday

RIPE Atlas API Integration testing
API Backend Testing priority medium
#52 1 Further testing

Testing front-end utility functions
Frontend Testing priority medium
#53 1 Further testing Tomorrow

In progress 2 0

World Map Update
API Frontend priority medium
#54 Enhance and R... Friday

Send coordiantes via JSONResponse
API Backend priority high
#55 Last touches to... Tomorrow

For review 1 0

Update README
Backend priority low
#18

Closed 46 23

Update main page information
API Frontend Measurements priority high
#39 2 Enhance and R... Jun 9

Add root dispersion and poll and vantage point it to response
API Backend For review priority high
#43 RIPE Atlas Integ... Jun 7

MaxMind integration
API Backend For review Measurements priority medium
#42 RIPE Atlas Integ... Jun 8

Change how you get the probes from RIPE Atlas
Backend Measurements priority high
#37 2 RIPE Atlas Integ... Jun 7

Change request payload and measurement re-sponses
API Backend For review Measurements priority medium
#38 1 RIPE Atlas Integ... Jun 6

Change probes received check from RIPE Atlas
API Backend priority medium
#36 RIPE Atlas Integ... Jun 5

Tab system and comparing servers
For review Frontend priority medium
#32 Enhance and R... Jun 4

Add configurable options
Backend For review priority medium
#29 1 Jun 6

Change jitter logic
Backend priority medium

Figure 5.1: Overview of the issue board on GitLab

For official communication with the university, such as coordination with teaching assistants, coach, and course coordinators, we used Mattermost. This platform ensured that important updates were not missed and scheduling meetings is straightforward.

This setup ensured that all team members were consistently informed, and all project-related discussions were properly organized and accessible. The use of multiple communication channels ensured flexibility, while the clear structure helped maintain transparency and accountability.

5.4 Policies

We are a team that values transparency, motivation, and trustworthiness. Each merge request had to be be reviewed and approved by at least 2 people before being merged to ensure that everyone maintained full knowledge about the content of our project. Before being merged or reviewed, each merge request was required to pass the The CI pipeline consisting of three main stages: Type checking, Linting and Testing, detailed further in Section 8.

Another policy was to use meaningful commit messages when significant changes were made. This helped the developers locate the parts of the project that the author of the commit changed.

Every feature was required to include unit tests and have at least 80% branch coverage.

Reviewers had to read the full merge request description, analyze the changes and how they were tested and only then to approve if everything was alright, or to contact the author and to communicate the issues they found in case something seemed wrong.

6 Product Design & Implementation

This chapter presents the way the project was structured into components and goes into detail about the implementation and technologies used for each of them. Section 6.1 describes the architecture of the system. In Section 6.2, the implementation and design of the front-end component are presented. The same is done in Section 6.3 for the back-end component. Section 6.4 then discusses the frameworks and technologies that we used throughout the project.

6.1 Architecture & Components

The project uses a Client and Server architecture. The application is split into two main components - the front-end (client) and the back-end (server). The front-end component is represented by the User Interface which makes API calls to the back-end component, where all the business logic and validation of the data happens. In isolation, the back-end component employs a layered architecture - an API Layer that manages the calls from the client, a Business Logic layer represented by the NTP Measurement Engine, and a Data Layer that holds the Database schema. A visualization of the structure of our project can be seen in Figure 6.1.

6.2 Front-end Component

6.2.1 User interface

The User Interface is represented by a web application, composed of four tabs. On the Home tab, users can query an NTP server and get different measurements such as offset, round-trip time, stratum, and jitter, as seen in Figure 6.2. The two different result boxes represent our NTP client and the RIPE Atlas measurements. Along with the results, a visualization is shown in the form of a graph. The user then has the option of downloading the results in either CSV or JSON format. A map is shown to display the vantage points and the NTP server that was queried, as seen in Figure 6.3. The second tab aims to visualize detailed historical data in the form of line charts. Users can query an NTP server by its domain name or IP address and select either a preset or custom time interval to visualize the historical data of that server during that particular interval, as well as aggregated data such as the average, miminum and maximum values of the selected metric, as seen in Figure 6.4. The third tab is dedicated to comparing two or more NTP servers. After entering the servers and selecting a time interval, users can compare the evolution of their historical data for that particular time, as seen in Figure 6.5. The fourth and the last page is the About tab, which includes general information about our application and the team behind it.

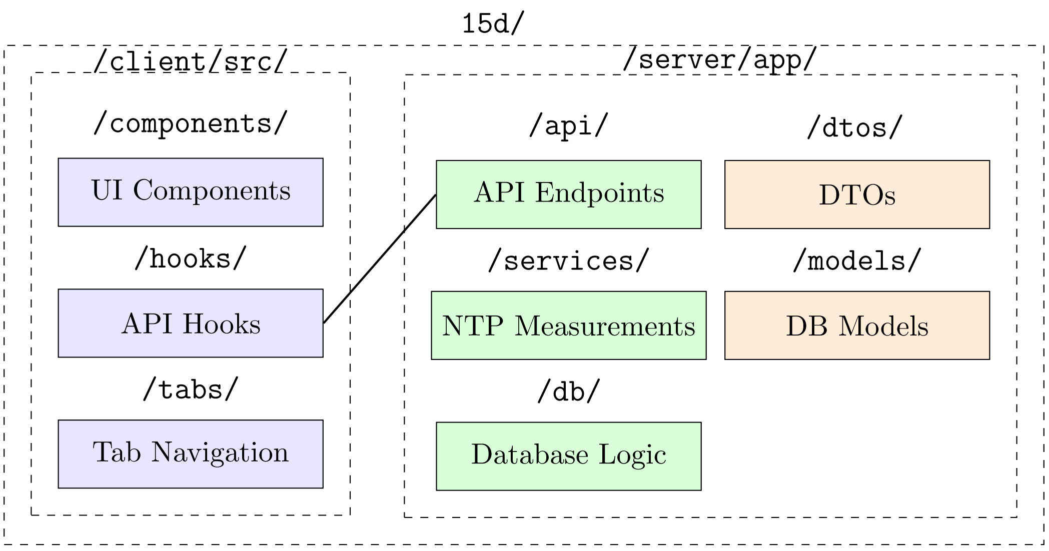
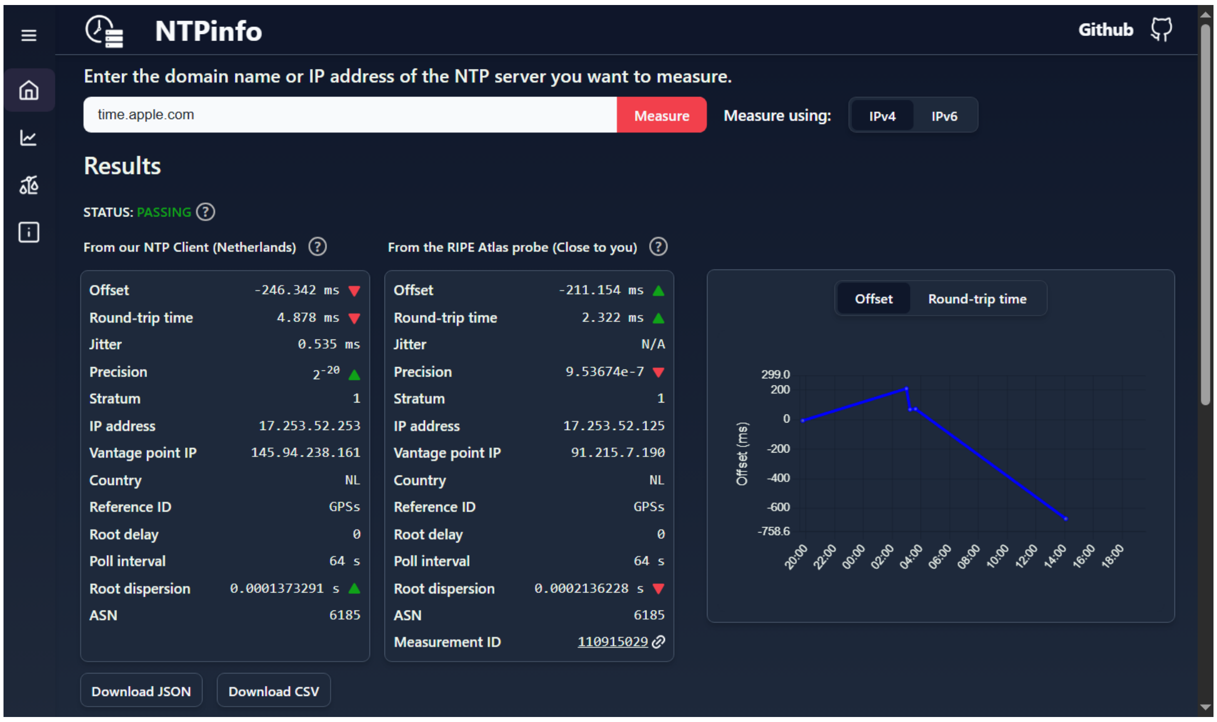


Figure 6.1: Structure of our Git repository



Enter the domain name or IP address of the NTP server you want to measure.

time.apple.com

Results

STATUS: PASSING

From our NTP Client (Netherlands)

Offset	-246.342 ms ▼
Round-trip time	4.878 ms ▼
Jitter	0.535 ms
Precision	2^{-20} ▲
Stratum	1
IP address	17.253.52.253
Vantage point IP	145.94.238.161
Country	NL
Reference ID	GPSs
Root delay	0
Poll interval	64 s
Root dispersion	0.0001373291 s ▲
ASN	6185

From the RIPE Atlas probe (Close to you)

Offset	-211.154 ms ▲
Round-trip time	2.322 ms ▲
Jitter	N/A
Precision	9.53674e-7 ▼
Stratum	1
IP address	17.253.52.125
Vantage point IP	91.215.7.190
Country	NL
Reference ID	GPSs
Root delay	0
Poll interval	64 s
Root dispersion	0.0002136228 s ▼
ASN	6185
Measurement ID	110915029

Figure 6.2: Main page after querying domain name time.apple.com

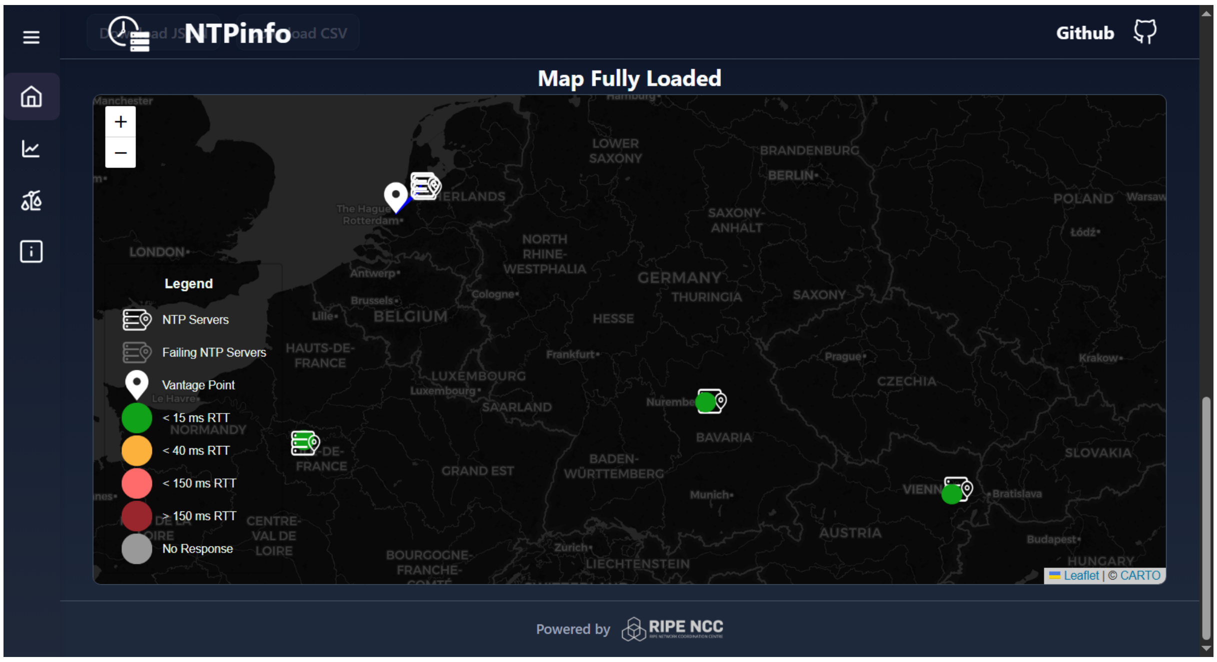
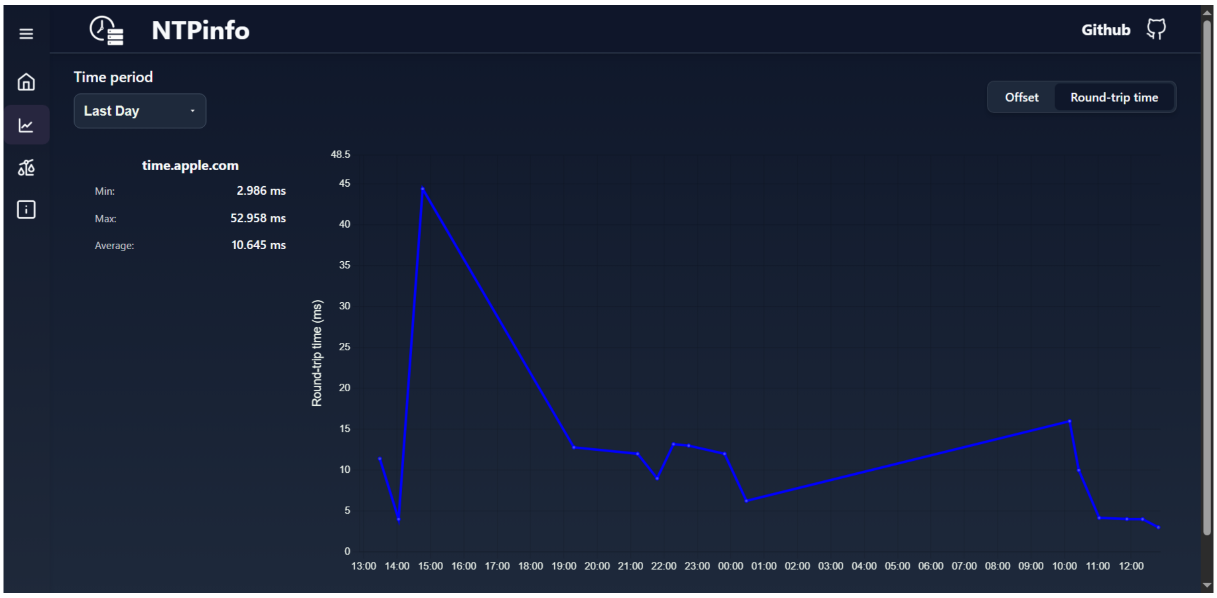


