OST
Eastern Switzerland
University of Applied Sciences

Master's Thesis

Efficient Public Transit Routing with RAPTOR:
A Production-Ready Solution Leveraging GTFS Data

Authors:
Merlin Unterfinger
Michael Brunner
Lukas Connolly

Supervised by:
Thomas Letsch

MAS Software Engineering 2022-2024
Eastern Switzerland University of Applied Sciences
September 27, 2024

Abstract

Timetable queries in public transportation present a complex problem. Traditional graph-based methods encounter their limitations, especially when it comes to correctly representing transfers between different routes. The temporal complexity of transfers makes a trivial representation nearly impossible. A promising alternative to these traditional approaches is the Round-based Public Transit Optimized Router (RAPTOR) algorithm, developed by a team of Microsoft engineers in collaboration with the Karlsruher Institute for Technology (KIT). This round-based algorithm does not require conventional graph construction and operates directly on timetable data. Starting from the departure location, the departures of all routes passing through are scanned. In subsequent rounds, all routes from stops newly reached in the previous round are considered, until the destination is reached.

The goal of our work was to develop our own implementation of this algorithm and to use it with timetables based on the General Transit Feed Specification (GTFS). The results of our work can be divided into two main areas:

1. **Productive Solution:** We extended the RAPTOR algorithm to make it suitable for everyday, productive systems. The algorithm was supplemented with features such as prioritizing routes by departure or arrival times, defining custom query criteria (e.g., mode of transportation, number of transfers, maximum walking distance, minimum transfer time, wheelchair accessibility, or bicycle accommodations), and supporting multi-day connections. This extended version of the algorithm was integrated into a Spring application with REST API, allowing it to be used by a web application.
2. **Performance Optimization:** Another aspect of our work was optimizing performance. We compared different implementations of the RAPTOR algorithm: the basic version, as proposed in the original paper, once in Java and once in C++. Additionally, we investigated other implementations, such as the multi-day RAPTOR we developed, which enables scanning timetables over multiple service days, as well as the range RAPTOR, which was also described in the original paper.

Both aspects of our work were successfully implemented and contribute to advancing the RAPTOR algorithm both in theory and practice.

Keywords RAPTOR algorithm, GTFS, timetable routing, isoline routing, performance optimization, public transportation

Acknowledgements

We would like to express our sincere gratitude to our supervisor, Thomas Letsch, for his invaluable guidance, insightful feedback, and continuous support throughout this project. His expertise and encouragement were instrumental in shaping the direction of our work.

Additionally, we extend our thanks to the Eastern Switzerland University of Applied Sciences and its dedicated lecturers for providing us with the resources and learning environment that made this project possible. We also want to acknowledge the open-source community, whose contributions to the development of tools and libraries played a significant role in the success of our project.

Finally, we are deeply grateful to our friends and families for their unwavering support and understanding during the long hours and late nights spent on this thesis. Without their patience and encouragement, this work would not have been possible.

Contents

List of Figures

List of Tables

Glossary

In the context of this thesis, certain terms have different meanings depending on whether they refer to public transit in general, the RAPTOR algorithm, or GTFS. As a result, some terms are defined multiple times in the glossary to reflect their specific usage in each area. While we aim for clarity in the text, some terms in the code may carry different meanings depending on the layer or package they are associated with.

Public Transit

Term	**Description**
Agency	An organization providing transit services.
Connection	An option for travel that includes details like fare and possible routes and transfers.
Isoline	Reachable stops within a time frame, forming accessibility boundaries from a source stop, exploring routes without a predefined destination.
Journey	A connection is realized as a journey when the passenger selects it and begins traveling.
Leg	A part of a connection in one mode of transportation or vehicle.
Platform	A designated area within a station where passengers wait to board or alight from transit vehicles.
Schedule	The planned timetable for transit services, detailing stops, routes, and times.
Service	Transit offerings by agencies, consisting of trips operating on specific routes.
Station	A station is a large stop facility, typically serving rail. It can also consist of multiple smaller stop facilities (e.g., platforms).
Stop	A location where passengers board or alight from transit vehicles.
Stop facility	Physical infrastructure where passengers wait and board transit vehicles.
Transfer	A switch between different routes or a footpath between two stops.

RAPTOR

Term	**Description**
Footpath	*Between stop transfer*: A walking connection between two different stops that can be used during transfers.
Marked stop	A stop where the arrival time improved in the last round, requiring evaluation of routes or footpaths connected to this stop in the current round.
Pareto-optimal	A set of solutions where no journey is strictly worse in all criteria compared to any other journey.
Relaxing	The process of applying footpaths to other stops connected to improved, marked stops in the current round.

Continued on next page

Term	Description
Round	An iteration in the RAPTOR algorithm, evaluating routes and footpaths at each stop reached within the current number of allowed transfers.
Route scanning	The process of evaluating possible transit routes that pass through marked stops to find optimal paths.
Transfer	*Same stop transfer*: Switching between routes at the same stop during a journey.

GTFS

Term	Description
Agency	Transit agencies with service represented in this dataset.
Calendar	Service dates specified using a weekly schedule with start and end dates.
Calendar date	Exception for a service defined in the calendar.
Feed info	Dataset metadata, including publisher, version, and expiration information.
Frequency	Headway (time between trips) for headway-based service or a compressed representation of fixed-schedule service.
Journey	Overall travel from origin to destination, including all legs and transfers in-between.
Leg	Travel in which a passenger boards and alights between a pair of subsequent locations along a trip.
Location	Zone for passenger pickup or drop-off requests by on-demand services, represented as GeoJSON polygon.
Route	A transit route is a group of trips that are displayed to passengers as a single service.
Stop area	Criteria to assign stops to an area.
Stop time	Time that a vehicle arrives at and departs from stops for each trip.
Stop	Location where vehicles pick up or drop off passengers. Also defines station and station entrances.
Transfer	Rule for making connections at transfer points between routes.
Trip	A trip is a sequence of two or more stops that occur during a specific time period.

1 Introduction

Public transportation systems are integral to urban mobility, facilitating the movement of millions of passengers daily. The efficiency of these systems relies heavily on robust routing algorithms that optimize travel routes, minimize travel times, and enhance the overall user experience. This thesis investigates the challenges associated with public transit routing and explores innovative approaches to address these challenges, leading to the release of the production-ready solution named *Naviqore* [2].

1.1 Motivation

This thesis is driven by several key challenges in public transit routing. In recent years, many public transit agencies have begun publishing their schedules using the General Transit Feed Specification (GTFS) format as open data [3]. Maintaining accurate public transit schedules is a resource-intensive task. The volume and complexity of GTFS data present significant challenges for developing efficient data structures and processing methodologies.

Public transit routing is inherently complex due to the time-dependent nature of transit networks. Transit schedules vary based on the time of day, day of the week, and service exceptions for specific dates, adding a layer of complexity to the routing process. Developing efficient algorithms that account for this time-dependency remains a challenging task in transportation research. Traditional graph-based routing algorithms often face performance issues when dealing with large-scale transit networks, leading to very large graph structures and computational cost in preparing such graphs. The RAPTOR (**R**ound-b**a**sed **P**ublic **T**ransit **O**ptimized **R**outer) algorithm, recognized for its speed and accuracy, offers a promising solution to these performance limitations [4] as it can operate directly on the schedule information and requires little pre-processing.

Developing a public transit routing system was found to be an optimal challenge as it integrates various aspects of the Master of Advanced Studies in Software Engineering curriculum, including software architecture, algorithms and data structures, automation, distributed systems communication, testing, and containerized deployment. This project adopts an agile, iterative development methodology, providing an opportunity to address complex challenges while gaining practical experience in software engineering.

1.2 Research Questions

The primary research question addressed by this thesis is how the RAPTOR algorithm can be implemented and validated within the context of the Swiss public transit network. This question is particularly relevant given the complexity and scale of the network, which poses significant challenges for routing algorithms. A related research question focuses on the optimization of the RAPTOR algorithm in Java and C++ and comparing the performance of both programming environments for such a task.

These research questions are framed within the context of the current state of public transit networks. The complexity of these networks, which involve multiple routes, stops, footpaths, and transfer points, necessitates the development of routing algorithms that account for factors such as transfer times, service frequencies, and real-time constraints. The research seeks to address these challenges.

1.3 Approach and Expected Outcomes

To address the identified challenges, a comprehensive approach was defined, beginning with the utilization of GTFS data [3] as the foundation for the Naviqore transit routing system. A memory-efficient library for reading and processing GTFS schedules will be developed to enable scalable management of large datasets, ensuring the system can handle large-scale transit networks effectively.

One of the core aspects of this thesis is the implementation of the RAPTOR algorithm [4], which is designed to efficiently compute transit routes by considering time dependencies such as schedules and transfer windows. The RAPTOR algorithm employs a round-based approach, where transit routes are organized into discrete rounds, each representing the number of trips needed. This structure simplifies the traversal of the network by enabling the algorithm to explore transit options in rounds and return Pareto-optimal solutions, minimizing both arrival time and the number of transfers.

The system will be complemented with an integration layer that manages both the routing algorithm and the schedule, while handling all necessary conversions between the two. This design maintains the independence of the schedule and the routing algorithm, enabling a modular architecture. The modularity allows for system components to be modified or replaced without affecting external dependencies, ensuring the system's adaptability and long-term maintainability.

To facilitate interaction with the system, a REST API will be developed, allowing external applications to access the routing and scheduling services. This API will provide the foundation for seamless integration between the routing algorithm and external systems. As a proof of concept, a simple web application will be created to visualize search results and demonstrate the capabilities of the backend service. While the web application will serve as a proof of concept, the primary focus of this thesis remains on developing the backend service.

1.4 Scope and Delimitation

This thesis is part of the Master of Advanced Studies in Software Engineering at the Eastern Switzerland University of Applied Sciences (OST). It focuses on applying the knowledge gained from the program to further advance methodologies in public transit routing, with no external company interests influencing the scope or outcomes of this research.

2 Project Management

The development process followed an iterative Scrum-like approach [5], with 4-week sprints supported by Jira for task management. Weekly meetings were held to synchronize progress, and monthly supervisor syncs were conducted to review completed work and plan for the next sprint. Each sprint concluded with demos and reviews, ensuring continuous feedback and iteration. The team operated without daily stand-ups, relying on weekly meetings for coordination and issue resolution.

The project was divided into three distinct phases: the *walking skeleton (v0.2.0)*, *main implementation (v1.0.0)*, and the *feature freeze* for final documentation and presentation. Each phase, divided into smaller iterations, incrementally contributed to the overall project goals, progressing through clear milestones to ensure steady and structured development.

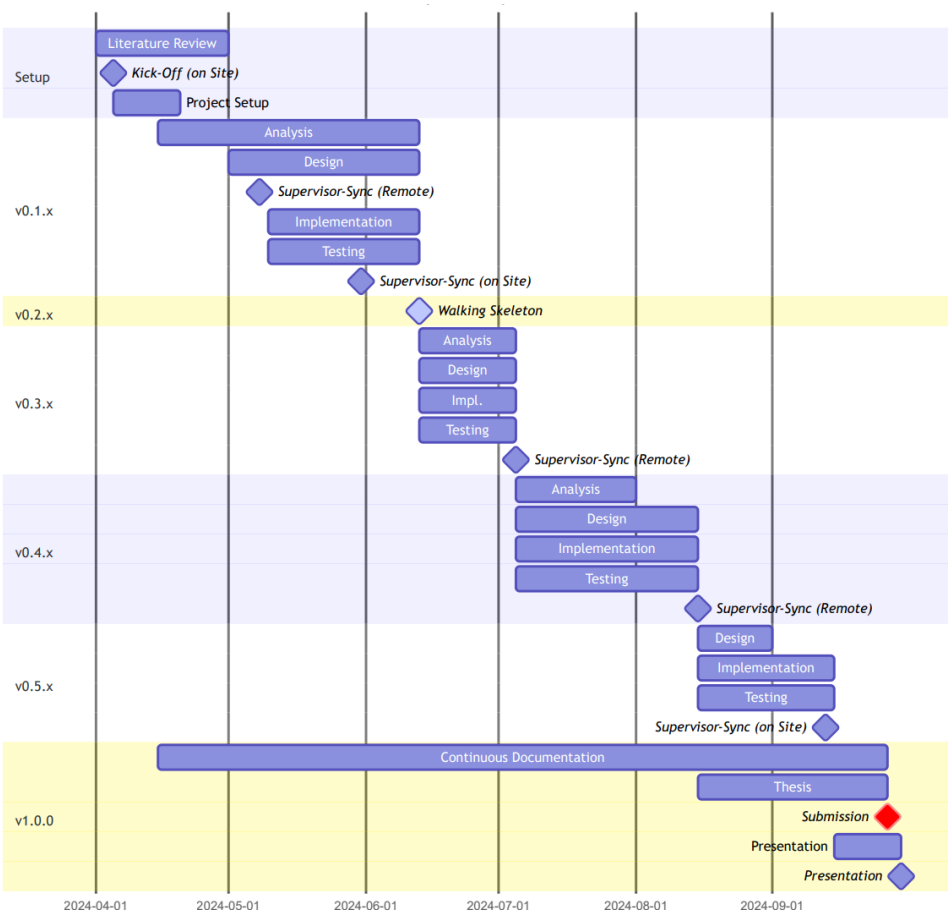

Figure 2.1: Outline of project iterations and development phases.

2.1 Best Practices and Guidelines

Guidelines and best practices were established early in the project to ensure consistency and quality throughout the project. These included:

- **Code Reviews:** All pull requests underwent thorough code reviews to maintain code quality and ensure consistency across the project.
- **Automated Testing:** Both unit and integration tests were integrated into the CI pipeline via GitHub Actions. This ensured reliable code with adequate test coverage, especially for edge cases.
- **Continuous Documentation:** Documentation was updated continuously using WriterSide and published daily on GitHub Pages. This practice ensured clarity, traceability, and accessibility, with all content written in English.
- **Task Estimation:** Collaborative task estimation was done by categorizing tasks in T-shirt sizes: Small (1 hour), medium (4 hours), large (8 hours), and extra large (2 days). This method ensured effective workload distribution and realistic sprint goals.
- **Sprint Planning and Review:** Regular sprint planning and review meetings were conducted to monitor progress, adjust priorities, and review the accuracy of previous estimations. This allowed the team to refine the estimation process and improve planning over time.

More detailed code, Git and release guidelines are added to Appendix A.1, A.2, and A.3, respectively.

2.2 Time Tracking

Time tracking for this project was meticulously managed using Jira, with all team members logging their hours against individual issues, which were later aggregated by epics. Each team member was required to log a minimum of 375 hours over the course of the project, spanning from March to the final submission in late September 2024, equating to approximately 16 hours per week. This system ensured accountability and provided a clear overview of time allocation across different phases of the project.

Appendix B presents detailed visualizations of the logged hours, including the time distribution per developer and the hours spent on each epic, offering insights into workload balance and project progress.

3 Analysis

3.1 Use Cases

Users engage with the Naviqore public transit service to enhance their transit experience. They can explore transit schedules, locate nearby stops, customize preferences for accessibility and travel modes, and plan optimal routes between different locations to meet their personal mobility needs.

Analysts request large volumes of connections or isolines to conduct analyses related to public transit and mobility behavior, such as determining how far one can travel within a given time budget, evaluating the accessibility of a region, and assessing alternative routes available at the time of decision-making.

Transit Agencies publish GTFS schedules, which are then consumed by the service to provide up-to-date public transit information and connections.

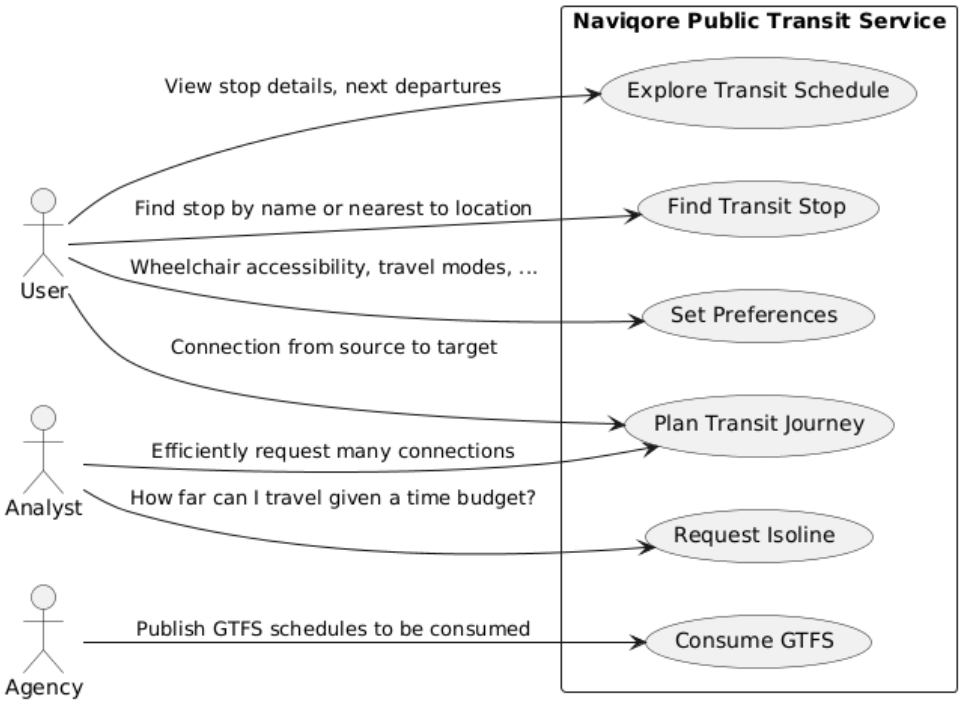

Figure 3.1: Use cases for this project.

3.2 Requirements

Requirements were defined early in the process and were categorized into three groups based on their level of importance: must-have, should-have, and nice-to-have features.

- **Must-have** requirements represented the core functionalities that were critical for the system to operate effectively. These features were essential and had to be implemented to ensure the service met its primary objectives.
- **Should-have** requirements were highly desirable features that, although not mandatory, significantly improved the user experience and operational efficiency. While not required for the initial release, implementing these features was strongly recommended.
- **Nice-to-have** requirements were additional features that would have enhanced the service. They were not critical for the system's basic functionality, but they offered added value and could have been considered for future iterations to provide a more refined user experience.

For a detailed breakdown of the specific requirements, including descriptions of individual features and their priority levels, please refer to Appendix C.

3.3 Domains

The domain analysis identified four key domains, each representing distinct perspectives and functional areas critical to the system's design. These domains include the user perspective, which focuses on the interactions and requirements of end-users engaging with the transit service; the agency perspective, which encompasses the needs and operational considerations of the transit agencies that manage and provide transit data; and two external domains defined by widely accepted standards: the RAPTOR algorithm for efficient transit routing and the General Transit Feed Specification (GTFS) for handling public transit schedules. By analyzing each of these domains, the system can be designed to effectively meet the diverse needs of its stakeholders while maintaining compatibility with industry standards.