Figure 6.3: Map after querying 2.android.pool.ntp.org, showing the vantage point (white location point), probes (green dots), NTP servers (server icons)



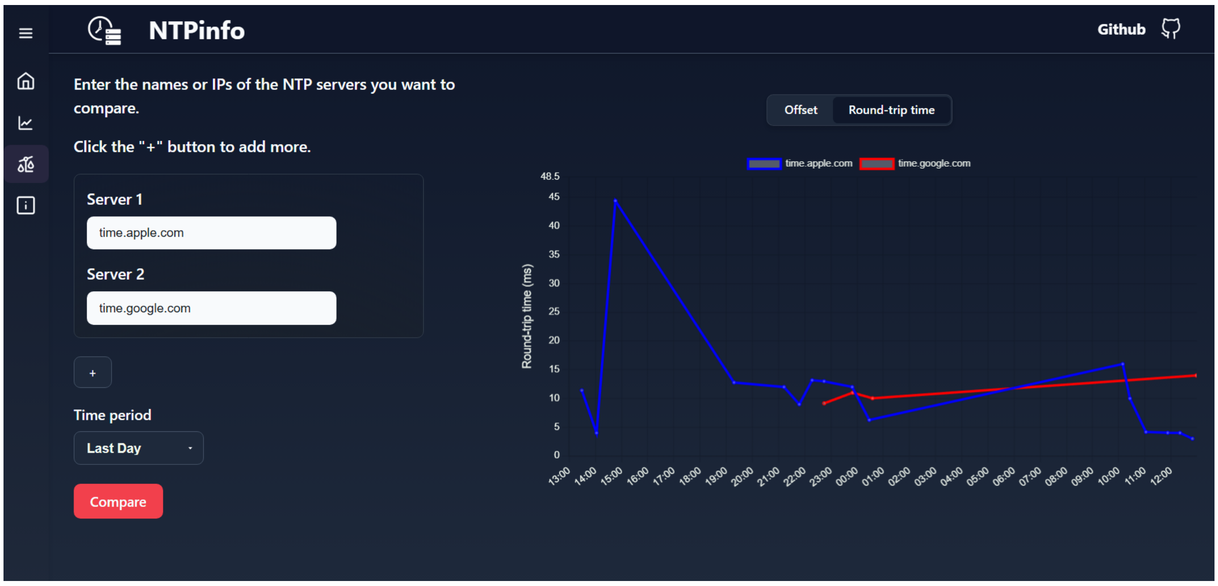
time.apple.com

Min: 2.986 ms

Max: 52.958 ms

Average: 10.645 ms

Figure 6.4: Historical measurements made in the last 24 hours for the Round-trip time of time.apple.com



Enter the names or IPs of the NTP servers you want to compare.

Click the "+" button to add more.

Server 1

time.apple.com

Server 2

time.google.com

Time period

Last Day

Figure 6.5: Graph comparing the Round-trip time results of time.apple.com and time.google.com from the last 24 hours

6.2.2 API Hooks

The Application Programming Interface (API) on the front-end is how it can communicate and transfer information to the server. Since the front-end cannot do the measurements by itself, it relies on the back-end to do it in its place, and then displays that information for the user to see. The API is implemented using hooks made available by the React library. These hooks allow for the state of the call and the data to be both saved and dynamically updated with each new call. Specifically, it allows for the storage of variables that indicate the state of the call, if its still processing or not, that indicate errors with the call, and that store and can alter the data received or transmitted. The application has four API hooks that it uses:

- **Fetching Measurement Data**
  To receive measurement data about the server chosen by the user, the hook used is `useFetchIPData`. This hook calls the server endpoint responsible for measuring data of a specific server and receives back the processed information. To make the specific call, it first uses the domain of the server as the main part of the link to the endpoint. Another component of the call is the payload, which contains the IP or DNS given by the user and if the measurement should be done using IPv6. A payload was chosen for this instead of embedding them into the link as it allows for scalability in the data that is transmitted to the back-end, and for easy interpretation. The last part of the call is the header, which specifies the format which the payload will be transmitted as. The response is a JSON which contains all the relevant measurement information collected by the back-end, which is then parsed for a better representation on the website.

- **Fetching Historical Data**
  To receive historical data about the server chosen by the user for a specific time period, the `useFetchHistoricalIPData` hook is used. This hook interacts with the back-end similarly to the other one, however, the information is transmitted differently. Instead of using a JSON payload to transfer information, this data

is instead transmitted as URL parameters. These parameters are the server from which the user wants the historical data, the start time of the time period, and the end time of the desired time period. The format for the transmitted date is ISO8601 format, which is the same as used by Python. To facilitate conversion from UNIX time to that format, a helper function called `dateFormatConversion` was made and used. Compared to the measurement API call, this returns an array of JSON responses that are then parsed to both be used for display on the charts on the website, as well as be available for download by the user.

- **Fetching RIPE Atlas Data**
  To receive the probe information, NTP server used by the probe and the measurements of the probe on the front-end, the hooks used are `triggerRipeMeasurement` and `useFetchRipeData`. First, the `triggerRipeMeasurement` hook initiates the measurement and receive a measurement ID, and uses the same payload as the measurement hook. From there, the `useFetchRipeData` hook polls data from the back-end, updating the response each time new data is received until the measurement is fully completed.

6.2.3 Data visualization

- **Charts for historical data**
  To ensure better data visualization for the user, the application uses ChartJS for its historical data representation. ChartJS is used over other technologies such as Recharts and Victory, due to them requiring more setup, being harder to integrate, and having less customization options. The main chart used is the line graph offered by ChartJS. This is used to display historical data in a more legible way for the user. It directly uses the data offered by the API calls done by the front-end, having options for showing offset and delay.

- **Map for RIPE Atlas and measurement data**
  To visualize the data offered by RIPE Atlas and our own measurements, a world map using Leaflet was implemented. On this map, the location of the NTP server or servers used, the probe and our vantage point are shown. The map component itself is rather simple in nature. The map is used only on the main page and uses the data from the API calls to display the locations of the points. Additionally, the map is set to automatically adjust its zoom depending on the location of the points that are loaded, which allows for the user to properly visualize where everything is located.

6.3 Back-end Component

6.3.1 API Layer

For a smooth development, the back-end and front-end components are be decoupled. The interaction between the two components is handled through the Application Programming Interface (API), which follows the Representational State Transfer (REST) architectural style. This architectural style focuses on standardized interfaces, allows components to be deployed independently, and supports scalable interactions among them. There are four key endpoints for communicating between the front-end and the back-end:

- **Performing a measurement**

  Users can input either a domain name or an IP address to begin an NTP measurement. This is sent as a `POST` request to the `/measurements` endpoint, in the form of a JSON payload. The payload consists of two components: the server name or IP to be measured and if the measurement should be done using IPv6. The response includes a list of multiple measurements if a domain name was requested by the user, as it can map to multiple IP addresses. In the case of querying just an IP address, a list of only one element is returned. Additionally, it records four key timestamps: when the request was sent by the client, when it was received by the NTP server, when the server sent the response, and when the client received that response. Based on these timestamps, the measurement calculates synchronization metrics including the offset (the estimated time difference between the client and the server), the round-trip time (RTT) and many more. Additionally, jitter is calculated based on the last eight measurements, as this was found to provide the most reliable results. If fewer than eight measurements are available, the jitter is computed using the measurements collected so far.

- **Querying Historical Data**

  After getting the results of a measurement, there is also the option of seeing historical data for different properties (such as offset and RTT), in the form of charts. A `GET` request is be sent to the `/measurements/history` endpoint. This request has three parameters: The server name or IP and the starting and ending points of the interval from which the measurements will be shown. If the ending point is left empty, the current time will be considered. The response from the server to the client consists of a list of specific values for different fields of an NTP measurement, ordered chronologically. They are then displayed in the form of a graph, showing its evolution over time.

- **Triggering a measurement with RIPE Atlas**

  As it was mentioned in the research RIPE Atlas is used for getting more data on the NTP servers that are measured. A `POST` request to `/measurements/ripe/trigger` triggers an external measurement. The payload includes the target server and if the measurement will be done using IPv6. The application then sends a confirmation response that includes: the RIPE measurement ID of the ripe measurement, the IP and geolocation of our the vantage point, a status message that tells the user that the measurement has been started and an instruction telling where they can asynchronously ask for data.

- **Querying the data from RIPE Atlas**

  Since RIPE Atlas measurements are asynchronous, the client must poll for data on the `/measurements/ripe/{measurement_id}` endpoint using a `GET` request. The server checks whether the data is complete, partially complete, or still pending. The response includes either the measurement results or a status update, allowing the client to retry until completion. Compared to the measurements returned on other endpoints, the ones did with RIPE also include the coordinates of the probe.

6.3.2 NTP Measurement Engine

The NTP Measurement Engine is designed to perform time synchronization measurements using the Network Time Protocol. It connects to NTP servers via IP or domain names, collects precise timestamp data, computes time offsets, delays, stratum, and jitter, and formats and stores the results for in depth analysis, that is then displayed via the front-end component to the user.

This engine is implemented in Python using the open-source `ntplib` Python library which is used for connecting and fetching basic information about the time servers. This information is then used on further data analysis which is then presented to the user.

The client recommended this library to the team, as they had a good experience with it in the past. On top of this, they requested a solution that is easy to maintain and extend. Therefore research was done in determining which would be the most feasible solution. In the end, `ntplib` was chosen despite its known limitations in measurement accuracy. While it may not offer the highest precision compared to lower-level or more specialized NTP implementations, the decision was driven by other priorities such as: simplicity, reliability, and maintainability. As a lightweight and well-documented client library, `ntplib` enables accurate-enough NTP queries with minimal configuration, making it well-suited for rapid development and straightforward integration. Its ease of use, lack of need for system-level privileges, and broad platform compatibility outweighed the marginal gains in precision offered by more complex alternatives.

To compute clock offset and RTT of the NTP server, the engine follows the standard four-timestamp approach defined by the NTP protocol. It uses the difference between send and receive times on both client and server sides to estimate the offset and Round-Trip time. Internally, timestamps are represented with a custom PreciseTime class to maintain high precision by separating seconds and fractional seconds. While `ntplib` provides built-in floating-point values for timestamps, it has an open issue related to floating-point rounding errors when calculating offset and delay directly. To ensure accuracy and avoid precision loss, those calculations are handled manually using these formulas on the raw NTP timestamps:

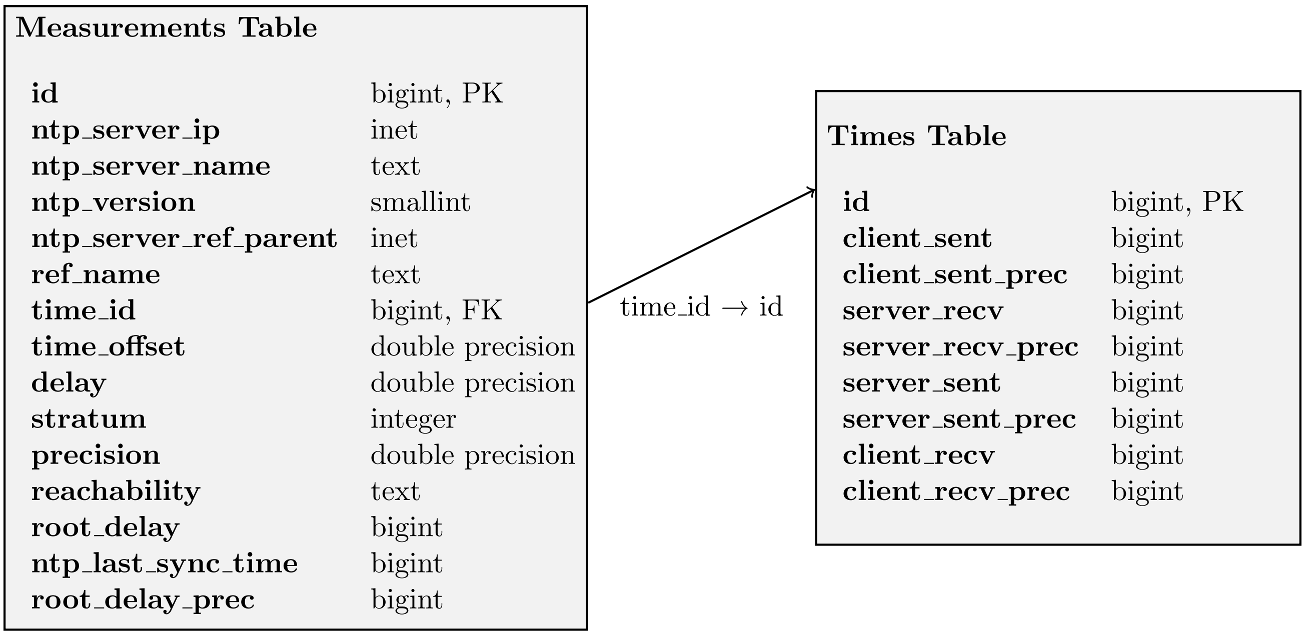
$$\text{Offset} = \frac{(T_2 - T_1) + (T_3 - T_4)}{2} \qquad \text{RTT} = (T_4 - T_1) - (T_3 - T_2)$$

$$\begin{aligned} &T_1 : \text{Client sends request (Client Transmit Time)} \\ &T_2 : \text{Server receives request (Server Receive Time)} \\ &T_3 : \text{Server sends response (Server Transmit Time)} \\ &T_4 : \text{Client receives response (Client Receive Time)} \end{aligned}$$

6.3.3 Data Storage

Our data storage solution is powered by PostgreSQL, a robust relational database system known for its reliability, scalability, and support for complex queries. The design is optimized for efficient storage, retrieval, and analysis of NTP measurement data. An overview of the database schema, consisting of two primary tables, can be seen in Figure 6.6.

Data integrity is ensured through the use of foreign key constraints, linking the measurements table to the times table, maintaining consistency. Indexing on the primary key of the measurements table accelerates data retrieval, while database transactions guarantee atomicity for multi-step operations, such as the insertion of measurement and associated timestamps. Regular maintenance tasks, including index optimization and vacuuming, are scheduled to maintain consistent performance.



Measurements Table

id	bigint, PK
ntp_server_ip	inet
ntp_server_name	text
ntp_version	smallint
ntp_server_ref_parent	inet
ref_name	text
time_id	bigint, FK
time_offset	double precision
delay	double precision
stratum	integer
precision	double precision
reachability	text
root_delay	bigint
ntp_last_sync_time	bigint
root_delay_prec	bigint

Times Table

id	bigint, PK
client_sent	bigint
client_sent_prec	bigint
server_recv	bigint
server_recv_prec	bigint
server_sent	bigint
server_sent_prec	bigint
client_recv	bigint
client_recv_prec	bigint

Figure 6.6: PostgreSQL Database Schema

6.3.4 External Data Integration

- **MaxMind database integration**
  The MaxMind database was used to fetch an approximate location of the client in order to select the closest probes next to the user. This publicly available database was chose over external API due to several advantages. MaxMind's local database is not subject to rate limits, allowing an unlimited number of queries without additional cost, unlike many exteral APIs, such as IPInfo. Hosting the database locally also ensures that no user IP data is sent to third-party services, enhancing data control and addressing privacy concerns.

- **Fetching data from RIPE Atlas**
  After the probes are successfully selected and the measurement is scheduled, the back-end starts to fetch the raw results by sending a `GET` request to:
  `https://atlas.ripe.net/api/v2/measurements/{measurement_id}/results`.

  These results are then parsed into structured `RipeMeasurement` objects. Each object encapsulates data from a single probe, including an associated `NtpMeasurement`, which contains details about the measurements (i.e offset, stratum, etc.).

  Additionally, for each probe, metadata is fetched with a `GET` request to:
  `https://atlas.ripe.net/api/v2/probes/{probe_id}`, that also returns raw data that is then parsed into a `ProbeData` object, that includes the IP addresses (both IPv4 and IPv6 if available), country code, and geolocation coordinates.

The back-end also checks the measurement status by sending a `GET` request to `https://atlas.ripe.net/api/v2/measurements/{measurement_id}`. This polling continues until all probes have either completed or the measurement is marked as stopped. Because RIPE Atlas sends measurement data asynchronously, polling provides a practical way to determine when results are ready. If a measurement times out without receiving all expected results, an appropriate timeout status is returned to the front-end.

To ensure robust parsing, the system handles failed measurements by checking if all result entries are marked with an error indicator (typically the `"x":"*"` field). In such cases, default values are inserted, as in the case of missing fields. For successful measurements, the entry with the lowest offset is selected as representative.

As RIPE Atlas measurements are initiated externally and their results become available asynchronously at unpredictable times, we opted for client-side polling as the method to retrieve the data. This decision was based on various considerations. Infrastructure independence is one of them. Alternatives such as WebSockets or Server-Sent events would require long-lived connections and more complex state management. Webhooks, while efficient are not viable in this case because the back-end has no control over when RIPE completes a measurement or where to send a callback. Another aspect that was considered was resilience. Measurements can take varying amounts of time, and polling provides a robust way for clients to check for completion without depending on event streams or external triggers. Additionally, what makes polling viable is the presence of server-side rate-limiting, which ensures the server remains protected from overload even if clients repeatedly query for updates.

6.3.5 Rate Limiting

To prevent abuse and ensure fair usage, the back-end enforces rate limiting on critical endpoints using the `slowapi` middle-ware, which is based on the `limits` library. Each of the main API endpoints is protected with a limit of five requests per second per client IP.

The rate limiter is applied with the `@limiter.limit("5/second")` decorator on each route.

Client identification is based on the IP address extracted from the `X-Forwarded-For` header or, if unavailable, from the connection metadata provided by the FastAPI `Request` object.

If the request rate exceeds the defined threshold, the server responds with HTTP status code `429 Too Many Requests`, informing the client to slow down.

With this mechanism in place, issues like: excessive or malicious requests from individual clients, back-end overload due to repeated retries, resource exhaustion during database queries and RIPE Atlas measurements can be mitigated. Additionally, this also provides lightweight protection without the complexity of full authentication or quota systems, making it suitable for our use case.

6.4 Technologies Used

When choosing the technologies for this project, we tried to stick to a few main principles. Firstly, since the application will be open-source, it needs to be maintainable and scalable. Secondly, since we are dealing with the very precise field of NTP servers, accuracy is essential, so we need to focus on performance and data robustness. An overview of the technologies we used can be seen in Figure 6.7.

6.4.1 Front-end

React.js is the main framework chosen for the Web application. Given this is an open-source project, we decided to put maintainability and scalability first. React's comprehensive documentation and robust community support made it an ideal choice. Furthermore, its modular and component-based nature is particularly suited for team projects, as it allows components to be developed independently and integrated seamlessly.

Another important choice for the front-end component is the usage of **TypeScript**, as opposed to the more traditional **JavaScript**. As mentioned earlier, our client specified the importance of very exact data types and type-checking, which is the main advantage of the static type system that TypeScript provides. Static typing also makes the code more robust in this case, as interactions are easier to model and verify, which allows for better detection of possible bugs. This makes the code easier to maintain and easier to understand for potential future developers (who might see the codebase for the first time).

Vite is used to actually serve the page(s) to the React application. Our client specifically mentioned that we should avoid using a custom Node.js / Express.js server, as the pages would be rendered server-side, which impacts performance. With React and Vite, Node.js is only used during development, while in production the pages are served as static HTML/CSS/JS files.

For facilitating HTTP requests, **Axios** was used over the conventional **Fetch**. It is a more convenient way to receive and transmit requests. The main features that it provides are that requests are directly handled as JSON requests instead of having to be converted manually, it throws errors directly instead of having to be checked manually, and it has a simpler syntax. Moreover, it provides more options in the event of future expansion of the app, which makes it a more robust option, enhancing the maintainability and scalability of the app.

6.4.2 Back-end

For the programming language, we considered **Python** to be the best choice, specifically for the `nptlib` library. This library was specifically recommended by our client due to its efficiency, low overhead, and lack of external dependencies, making it a lightweight and dependable choice for querying NTP servers, as mentioned in Section 6.3.2. The community support for the libraries we chose to use is another factor that we took into consideration.

To guarantee data integrity and reliability, we opted for **PostgreSQL** as our database. This choice aligns with our principle of robustness by providing strict data typing, transactional support, and advanced querying capabilities that are crucial for managing precise and reliable data. It can also be seamlessly integrated with Python, reducing the probability of unexpected compatibility problems. Our client suggested using DuckDB as an alternative, however we opted for PostgreSQL due to the reasons mentioned above and due to concurrency issues in DuckDB. Another problem with DuckDB is that being on disk means that the server of the application represents a single point of failure, while PostgreSQL runs on a separate server.

Python was used for the API as well, specifically the **FastAPI** framework. As mentioned, we wanted to focus on performance, so the simplicity, lightweight nature, and asynchronous capabilities of FastAPI made it the best choice for our application (Wu, 2024). The asynchronous capabilities are essential for efficiently handling multiple concurrent NTP queries and calls to other external APIs, such as RIPE Atlas API. As mentioned earlier, the API consists of only four endpoints and the data is relatively simplistic, so there is no need to add unnecessary complexity or overhead that come with more popular options such as Django and Flask.

To protect the back-end from abuse and ensure fair resource usage, we integrated **SlowAPI** for rate limiting. This library was chosen for its compatibility with FastAPI, with simple decorator-based syntax, and built-in support for IP-based throttling. By limiting requests to a fixed rate (e.g., 5 requests per second), it helps prevent server overload and ensures a stable user experience, especially when multiple concurrent queries are issued or expensive operations (like RIPE measurements) are involved.

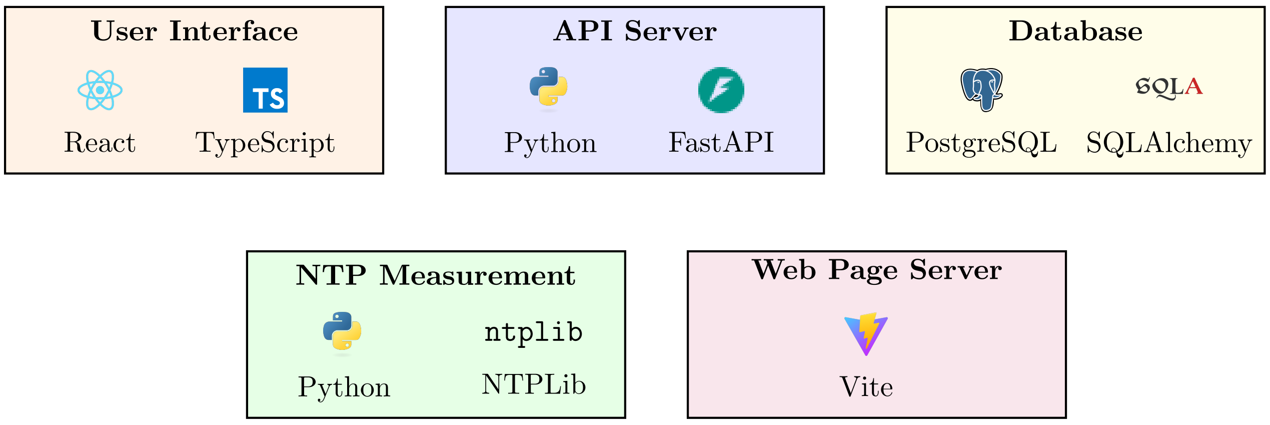


Figure 6.7: Overview of the technologies used

6.4.3 Docker containerization

To deploy the app and ensure a consistent and reproducible environment, the entire application stack has been containerized using **Docker**. This approach abstracts away machine-specific configurations and enables seamless deployment to any platform that supports **Docker**. The application is composed of three main services, each encapsulated in its own container and orchestrated using **docker-compose**.