3.3.1 User Perspective

The *user* interacts with the *schedule* to request *connections*, which are options for travel that include details like fare and possible routes. A connection is realized as a *journey* when the user selects it and begins traveling. A connection consists of *legs*, which are defined by departure and arrival times, transport mode, and distance. Each leg begins and ends at a specific *stop facility*.

The domain model for this perspective is shown in Figure 3.2.

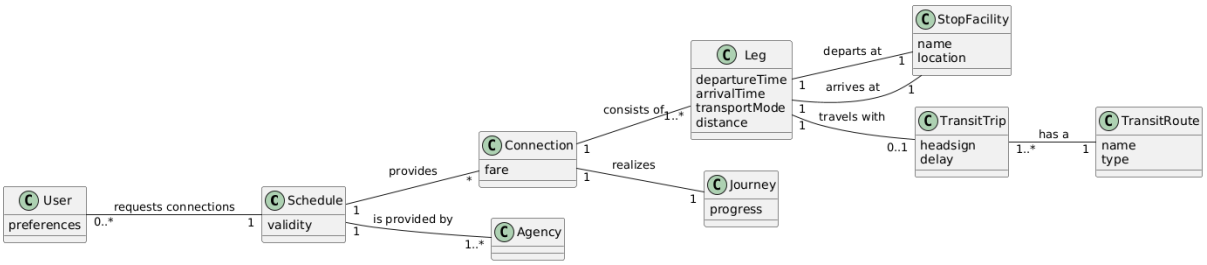

Figure 3.2: Domain model for user perspective.

3.3.2 Agency Perspective

The *agency* offers public transit *services*, made up of *trips* that run along defined *routes*. Each trip follows the series of stop facilities on its route to allow passengers to board and alight. At stops, passengers can *transfer* between trips on different routes or walk to nearby stop facilities for further trips. A schedule may include services from multiple transit agencies.

The domain model for this perspective is very similar to the domain model defined by the GTFS standard, as shown in Figure 3.3.

3.3.3 GTFS

GTFS, or General Transit Feed Specification, is a standardized data format used for sharing public transportation schedules and related geographic information. It was originally developed by Google in partnership with the public transportation agency TriMet, to enable public transit agencies to publish their data in a format that is easily consumed by applications like Google Maps, making public transit data more accessible and useful for riders.

Key Components

GTFS consists of a series of text files stored in a ZIP archive, each containing specific information about different aspects of a public transit network. These files define various entities, such as *stops.txt* for station locations, *routes.txt* for transit routes, and *trips.txt* for individual trips along those routes. Figure 3.3 presents the domain model for GTFS files provided by the Open Data Platform Mobility Switzerland [1], illustrating how the fields in each file interact to form a cohesive public transport network. While this example does not cover all GTFS files or fields, it offers a clear overview of the most essential components. For a complete list of GTFS files and their fields, refer to the official GTFS reference [3].

Usage

GTFS data is used in various ways to enhance public transit systems and improve user experience. Public transit apps like Google Maps and Apple Maps use GTFS feeds for route planning, schedule information, and real-time transit updates. Transit agencies utilize GTFS for scheduling, route optimization, and operational analysis. Data analysts and urban planners rely on GTFS for performance analysis, accessibility studies, and transit network modeling. Information systems at stations and stops leverage GTFS data to display real-time arrival and departure information, route details, and service updates.

File Format - Example: *routes.txt*

```
route_id,agency_id,route_short_name,route_long_name,route_desc,route_type
"91-10-A-j22-1","37","10","","T","900"
"91-10-B-j22-1","78","S10","","S","109"
"91-10-C-j22-1","11","S10","","S","109"
"91-10-E-j22-1","65","S10","","S","109"
"91-10-F-j22-1","11","RE10","","RE","106"
"91-10-G-j22-1","11","SN10","","SN","109"
"91-10-j22-1","3849","10","","T","900"
"91-10-Y-j22-1","82","IR","","IR","103"
```

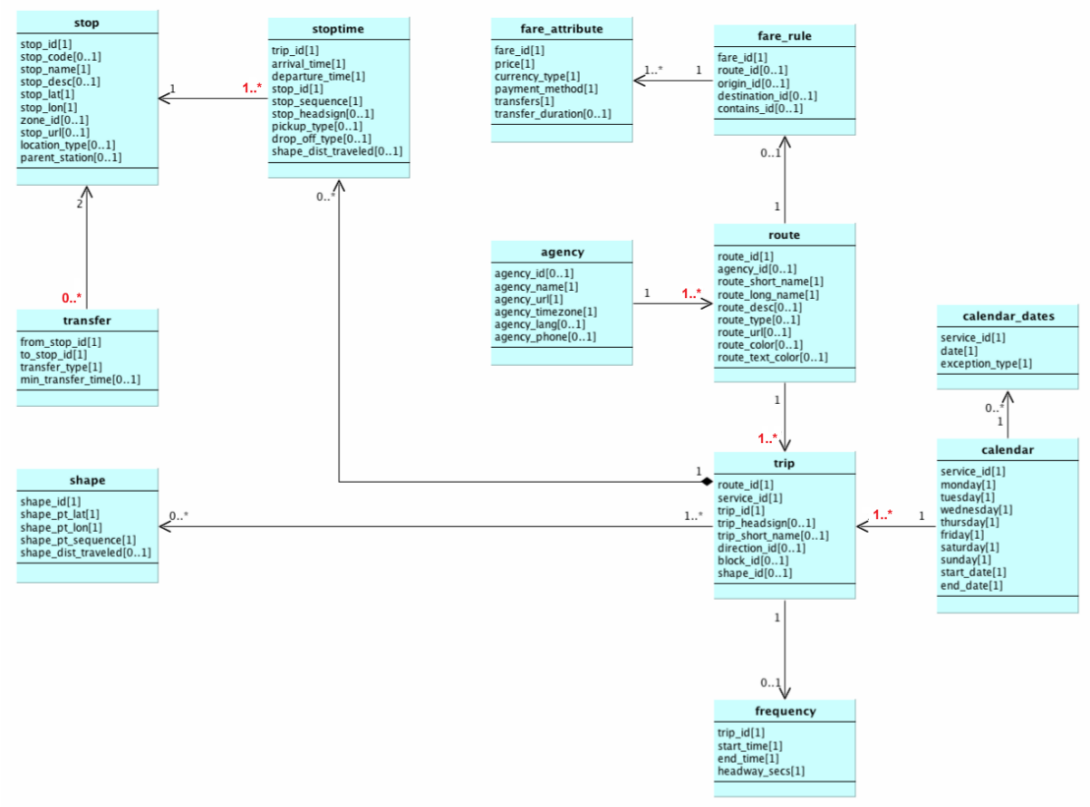

Figure 3.3: Domain model for GTFS files. Source: Open Data Platform Mobility Switzerland [1], with corrections highlighted in *red*.

3.3.4 RAPTOR

The RAPTOR (**R**ound-b**A**sed **P**ublic **T**ransit **O**ptimized **R**outer) algorithm is a powerful tool for computing optimal routes in public transportation systems, particularly for large and complex networks. Its primary goal is to efficiently determine the best route options based on factors like arrival time and the number of transfers. RAPTOR is especially effective for real-time journey planning due to its speed and adaptability [6].

The data used by the RAPTOR algorithm is significantly simplified compared to the full GTFS schedule. The algorithm focuses on just five key components. As illustrated in Figure 3.4, a public transit network consists of a set of routes and stops. Each route is made up of trips, all of which follow the same sequence of stops. A trip is essentially a container that includes a set of stop times, which provide the arrival and departure times for each stop along the trip. Additionally, stops contain information about potential transfers to other stops, facilitating passenger connections within the network.

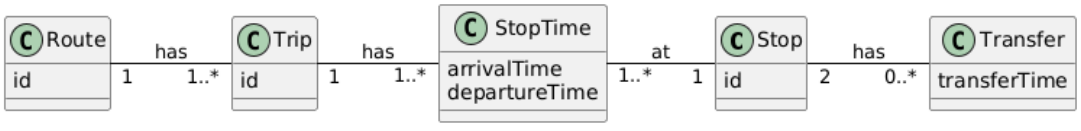

Figure 3.4: Domain model for RAPTOR.

How RAPTOR Works

- **Initialization**: The algorithm starts at the origin station and time specified by the user. Initially, it only considers the starting station, marking it for the first round of exploration.

- **Initial Footpath Relaxation**: Before starting the main loop, the algorithm relaxes footpaths from the starting station to other nearby stations. This step allows the algorithm to consider other stations that are within walking distance of the starting station.
- **Round-based Search**: The search process is broken into several rounds. In each round, the algorithm evaluates all transport lines connected to the current list of stations. It computes the earliest possible arrival times at the stations along each line and marks stops for the next round if an improvement is found.
- **Footpath Relaxation**: After each round, the algorithm relaxes footpaths from the newly marked stops to other nearby stations. This allows adding further stations to the list of stops to be considered in the next round.
- **Termination Condition**: The algorithm stops when it reaches the destination station and none of the marked stops have the ability to improve the arrival time of the destination station (all possible routes have been explored).
- **Tracing the Optimal Route**: Once the algorithm concludes, the optimal route is reconstructed by tracing the updated arrival times back from the destination to the start station.

RAPTOR is highly efficient because it operates directly on schedule information without the need to build extensive graphs that incorporate the complexity of transfers. This characteristic makes RAPTOR particularly suitable for networks of any size, as it avoids the need for time-consuming preprocessing. This efficiency allows us to enable features like accessibility and bike information on our routes.

Pareto-optimal Connections

One of the core features of the native RAPTOR algorithm is its ability to compute Pareto-optimal connections. A connection is Pareto-optimal if no objective (e.g., arrival time or the number of transfers) can be improved without worsening the other. RAPTOR achieves this by minimizing the earliest arrival time while also considering the number of transfers. It processes the network in rounds, ensuring that it returns the fastest routes with the fewest transfers. As a result, RAPTOR guarantees that no further improvement can be made on any route without negatively impacting another factor, such as increasing transfers or delaying arrival time.

Range Queries

Range RAPTOR (rRAPTOR) is an extension of the classic RAPTOR algorithm, specifically designed to handle multiple departure times efficiently. Instead of calculating the optimal route for just a single departure time, it determines the best route over a range of potential departure times. This makes it particularly useful for trip planning within a time window, where the goal is to find the earliest possible arrival time given any departure between specified hours (e.g., "find the earliest arrival if I leave anytime between 8:00 and 10:00 AM"), ultimately reducing travel time. The algorithm efficiently handles these multiple scenarios by leveraging the underlying RAPTOR structure, applying it over the entire range while ensuring that the returned connections are still Pareto-optimal.

Multi-criteria Queries

Multi-criteria RAPTOR (mcRAPTOR) extends the original RAPTOR algorithm by considering additional optimization criteria, such as fare, comfort, or walking distance, alongside the traditional criteria of arrival time and number of transfers. This multi-criteria extension computes Pareto-optimal solutions that account for these extra dimensions. The challenge with mcRAPTOR lies in balancing these additional criteria without sacrificing computational efficiency. However, it still adheres to RAPTOR's core principle of fast, round-based processing, providing more tailored route suggestions based on multiple user preferences.

4 Design

4.1 Architecture

We have designed our architecture based on the principles of Clean Architecture [7] and Onion Architecture [8]. The application is divided into the following layers: Core, use case, application, and infrastructure/UI. Dependencies only flow inward via dependency injection.

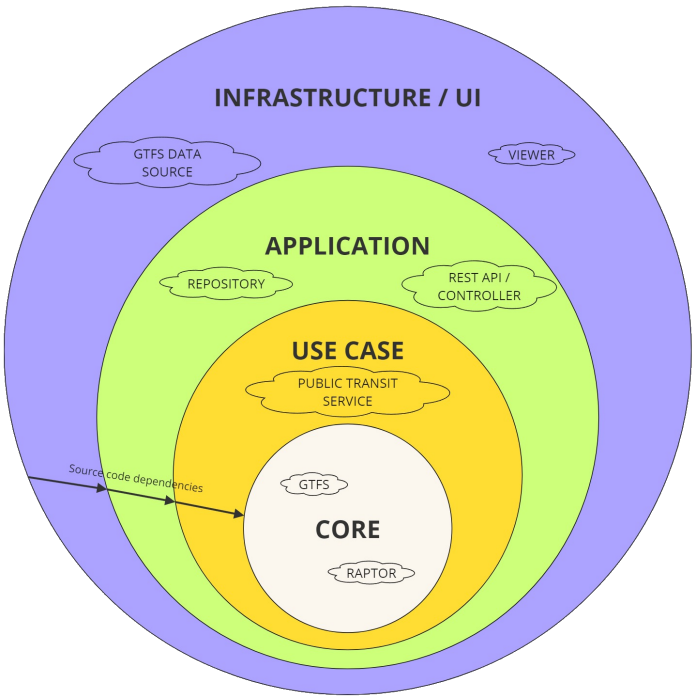

Figure 4.1: Architectural overview.

4.1.1 Dependency Flow

Since we have adopted an Onion Architecture, all dependencies point inward. The outer layers interact with abstractions defined in the inner layers or implement interfaces provided by the inner layers. Each layer exposes a set of public interfaces, serving as its outward-facing API. This architecture promotes loose coupling because the implementation of any layer can be changed with minimal modifications to the outer layers.

The service layer, also referred to as *Use Case* layer in Figure 4.1, provides the `PublicTransitService` interface along with relevant parameter and return types, such as `ConnectionQueryConfig` and `Connection`. The actual implementation of this service, `GtfsRaptorService`, is abstracted away from the application

layer, making it easily replaceable. In the application layer, `PublicTransitSpringService` also implements the `PublicTransitService` interface and uses the delegate pattern to forward requests from the REST controllers to the service. This encapsulates all the Spring [9] components, configurations, and dependencies within the outermost app layer, keeping them isolated from the service and core logic.

Similarly, the `GtfsScheduleRepository` interface in the service layer is implemented in the application layer by classes such as `GtfsScheduleFile` and `GtfsScheduleUrl`. For the service itself, it is not relevant how the GTFS schedule is provided, as it only expects a concrete implementation of the repository interface from the instantiator, which is responsible for providing the GTFS schedule.

4.1.2 Abstractions and Type Mapping

This concept is applied consistently across the project, with a few deliberate exceptions. One notable drawback of this architecture is that each layer defines its own representation of the same real-world entities, leading to frequent type mapping as requests and responses traverse the layers. A good example is the entity of a transit stop, which is abstracted differently in various layers. In the core layer, we maintain two distinct abstractions: one for GTFS stops and another for stops used by the RAPTOR algorithm. This separation exists because the concept of a transit stop differs based on the context of the layer. For instance, in the RAPTOR layer, technical details such as possible transfers are prioritized, while in the service layer, information from the passenger's perspective, such as the stop's full name and available routes, takes precedence. In these cases, maintaining separate abstractions for transit stops is justified due to the distinct roles they play across layers.

However, for more stable concepts like locations and coordinates, a unified representation suffices across all layers. Instead of duplicating these entities within each layer, we extracted them into a cross-cutting `utils.spatial` module. These data types are shared across layers without any need for mapping, improving both performance and maintainability.

4.2 C4 Model

The C4 model is a simple, hierarchical way of visualizing software architecture [10, 11]. It consists of four layers:

- **System Context**: Shows how the system interacts with its environment.
- **Containers**: Defines high-level containers (applications, services, databases) that make up the system.
- **Components**: Focuses on the internal structure of each container, showing the components and how they communicate.
- **Code**: Zooms in on the details of individual components (often omitted in higher-level diagrams). In general, it is not recommended to create this layer, since it can be generated in modern IDEs.

4.2.1 System Context

Figure 4.2 visualizes the Naviqore system and its main actors. These actors include the *User*, *Researcher*, and *Public Transit Agency*, which interact with the system, and the GTFS schedule which is provided by the agency.

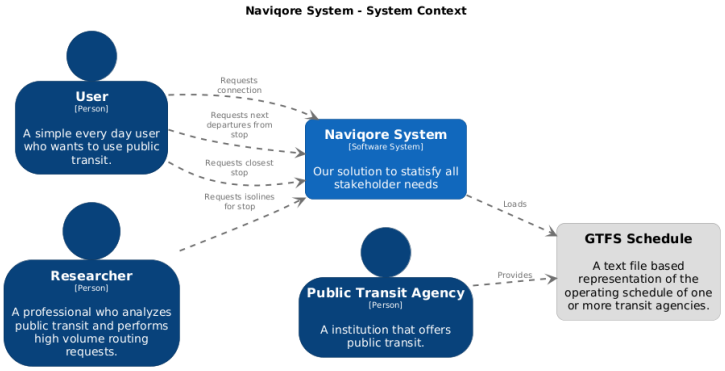

Figure 4.2: C4 context diagram.

4.2.2 Container

Figure 4.3 breaks down the Naviqore system into its primary containers. These include the *Web Application UI* and the *Service*, which depend on the GTFS schedule to fulfill requests.

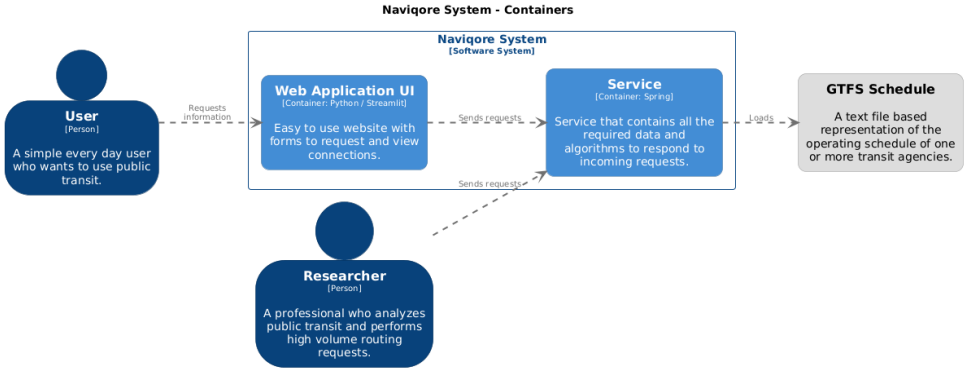

Figure 4.3: C4 container diagram.

4.2.3 Component

Figure 4.4 drills into the *Service* container, showing its key internal components such as the *RAPTOR*, *GTFS*, *REST API*, and *Public Transit Service*.

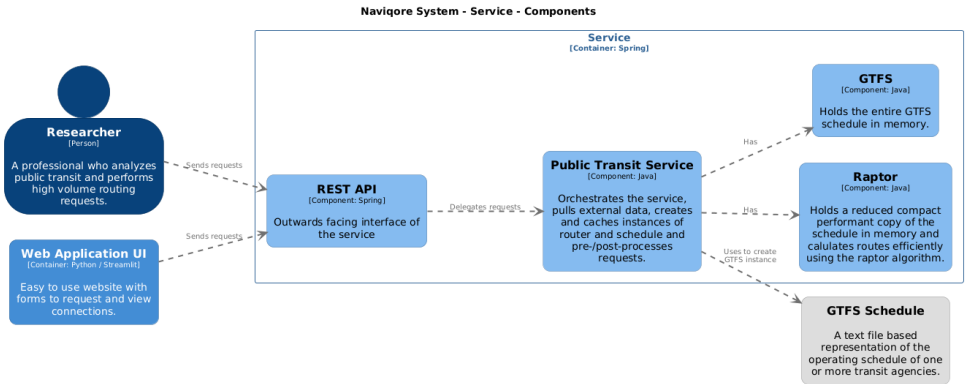

Figure 4.4: C4 component diagram.

4.2.4 Code

As mentioned earlier, the code layer is not included in this document since it can be automatically generated by modern IDEs and is prone to frequent changes. Instead, we focus on the higher-level architectural views of the system. Key code structures and implementation details will be highlighted in Chapter 5, which focuses on the implementation.

4.3 Sequence Diagrams

The following sequence diagrams illustrate the key interactions and processes within the Naviqore system.

4.3.1 Application Startup

Sequence Diagram 4.5 illustrates how the backend Spring application (REST API) initializes during the startup of the Naviqore system. Upon startup, the *Public Transit Service* loads the *GTFS Static* schedule either from a file or a URL. While running, the service periodically updates the *GTFS Static* schedule from the given source.

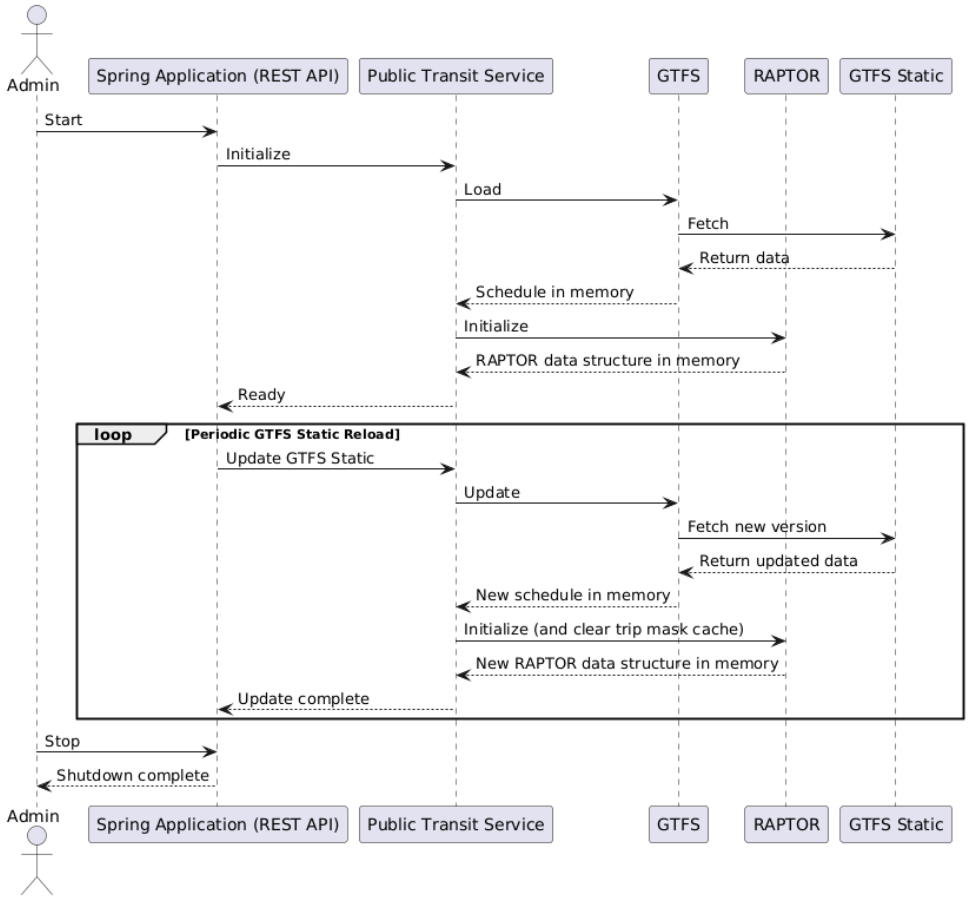

Figure 4.5: Application startup sequence diagram.

4.3.2 Connection Request

Sequence Diagram 4.6 illustrates how the Naviqore system processes a connection request by a user. The system first validates whether the requested stops are valid and retrieves parent or child stops if necessary.

If the stops are valid, the *RAPTOR* is used to query connections, and the connection results are enriched with information from the *GTFS* before being returned to the user.

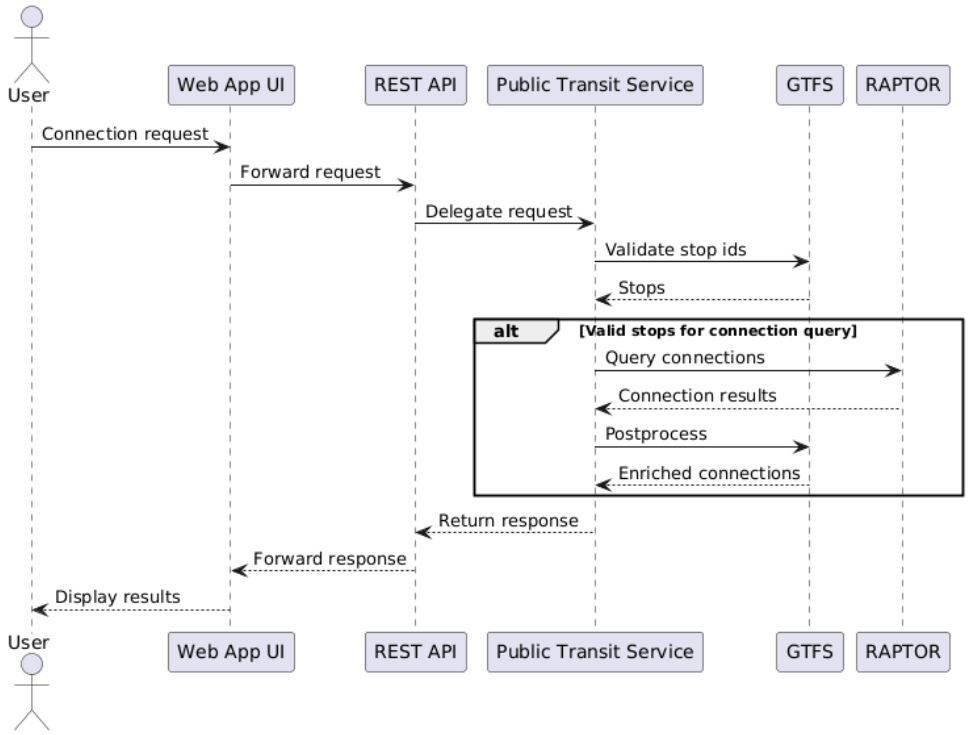

Figure 4.6: Connection request sequence diagram.

5 Implementation

5.1 GTFS Schedule

In the public transit service, only GTFS schedule data is utilized, which provides static transit service information for a specified validity period, without incorporating GTFS realtime data. The `GtfsScheduleReader` class processes all required files and fields, as well as some optional fields (accessibility) and optional files (transfers). The reader scans the corresponding CSV files and, in order to minimize memory usage, calls the `GtfsScheduleParser` to parse each record on the fly. The parser converts the string values from the CSV file into the appropriate data types and passes them to the `GtfsScheduleBuilder`, which constructs a valid `GtfsSchedule`.

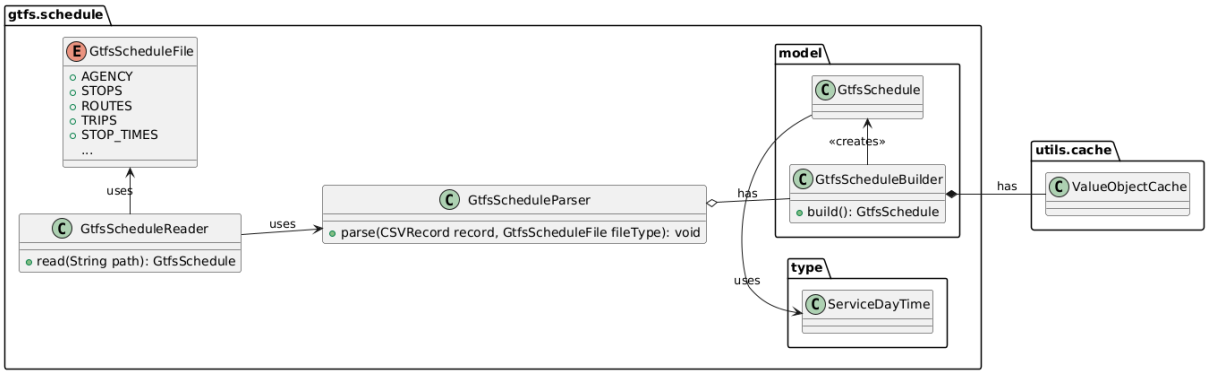

Figure 5.1: GTFS schedule class diagram.

The `GtfsScheduleBuilder` applies the *fail-fast principle*, ensuring that it only accepts valid inputs and will fail immediately if any invalid or missing data is encountered. Error handling is delegated to the `GtfsScheduleParser`, which catches and logs any errors, such as malformed data, during the parsing process. This guarantees that the builder only receives valid data. As a result, by the time the GTFS data is fully processed, the system can confidently expect a valid GTFS schedule to be stored in memory.

Since no changes are expected after reading the GTFS schedule data, the data is sorted and transformed into immutable data structures. This means that the various collections in the `GtfsSchedule` class, such as `ArrayLists`, are trimmed to the size of their contained objects. Similarly, the hash functions and load factors of the `HashMaps` are optimized based on their final size. To further reduce the memory footprint, frequently occurring value objects, such as service day times in stop times, local dates in calendar dates or calendar date exceptions, and string values (e.g., trip head signs), are cached using a value object cache. This caching mechanism ensures that repeated values are stored efficiently.

By employing these memory-optimized data structures and value object caching, the requirement **NF-SC-M1** is satisfied, allowing the GTFS static data for Switzerland to be stored in memory with a consumption of under 1.5GB.

5.2 Public Transit Service

In the service layer of our architecture, use cases intersect with the technical details of transit schedules and routing. These details are abstracted behind the `PublicTransitService` interface, ensuring that the application only interacts with well-defined interfaces without needing to know the underlying complexity. This package has two primary objectives:

1. **Stable Service Interface**: To provide a consistent and abstract interface that remains independent of the specific public transit schedule or routing implementation and potentially allows changing the implementation.
2. **Concrete Service Implementation**: To integrate a concrete data source and routing engine, in our case, the `GtfsRaptorService`, which implements the `PublicTransitService` interface.

The `PublicTransitService` is divided into two responsibilities covered by the `ScheduleInformationService` and the `ConnectionRoutingService` interfaces.

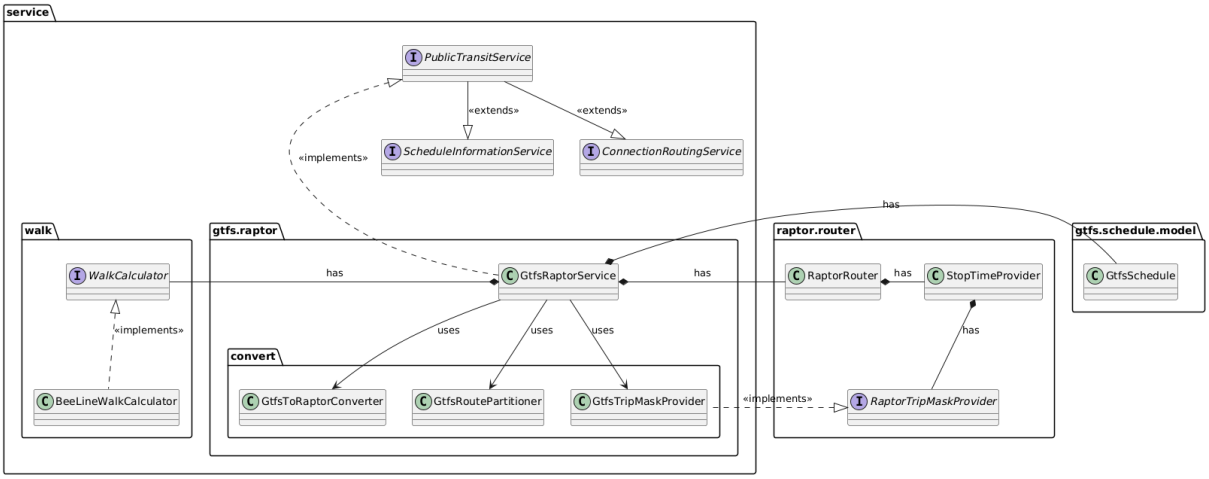

Figure 5.2: Public transit service class diagram.

5.2.1 Schedule Information Service

The schedule information service assists users in navigating public transit schedules. Users may need to search for stops by name or find nearby stops based on their current location. Once a stop is selected, the user can retrieve information about routes and upcoming departures.

To optimize these operations, the `GtfsRaptorService` employs various indexing structures:

1. **KDTree for Geospatial Queries**: To efficiently answer proximity queries (e.g., *"What are the nearest stops to my location?"*), we use a k-dimensional tree (KDTree) that operates on geographic coordinates (latitude and longitude). The distances between these coordinates are approximated using the Haversine formula [12], which provides sufficient accuracy for most inhabited areas of the Earth (excluding extreme polar regions).
2. **Compressed Trie for String Matching**: For searching stops by name, we implemented a compressed trie data structure. This allows for efficient storage and retrieval of stop names based on prefix matching. The trie is filled with all the stop names by inserting suffixes of the stop names into the structure [13]. This approach enables fast lookups for queries like *"find stops whose names start with, contain, or end with this substring."* Example of how the `suffixTrie` is filled:

```
public class SearchIndex<T> {

    // ...

    public void add(String key, T value) {

        for (int i = key.length() - 1; i >= 0; i--) {
            suffixTrie.insert(key.substring(i), entry);
        }

    }

}
```

Using these indices, queries for stops are highly efficient. The trade-off, however, is that these data structures consume more memory.

5.2.2 Connection Routing Service

The `ConnectionRoutingService` is responsible for finding connections between transit stops. In our `GtfsRaptorService` implementation, this service interacts with the extended RAPTOR routing engine (see Section 5.4 for details), which processes routing requests based on the GTFS schedule data.

Route Partitioning

The GTFS model and RAPTOR algorithm have differing perspectives on routes. In GTFS, a route refers to a user-friendly view of trips that may share a common line number or route name, even if the stop sequences differ slightly. For instance, a regional train might have trips that skip certain stops on later trips while still being part of the same route from a user's point of view.

However, RAPTOR requires that all trips associated with a route must have the exact same stop sequence. This discrepancy is resolved by the `GtfsToRaptorConverter` by applying the `GtfsRoutePartitioner`, which splits GTFS routes into sub-routes. These sub-routes group trips that share identical stop sequences, ensuring compatibility with RAPTOR's routing model. During post-processing, when the routing results are returned to the user, the sub-routes are converted back into the original GTFS routes to maintain user-friendly responses.

Transfer Generation

In GTFS, the transfer file is optional, meaning not all transfers between stops may be defined. To handle this, we introduced an optional transfer generation step in the `GtfsToRaptorConverter`. This step uses the `WalkTransferGenerator` to generate walking transfers between stops that are within a reasonable walking distance but are missing from the GTFS transfer data. Even when GTFS transfer data is provided, the generator can add missing, but viable, transfers and thus improve the connection results.

The walking distances between stops are calculated using a `WalkCalculator`. This implementation uses a beeline distance factor, typically set to 1.3 based on literature [14, p. 43], to account for real-world walking conditions such as obstacles, detours, or building access routes. The calculator multiplies the straight-line distance by this factor to approximate a more realistic walking time. The `WalkCalculator` interface allows for easy replacement with more advanced footpath routing solutions, such as A* or A$^+$ Landmark algorithms, if needed.

Handling Coordinates

In routing requests, users can specify coordinates (latitude/longitude) or stops as the origin or destination. When coordinates are provided, the `GtfsRaptorService` first identifies the nearest stops using an existing KDTree that indexes all the stops. The walking time to these stops is calculated using the `WalkCalculator` instance, and only stops within a reasonable walking duration, defined in the service configuration at startup, are considered in the routing request.

Once the RAPTOR routing engine processes the request, it calculates routes between the identified stops and returns the connection results. The `GtfsRaptorService` then enriches these results with walking segments (first mile and last mile), which represent the walk from the start coordinate to the selected origin stop and from the destination stop to the final coordinate.

In summary, the `ConnectionRoutingService` performs pre-processing to identify nearby stops, routes the request using the RAPTOR engine, and post-processes the results by adding walking segments and re-aggregating sub-routes to provide user-friendly routing information.

Note: Since we also support routing based on the latest departure, which involves routing backwards in time, we use the terms "source" and "target" stop in the code. However, in this explanation, we use the more common terms "origin" and "destination" for clarity.

5.3 Simple RAPTOR

The RAPTOR algorithm is a public transit routing algorithm designed for efficiently computing the fastest routes through a public transit network. This algorithm was initially implemented in Java as part of the early versions of this project (v0.1.0+), inspired by the work of Delling et al. [4]. The implementation relies on a series of well-structured data arrays that optimize route traversal and trip selection, as detailed in the appendix of their publication.

5.3.1 Builder

To streamline the construction of the data structures proposed by Delling et al. [4], a RAPTOR builder object was developed. This builder facilitated the inclusion of all active trips in a schedule for a specific service day. The process involved deriving the relevant routes, stops, stop times, and transfers from these active trips. Once all active objects were defined, they were sorted, and the proposed lookup arrays were instantiated.

Note: In the initial RAPTOR implementation, only stops served on the specific date for which the RAPTOR was built were included in the lookup objects. This approach was later modified in the extended implementation to incorporate all stops, regardless of service on the specific date.

5.3.2 Data Structures

The core data structures were implemented through the `RaptorData` interface, which is designed to return three primary records, each containing more detailed structures. These data structures were optimized with cache locality in mind to enhance routing performance. Instead of maintaining collections of object references, which would lead to scattered memory usage, the design involves storing indexes that cross-reference relevant objects across different arrays. A class diagram of the data structures is shown in Figure 5.3.