Front-end: The containerization process uses a custom **Dockerfile** and is split into a two-stage build. In the first stage, a **Node.js** environment is used to install dependencies and build the front-end app. In the second stage, the built static files are served by a lightweight **Nginx** container. Environment-specific configuration values, such as the server host address and status threshold, are passed as build arguments to ensure flexibility across different deployment environments.

Back-end: The back-end is containerized using a custom **Dockerfile** based on the **python:3.11-slim** image, and includes additional functionality beyond a standard FastAPI runtime. Because the app relies on external dependencies (**MaxMind** and **BGP Tools** databases) that have to be updated daily using, a dynamic **cron** job is configured at startup using an entrypoint script, with the schedule injected via an environment variable. Sensitive values like account IDs and license keys are stored in a custom environment file (/etc/cron.env) and securely sourced by the **cron** job. A pre-start script is executed on container launch to set everything up, after which the **FastAPI** application is started.

Database: A standard **PostgreSQL** container is used, based on the official image. Database credentials and initialization options are configured using environment variables provided in the **docker-compose.yml**.

Network configuration: All services are connected via a custom Docker network named `my-net`. This network uses the bridge driver and is explicitly configured to support IPv6 by specifying a custom subnet and gateway. The subnet and gateway values are sourced from environment variables, making it more configurable, without the need of changing the compose file. A custom network was needed because of **Docker's** limitation in handling IPv6 by default. Unlike IPv4, where containers can communicate with external servers via **Network Address Translation (NAT)**, IPv6 does not support **NAT** in the same way. As a result, containers require globally routable IPv6 addresses to receive responses from external servers, as in most default **Docker** setups, containers can initiate outbound IPv6 connections, but external responses are dropped because the return path is not routable. By defining this dedicated **Docker** network, assuming that the host system itself has an IPv6 address, containers are able to communicate properly with external IPv6 services.

7 Ethical Implications

To build an accurate tool for measuring NTP servers, we need to process certain user data. This chapter explores the ethical implications of doing measurements on behalf of the user. Section 7.1 focuses on privacy concerns, especially around the use of IP addresses to improve result relevance. Section 7.2 discusses how automatic measurements performed on the user's behalf could affect how the user interprets or uses the data. Throughout, we aim to identify and mitigate potential risks while maintaining the tool's effectiveness.

7.1 Handling of User IP

For a better evaluation of NTP server quality the application processes the IP of the user to select RIPE probes in a manner that is more relevant to the user. This is made apparent to the user by a popup first appearing on the page, stating how the IP is used. The specifics are that the IP is always prefixed, meaning that the exact IP address is never processed. It is also not transmitted to any external resources, as it is only used for geolocation data using Maxmind, and the IP is never stored. As mentioned in Chapter 3, the geolocation information is only used to select RIPE probes that are closest to the user in the internet space, which allows for data more relevant to the user to be shown, as probes further away might offer data unreflective of their own use case.

There are multiple methods set in place to prevent the leakage of personal information. As mentioned before, only a masked version of the IP will be used, and the data is not saved at any point in the process of gathering which probes are closest to the user. A concern of the user could be the grabbing of the IP when it is sent from the front-end to the back-end in the header of the API request. However, this depends entirely on the deployment of the app. If deployed properly, with a private back-end server, and a public front-end server that has a closed communication with said back-end, the data sent between the two should not be able to get intercepted. Moreover, if the web app is not deployed properly and does not use HTTPS for secure communication, the concern increases, as is it easier for an attacker to attack the user. However, since this product is open-source and depends on the specific deployment practices of the individuals, which in theory could also modify the app themselves to facilitate the grabbing of personal data, it is out of the scope of our product since it is not a controllable variable. Despite this, we do recommend that the future users who deploy the app will take into consideration all safety precautions in order to protect the information of the people using it.

7.2 Accuracy of Results and Generated Data

In regards to the contributions of the application to the users of the app, we are responsible for offering reliable and accurate data. Our app can be deployed, and since its function is to facilitate data collection for the user, accurate and relevant results will be able to enhance the user experience. However, since it provides an easy alternative to people gathering the data themselves, it might skew their interpretations.

For the retrieval of data from NTP servers, the `ntplib` python library is used, and for measurements from the RIPE probes, information offered directly by RIPE Atlas is used. For general usage, this information is accurate enough, offering consistent millisecond precision, which is then made public for access and downloading to anybody using the application. Due to the increased accessibility of the app, some users could choose to only gather the data we provide and use it as is. However, for other applications of accurate time data, such as financial transactions or scientific measurements, this accuracy is not enough. In cases like these, users would be required to also go to other sources for data with a higher accuracy for their uses.

In addition to everything mentioned before, there is also an issue about offering this data with such widespread availability, specifically for the extractions of information for malicious purposes. This, while not being a point of concern, is also outside of the scope of the project. All the tools that were used for gathering data in the application, as well as the application itself are open-source and free to use. This means that all information gathered can also be replicated, and even tampered by individuals, which is outside of the realm of possibility for being controlled. The only possible prevention would be the taking down of specific websites, similarly to other internet applications, not just this specific project.

8 Testing & Quality Control

This chapter discusses the techniques and tools that we used to ensure that our product has the best quality. Section 8.1 explains how we approached testing the system, from individual components to the way they interact. Section 8.2 dives into how the project was documented to ensure users and future developers can easily understand it. Section 8.3 presents the different code checks we have put in place to ensure the robustness of our codebase.

8.1 Testing

8.1.1 Unit Testing

Before testing how the different components of the system work together, we need to make sure they each function as expected individually. For unit testing, we used `Pytest` on the back-end. `Pytest` is the most common testing library for Python and it offers a large set of features that make it well-suited for writing scalable and maintainable tests, such as fixtures, mocking, parameterization and detailed assertions. Some parts of the application, such as the database logic and the front-end logic, were hard to unit test, require more complex interaction with other modules, and as such are difficult to test in isolation. This is where we switched our attention to integration testing.

8.1.2 Integration Testing

After exhaustively unit testing every part of the system, integration testing has been used to test complex interactions between system components. There were two main areas where integration testing was employed: testing the integration of the API endpoints with the database and testing the integration of the RIPE Atlas API with the rest of the system.

For testing the integration of the FastAPI endpoints and the PostgreSQL database, instead of mocking the interaction, like we did for unit testing, we set up a test database and we used Pytest fixtures to simulate test versions of the application and the client, that depend on the test database instead of the real one. At one point, we considered using an in-memory SQLite database, however that seemed more like enhanced unit testing, so we chose to create a PostgreSQL test database for real integration testing.

Another important aspect was testing the integration of our application with the external RIPE Atlas API. Since we had a large amount of credits for performing RIPE measurements, there was no need to set up a separate account for testing. Due to time limitations (RIPE measurements taking very long to actually give full data), it was nearly impossible to test the full "lifetime" of a measurement, from triggering it to fetching data. Instead, we tested the status of a measurement after triggering it and whether the actual measurement ID was received or not. For actually fetching the RIPE Data, we chose some actual past measurements done by us, which had known behaviour, to actually test the data that was received.

8.1.3 Front-end Testing

On the front-end, `Vitest` was the main testing library used, as it incorporates well with Vite and allows for very flexible configurations (e.g. running with coverage or getting a nice UI when running the tests). The first part we had to test were the utility functions, which are used in many different parts of the front-end component and mostly deal with data conversion.

Furthermore, we tested the React API Hooks to ensure the correctness of the interaction between the front-end and the back-end. Initially, we tried to do this by running the back-end with a script and directly testing the calls. However, this proved to be very inefficient, as tests could take a very long time to give results in some cases and they had to run sequentially. This not only wasted time, but also made the tests flaky. We then switched to using Mock Service Worker to mock the back-end and only focus on the actual interaction. Not relying on the real back-end led to faster tests and more consistent results, as we only tested the calls and checked if they received and transformed the data correctly.

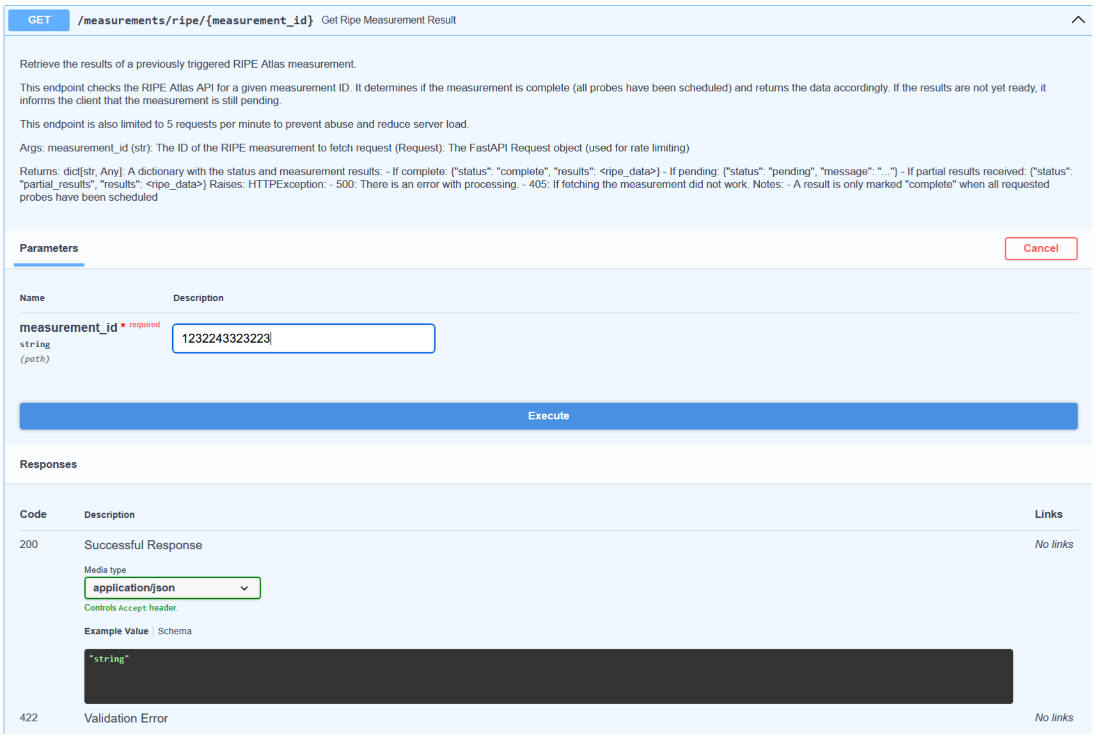
Finally, we moved on to testing the React UI Components, using `React Testing Library` together with Vitest. Due to lack of time, only components such as buttons and input forms, containing behaviour and functionality, were able to be tested. Interactions were mocked to ensure proper isolation. After these were properly tested, we could start with integration testing.

Appendix C shows more details about the line and branch coverage of the tests.

8.2 Documentation

To support not only seamless collaboration, but also future development, the codebase has been extensively documented. Every class and function is documented using inline comments, Python Docstrings (back-end) and TSDoc (front-end). Furthermore, there is a `README.md` file in the repository, containing a detailed description of the project and how to use it.

The API endpoints were documented according to OpenAPI standards. This allows for better visualization of the whole endpoint architecture, with the documentation of each endpoint including the following: A short description, the type of request, the expected parameters and/or payload, the response format, the expected status codes, and examples of requests and responses. An example of this can be seen in Figure 8.1.



GET `/measurements/ripe/{measurement_id}` Get Ripe Measurement Result

Retrieve the results of a previously triggered RIPE Atlas measurement.

This endpoint checks the RIPE Atlas API for a given measurement ID. It determines if the measurement is complete (all probes have been scheduled) and returns the data accordingly. If the results are not yet ready, it informs the client that the measurement is still pending.

This endpoint is also limited to 5 requests per minute to prevent abuse and reduce server load.