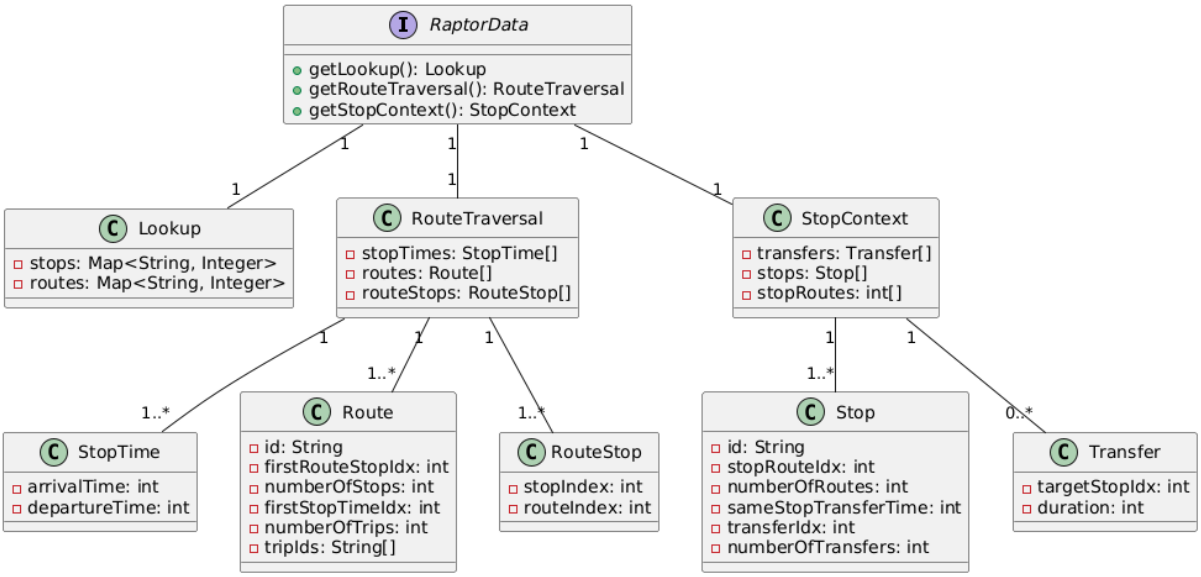

Figure 5.3: RAPTOR data class diagram.

Route Traversal

For efficient route traversal during the algorithm's route scanning loop, the following data structures are utilized:

- **Routes Array**: Each entry corresponds to a specific route and includes:
  - The number of trips associated with the route.
  - The number of stops in the route (identical for all its trips).
  - Pointers to two lists: one representing the sequence of stops along the route and another for the list of all trips operating on that route (stop times).
- **RouteStops Array**: Instead of maintaining separate lists of stops for each route, all stops are stored in a single array. Each route's stops are stored consecutively, with pointers in the `Route` object indicating the starting position for each route's stops.
- **StopTimes Array**: This array contains the trip times (arrival and departure) for each route, organized into blocks by route and sorted by departure time. Within a block, the trips are sorted by their departure times from the first stop on the route.

Stop Information

To support operations outside the root scanning loop, the following structures are used:

- **Stops Array**: This array contains information about each stop, including:
  - A list of all routes that serve the stop, crucial for route traversal and improvement operations.
  - A list of all footpaths (transfers) that can be taken from the stop, along with their corresponding transfer times.
- **StopRoutes Array**: Aggregates the routes associated with each stop into a single array, ensuring efficient access during the algorithm's execution.
- **Transfers Array**: Aggregates all footpaths available from each stop, along with their corresponding transfer times, into a single array.

By organizing these data structures in contiguous memory blocks, the implementation ensures efficient access and processing, critical for the performance of the RAPTOR algorithm.

Look Up

This record contains two maps that link the ID (type `String`) of a stop or route to the index in the corresponding `stops` array (part of `StopContext`) or `routes` array (part of `RouteTraversal`). This setup enables quick access to the `Stop` or `Route` objects.

This is used for pre- and post-processing routing requests. Requests are made using stop ids and the requester expects ids to be returned, since internal index numbers are not visible outside the RAPTOR implementation.

5.3.3 Router

The `RaptorRouter` implementation integrates both the `RaptorAlgorithm` interface, which handles routing operations, and the `RaptorData` interface, which stores all relevant data required for routing. This dual implementation ensures that `RaptorRouter` not only provides the necessary algorithmic logic for finding routes but also maintains access to the essential data needed to perform these operations efficiently.

Query Object Creation and Route Calculation

Upon receiving a routing request, a new `Query` object is instantiated. The route calculation is then initiated by invoking the `run()` method on this `Query` object. The `Query` object plays a central role in the routing process, as it contains references to several crucial components, as shown in Figure 5.4 and explained in the following list:

- **QueryState**: This object stores all the labels generated during the routing process. It also tracks the best times for each stop, which are essential for determining optimal routes.
- **RouteScanner**: This component handles everything related to scanning routes. It processes the routes based on the current state and round, marking relevant stops as needed.
- **FootpathRelaxer**: This object is used to evaluate possible transfers between stops. It checks if walking paths and transfers can be utilized to improve the route.

A sequence diagram illustrating the route calculation process is shown in Figure 5.5.

Isolines

The `routeIsolines()` method uses the same `run()` method as regular route calculations, but with one key difference: the `targetStops` are not defined in the `Query` object. Because of this, the usual comparison of routing labels against the best times for stops does not occur, allowing for a more generalized exploration of possible routes without focusing on specific destination stops.

Label Post Processing

The `LabelPostprocessor` class is responsible for post-processing the results of the RAPTOR algorithm by reconstructing connections from the labels generated during the routing process. It uses the labels from each round of the algorithm to build meaningful connections, which can then be returned as a list of routes or isolines.

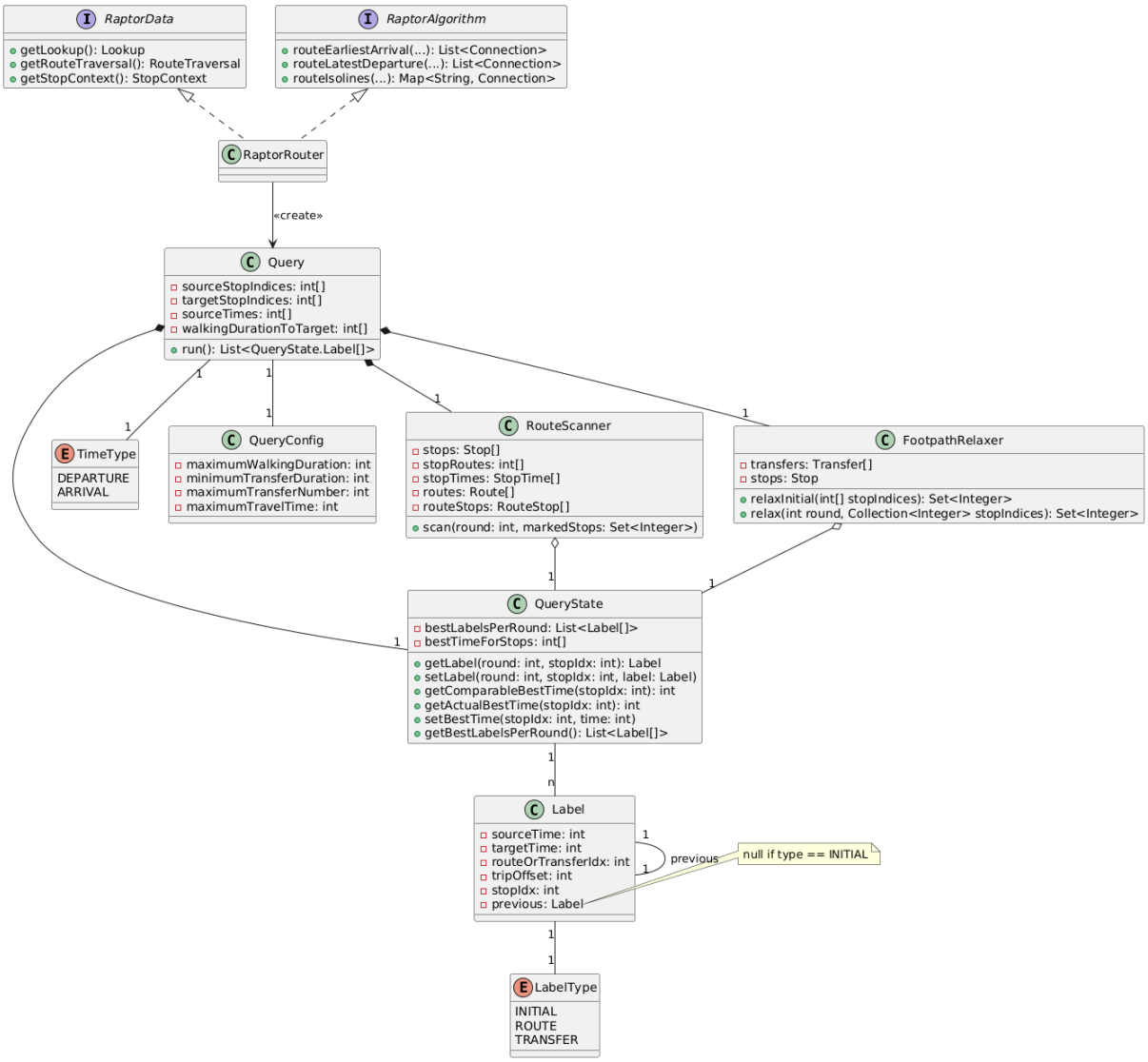

Figure 5.4: Query object and route calculation class diagram.

5.3.4 Range RAPTOR

Range RAPTOR is an extension of the classic RAPTOR algorithm, specifically designed to handle multiple departure times efficiently. Instead of calculating the optimal route for just a single departure time, it determines the best route over a range of potential departure times. This makes it particularly useful for trip planning within a time window, where the goal is to find the earliest possible arrival time given any departure between specified hours (e.g., "find the earliest arrival if I leave anytime between 8:00 and 10:00 AM"), ultimately reducing travel time.

Range RAPTOR identifies all departures within the specified time range from marked stops (i.e., departure stops and any potential walk transfers) after the initial round. It starts scanning routes from the latest possible departure and then progressively scans earlier departures. Through a process of self-pruning, previously computed labels (routes and their associated travel times) are only overwritten if an earlier departure results in a better (earlier) arrival time.

This approach offers a significant advantage: it minimizes idle time between connections. If a later connection can still catch the same subsequent leg of the journey, it effectively reduces travel time by pushing the departure time closer to the actual trip start without sacrificing the best arrival time.

The range RAPTOR algorithm was implemented as part of this project and benchmarked for performance.

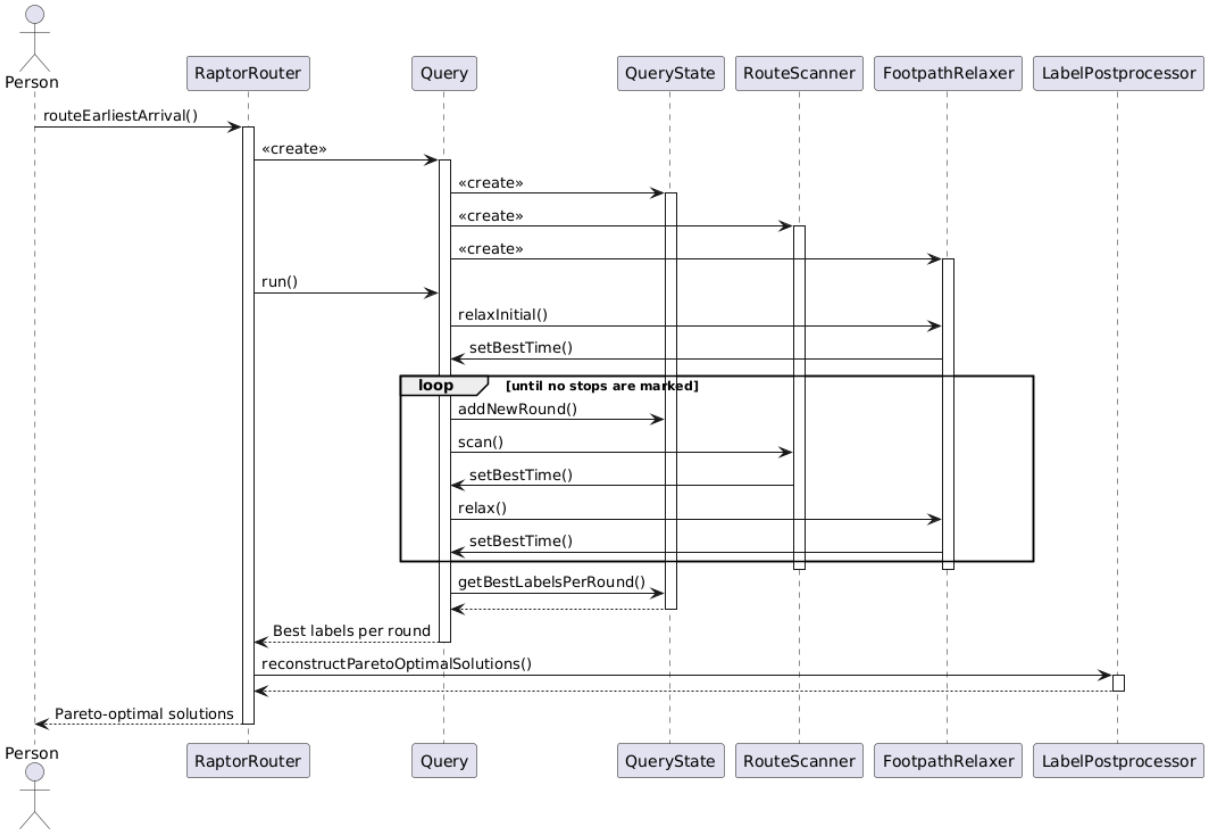

Figure 5.5: Sequence diagram for route calculation.

Since the implementation closely follows the textbook approach, no additional technical details are provided here.

5.4 Extended RAPTOR

The theoretical RAPTOR algorithm proposed by Delling et al. [4], while innovative, lacks several practical features necessary for real-world applications. These missing elements include reverse time routing (i.e., routing backwards from the arrival time and stop), key query configurations like setting a maximum walking distance between stops, defining minimum transfer times, and limiting the number of allowable transfers. Additionally, the original implementation does not account for diverse travel modes (e.g., tram, bus, rail), accessibility options for wheelchairs and bicycles, or the ability to route across multiple schedules spanning different service days.

In our extended RAPTOR implementation, we have addressed all of these gaps by incorporating these critical features, enhancing the algorithm's usability for real-world transit planning.

5.4.1 Multi-stop Routing

As mentioned in Section 5.2, the service can route to and from geographic coordinates by querying the nearest stops relative to those locations. However, the closest stops may be in opposite directions, and the service currently does not determine which stop is best suited for routing. To address this, an optimal approach is to route from multiple stops with different departure times, factoring in the varying walking times. Similarly, the service can route to multiple potential arrival stops, accounting for the walking times to the final destination from each stop.

This functionality was straightforward to implement and has been integrated into the extended RAPTOR. Now, routing requests expect a map where stop IDs serve as keys and the corresponding departure times or remaining walking durations serve as values for both the source and target stops.

In the routing process, only two changes were necessary. First, in round zero, multiple stops may need to be marked as starting points. Second, when determining the best arrival time at the target stops, the algorithm now checks multiple stops, factoring in the walking times (handicaps) before calculating the overall best time for each round.

5.4.2 Latest Departure

Implementing latest departure routing, which calculates the latest possible departure from a departure stop given a specified latest arrival time, did not introduce any new algorithmic concepts. However, achieving this in a way that minimizes code duplication while maintaining readability presented significant challenges. The primary difficulty lay in adapting the logic for reverse-time routing. In contrast to the earliest arrival routing, where the source stop is always the departure stop and the target stop is the arrival stop, these roles are reversed in latest departure routing. The source stop becomes the arrival stop, and the target stop is now the departure stop, as routing progresses backward in time.

This shift required changes to the variable naming conventions in the code. To avoid confusion, we adopted a generalized terminology where the "source stop" refers to the stop from which the scanning process begins (regardless of the direction), and the "target stop" refers to the destination of the scan. Additionally, loops that previously scanned for the earliest possible departure now had to scan for the latest possible arrival, requiring trips to be processed in reverse, starting from the latest possible trip rather than the earliest.

One of the biggest challenges was handling the conditional logic, which differs between earliest arrival and latest departure objectives. For earliest arrival, the algorithm marks the earliest arrival at each stop for further scanning, while for latest departure, the latest departure is marked. Implementing this distinction without duplicating large portions of code was nontrivial.

To strike a balance between maintainability and readability, we opted for an approach that introduces checks based on the current routing direction (i.e., earliest arrival vs. latest departure) within the same codebase, despite the added complexity. This approach was chosen over duplicating the entire algorithm with minor changes for each routing type, as the latter would have made long-term maintenance significantly more difficult.

5.4.3 Multi-day

The standard RAPTOR algorithm does not account for service days in a schedule, as it is primarily designed to scan routes based solely on departure times in ascending order. Given that GTFS schedules typically span a full year, directly chaining all departures in the stop times array would have resulted in an excessively large array, severely impacting performance.

In the early iterations of our RAPTOR implementation, we opted to build RAPTOR data structures for a single service day to address this issue. However, this approach introduced several limitations. For instance, a typical service day begins at 5 AM and extends into the early hours of the next calendar day (around 1 AM or 5 AM, depending on the availability of night services). As a result, routing requests for a local trip with a departure time at 12:00 AM might ideally be served by trips from the previous service day, but the algorithm would only display departures starting from 5 AM onward. Similarly, long-distance trips departing later in the afternoon would not be fully accommodated within the same service day, requiring the journey to extend into the next service day. This setup created gaps in service availability for both late-night and long-distance trips, necessitating further refinement of the algorithm.

An additional constraint for the RAPTOR implementation was that it should remain agnostic of the underlying GTFS schedule. This separation needed to be preserved in our design. To achieve this, we

introduced a `RaptorTripMaskProvider` interface that could be injected into the RAPTOR implementation via dependency injection. This allowed the RAPTOR algorithm to process schedule data without directly interacting with the GTFS implementation.

Trip Mask Provider

In our solution, the RAPTOR algorithm queries the `RaptorTripMaskProvider` for trip masks corresponding to different service days when handling a routing request. Typically, the algorithm requests trip masks for three days: the previous day, the current day, and the next day. Once the trip masks are retrieved, the routing process can begin, ensuring RAPTOR operates independently of how the schedule data is provided. A simplified UML diagram of the `RaptorTripMaskProvider` is shown in Figure 5.6.

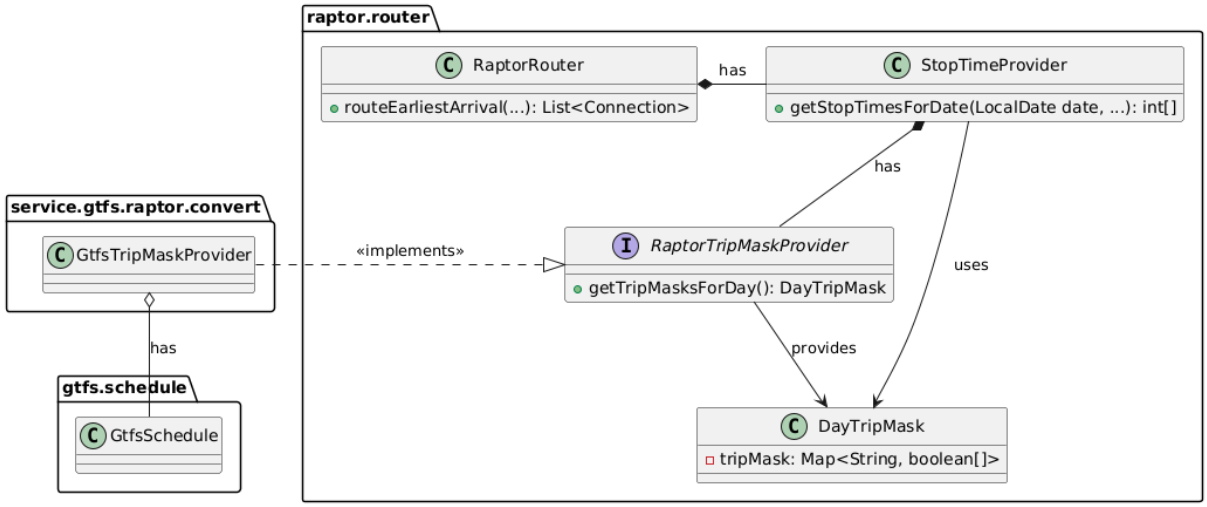

Figure 5.6: Trip mask provider class diagram.

The `DayTripMasks` provided by the `RaptorTripMaskProvider` consisted of a map, where the keys represented route IDs, and the values were boolean arrays that acted as masks. These arrays indicated which trips on a route were active on a given day. During route scanning, the process would typically begin by scanning the day trip mask for the previous day, then move to the current day's mask, and, if necessary, to the following day's mask if no suitable departure was found earlier. In the route scanner code, an additional check was implemented to ensure that the trip at the specified offset was active by verifying it against the day trip mask.

However, these modifications had a notable impact on performance. Routing requests, which previously met the target, now took around 250 milliseconds, falling short of the requirement for a response time of under 200 milliseconds. This highlighted the need for further optimizations to meet performance expectations.

New Stop Times Array Layout and Stop Times Provider

To address the performance issues, several improvements were identified and implemented. These included optimizing how departure times were accessed and restructuring the internal data to improve memory usage and scanning efficiency.

- **Easy Lookup Variables:** To reduce the number of stop time lookups, new variables were added to quickly determine if a route had any departures after the first available departure time.
- **Memory Optimization:** The stop times array was changed from an array of `StopTime[]` objects to an array of `int[]` to take advantage of cache locality. Since both arrival and departure times were integer values, this conversion allowed for more efficient memory access.

- **Pre-building Stop Times Array:** The stop times array is now pre-built before routing began, using the `DayTripMask` to mask invalid stop times by setting them to `Integer.MIN_VALUE`. This eliminated the need for multiple lookups during route scanning.

To implement these changes, an additional class called `StopTimeProvider` was introduced. It takes the `RaptorTripMaskProvider` as an injected dependency and is responsible for creating `int[]` stop time arrays for a given service day. These arrays contain all trips for each route, sorted by departure time, with additional improvements:

- **Service Day-level Information:** At indices 0 and 1 of the stop times array, data is included about the earliest departure and latest arrival for the entire service day. This allows the algorithm to skip scanning the previous day if, for instance, the routing request is for 8 AM and the previous day's service ends at 5 AM.
- **Route-level Information:** Each route's partition in the array includes two additional values at the start, indicating the earliest departure and latest arrival for that route. This further optimizes the process by allowing the algorithm to avoid scanning trips on a route if it is inactive at the requested time.
- **Simplified Stop Times:** Each stop time now consists of two integer values (arrival and departure times), replacing the previous `StopTime` object, leading to faster data access.

This new structure enabled the stop times array to serve multiple purposes while significantly improving routing performance, from 250 ms down to approximately 90 ms. However, accessing the correct information in the array became more complex, as shown in Figure 5.7.

5.4.4 Travel Mode, Accessibility and Bike Information

After implementing the multi-day logic for routing and adding functionality to the RAPTOR module to allow external masking of trips using schedule information, it became straightforward to extend the `QueryConfig` values to accommodate additional routing request parameters. These new parameters include options such as preferred travel modes (e.g., bus, train, ship) and specific requirements like ensuring that all trips are wheelchair accessible or allow bikes.

Technically, nothing significant changed in the core logic of the RAPTOR algorithm. However, the stop time arrays now became specific not only to the service day but also to the query configuration, meaning each set of preferences (e.g., travel mode, accessibility) would result in a unique stop time array. This led to an increase in the number of potential stop time arrays that needed to be cached, but it allowed for greater flexibility in handling diverse routing requests.

5.4.5 Caching

As expected, computing stop time arrays from schedule information using the `StopTimeProvider` and `RaptorTripMaskProvider` introduces significant overhead. Typically, this computation takes around 400-500 milliseconds per service day and query configuration, resulting in approximately 1200-1500 milliseconds for a multi-day routing request. To avoid recalculating these arrays for each new request, caching was identified as an effective solution.

To address this, a Least Recently Used (LRU) eviction cache was implemented to store the computed stop time arrays. This allows frequently accessed stop time arrays to remain in memory, while less-used ones are evicted to free up space.

However, it's important to note that for the Swiss GTFS schedule, the stop time arrays can be quite large, around 16 million integer values, corresponding to a memory footprint of 64 MB per service day and query configuration combination. This substantial memory requirement means that the application needs to run

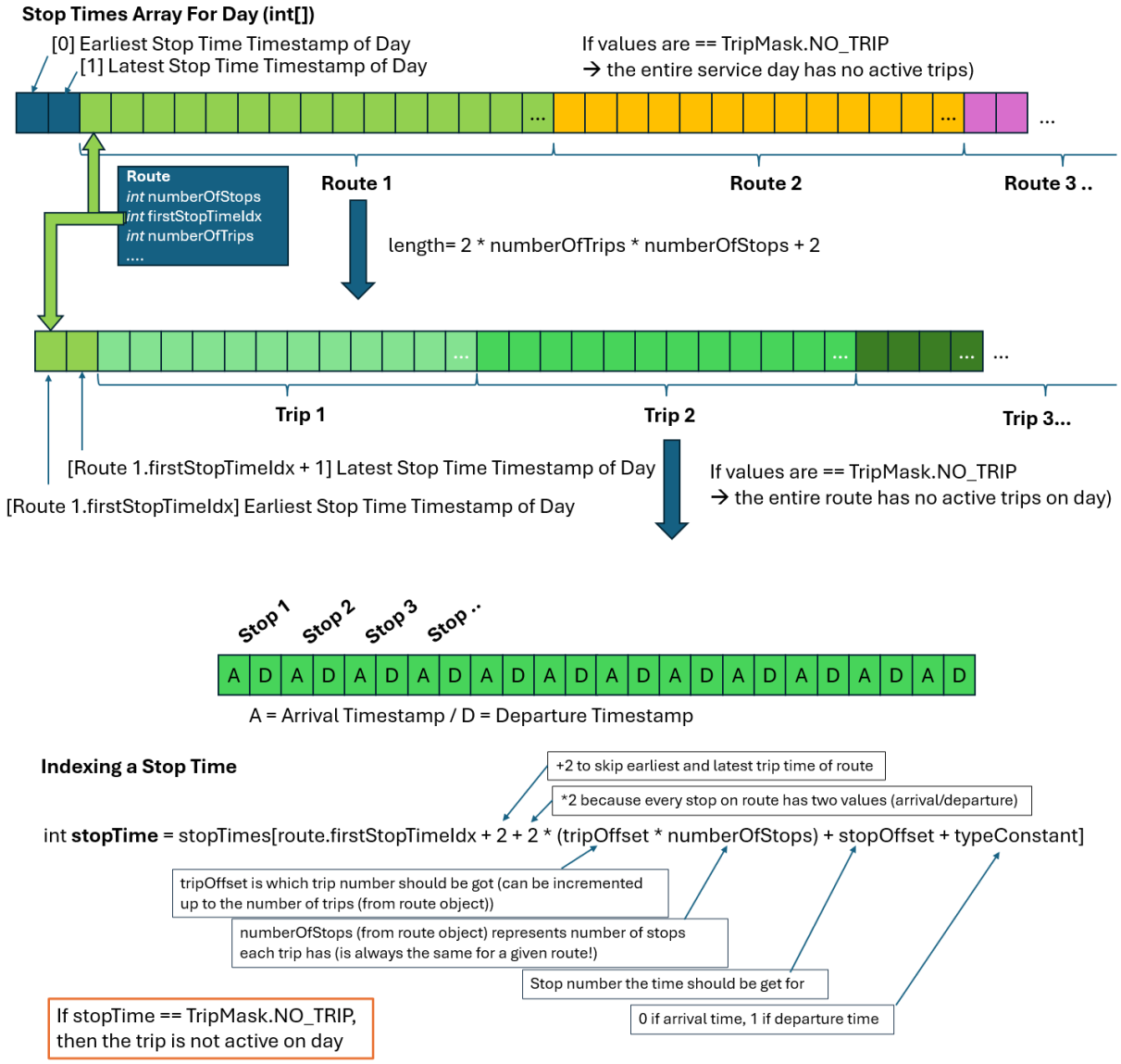

Figure 5.7: New stop times array layout.

on machines with significant memory resources or, ideally, in a distributed system for production. In such a system, a load balancer would distribute routing requests from the front-end layer across replicas of the router based on the available cached stop time arrays, ensuring that the application can run efficiently without excessive recalculations or memory strain. Unfortunately, a setup like this sacrifices the stateless nature of the application. To account for this, the stop time arrays and the GTFS schedule could be stored in a database, encapsulating the state within the database rather than the application, allowing for stateless load balancing.

5.5 C++ RAPTOR

This section discusses the design, implementation, and optimization of a public transport routing system, developed using modern C++20/C++23 standards. The project incorporates advanced data structures and algorithms to enable efficient route planning, with an emphasis on cross-platform compatibility and robust build configurations via CMake.

The C++ implementation is heavily inspired by the Java Simple RAPTOR Implementation, explained in Section 5.3, and utilizes modern C++ features. This section focuses on the C++ implementation, detailing the key components and challenges encountered during development.

The C++ implementation comprises:

1. A schedule reader.
2. A module for converting GTFS data into the RAPTOR structure.
3. A RAPTOR algorithm module.
4. A Geometry helper module.
5. A wrapper of spdlog for logging purposes.

A basic overview of the C++ RAPTOR implementation is shown in Figure 5.8.

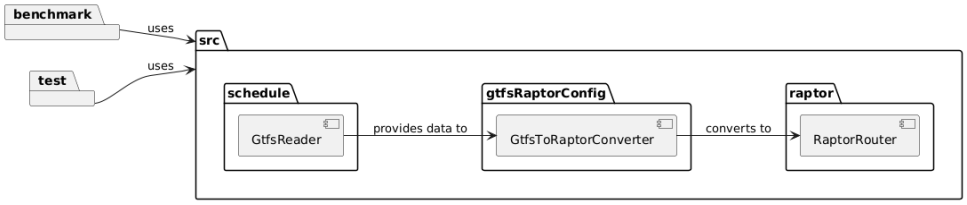

Figure 5.8: C++ RAPTOR package overview.

We aimed to utilize the most modern C++ features possible. Although we initially attempted to implement the project using the new C++ module system, limited compiler support (primarily from MSVC on Windows) posed challenges, which are further discussed in Section 6.3.

5.5.1 Code Efficiency: Patterns and Idioms

In an effort to make the code as efficient and maintainable as possible, we employed a variety of design patterns and modern C++ idioms. These strategies help in structuring the code for performance, clarity, and extensibility. Below are some of the key patterns and techniques we utilized.

Design Patterns Used in GTFS-Reader

1. Value-Based Strategy Pattern: We implemented the value-based strategy pattern to enhance flexibility and performance in reading the GTFS data. This design pattern allows runtime selection of a class's behavior without relying on traditional virtual methods and inheritance. Instead, we use callable objects such as functors, lambdas, or `std::function` to encapsulate the strategy. This approach avoids the overhead of virtual function calls, leading to better performance while maintaining flexibility.

2. Factory Pattern: The factory pattern was employed to simplify the creation of complex objects, decoupling the creation logic from the client code. This made our codebase more modular and easier to extend, particularly when integrating new components without impacting existing logic.

Ranges Views Library

We also leveraged the Ranges Views library from C++20 to optimize the handling and transformation of data collections. The Ranges Views library provides a powerful, declarative way to work with ranges of data, such as arrays, vectors, and other containers, without the need to manually write loops or intermediate data transformations.

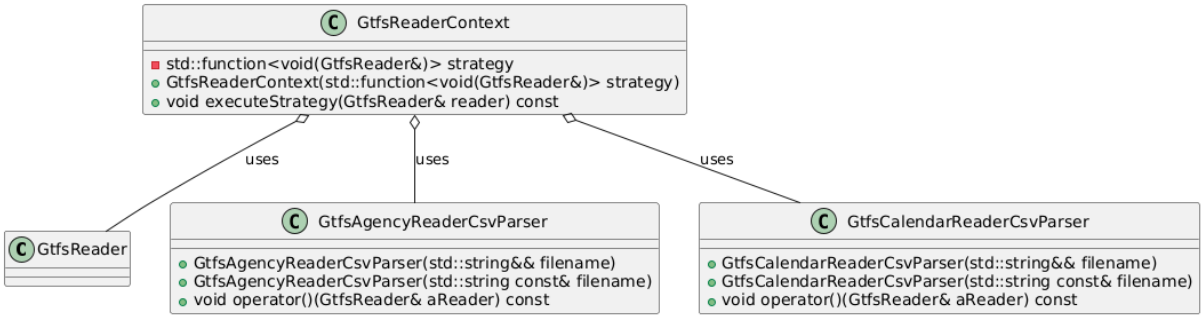

Figure 5.9: Value-based strategy pattern class diagram.

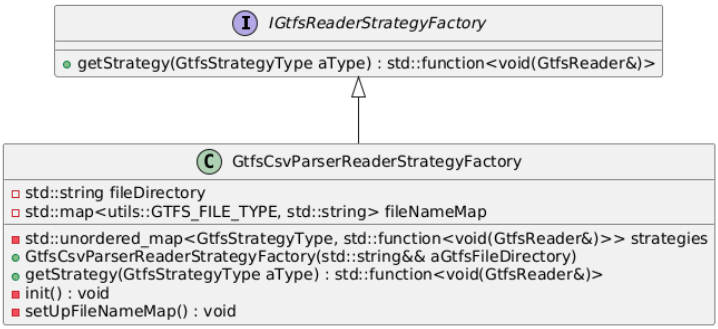

Figure 5.10: Factory pattern class diagram.

Advantages of Ranges Views:

- **Lazy Evaluation:** One of the primary advantages of ranges views is lazy evaluation. Unlike traditional containers, views are not eager; they do not perform operations immediately. Instead, they build a pipeline of transformations, only performing computations when the result is needed. This reduces unnecessary memory allocation and computation, making code both faster and more memory-efficient.
- **Composability:** Ranges Views allow you to chain multiple operations (e.g., filtering, transforming, sorting) in a clear and concise manner. This composability leads to cleaner, more readable code, reducing the risk of errors during iteration and transformation.
- **Improved Expressiveness:** With Ranges Views, the code becomes more expressive, focusing on *what* is being done (e.g., filtering, mapping) rather than *how* it's being done (i.e., the loop mechanics). This declarative approach improves code readability and maintainability.
- **Avoiding Intermediate Copies:** Ranges Views transformations occur on the original range without creating new containers or copies unless explicitly needed, thereby improving both memory and runtime efficiency.

Example Usage of Ranges Views in Our Code: In our implementation, Ranges Views allowed us to simplify complex operations like filtering trips, sorting schedules, or transforming datasets. For example:

```
const auto servedDates = data->calendarDates
    | std::views::values
    | std::views::join
    | std::views::filter(
        [this, &localDateTime]
        (const std::shared_ptr<schedule::gtfs::CalendarDate>& date) {
```

```
            return this->isServiceAvailable(
                date->serviceId, localDateTime
            );
        })
    | std::views::transform(
        [](const std::shared_ptr<schedule::gtfs::CalendarDate>& date) {
            return date->serviceId;
        })
    | std::ranges::to<std::set<std::string>>();
```

5.6 Client and Viewer

To demonstrate potential applications of our public transit service, we developed two Python packages.

5.6.1 Client

The `public-transit-client` package was created to encapsulate the entire interaction with the Spring REST API, offering pythonic functions and typed class objects for both request and response parameters. This package was built using `poetry` and published to PyPI, adhering to best practices, including the inclusion of the `py.typed` file to support type hinting. The client is designed for both frontend applications that integrate with the routing service and researchers who need to perform bulk routing requests directly from Python without relying on a graphical user interface.

5.6.2 Viewer

The second package, `public-transit-viewer`, is a mockup frontend application built using Streamlit, an open-source Python framework designed for quickly building and deploying data-driven web applications with minimal effort. It leverages the `public-transit-client` package to communicate with the Java-based public transit service. The viewer provides a graphical interface, as shown in Figure 5.11, to easily request connections and visualize results, making it useful for both demonstration and debugging purposes. The viewer is published as an image on the GitHub Container Registry for easy deployment.

While Streamlit allowed us to rapidly develop the application, its limited potential for customization and linear, non-modular code structure makes it less ideal for a fully custom, branded frontend. For projects requiring a more powerful and customizable user interface, we would recommend using a JavaScript framework. However, given the objectives of this study, Streamlit's simplicity and rapid development capabilities made it an adequate choice for our needs.

5.7 Testing

In general, tests are divided into unit test cases and integration test cases. Unit tests specifically target one unit of code (typically a class) in isolation, often with mocked dependencies when necessary. Integration tests, on the other hand, assess multiple components interacting together (e.g., a file reader working with a records parser and test data). In our case, integration tests may also include running external dependencies, such as in the client, where a service instance must be started using Docker Compose. We acknowledge that the boundary between integration and system tests can be blurred, but we have chosen to categorize these as integration tests rather than introducing a third test category. It is important that unit tests and integration tests can be executed in distinct phases.