Args: measurement_id (str): The ID of the RIPE measurement to fetch request (Request): The FastAPI Request object (used for rate limiting)

Returns: dict[str, Any]: A dictionary with the status and measurement results: - If complete: {"status": "complete", "results": <ripe_data>} - If pending: {"status": "pending", "message": "..."} - If partial results received: {"status": "partial_results", "results": <ripe_data>} Raises: HTTPException: - 500: There is an error with processing. - 405: If fetching the measurement did not work. Notes: - A result is only marked "complete" when all requested probes have been scheduled

Parameters

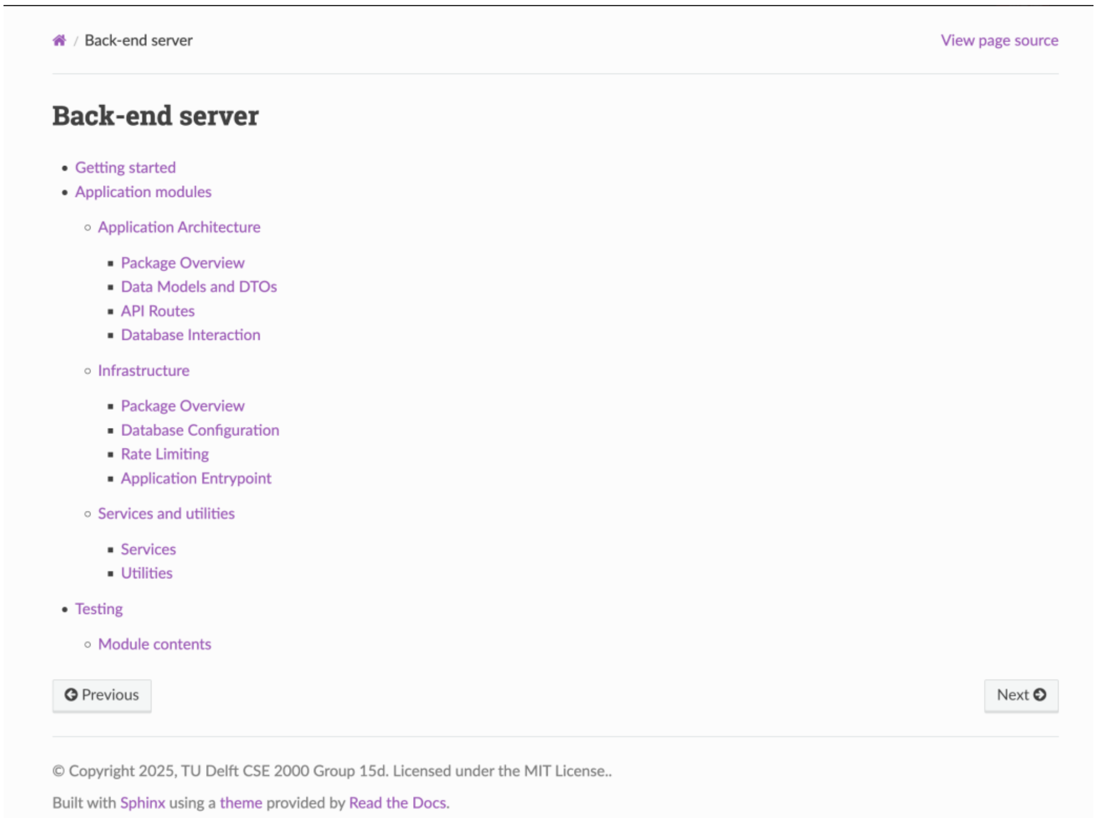
Name	Description
measurement_id * required string (path)	1232243323223

Responses

Code	Description	Links
200	Successful Response Media type: application/json Controls Accept header. Example Value | Schema "string"	No links
422	Validation Error	No links

Figure 8.1: OpenAPI Documentation of the `/measurements/ripe` endpoint for fetching RIPE Data, including a description and example input and output

The back-end component was additionally documented using `Sphinx`. The documentation includes not only the Python docstrings present in the code, but also an overview of the back-end architecture and introductions to all the different parts of it, as well as more detailed descriptions of the functions and classes. Figure 8.2 shows the architecture overview, while Figure 8.3 displays an example of the documentation of a function.


Back-end server   View page source

Back-end server

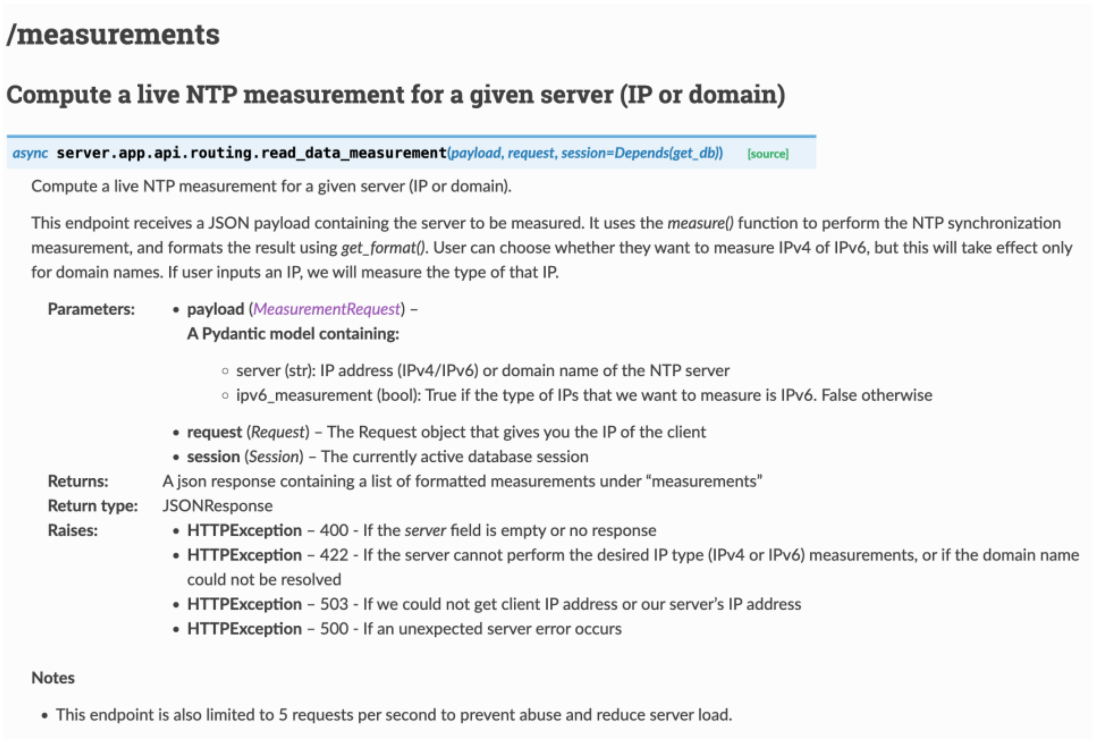
- Getting started
- Application modules
  - Application Architecture
    - Package Overview
    - Data Models and DTOs
    - API Routes
    - Database Interaction
  - Infrastructure
    - Package Overview
    - Database Configuration
    - Rate Limiting
    - Application Entrypoint
  - Services and utilities
    - Services
    - Utilities
- Testing
  - Module contents

Previous   Next

© Copyright 2025, TU Delft CSE 2000 Group 15d. Licensed under the MIT License..

Built with Sphinx using a theme provided by Read the Docs.

Figure 8.2: The overview of the Sphinx back-end documentation


/measurements

Compute a live NTP measurement for a given server (IP or domain)

```
async server.app.api.routing.read_data_measurement(payload, request, session=Depends(get_db))    [source]
```

Compute a live NTP measurement for a given server (IP or domain).

This endpoint receives a JSON payload containing the server to be measured. It uses the *measure()* function to perform the NTP synchronization measurement, and formats the result using *get_format()*. User can choose whether they want to measure IPv4 of IPv6, but this will take effect only for domain names. If user inputs an IP, we will measure the type of that IP.

Parameters:
- **payload** (*MeasurementRequest*) –
  A Pydantic model containing:
  - server (str): IP address (IPv4/IPv6) or domain name of the NTP server
  - ipv6_measurement (bool): True if the type of IPs that we want to measure is IPv6. False otherwise
- **request** (*Request*) – The Request object that gives you the IP of the client
- **session** (*Session*) – The currently active database session

Returns: A json response containing a list of formatted measurements under "measurements"

Return type: JSONResponse

Raises:
- **HTTPException** – 400 - If the *server* field is empty or no response
- **HTTPException** – 422 - If the server cannot perform the desired IP type (IPv4 or IPv6) measurements, or if the domain name could not be resolved
- **HTTPException** – 503 - If we could not get client IP address or our server's IP address
- **HTTPException** – 500 - If an unexpected server error occurs

Notes
- This endpoint is also limited to 5 requests per second to prevent abuse and reduce server load.

Figure 8.3: Sphinx documentation of the `read_data_measurement` API function

8.3 Code checks

The first type of check we have employed for our code is type checking. Our client mentioned we need to have very exact types, as the field of time server relies on precision. Since Python is not naturally a strongly-typed language, we decided to use the `mypy` static type checker to enforce types of parameters and returned values. On the front-end, the use of Typescript, as opposed to Javascript, already allowed us to enforce types more strictly.

We also enforced consistent Checkstyle rules, using Flake8 on the back-end. We focused on enforcing proper indentation, spacing, and cyclomatic complexity (with a limit of 10), as well as docstring for every function. These rules ensured readability and maintainability, but we also added project-specific constraints such as limiting the maximum line length to 140, and excluding virtual environments and comments from checks to keep Checkstyle relevant. On the front-end, ESLint came packaged with React and enforced similar rules, but also ensured no unused imports were present.

9 Conclusion & Further Recommendations

9.1 Summary & Insights

The aim of this report was to analyze how a public, open-source web application could be developed to measure the accuracy and quality of NTP servers in real-time, as well as describing the development process of this tool while taking into consideration technical aspects and usability. The solution that was achieved consists of 2 main parts, the front-end and back-end, which each interact with each other to send data. The front-end contains the actual website that is displayed to the user, which is where they can interact with the app, and shows the data it gets from the back-end via the Application Programming Interface (API). This allows them to view and use the data we provide for any purposes they might need them for. The back-end is the part of the app that does the calculations, retrieves the data from other sources, and then processes them to send to the user. It uses the API to send these data to the front-end after performing queries to the user's choice of server.

In terms of technologies used, research was done in each situation to find the best method of achieving the intended results, while also keeping in mind the scope of the project and the future scalability requirements enforced. This led to a lot of open-source, lightweight and well-maintained libraries and dependencies being used, such as `ntplib` for gathering data from NTP servers, or `ChartJS` for making graphs to properly let the user visualize the data we transmit.

Regarding the requirements set at the beginning of the project, all of the must and should haves have been implemented with great success, and the additional could haves that were done have been chosen by priority according to the client's wishes. The client has been very satisfied with the product that has been created. The primary goal of receiving data from NTP servers, and displaying them with proper visualizations has been fully delivered, with additional information being provided with the RIPE Atlas API data we gather, allowing the user to receive additional information that pertains to their situation more closely. The requirements that were not reached were all lower priority, with the main goal of them being improving the user experience, such as offering different languages or having a typo checker for the user's input.

9.2 Possible future improvements

The first improvements that can be tackled are the parts that could not be finished in the "could have" section of the requirements (see Appendix A). This includes adding a light mode and high contrast theme, as well as a typo checker to help users correct common mistakes. Some other improvements could be made to the HCI of the application, such as adding support for a color-blind mode or improving accessibility for people with disabilities in general. Another possible improvement would be to implement some sort of caching for recent results, so users can get more accurate data about the jitter and query recent historical data without putting too much of a load on the server. A possible new feature could be the addition of a leaderboard for the best performing NTP servers.

This leaderboard would rank NTP servers based on their performance metrics such as offset, delay, and jitter. This feature would not only provide valuable insight into the best publicly available NTP servers but would also encourage healthy competition among NTP server hosts to improve their services. Another significant enhancement we would have liked to implement is an advanced charting system that allows users to dynamically adjust the displayed time period through interactive zooming capabilities. This improvement would substantially improve the utility and convenience of our data visualization, enabling users to focus on specific time intervals of interest with greater precision and ease. The zooming functionality would allow for a more detailed examination of measurement trends, making it easier to identify patterns, anomalies, and performance fluctuations over time.