We prioritized testing the most relevant and complex features over simply achieving high line coverage. A core principle during development was that no new or modified features should be implemented without

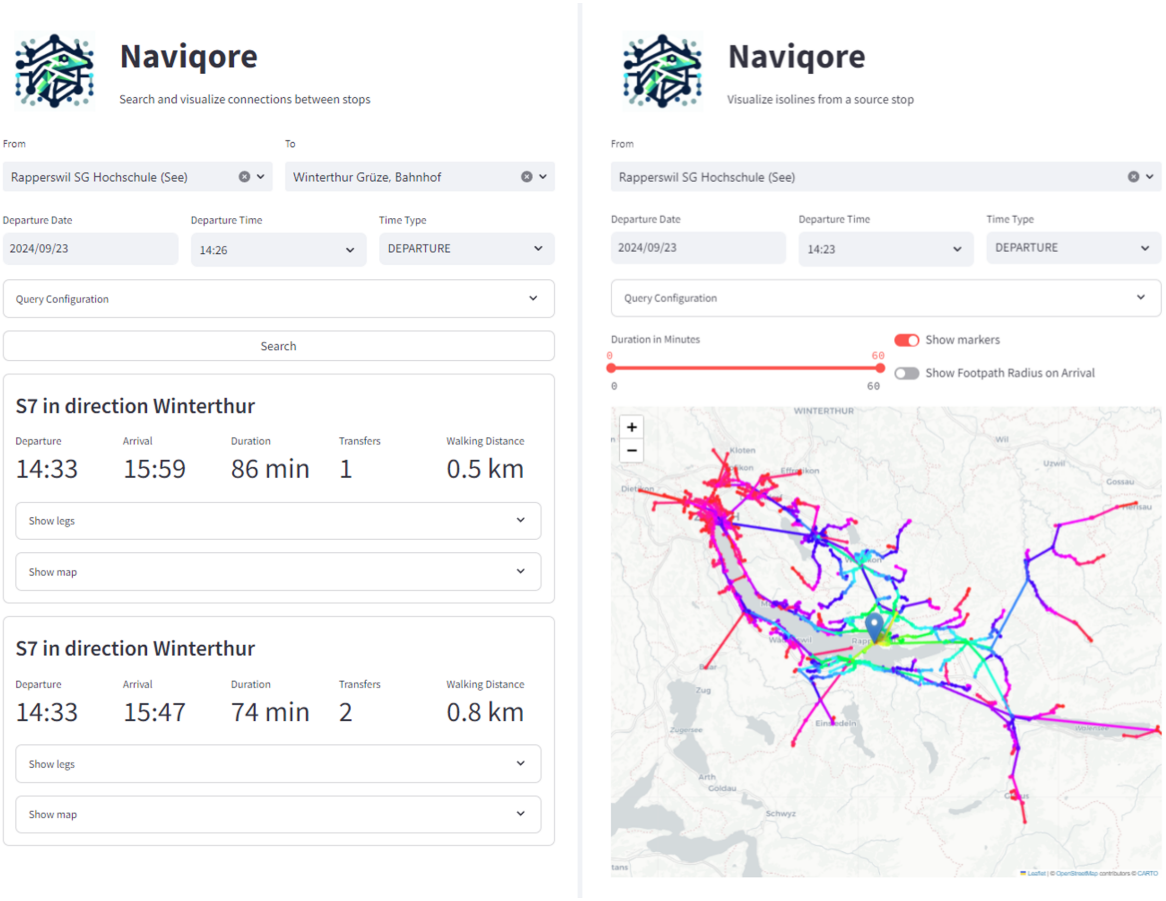

Figure 5.11: Viewer: Connections (left) and isolines (right).

corresponding tests. Writing tests after implementation was discouraged, as testing was an integral part of the development process. During code reviews, ensuring proper testing of new or modified components was a mandatory part of the review process, as outlined in the code guidelines (Appendix A.1).

The testing frameworks and tools used in each repository are as follows:

- **Java, Maven**: We use JUnit 5, the Maven Surefire Plugin for unit tests (executed in the Maven `test` phase, with a `<TestName>Test` suffix), and the Maven Failsafe Plugin (executed in the Maven `verify` phase, with a `<TestName>IT` suffix) for integration tests. This allows unit and integration tests to be run in separate phases.
- **Python, Poetry**: Pytest is used as the test runner, with Pytest markers to distinguish between unit and integration test categories.
- **C++, CMake, and vcpkg**: GoogleTest is used for testing.

5.8 Deployment

For instructions on deploying both the public transit service and the viewer, including details on configuration and environment variables, please refer to Appendix D.

6 Results and Discussion

6.1 Code Metrics

Table 6.1 provides an overview of some key code metrics for the main repositories involved in this thesis. The lines of code were counted using the command-line tool `cloc` and include both production and test code, excluding inline comments and documentation (such as Javadoc or docstrings). Test coverage is measured in terms of line coverage.

The repositories are:

- **public-transit-service** (Java): Public transit schedule and routing service based on GTFS data and the RAPTOR algorithm.
- **public-transit-client** (Python): Client to access the public transit service API endpoints.
- **public-transit-viewer** (Python): Viewer to interact with the public transit service.
- **raptorxx** (C++): Implementation of the RAPTOR algorithm in C++ for benchmarking.

Repository Name	Commits	Lines of Code *(Test)*	Test Cases *(Integration)*	Test Coverage
public-transit-service	751	12116 *(5662)*	569 *(51)*	86%
public-transit-client	72	822 *(419)*	23 *(15)*	80%
public-transit-viewer	125	1007 *(21)*	1 *(1)*	-
raptorxx	264	9164 *(1018)*	47 *(0)*	-

Table 6.1: Code metrics overview.

Since the focus of this thesis is on the RAPTOR algorithm, the service received the most testing attention. The viewer, primarily developed for demonstration purposes, has only one integration test to verify the application can boot, but lacks unit tests. For a production-ready frontend, additional unit tests would need to be implemented.

Note: The integration test coverage of the *public-transit-viewer* cannot be measured because Streamlit runs in a separate subprocess, preventing Pytest from tracking the executed lines. Further, no coverage can be calculated for the *raptorxx* project, since Clion only supports coverage for GCC and Clang and not MSVC. Visual Studio does not yet support coverage for CMake projects.

6.2 Benchmarking

6.2.1 Extended RAPTOR for Production

According to **NF-RO-M2**, the benchmarking of the extended RAPTOR algorithm (supporting multi-day connections, querying by departure or arrival times, and allowing for custom query criteria such as

transport modes, number of transfers, maximum walking distance, minimum transfer time, accessibility, or the possibility of carrying a bicycle) was conducted using the current GTFS data for the whole of Switzerland. The results were continuously versioned to files, which allowed us to track the performance of our algorithm during development. The machine used for the benchmark meets the requirements outlined in **NF-RO-M2**.

Figure 6.1: Extended RAPTOR benchmark results.

The benchmark results are visualized in nine plots. Processing time is shown to increase with both the number of transfers and the number of connection results, with median times remaining under 100 ms for most cases. The histogram of processing times reveals a mean of 48 ms and a median of 50 ms per request, confirming efficient connection query handling. Scatter plots show a positive correlation between processing time and both beeline distance and connection duration. Most queries involve 1-2 transfers (80%), and departure time accuracy is high, with minimal deviation between requested and actual times.

In regular use, on a large public transit schedule dataset on an average machine, the router performs efficiently and fully satisfies the requirements **UC-SE-M3** (*Efficiently request connections between two stops, in mean <250ms*) and **NF-SE-M1** (*The complete service for Switzerland must be able to run on a standard machine with 8 cores and 16GB of RAM*).

When comparing our system to the RAPTOR implementation (*SwissRailRaptor*) by Rieser et al. [15], we found that our router performed comparably well, though slightly slower when tested on Switzerland's national public transit schedule. This performance trade-off is acceptable, given that our system is designed for real-world applications, such as handling diverse service days, accessibility information, multi-day trips, and latest-departure routing. In contrast, *SwissRailRaptor* is optimized for simulation efficiency within the MATSim framework.

As specified in **NF-RO-M1**, more than five connections were manually validated using the SBB app as a reference. There were generally no discrepancies, and when differences did occur, they were attributable to service configuration settings, such as minimum transfer times or maximum walking distances. Requirement **UC-RO-M2** is also fulfilled, as all compared connection results are Pareto-optimal.

Performance Improvements

During profiling of connection requests, it was discovered that approximately 25% of the CPU time per routing request was consumed by inserting new stops into the marked stops hashset in the main loop of the RAPTOR router. Since the set of marked stops is relatively small, we hypothesized that frequent collisions might occur in the hash function, leading to inefficiencies. However, attempts to optimize the hash function did not result in significant performance gains.

To address this, we replaced the hashset with a boolean array, where each index corresponds to a stop, and stops to be scanned in the next round are flagged as true. This improved performance considerably, reducing the time per request from 76 ms to 48 ms, a 37% performance improvement, on the GTFS dataset for Switzerland.

6.2.2 RAPTOR Versions for Comparison

After adding multiple configurations for the RAPTOR algorithm, it was interesting to compare the performance of different configurations. All of the compared configurations were benchmarked with 1000 identical routing requests.

Tested Implementations

- **Simple RAPTOR**: A minimal implementation of the algorithm, without support for multi-day connections (no `TripMaskProvider`) and only supporting earliest arrival queries. Since it only includes routes and trips operating on the selected service day, the number of routes and stop times is reduced. This results in fewer checks and lookups, making this version expected to be the fastest.
- **Regular Extended RAPTOR**: This version operates on the same core code as the following implementations but is configured in the most basic way. It does not support departure ranges (i.e., no range RAPTOR) and does not use multi-day masks (only loading the service day mask for the active day). This should be the fastest of the "extended" implementations.
- **Multi-day RAPTORs**: For these tests, the range was set to 0, and different numbers of days were scanned. Here, *1 day* refers to scanning only the active service day, *2 days* means scanning the active and previous day, and everything greater than *3 days* means scanning the previous, the current, and *n-2 days* in the future.
- **Range RAPTORs**: Basic range RAPTORs were run with different range values but without multi-day masks. Tested ranges were 30 minutes (1800 s), 1 hour (3600 s), 3 hour (7200 s), and 4 hour (14400 s).
- **Multi-day Range RAPTORs**: To assess the impact of combining multi-day scans and range RAPTORs, the same range values used in the range RAPTOR tests were tested again, but with the scan set to 3 days.

Results and Discussions

As expected, the simple RAPTOR implementation was the fastest, with an average runtime of 32 ms (± 21 ms), since it required the fewest comparisons and lookups. The additional overhead of loading the full stop times array and masking non-active trips with `Integer.MIN_VALUE` added around 7 ms, resulting in an average of 39 ms (± 25 ms).

The performance comparison between range RAPTOR and Multi-day RAPTOR was more intriguing. Adding small ranges to the RAPTOR configuration did not significantly increase computational time (+7% for a 30-minute range, +42% for a 1 hour range). This can be attributed to the benchmarking process, which sampled stops randomly across Switzerland. Small stops with limited service were overrepresented, leading to fewer departures within the specified range, and often resulting in scanning only one set of

departures. However, for larger ranges, the cost became more pronounced (+112% for a 2-hour range, +287% for a 4-hour range).

Similarly, increasing the number of days to scan had a noticeable but less dramatic impact. Scanning across 3 days (the previous, current, and next day) increased the runtime by +48%, while scanning across 5 days resulted in a 100% increase in runtime.

When combining both range and multi-day scans in one RAPTOR implementation, the performance costs were cumulative, as both factors added their respective overheads. All performance metrics are summarized in Table 6.2, and the boxplot in Figure 6.2 visualizes the distribution of processing times for each RAPTOR implementation.

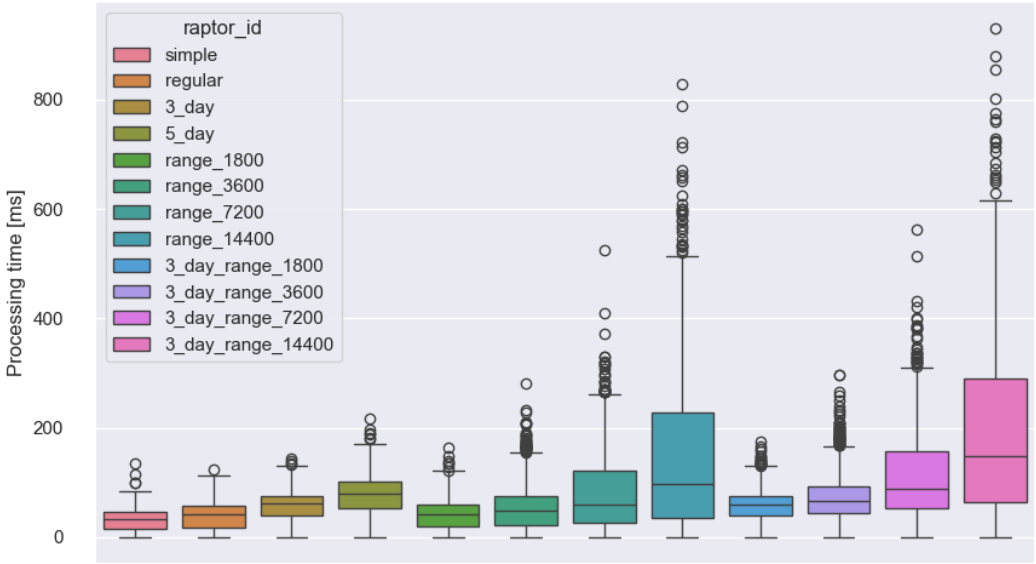

Figure 6.2: Router comparison.

router_id	count	mean (ms)	std (ms)	median (ms)	max (ms)
simple	1000	32	21	34	135
regular	1000	39	25	42	123
3_day	1000	58	28	62	144
5_day	1000	78	36	80	218
range_1800	1000	42	29	43	163
range_3600	1000	55	44	48	282
range_7200	1000	83	76	59	525
range_14400	1000	151	152	98	828
3_day_range_1800	1000	59	31	60	175
3_day_range_3600	1000	74	47	67	296
3_day_range_7200	1000	113	86	90	562
3_day_range_14400	1000	198	170	149	930

Table 6.2: Benchmark results for different RAPTOR implementations.

Of course, adding more features to the RAPTOR algorithm not only increased computational costs but also provided better results and enhanced flexibility for more specific queries (e.g., travel modes, wheelchair accessibility, etc.). Most notably, the range RAPTOR enabled finding more time-efficient connections by allowing later departures while still achieving the same earliest arrival. Meanwhile, the multi-day RAPTOR made it possible to find earlier arrivals, as trips from the previous service day could be utilized. It also increased the success rate for connections with limited service, where trips from the following days were necessary.

Table 6.3 and Figure 6.3 illustrate these results:

- **Success Rate**: The percentage of requests that successfully found a connection between the departure and arrival stops.
- **Earliest Arrival Rate**: The proportion of requests that resulted in the earliest possible arrival.
- **Shortest Duration Rate**: The percentage of requests that returned the latest possible departure with the shortest overall travel time.

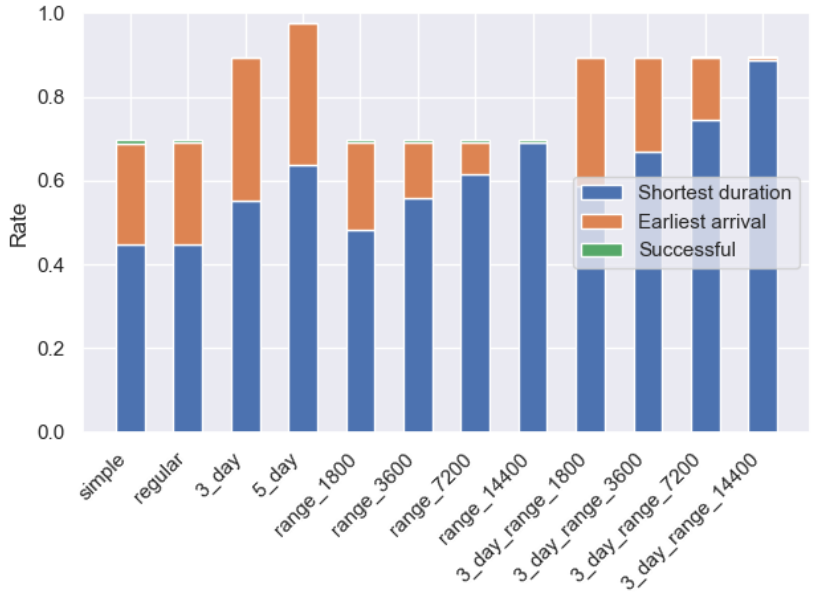

Figure 6.3: Router success rates.

raptor_id	Successful	Shortest Duration	Earliest Arrival
simple	69%	44%	69%
regular	69%	44%	69%
3_day	89%	55%	89%
5_day	97%	63%	97%
range_1800	69%	48%	69%
range_3600	69%	55%	69%
range_7200	69%	61%	69%
range_14400	69%	69%	69%
3_day_range_1800	89%	58%	89%
3_day_range_3600	89%	66%	89%
3_day_range_7200	89%	74%	89%
3_day_range_14400	89%	88%	89%

Table 6.3: Performance metrics for different RAPTOR implementations.

In summary, adding a greater range and more days to scan significantly improves the routing responses. However, this effect is somewhat biased due to the dataset used, which includes many small stops with only a few departures per week. This makes the impact of adding larger ranges and more days appear more extreme. In contrast, the differences would be less pronounced in an urban network with high-frequency service.

Therefore, when configuring the RAPTOR algorithm, it's important to select parameters based on the characteristics of the typical user and the specific transit network being routed, to ensure optimal performance.

6.3 C++ and Java Comparison

6.3.1 Performance

In our development of the public transit routing system Raptor, we aimed to leverage the performance advantages of C++. However, our analysis indicated that while C++ is indeed faster in several cases, Java's performance is impressively close. The Java Virtual Machine (JVM) optimizations, along with its automatic memory management and just-in-time compilation, enabled Java to execute routing requests with competitive efficiency.

The testing results (Table 6.4) showed that C++ performed better in many scenarios, often demonstrating lower elapsed times. However, Java consistently delivered strong performance, falling just short in a few cases. This indicates that, although C++ can offer high performance with fine-grained control over system resources, achieving that performance requires careful management of various factors, such as memory allocation and data structure handling.

Additionally, potential inefficiencies or errors in the C++ implementation can affect overall performance. The challenges of adapting the C++ code to align with Java's memory layout further complicate the process.

Ultimately, while C++ presents a powerful option for high-performance applications, Java's optimizations and robust ecosystem often provide a more practical and reliable solution for our routing needs. This highlights that sometimes high-level abstractions can lead to results that are not only competitive in speed but also more manageable in terms of development and maintenance.

Source Stop	**Target Stop**	**Iterations**	**Java Elapsed Time (ms)**	**C++ Elapsed Time (ms)**	**Average Difference (ms)**	**Average Difference (%)**
8589640 (St. Gallen, Vonwil)	8579885 (Mels, Bahnhof)	100	13	19	-6	-31.6%
8574563 (Maienfeld, Bahnhof)	8587276 (Biel/Bienne, Taubenloch)	100	55	38	17	44.7%
8588524 (Sion, Hôpital Sud)	8508896 (Stans, Bahnhof)	100	45	20	25	125.0%
8510709 (Lugano, Via Domenico Fontana)	8579255 (Lausanne, Pont-de-Chailly)	100	46	29	17	58.6%
8574848 (Davos Dorf, Bahnhof)	8576079 (Rapperswil SG, Sonnenhof)	100	12	15	-3	-20.0%

Table 6.4: Comparison of Java and C++ performance.

Summary of Average Differences

The *Difference (ms)* column reflects the absolute time difference in milliseconds between Java and C++ for each routing request. A negative value indicates that Java was faster than C++, while a positive value shows that C++ performed better.

The *Difference (%)* column presents the percentage difference in performance between Java and C++.

Positive percentages indicate that C++ performed better (was faster), while negative percentages indicate instances where Java was faster.

$$\frac{\text{Java Elapsed Time} - \text{C++ Elapsed Time}}{\text{Java Elapsed Time}} \times 100$$

Benchmarking System Specifications

The following table summarizes the key specifications of the machine used for testing the performance of Java and C++ implementations of RAPTOR. The tests were conducted on a high-performance Windows laptop designed for heavy computational tasks, ensuring a robust environment for comparing the performance of both languages.

Specification	Details
Operating System	Microsoft Windows 11 Pro
System Model	HP ZBook Studio x360 G5
Processor	Intel® Core™ i9-9980HK CPU @ 2.4 GHz
CPU Cores	8 physical cores
Logical Processors	16 logical processors (Hyper-threading)
Installed Physical Memory	64 GB

Table 6.5: Benchmark system specifications.

6.3.2 Challenges

The implementation of RAPTOR in C++ was aimed at leveraging the language's potential for superior performance compared to Java. However, this endeavor turned out to be more challenging than initially expected, particularly because of Java's highly optimized Java Virtual Machine (JVM), which often provides excellent performance through Just-In-Time (JIT) compilation and other built-in optimization strategies. While C++ offers more granular control over system performance, several complex aspects of development in C++, including cross-platform compatibility, memory management, and build management, introduced significant hurdles.

Cross-platform Compatibility: One of the primary challenges in implementing RaptorXX in C++ is ensuring cross-platform compatibility. Unlike Java's "write once, run anywhere" philosophy, which abstracts away platform differences through the JVM, C++ developers must account for variations in compilers, libraries, and system calls across operating systems like Windows, Linux, and macOS. This requires conditional compilation and platform-specific code to handle these differences, adding complexity to the project.

Additionally, the C++ Standard Library (STL), while powerful, does not offer built-in support for many common tasks, such as logging or CSV parsing. In contrast, Java's comprehensive standard library offers out-of-the-box solutions. In C++, developers often need to rely on third-party libraries like `spdlog` for logging or `csv-parser` for file handling, further complicating development. Furthermore, C++ compilers do not universally support all C++20 features, requiring careful attention to compatibility issues for seamless cross-platform development.

Memory Management: Another area where C++ introduces complexity compared to Java is memory management. Java automatically handles memory management via garbage collection, allowing developers to focus on high-level design without worrying about manual resource allocation. In contrast, C++ requires developers to manually manage memory, which introduces risks such as memory leaks, buffer overflows, and dangling pointers.

Though modern C++ provides tools like smart pointers (`std::unique_ptr`, `std::shared_ptr`) to alleviate some of these concerns, their correct use requires meticulous planning. The burden of manual memory management in C++ contrasts sharply with the ease of Java's automated garbage collection, where such issues are mostly abstracted away.

Error Handling: While both C++ and Java use exceptions for error handling, C++ introduces additional complexity in ensuring exception safety and resource cleanup. In Java, the combination of try-catch blocks and automatic memory management simplifies this process. However, in C++, developers must carefully design their code to release resources manually, especially in the presence of exceptions. Although C++ has best practices like RAII (Resource Acquisition Is Initialization) to manage resources efficiently, implementing them correctly requires more effort than in Java.

Dependency and Build Management: Another significant challenge with C++ development lies in managing dependencies and the build process. Java's mature ecosystem, featuring tools like Maven and Gradle, simplifies dependency management and build configuration, providing tight integration with development environments. In contrast, C++ lacks such streamlined tools. Although CMake is a widely used tool for build management in C++, it often requires extensive manual configuration, especially for large projects.

To manage dependencies, we relied on Microsoft's `vcpkg`, a package manager that simplifies the process to some extent. However, tools like `vcpkg` and `Conan` are still not as seamless or well-integrated as Java's solutions. In addition, setting up and configuring debugging tools, profilers, and other development resources in C++ requires more effort and familiarity with the toolchain. In our case, we primarily used CLion from JetBrains, but we occasionally had to switch to Visual Studio for specific tasks, adding to the complexity.

Performance Optimization: One of the theoretical advantages of C++ over Java is its potential for superior performance, thanks to fine-grained control over system resources. However, achieving this level of optimization requires deep knowledge of both C++ and the underlying hardware. In contrast, Java abstracts much of this complexity, enabling developers to focus on application logic while the JVM handles performance optimizations in the background.

Although C++ can outperform Java in specific scenarios, writing C++ code that is both highly optimized and maintainable presents significant challenges. The need to balance performance optimizations with code readability and maintainability often makes development in C++ slower and more complex compared to Java.

6.3.3 Foreign Function and Memory (FFM) API

Initially, we explored the possibility of calling the C++ RAPTOR implementation from Java using the Foreign Function and Memory (FFM) API [16]. However, we faced significant challenges due to the complexity of mapping intricate data structures between C++ and Java. Ultimately, we decided against this approach.

The FFM API (Foreign Function & Memory) was not well suited for our application. Sending routing requests from Java to C++ required C++ to interact with a memory structure created in Java, which would have been highly complex to implement. Since Java was the primary application, the alternative would have been to recreate the schedule for each request from files in C++, which would have been extremely inefficient.

A more effective approach would have been to implement a special C++ service that Java could call to forward tasks. While the FFM API is useful for certain types of function substitutions, it was not designed to manage the larger, integrated workflows we needed and was therefore unsuitable for our use case.

6.3.4 Summary

While C++ offers greater control and the potential for higher performance, these advantages come with significant trade-offs in complexity. Ensuring cross-platform compatibility, managing memory manually, and handling dependencies and builds are all more challenging in C++ than in Java. Additionally, the time and effort required to fully leverage C++'s performance advantages can outweigh the potential gains, especially given Java's highly optimized JVM, which often delivers comparable or better performance in many real-world scenarios. Ultimately, while C++ has its strengths, the ease of development and robust tooling in Java make it a more practical choice for many applications, including the RAPTOR algorithm.

6.4 Limitations

While this thesis demonstrates the potential and applicability of RAPTOR for public transit routing using GTFS data, there are several limitations to the approach and implementation that need to be acknowledged.

One significant limitation is the absence of useful optional components in the GTFS publications by various national transit agencies. While the mandatory fields are generally present, many agencies do not provide certain optional fields that could enhance the service's capabilities. For example, the Swiss GTFS data lacks some footpaths between stops, even though these transfers are possible. Additionally, there is no information regarding accessibility features or bike transportation, which limits the service's adaptability for users with specific needs.

International routing presents another significant challenge, as it requires working with multiple GTFS feeds. Merging datasets from regions within the same timezone is technically feasible but introduces its own complexity, such as resolving duplicate routes and stops that may have different identifiers. When dealing with multiple timezones, all stop times and calendar dates must be converted to UTC before merging schedules.

The trade-off between memory usage and performance is another limitation of the system. Our RAPTOR router requires a significant amount of memory, particularly for building and caching various stop time arrays, as a new array is generated for each combination of service days, transport modes, accessibility options, and bike transport settings. However, as the size of the GTFS dataset increases, so does the cache, potentially leading to memory exhaustion. In resource-constrained environments, the cache size must be reduced, negatively impacting performance, especially when users request a wide range of query combinations.

Additionally, the implementation of the RAPTOR in C++ brought several technical obstacles. The complexity of working with the Foreign Function and Memory (FFM) API, multi-operating system builds, and dependency management through CMake and vcpkg presented significant hurdles. Moreover, setting up a logging framework without introducing dependencies throughout the project required extensive effort, with the bridge pattern ultimately being adopted to isolate logging logic.

Lastly, the initial goal of using the FFM API to pass routing requests from the benchmarking setup to the C++ RAPTOR was abandoned. The need to reduce the complex RAPTOR data structures in Java to primitive types for the API, and then rebuild them on the C++ side, would have significantly impacted performance. Consequently, the C++ benchmarking was conducted independently of the Java implementation.

7 Conclusion and Outlook

In conclusion, the project successfully fulfilled all the must-have, most of the should-have, and some of the nice-to-have requirements, delivering a production-ready public transit routing solution.

One of the key strengths of this project was our ability to extend the RAPTOR algorithm to accommodate more complex real-world requirements, such as calculating connections spanning multiple days, incorporating latest departure routing, and considering accessibility or bike transportation constraints in connection requests. These aspects were only identified as requirements during the course of the project. The iterative approach we adopted, along with careful task management, allowed us to address newly identified requirements while maintaining progress. This demonstrated the flexibility and adaptability of both the algorithm and the chosen software development process.

The team's strong motivation and collaborative spirit were essential to our success, creating a productive environment driven by a compelling and challenging topic. Our extensive use of pull request reviews helped ensure high code quality, leading to solid design decisions and robust testing practices. In addition to these technical achievements, the final product is available as open-source under the MIT license, which encourages future contributions and improvements. Each team member also expanded their expertise in key areas such as data structures, algorithms, language specifics, and software architecture, making this project a valuable learning experience.

However, there were areas for improvement. One notable challenge was the minimal overlap between the C++ and Java components, which led to reduced synergy during the development process. Additionally, the inconsistent implementation of GTFS by national transit agencies posed challenges, such as invalid ID references, mostly missing accessibility information, and incomplete transfer data between stops. The intense project schedule, with long hours spent coding during evenings and weekends, also placed considerable strain on the personal lives of the team members, highlighting a work-life balance issue.

Looking to the future, several areas of potential development could further enhance the product. Introducing fare-based multi-criteria routing through mcRAPTOR could offer users more tailored, cost-effective options. Incorporating real-time GTFS updates would make the system more responsive to current conditions, improving its accuracy. Implementing an A*-based routing for footpaths would enhance the precision of walking routes between transit stops. Implementing a common storage architecture for loaded schedules and router stop times within the public transit service would enhance horizontal scalability and encapsulate the state within a new, dedicated data layer.

Overall, we believe this project has established a strong foundation for future development. As we conclude our MAS in Software Engineering at the Eastern Switzerland University of Applied Sciences, we are excited to see how the project evolves and progresses in the future.

Bibliography

[1] "Gtfs cookbook," https://opentransportdata.swiss/de/cookbook/gtfs, n.d., retrieved May 25, 2024.

[2] M. Unterfinger, M. Brunner, and L. Connolly, "Efficient public transit routing with raptor: A production-ready solution leveraging gtfs data," https://github.com/naviqore, 2024, retrieved September 26, 2024.

[3] "General transit feed specification," https://gtfs.org, n.d., retrieved May 25, 2024.

[4] D. Delling, T. Pajor, and R. F. Werneck, "Round-based public transit routing," in *Proceedings of the Meeting on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 2012, pp. 130–140. [Online]. Available: https://doi.org/10.1137/1.9781611972924.13

[5] K. Schwaber and J. Sutherland, "The scrum guide: The definitive guide to scrum: The rules of the game," https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-US.pdf, 2020, retrieved May 25, 2024.

[6] J. Sauer, "Public transit journey planning," Lecture slides, Karlsruhe Institute of Technology (KIT), 2021, retrieved May 21, 2024.

[7] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, 1st ed. USA: Prentice Hall Press, 2017. [Online]. Available: https://doi.org/10.5555/3175742

[8] J. Palermo, "The onion architecture, part 1," https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1, 2008, retrieved June 8, 2024.

[9] "Spring boot reference documentation," https://spring.io/projects/spring-boot, n.d., retrieved May 31, 2024.

[10] S. Brown, "The c4 model for visualising software architecture," https://c4model.com, n.d., retrieved June 14, 2024.

[11] ——, "The c4 model for software architecture," https://www.infoq.com/articles/C4-architecture-model, 2018, retrieved June 14, 2024.

[12] "Haversine formula," https://en.wikipedia.org/wiki/Haversine_formula, n.d., retrieved May 25, 2024.

[13] M. T. Goodrich and R. Tamassia, *Algorithm Design and Applications*, 1st ed. Wiley, 2014. [Online]. Available: https://www.wiley.com/en-us/Algorithm+Design+and+Applications-p-9781118335918

[14] A. Horni, K. Nagel, and K. W. Axhausen, *The Multi-Agent Transport Simulation MATSim*. London: Ubiquity Press, 2016. [Online]. Available: https://doi.org/10.5334/baw

[15] M. Rieser, D. Métrailler, and J. Lieberherr, "Adding realism and efficiency to public transportation in MATSim," in *Proceedings of the 18th Swiss Transport Research Conference (STRC)*, Monte Verità, Ascona, May 2018. [Online]. Available: https://www.strc.ch/2018/Metrailler_Lieberherr.pdf

[16] OpenJDK, "Jep 454: Foreign function & memory api (second preview)," https://openjdk.org/jeps/454, 2023, retrieved September 17, 2024.

[17] "Project lombok: Cleaner, faster java," https://projectlombok.org, n.d., retrieved May 25, 2024.

A Guidelines

A.1 Code Guidelines

A.1.1 General

- Always use the formatter defined per project before committing.
- Restrict visibility strictly; only allow access from outside the package if absolutely necessary.
- Program to interfaces rather than implementations.
- Follow the SOLID principles to structure the code (classes).
- Adhere to the DRY (Don't Repeat Yourself) principle.
- Avoid magic numbers or literals; use constants instead.
- The fail-fast concept is preferred. Given the complexity of the project, failing quickly allows for early identification and resolution of issues, ensuring a more efficient development process.

A.1.2 Java

- Use the formatter defined for IntelliJ.
- Utilize Maven as the build tool and for dependency management.
- Avoid boilerplate code by using Lombok [17] to generate getters, setters, and constructors.
- Add spaces after class definitions and before return statements if a method has more than three statements. Use spaces to clarify blocks that belong together (e.g., before if-control flow statements).
- Group imports into packages after five classes from the same package.
- Follow naming conventions: `camelCase` for methods, `UpperCamelCase` for classes, and `ALL_UPPER_CASE` for static constants.
- Do not use `Optional<>` for parameters; it is allowed for return types and internal usage.

Code Example

```
public class ExampleService {

    private static final int MAX_ATTEMPTS = 5;
    private final ExampleRepository repository;

    public ExampleService(ExampleRepository repository) {
        this.repository = repository;
    }

    public Optional<Example> findById(String id) {
```

```
        return repository.findById(id);
    }
}
```

Documentation

- Document only non-obvious public members with Javadoc (e.g., do not document getters or setters).
- Ensure non-obvious public members are well documented.
- Avoid comments in code, except when the code or case structure is complex. If commenting, write clear and concise comments.

Testing

- Use JUnit 5 for testing. AssertJ and Mockito are allowed.
- The Surefire plugin runs unit tests (`ExampleTest`), and the Maven Failsafe plugin runs integration tests (`ExampleIT`). Follow the naming convention with the postfix `Test` and `IT`.
- Use descriptive names for test cases: `should...`, `shouldNot...`, `shouldThrow...`. The character _ is allowed in test case names to differentiate test cases.
- Use nested test classes to group thematically related cases together.
- If tests are complex in setup, use the test builder pattern and a Jupiter extension to reuse it.

Test Example

```
public class ExampleServiceTest {

    @Test
    void shouldReturnExampleWhenIdExists() {
        // Arrange
        ExampleRepository repository = mock(ExampleRepository.class);
        ExampleService service = new ExampleService(repository);
        when(repository.findById("1"))
                .thenReturn(Optional.of(new Example("1")));

        // Act
        Optional<Example> result = service.findById("1");

        // Assert
        assertThat(result).isPresent();
    }
}
```

A.1.3 C++

The following section outlines the C++ coding guideline focusing on formatting, naming conventions, and testing practices. The code example demonstrates how to implement and test a simple `Example` class.

Formatting

Always adhere to the project's defined code formatting. You should use the `.clang-format` file to enforce consistent styling across the codebase.

- Use the formatter defined in the `.clang-format` file.

Example of a `.clang-format` configuration:

```
Language: Cpp
AccessModifierOffset: -2
AlignAfterOpenBracket: Align
AlignConsecutiveAssignments: false
AlignConsecutiveDeclarations: false
AlignEscapedNewlines: Right
AlignOperands: true
BraceWrapping:
  AfterClass: true
  AfterControlStatement: true
  AfterEnum: true
  AfterFunction: false
  AfterNamespace: false
  AfterObjCDeclaration: false
...
```

Naming Conventions

`camelCase` for methods, `UpperCamelCase` for classes, and `ALL_UPPER_CASE` for static constants.

Example C++ implementation:

```
#include <string>
#include <stdexcept>

class Example {
public:
    explicit Example(const std::string& id)
    : id(id) {
     if (this->id.empty()) {
            throw std::invalid_argument("id must not be empty");
        }
    }
    const std::string& getId() const { return id; }
private:
    std::string id;
};
```

Testing

For testing, we use Google Test.

- Test cases should have descriptive names, starting with actions.
- Use the `TEST()` macro to define and name a test function.
- When using a fixture, use `TEST_F()` instead of `TEST()` as it allows you to access objects and subroutines in the test fixture.
- Avoid underscores (_) in test case names, unless necessary for differentiation.
- If possible, create free methods; if necessary, create classes (fixtures).

Example test code using Google Test:

```
#include <gtest/gtest.h>
#include "example.h"
```

```
// Test fixture for Example class
class ExampleTest : public ::testing::Test {
protected:
    void SetUp() override {}
    void TearDown() override {}
};

TEST_F(ExampleTest, shouldReturnIdWhenConstructedWithValidId) {
    // Arrange
    std::string validId = "valid_id";
    // Act
    Example example(validId);
    // Assert
    EXPECT_EQ(example.getId(), validId);
}

TEST_F(ExampleTest, shouldThrowWhenConstructedWithEmptyId) {
    // Arrange
    std::string emptyId = "";
    // Act & Assert
    EXPECT_THROW(Example example(emptyId), std::invalid_argument);
}
```

A.1.4 Python

Python code written for this module should follow the Google Python style guide `https://google.github.io/styleguide/pyguide.html` whenever possible. All modules and public functions must include a docstring as further described in `https://google.github.io/styleguide/pyguide.html#38-comments-and-docstrings`.

Further:

- Format code with Black and isort.
- Follow PEP8 guidelines.
- Use Poetry as the build tool.
- Follow naming conventions: `snake_case` for methods, `UpperCamelCase` for classes, and `ALL_UPPER_CASE` for constants.