References

1. Al-Dalky, R., & Rabinovich, M. (2020). *Revisiting comparative performance of DNS resolvers in the IPv6 and ECS era.* arXiv.
https://arxiv.org/pdf/2007.00651

2. Bajpai, V., Eravuchira, S. J., Schönwälder, J., Kisteleki, R., & Aben, E. (2017). Vantage point selection for IPv6 measurements: Benefits and limitations of RIPE Atlas tags. In *IFIP/IEEE Symposium on Integrated Network and Service Management (IM)* (pp. 37-44). IEEE.
https://ieeexplore-ieee-org.tudelft.idm.oclc.org/abstract/document/7987262

3. Contavalli, C., van der Gaast, W., Lawrence, D., & Kumari, W. (2016). *Client subnet in DNS queries* (RFC 7871). Internet Engineering Task Force.
https://datatracker.ietf.org/doc/html/rfc7871

4. Dash, P., & Hu, Y. C. (2021). How much battery does dark mode save? An accurate OLED display power profiler for modern smartphones. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '21)* (pp. 323–335). Association for Computing Machinery.
https://doi.org/10.1145/3458864.3467682

5. Izyumov, P. S., & Ivchenko, A. V. (2024). *Analysis of network state by RIPE Atlas distributed measurement system.* IEEE.
https://ieeexplore-ieee-org.tudelft.idm.oclc.org/stamp/stamp.jsp?tp=&arnumber=10510129

6. Morowczynski, M. (2020). Did your Active Directory domain time just jump to the year 2000? Microsoft Tech Community.
https://techcommunity.microsoft.com/blog/coreinfrastructureandsecurityblog/did-your-active-directory-domain-time-just-jump-to-the-year-2000/255873

7. Moura, G. C. M., Davids, M., Schutijser, C., Hesselman, C., Heidemann, J., & Smaragdakis, G. (2024). Deep dive into NTP pool's popularity and mapping. *Proceedings of the ACM on Measurement and Analysis of Computing Systems, 8*(1), 1–30.
https://doi.org/10.1145/3639041

8. Streibelt, F., Böttger, J., Chatzis, N., Smaragdakis, G., & Feldmann, A. (2013). Exploring EDNS-client-subnet adopters in your free time. *Proceedings of the 2013 Conference on Internet Measurement Conference (pp. 305-312).* ACM.
https://dl-acm-org.tudelft.idm.oclc.org/doi/abs/10.1145/2504730.2504767

9. Strowes, S. D. (2017). *Bootstrapping Active IPv6 Measurement with IPv4 and Public DNS.* arXiv.
https://arxiv.org/abs/1710.08536

10. Wu, X. (2024). *FastAPI as a backend framework* [Bachelor's thesis, Tampere University of Applied Sciences]. Theseus.
https://www.theseus.fi/bitstream/handle/10024/855956/Wu_Xiaomin.pdf?sequence=3

Appendix A: Requirements Overview

Must have

1. The system **must** include a back-end component that performs live NTP (Network Time Protocol) queries to user-specified servers.
2. The user **must** be able to use domain names or IP addresses as input for the NTP server.
3. The system **must** implement basic input validation to ensure that provided NTP server names or IP addresses are valid.
4. The front-end **must** provide a form interface allowing users to enter an NTP server address and initiate a test.
5. The system **must** use a JSON API endpoint to return NTP measurement results, including offset, delay, stratum, and other relevant data.
6. The front-end **must** display the results of NTP tests, including server status, offset, delay, and similar metrics.
7. The system **must** persistently store all NTP test results in a database.
8. The user interface **must** be clean and intuitive, with result visualizations such as tables and colored indicators to reflect server status and quality.

Should have

1. The system **should** be integrated with the RIPE Atlas API to enable additional measurements from multiple global vantage points (Moura, 2024).
2. The system **should** display results received from the RIPE Atlas API next to the ones received from the NTP measurement, for comparison.
3. The system **should** provide aggregated historical statistics per NTP server, such as average offset and minimum/maximum delay.
4. Users **should** be able to compare multiple NTP servers side by side.
5. The front-end **should** include graphical charts that display trends, such as offset over time.
6. The system **should** support querying historical data, including measurements from the last hour, day, or week.
7. Servers **should** be automatically tagged as “passing”, “caution”, or “failing” based on the offset of the measurements.
8. The system **should** support connecting to NTP servers that have pool modes and **should** indicate the IP of the specific server that was chosen.
9. The server should support doing measurements on both IPv4 and IPv6 IPs.

Could have

1. Users **could** be given the option to export results in formats such as JSON or CSV.
2. The front-end **could** support multiple languages, allowing users to change the interface language.
3. A light mode and high contrast theme **could** be implemented to enhance accessibility and user preference. Dark mode will be the base version (Dash, 2021).
4. The system **could** visualize the geographic locations of servers using IP-based geolocation data.
5. The system **could** provide a map-based visualization showing the distance and latency between the user's connection point and the queried NTP server. When using the RIPE Atlas API, it could additionally display the distance and latency between various global vantage points (probes) and the target NTP server.
6. If a test has been recently conducted on a server, the front-end **could** display the most recent result to avoid unnecessary re-queries.
7. The system **could** include a typo checker to help users correct common mistakes when entering server names.

Won't have

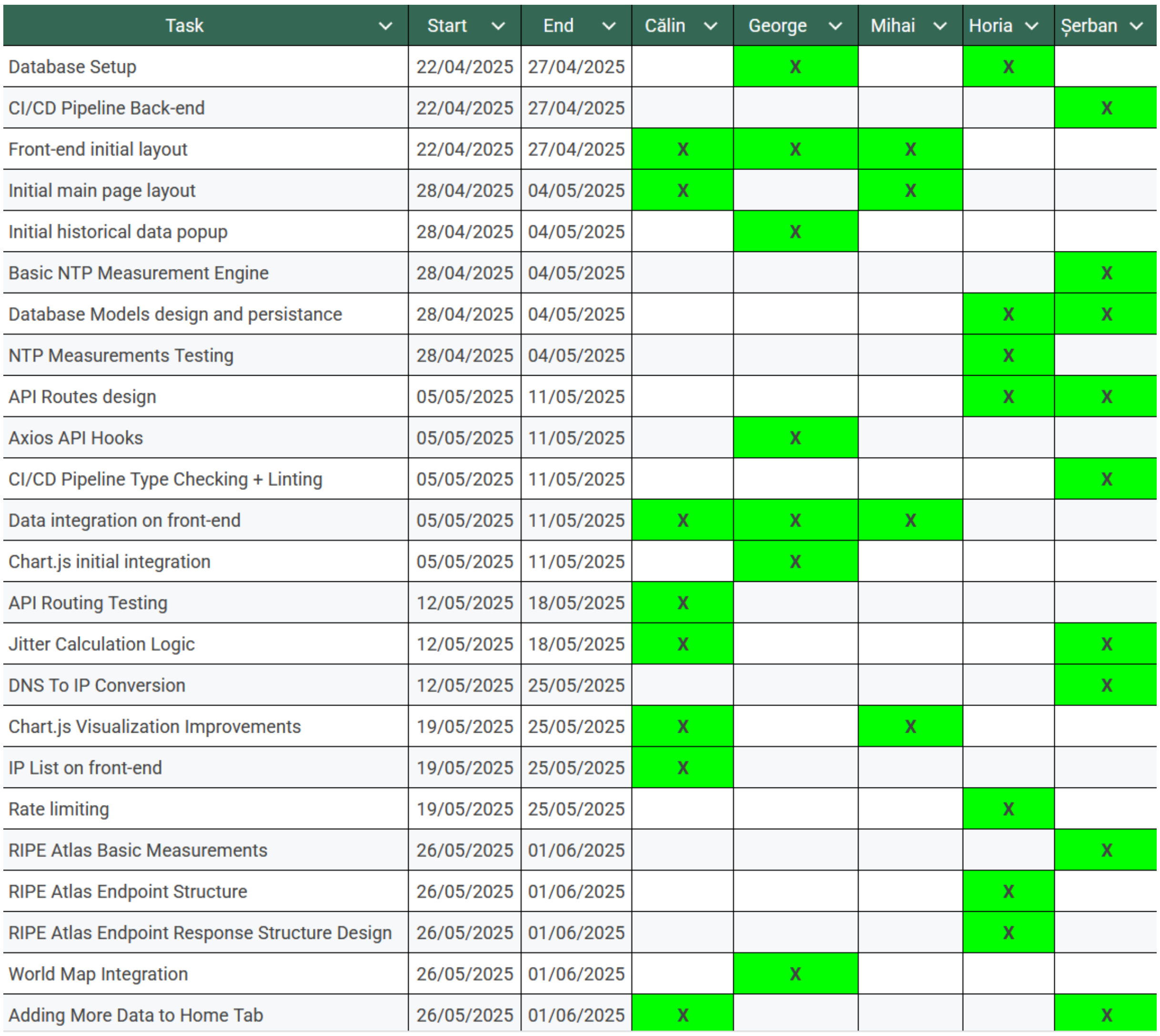
1. The system **will not** include a user login or account management system at this stage.
2. The current project scope **will not** contain a mobile app version.
3. The system **will not** contain advertisements, ensuring an ad-free user experience.
4. The system **will not** allow for users to put in requests to collect data over time.

Non-Functional Requirements

1. The entire project shall be made publicly available on GitHub after its completion, under an open-source license, with clear and complete documentation, to ensure transparency and the possibility of community collaboration.
2. The application shall implement a rate-limiting mechanism, both globally and per user, to prevent excessive load on the back-end and external NTP servers.
3. The code shall be written in a concise and modular fashion, as well as be properly documented, to ensure the sustainability and maintainability of the system.
4. The system shall only use libraries and dependencies that are actively maintained and have stable version releases to ensure long-term maintainability.
5. The back-end component shall be rigorously tested using `pytest`, including unit, integration, and system tests to validate the behaviour of modules in isolation, as well as together.

6. The front-end component shall be tested using React Testing Library to verify UI functionality.
7. The automated test suite shall achieve a minimum of 80% branch coverage, ensuring code correctness.
8. The system shall provide a visual response to the user within 6 seconds of initiating a measurement, ensuring low-latency performance. There will also be a single retry attempt in case of connection timeout, which will be properly conveyed on the front-end as well.
9. The system shall use the `ntplib` python library to query the NTP servers and obtain the accurate time information.
10. PostgreSQL shall be used to store the results of the measurements in a persistent database.
11. React.js shall be used for the front-end component, employing component-based design and state management for scalability.
12. The endpoint architecture shall be implemented using the FastAPI framework to enable asynchronous request handling.
13. The atomic clocks from SIDN Labs with which the other servers will be compared to are considered to be fully accurate as a ground truth.

Appendix B: Work Distribution



Task	Start	End	Călin	George	Mihai	Horia	Şerban
Database Setup	22/04/2025	27/04/2025		X		X	
CI/CD Pipeline Back-end	22/04/2025	27/04/2025					X
Front-end initial layout	22/04/2025	27/04/2025	X	X	X		
Initial main page layout	28/04/2025	04/05/2025	X		X		
Initial historical data popup	28/04/2025	04/05/2025		X			
Basic NTP Measurement Engine	28/04/2025	04/05/2025					X
Database Models design and persistance	28/04/2025	04/05/2025				X	X
NTP Measurements Testing	28/04/2025	04/05/2025				X	
API Routes design	05/05/2025	11/05/2025				X	X
Axios API Hooks	05/05/2025	11/05/2025		X			
CI/CD Pipeline Type Checking + Linting	05/05/2025	11/05/2025					X
Data integration on front-end	05/05/2025	11/05/2025	X	X	X		
Chart.js initial integration	05/05/2025	11/05/2025		X			
API Routing Testing	12/05/2025	18/05/2025	X				
Jitter Calculation Logic	12/05/2025	18/05/2025	X				X
DNS To IP Conversion	12/05/2025	25/05/2025					X
Chart.js Visualization Improvements	19/05/2025	25/05/2025	X		X		
IP List on front-end	19/05/2025	25/05/2025	X				
Rate limiting	19/05/2025	25/05/2025				X	
RIPE Atlas Basic Measurements	26/05/2025	01/06/2025					X
RIPE Atlas Endpoint Structure	26/05/2025	01/06/2025				X	
RIPE Atlas Endpoint Response Structure Design	26/05/2025	01/06/2025				X	
World Map Integration	26/05/2025	01/06/2025		X			
Adding More Data to Home Tab	26/05/2025	01/06/2025	X				X

Figure B.1: Work Matrix Tasks 1-25

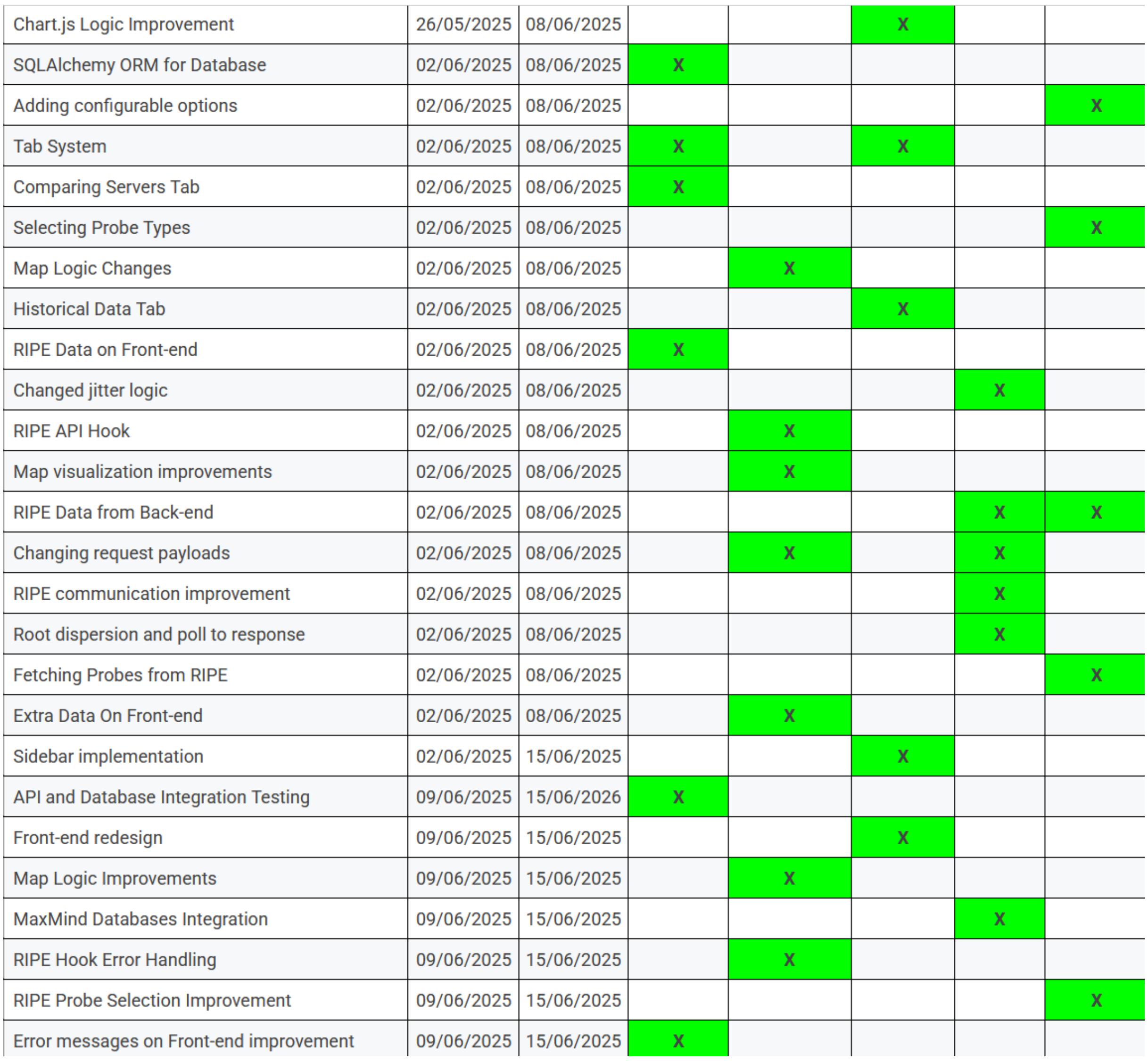
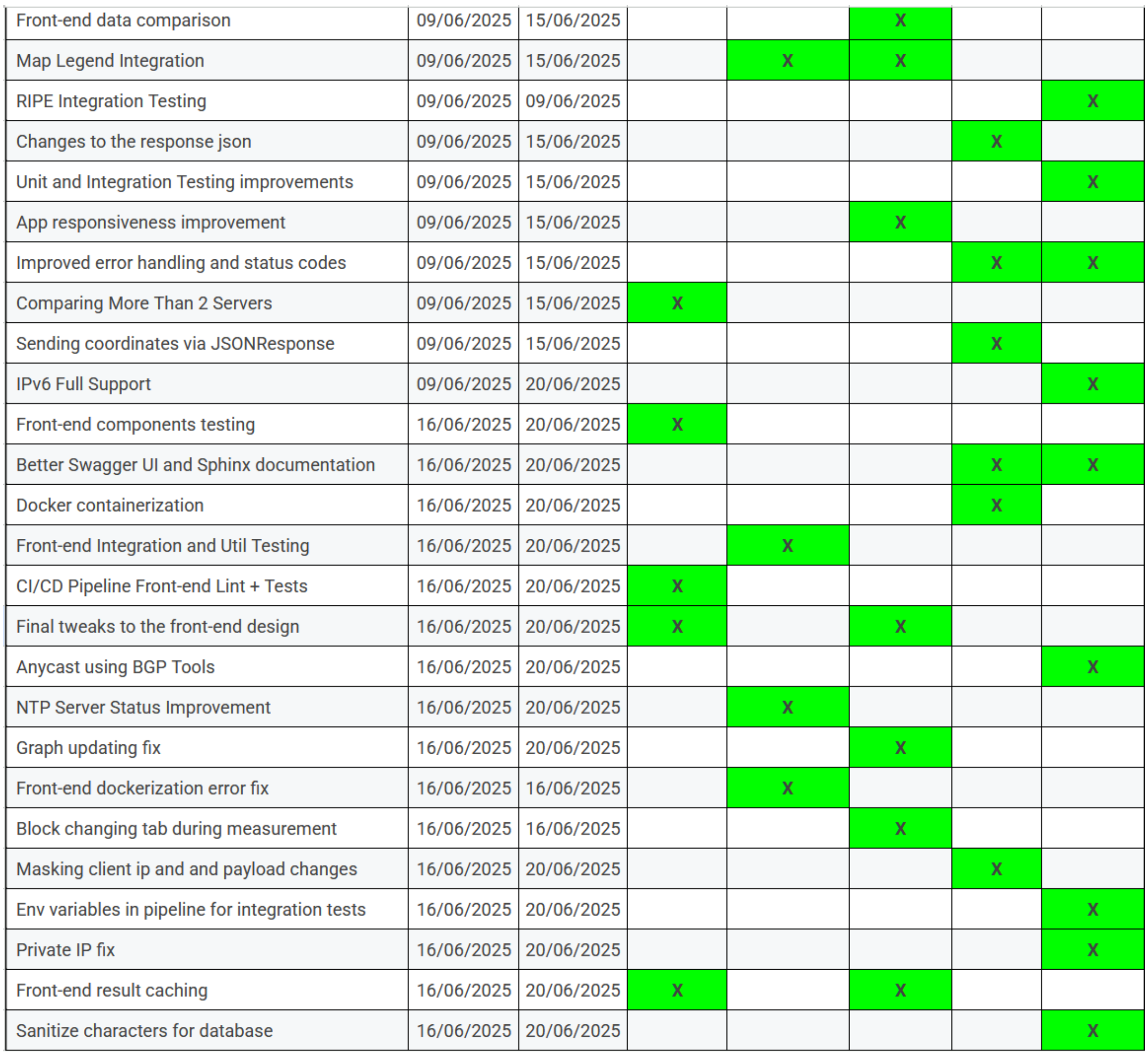


Chart.js Logic Improvement	26/05/2025	08/06/2025			X		
SQLAlchemy ORM for Database	02/06/2025	08/06/2025	X				
Adding configurable options	02/06/2025	08/06/2025					X
Tab System	02/06/2025	08/06/2025	X		X		
Comparing Servers Tab	02/06/2025	08/06/2025	X				
Selecting Probe Types	02/06/2025	08/06/2025					X
Map Logic Changes	02/06/2025	08/06/2025		X			
Historical Data Tab	02/06/2025	08/06/2025			X		
RIPE Data on Front-end	02/06/2025	08/06/2025	X				
Changed jitter logic	02/06/2025	08/06/2025				X	
RIPE API Hook	02/06/2025	08/06/2025		X			
Map visualization improvements	02/06/2025	08/06/2025		X			
RIPE Data from Back-end	02/06/2025	08/06/2025				X	X
Changing request payloads	02/06/2025	08/06/2025		X		X	
RIPE communication improvement	02/06/2025	08/06/2025				X	
Root dispersion and poll to response	02/06/2025	08/06/2025				X	
Fetching Probes from RIPE	02/06/2025	08/06/2025					X
Extra Data On Front-end	02/06/2025	08/06/2025		X			
Sidebar implementation	02/06/2025	15/06/2025			X		
API and Database Integration Testing	09/06/2025	15/06/2026	X				
Front-end redesign	09/06/2025	15/06/2025			X		
Map Logic Improvements	09/06/2025	15/06/2025		X			
MaxMind Databases Integration	09/06/2025	15/06/2025				X	
RIPE Hook Error Handling	09/06/2025	15/06/2025		X			
RIPE Probe Selection Improvement	09/06/2025	15/06/2025					X
Error messages on Front-end improvement	09/06/2025	15/06/2025	X				

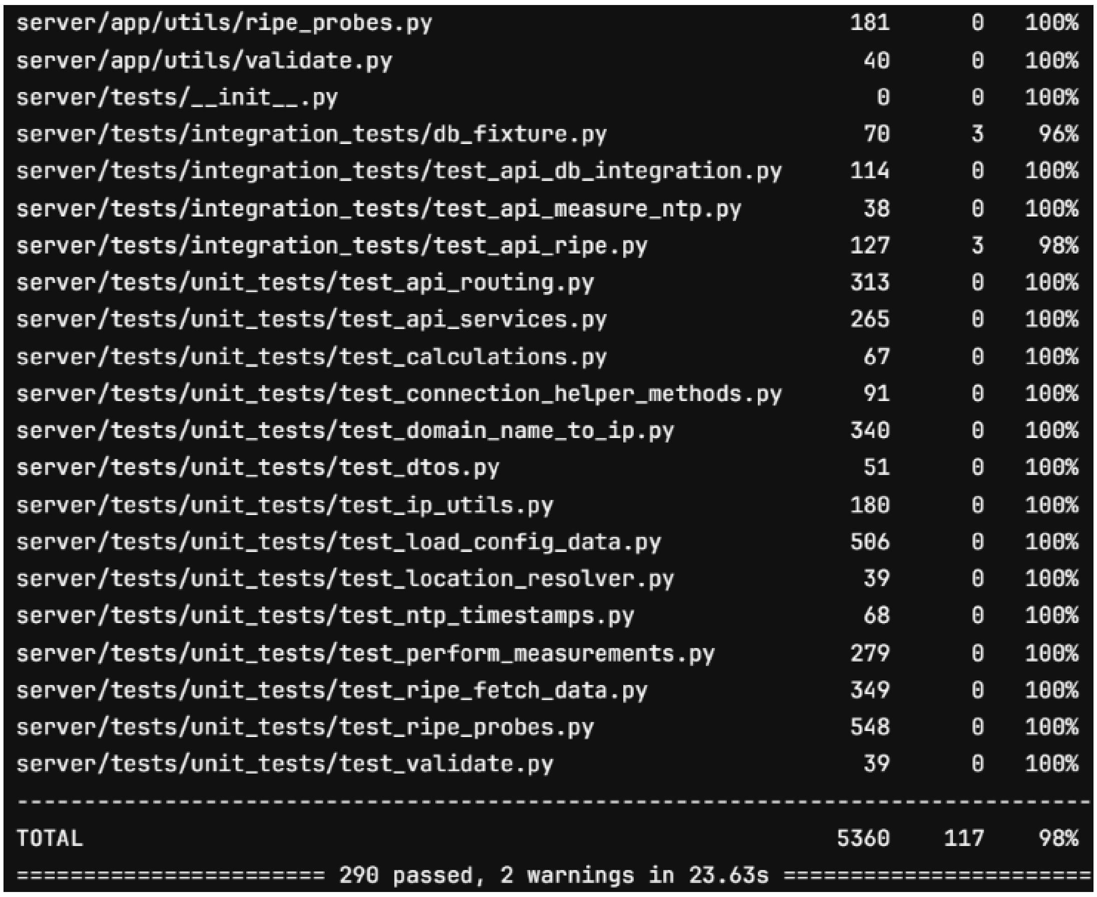
Figure B.2: Work Matrix Tasks 26-51



Front-end data comparison	09/06/2025	15/06/2025			X		
Map Legend Integration	09/06/2025	15/06/2025		X	X		
RIPE Integration Testing	09/06/2025	09/06/2025					X
Changes to the response json	09/06/2025	15/06/2025				X	
Unit and Integration Testing improvements	09/06/2025	15/06/2025					X
App responsiveness improvement	09/06/2025	15/06/2025			X		
Improved error handling and status codes	09/06/2025	15/06/2025				X	X
Comparing More Than 2 Servers	09/06/2025	15/06/2025	X				
Sending coordinates via JSONResponse	09/06/2025	15/06/2025				X	
IPv6 Full Support	09/06/2025	20/06/2025					X
Front-end components testing	16/06/2025	20/06/2025	X				
Better Swagger UI and Sphinx documentation	16/06/2025	20/06/2025				X	X
Docker containerization	16/06/2025	20/06/2025				X	
Front-end Integration and Util Testing	16/06/2025	20/06/2025		X			
CI/CD Pipeline Front-end Lint + Tests	16/06/2025	20/06/2025	X				
Final tweaks to the front-end design	16/06/2025	20/06/2025	X		X		
Anycast using BGP Tools	16/06/2025	20/06/2025					X
NTP Server Status Improvement	16/06/2025	20/06/2025		X			
Graph updating fix	16/06/2025	20/06/2025			X		
Front-end dockerization error fix	16/06/2025	16/06/2025		X			
Block changing tab during measurement	16/06/2025	16/06/2025			X		
Masking client ip and and payload changes	16/06/2025	20/06/2025				X	
Env variables in pipeline for integration tests	16/06/2025	20/06/2025					X
Private IP fix	16/06/2025	20/06/2025					X
Front-end result caching	16/06/2025	20/06/2025	X		X		
Sanitize characters for database	16/06/2025	20/06/2025					X

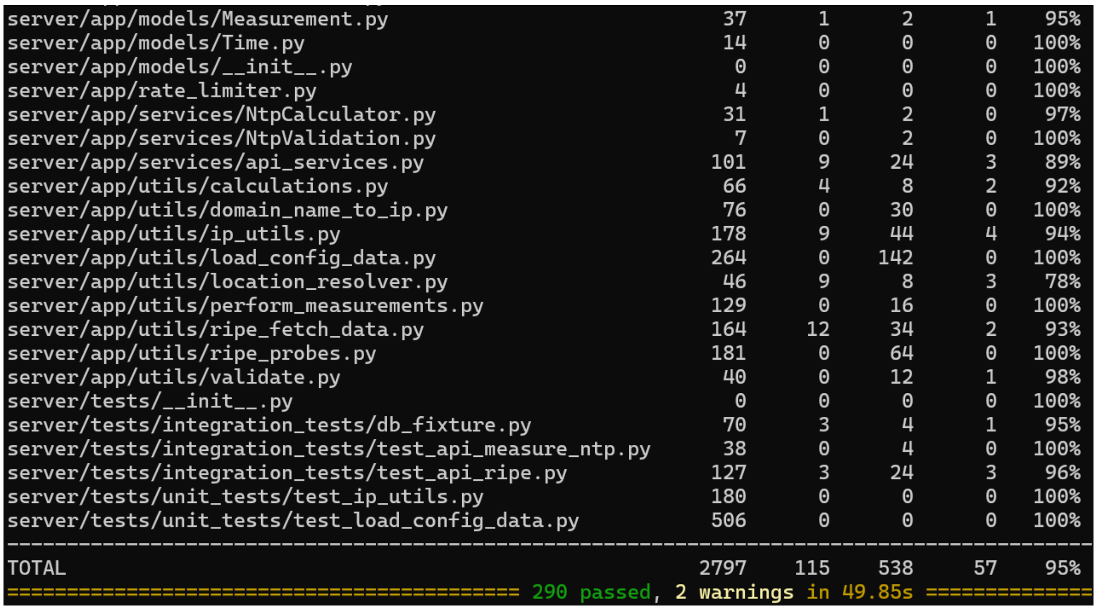
Figure B.3: Work Matrix Tasks 52-77

Appendix C: Code Coverage



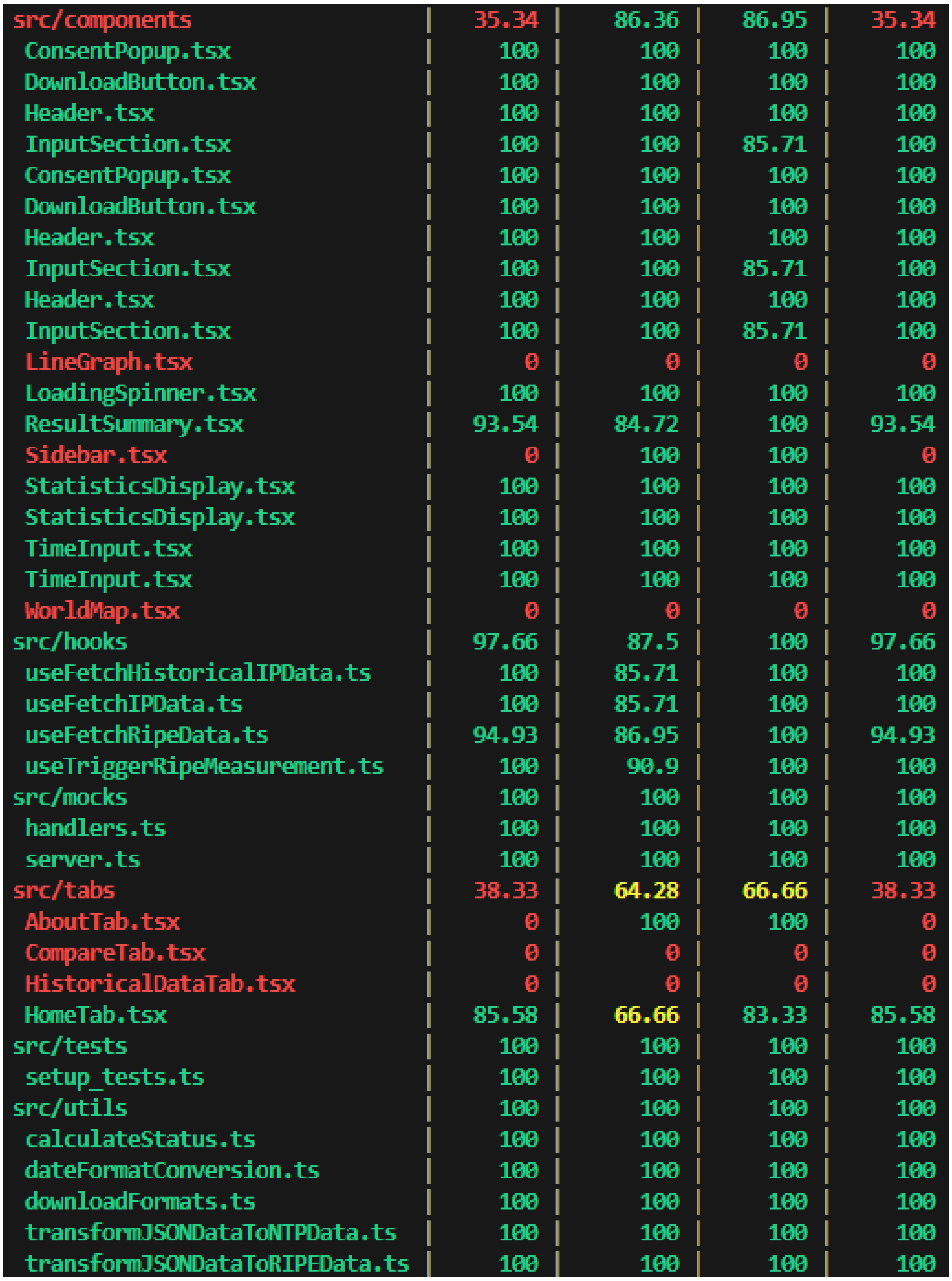
```
server/app/utils/ripe_probes.py                              181      0   100%
server/app/utils/validate.py                                  40      0   100%
server/tests/__init__.py                                       0      0   100%
server/tests/integration_tests/db_fixture.py                  70      3    96%
server/tests/integration_tests/test_api_db_integration.py    114      0   100%
server/tests/integration_tests/test_api_measure_ntp.py        38      0   100%
server/tests/integration_tests/test_api_ripe.py              127      3    98%
server/tests/unit_tests/test_api_routing.py                  313      0   100%
server/tests/unit_tests/test_api_services.py                 265      0   100%
server/tests/unit_tests/test_calculations.py                  67      0   100%
server/tests/unit_tests/test_connection_helper_methods.py     91      0   100%
server/tests/unit_tests/test_domain_name_to_ip.py            340      0   100%
server/tests/unit_tests/test_dtos.py                          51      0   100%
server/tests/unit_tests/test_ip_utils.py                     180      0   100%
server/tests/unit_tests/test_load_config_data.py             506      0   100%
server/tests/unit_tests/test_location_resolver.py             39      0   100%
server/tests/unit_tests/test_ntp_timestamps.py                68      0   100%
server/tests/unit_tests/test_perform_measurements.py         279      0   100%
server/tests/unit_tests/test_ripe_fetch_data.py              349      0   100%
server/tests/unit_tests/test_ripe_probes.py                  548      0   100%
server/tests/unit_tests/test_validate.py                      39      0   100%
---------------------------------------------------------------------------------
TOTAL                                                       5360    117    98%
======================= 290 passed, 2 warnings in 23.63s =======================
```

Figure C.1: Representation of our line coverage in the back-end (98%).



```
server/app/models/Measurement.py                          37      1      2      1    95%
server/app/models/Time.py                                 14      0      0      0   100%
server/app/models/__init__.py                              0      0      0      0   100%
server/app/rate_limiter.py                                 4      0      0      0   100%
server/app/services/NtpCalculator.py                      31      1      2      0    97%
server/app/services/NtpValidation.py                       7      0      2      0   100%
server/app/services/api_services.py                      101      9     24      3    89%
server/app/utils/calculations.py                          66      4      8      2    92%
server/app/utils/domain_name_to_ip.py                     76      0     30      0   100%
server/app/utils/ip_utils.py                             178      9     44      4    94%
server/app/utils/load_config_data.py                     264      0    142      0   100%
server/app/utils/location_resolver.py                     46      9      8      3    78%
server/app/utils/perform_measurements.py                 129      0     16      0   100%
server/app/utils/ripe_fetch_data.py                      164     12     34      2    93%
server/app/utils/ripe_probes.py                          181      0     64      0   100%
server/app/utils/validate.py                              40      0     12      1    98%
server/tests/__init__.py                                   0      0      0      0   100%
server/tests/integration_tests/db_fixture.py              70      3      4      1    95%
server/tests/integration_tests/test_api_measure_ntp.py    38      0      4      0   100%
server/tests/integration_tests/test_api_ripe.py          127      3     24      3    96%
server/tests/unit_tests/test_ip_utils.py                 180      0      0      0   100%
server/tests/unit_tests/test_load_config_data.py         506      0      0      0   100%
-------------------------------------------------------------------------------------------
TOTAL                                                   2797    115    538     57    95%
=========================================== 290 passed, 2 warnings in 49.85s =============
```

Figure C.2: Representation of our branch coverage in the back-end (95%).



```
src/components                  |   35.34 |   86.36 |   86.95 |   35.34
 ConsentPopup.tsx               |     100 |     100 |     100 |     100
 DownloadButton.tsx             |     100 |     100 |     100 |     100
 Header.tsx                     |     100 |     100 |     100 |     100
 InputSection.tsx               |     100 |     100 |   85.71 |     100
 ConsentPopup.tsx               |     100 |     100 |     100 |     100
 DownloadButton.tsx             |     100 |     100 |     100 |     100
 Header.tsx                     |     100 |     100 |     100 |     100
 InputSection.tsx               |     100 |     100 |   85.71 |     100
 Header.tsx                     |     100 |     100 |     100 |     100
 InputSection.tsx               |     100 |     100 |   85.71 |     100
 LineGraph.tsx                  |       0 |       0 |       0 |       0
 LoadingSpinner.tsx             |     100 |     100 |     100 |     100
 ResultSummary.tsx              |   93.54 |   84.72 |     100 |   93.54
 Sidebar.tsx                    |       0 |     100 |     100 |       0
 StatisticsDisplay.tsx          |     100 |     100 |     100 |     100
 StatisticsDisplay.tsx          |     100 |     100 |     100 |     100
 TimeInput.tsx                  |     100 |     100 |     100 |     100
 TimeInput.tsx                  |     100 |     100 |     100 |     100
 WorldMap.tsx                   |       0 |       0 |       0 |       0
src/hooks                       |   97.66 |    87.5 |     100 |   97.66
 useFetchHistoricalIPData.ts    |     100 |   85.71 |     100 |     100
 useFetchIPData.ts              |     100 |   85.71 |     100 |     100
 useFetchRipeData.ts            |   94.93 |   86.95 |     100 |   94.93
 useTriggerRipeMeasurement.ts   |     100 |    90.9 |     100 |     100
src/mocks                       |     100 |     100 |     100 |     100
 handlers.ts                    |     100 |     100 |     100 |     100
 server.ts                      |     100 |     100 |     100 |     100
src/tabs                        |   38.33 |   64.28 |   66.66 |   38.33
 AboutTab.tsx                   |       0 |     100 |     100 |       0
 CompareTab.tsx                 |       0 |       0 |       0 |       0
 HistoricalDataTab.tsx          |       0 |       0 |       0 |       0
 HomeTab.tsx                    |   85.58 |   66.66 |   83.33 |   85.58
src/tests                       |     100 |     100 |     100 |     100
 setup_tests.ts                 |     100 |     100 |     100 |     100
src/utils                       |     100 |     100 |     100 |     100
 calculateStatus.ts             |     100 |     100 |     100 |     100
 dateFormatConversion.ts        |     100 |     100 |     100 |     100
 downloadFormats.ts             |     100 |     100 |     100 |     100
 transformJSONDataToNTPData.ts  |     100 |     100 |     100 |     100
 transformJSONDataToRIPEData.ts |     100 |     100 |     100 |     100
```

Figure C.3: Representation of the statement, branch, function and line coverages in the front-end.