Code Example

```
class ExampleService:

    MAX_ATTEMPTS = 5

    def __init__(self, repository):
        self._repository = repository

    def find_by_id(self, id: str) -> Optional[Example]:
        return self._repository.find_by_id(id)
```

A.2 Git Guidelines

A.2.1 Commit Messages

Commit message format explained by a sample commit message:

```
Capitalized, short (50 chars or less) summary

More detailed explanatory text, if necessary. Wrap it to about 72 characters
or so. In some contexts, the first line is treated as the subject of an email
and the rest of the text as the body. The blank line separating the summary
from the body is critical (unless you omit the body entirely); tools like rebase
can get confused if you run the two together.

Write your commit message in the imperative: "Fix bug" and not "Fixed bug" or
"Fixes bug." This convention matches up with commit messages generated by
commands like git merge and git revert.

Further paragraphs come after blank lines.
- Bullet points are okay, too
- Typically a hyphen or asterisk is used for the bullet followed by a single space,
    with blank lines in between, but conventions vary here.
- Use a hanging indent
```

Source: `https://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html`

The summary line must be formatted as follows:

```
{TYPE}: {ISSUE} - {SUMMARY}.

{ISSUE}: Id of related Jira Issue. Example: NAV-2.
{SUMMARY}: See previous section
{TYPE}: Commit type, must be any of the following:
```

- **ENH**: New functionality / Feature Implementation (add, improve, or remove functionality; changes application behavior)
- **FIX**: Fixing something that does not work correctly
- **STYLE**: Minor cosmetic changes to code (e.g., rename variables for clarity) which do not change the behavior of the code
- **REFACTOR**: Improve code organization or implementation for maintainability purposes
- **DOC**: Documentation only changes (source code or user documentation)
- **ORG**: Organizational changes, e.g., adding `.gitignore` file or `.vscode/settings.json`
- **TEST**: Adding or changing a test; does not affect application behavior

Example:

```
ENH: NAV-2 - Add option to sort tasklist by date
```

A.2.2 Reviews

Reviewing typically happens in two stages, during pull requests and when signing off the Jira issue.

Review & Approval Order

- Developer creates a Pull Request (PR)
- Reviewer reviews PR, performs testing, proposes changes, etc.
- Once good, reviewer approves the PR
- Reviewer closes the Jira issue
- Reviewer merges the PR

A.2.3 Branching Strategy

We are using trunk-based development as shown in Figure A.1.

Key Principles of Trunk-Based Development

1. **Single Source of Truth**: Developers commit directly to the trunk/main branch.
2. **Frequent Commits**: Changes are integrated multiple times a day.
3. **Short-Lived Branches**: Feature branches, if used, are short-lived (a few hours to a couple of days).
4. **Automated Testing**: Continuous integration systems run tests on every commit to catch issues early.
5. **Feature Flags**: Incomplete features are integrated into the trunk but are hidden behind feature flags.

Benefits

- Faster feedback and resolution of integration issues.
- Simplified codebase management.
- Easier to maintain a releasable state.
- Encourages collaborative development and collective code ownership.

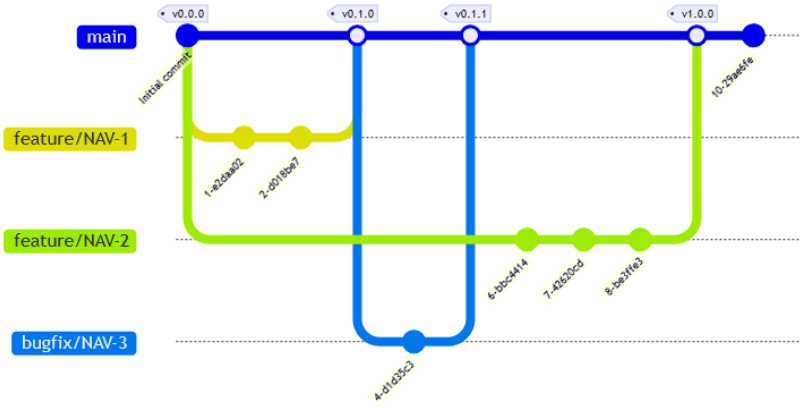

Figure A.1: Trunk-based development workflow.

A.3 Release Guidelines

A.3.1 Semantic Versioning

This project adheres to Semantic Versioning. We use a three-part version numbering system: `MAJOR.MINOR.PATCH`. Here is what each part represents:

- **MAJOR**: When we make incompatible API changes.
- **MINOR**: When we add functionality in a backwards compatible manner.
- **PATCH**: When we make backwards compatible bug fixes.

Version tags are prefixed with a `v`, e.g., `v3.1.4`.

A.3.2 Release Process

To ensure a smooth and consistent release process, follow these steps:

1. **Release Branch:**
   Create a new release branch from `main` with the semantic version number to be published:

   ```
   git checkout -b release/vX.X.X
   ```

2. **Version and Documentation:**
   Set the correct version number in the version file (`pom.xml`, `CMakeLists.txt`, `package.json`, `poetry.toml`, etc.). Update any other necessary documentation.

3. **Pull Request:**
   Submit a pull request with the new version number and documentation changes. Include a release description to be used later in the release. Since we are generating the changelog automatically, keep it concise and mention the most important aspects.

4. **Publish:**
   Upon review and merging of the pull request, use the GitHub web interface to create a new release named `<REPOSITORY-NAME> <X.X.X>` (e.g., `public-transit-service 0.2.1`). Set the version tag as `vX.X.X` and automatically generate the changelog for the release. Add the release description from the PR comment and hit publish, which triggers the appropriate GitHub actions (e.g., Maven or Docker Publish).

B Time Tracking

Throughout the project, each team member diligently logged a minimum of 375 hours, ensuring balanced contributions to different aspects of the work. Figure B.1 illustrates the total hours logged per developer across the project timeline.

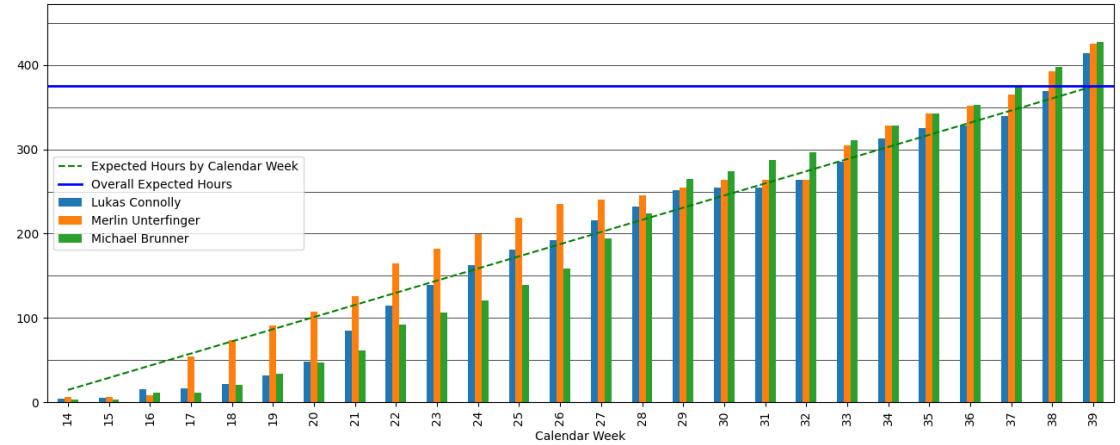

Figure B.1: Total time logged per developer by calendar week.

While the total hours were evenly distributed, the focus of each developer varied across different epics. As shown in Figure B.2, Michael Brunner dedicated a substantial portion of his time to the CXX RAPTOR epic, which involved the C++ implementation of the RAPTOR algorithm. Meanwhile, Merlin Unterfinger and Lukas Connolly contributed equally to epics focused on the Java implementation of the GTFS Schedule, the RAPTOR Module, and the integration service.

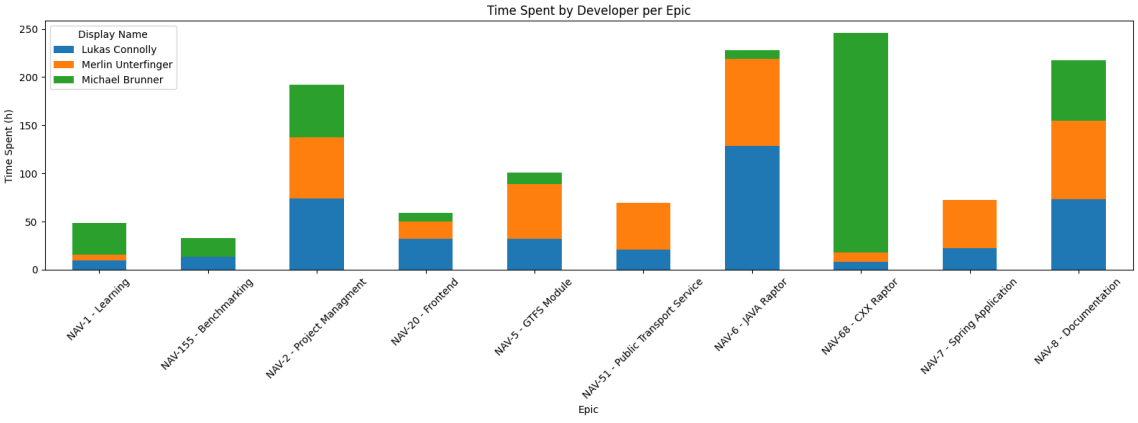

Figure B.2: Time allocation per developer by epic.

Administrative tasks, such as meetings, project documentation, and general coordination, were shared

equitably among all team members. Figure B.4 provides a detailed breakdown of the time spent on various epics, highlighting the week-to-week efforts across different components of the project.

Time allocation across the various epics showed significant variation, as illustrated in Figure B.3. The most resource-intensive tasks were the development of the RAPTOR Module, both in C++ and Java, which each accounted for nearly 20% of the total project hours. Documentation and administrative tasks also consumed a substantial portion of time both adding up to more than 15% of project time each. In contrast, simpler modules like the GTFS Schedule and the integration service required considerably fewer hours. As expected, the frontend tasks involved minimal effort, with the bulk of work concentrated on backend services.

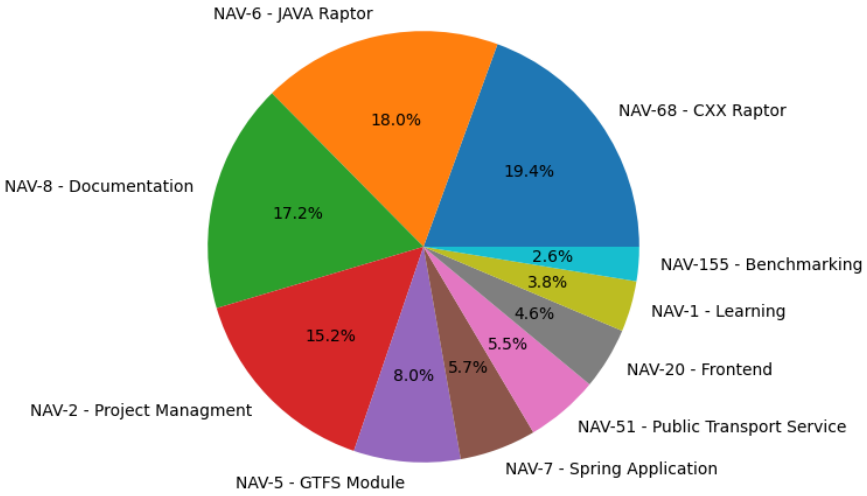

Figure B.3: Proportional time spent per epic.

The focus on epics evolved over time. The initial weeks were primarily dedicated to learning, domain analysis, and establishing the basic project structure and documentation. Once this foundation was set, attention shifted to the GTFS module, which served as a critical prerequisite for the subsequent RAPTOR module. By calendar week 22, with both modules in their initial working state, the implementation of the integration service began, leading to a significant spike in logged work hours. Afterward, the focus returned to the RAPTOR module, where additional features were developed, and the code underwent refactoring. In the final stages of the project, efforts were directed towards re-implementing the RAPTOR module in C++, alongside finalizing the project documentation, as illustrated in Figure B.4.

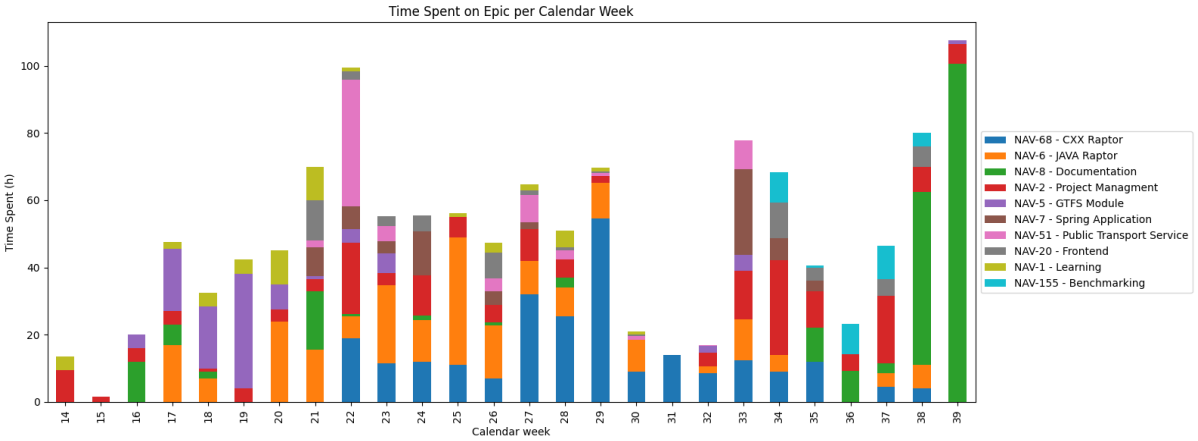

Figure B.4: Time spent on epics per week throughout the project.

C Requirements

This appendix outlines the requirements for the Naviqore public transit service and its User Interface (UI). The requirements are categorized into *must-have*, *should-have*, and *nice-to-have* features. The *must-have* requirements are essential and need to be met, while the *should-have* requirements are highly desirable, and the *nice-to-have* requirements are additional features that would enhance the service but are not critical.

The requirement ID is structured as: `{TYPE}-{COMPONENT}-{PRIORITY}{NUMBER}`

Requirement Types	**Components**	**Priority Levels**
UC: Use case	**SE**: Service	**M**: Must-have
NF: Non-function requirement	**SC**: Schedule	**S**: Should-have
	RO: Routing	**N**: Nice-to-have
	VW: Viewer	

C.1 Must-have Requirements

ID	Done	Description
UC-SE-M1	✓	Efficiently (<250ms) find the nearest stops (beeline distance) with a limit (max 5km) with stop information (coordinates, name, routes).
UC-SE-M2	✓	Find the next departures from a given stop with information (name, direction, if available head-sign).
UC-SE-M3	✓	Efficiently request (in mean <250ms) connections between two stops.
UC-SE-M4	✓	The minimum transfer time can be set by the user in the connection routing request. Already provided transfer times from the public transit schedule should be preferred, but can optionally be overridden by the minimum transfer time set by the user.
NF-SE-M1	✓	The complete service for Switzerland must be able to run on a normal machine (8 cores, 16GB).
NF-SE-M2	✓	The prototype implementation of the service, router, and GTFS reader is completely in Java.
NF-SE-M3	✓	The communication with the service must be stateless (caching of results is still allowed).
UC-SC-M1	✓	Use the GTFS (General Transit Feed Specification) as input for the transit schedule.
UC-SC-M2	✓	Read all required components from GTFS static schedule.
UC-SC-M3	✓	Read optional transfers from GTFS and make them available to the router.
NF-SC-M1	✓	The schedule is read into a memory-efficient structure, which takes less than 1.5GB (the size of the schedule file on disk) for all required fields of the GTFS Switzerland.
UC-RO-M1	✓	The routing algorithm RAPTOR (Round-Based Public Transit Routing) is used to find connections with the earliest arrival in the schedule, as its data structures are optimized for cache locality.

ID	Done	Description
UC-RO-M2	✓	The returned connection solutions have to be pareto-optimal (duration and number of transfers).
UC-RO-M3	✓	The integration in the performant programming language C++ with FFM API is evaluated.
NF-RO-M1	✓	At least 5 manually chosen connections from the SBB app are used to validate the solutions from the algorithm.
NF-RO-M2	✓	Efficiency of random route requests is measured in relation to the complete transit schedule of Switzerland of 2024 in a benchmark test. The results are written to a file. The file must have a date-time stamp to be able to track the progress of the development.
UC-VW-M1	✓	Simple frontend to display connections from origin to a destination stop for demo purposes.

Table C.1: Must-have requirements.

C.2 Should-have Requirements

ID	Done	Description
UC-SE-S1	✓	Update the static transit schedule at the defined interval (most often weekly) by the publisher (pull new GTFS static from URL and replace the current schedule in the service).
NF-SE-S1	✓	The public transit service should be able to process concurrent requests in parallel.
NF-SE-S2	✓	Containerized deployment ensures portability.
NF-SE-S3	✓	Open-Source release under a permissive or copy-left license.
UC-SC-S1	✗	Apply the realtime GTFS updates on the static GTFS schedule at the defined interval (most often 2 minutes) by the publisher. Since the login procedure is different for every publisher, the focus is set on Switzerland.
UC-RO-S1	✓	Implement range queries (rRAPTOR) to find the connections with the earliest arrivals in a departure time window.
UC-RO-S2	(✓)	If the effort of the C++ implementation does not justify the performance gain, the JNI integration should be terminated based on this reasoning.
UC-RO-S3	✓	The routing should support isoline requests: How far can someone travel from an origin stop given a time budget.
UC-RO-S4	✓	A generic footpath routing interface is provided, which allows for extension of the router in the future.
UC-RO-S5	✓	Default implementation of footpath routing, which approximates the footpaths between stations with a sensible factor times the beeline distance.
UC-RO-S6	✓	Connection requests should support setting the latest departure time in addition to the default case, which is the earliest arrival time.
UC-RO-S7	✓	The router should support connection requests that span multiple days (e.g., a connection combining a night bus with the first regular service of the following day).
UC-VW-S1	✓	The connection routing results should be displayed on a map or line network graph.

Table C.2: Should-have requirements.

C.3 Nice-to-have Requirements

ID	Done	Description
UC-SE-N1	✗	Implement user accounts with login, which allows saving favorite connections in a database.
NF-SE-N1	✗	Store the parsed static GTFS schedule and built RAPTOR data structures with a validity timestamp to a common storage (mounted volume or database), which is used by new instances of the service when started, instead of parsing the GTFS again.
NF-SE-N2	✓	The service should be horizontally scalable.
NF-SE-N3	(✓)	Deployment of public transit service in a cloud environment.
UC-SC-N1	✗	Read the optional fares from the GTFS static schedule and make them available to RAPTOR.
UC-RO-N1	✗	Implementation of the footpath routing interface with a graph-based routing algorithm (e.g. A* or A* Landmark), that uses a network read from OpenStreetMap data (e.g. from Geofabrik).
UC-RO-N2	✗	Supports efficient matrix route queries, and RAPTOR can potentially be optimized for this.
UC-RO-N3	✗	Implement multi-criteria queries (mcRAPTOR) to find the cheapest connections based on fares.
UC-RO-N4	✓	Supports availability and bike information in connection queries.
UC-VW-N1	(✓)	A more sophisticated user interface.
UC-VW-N1.1	✗	Explore and visualize the GTFS schedule.
UC-VW-N1.2	✗	Visualize the estimated vehicle positions from the GTFS realtime feed.
UC-VW-N1.3	✗	Display lines with departure times at selected stations.
UC-VW-N1.4	✓	Query connections using the RAPTOR router and display the results on a map.
UC-VW-N1.5	✓	Query reachability from a station within a given time budget.
UC-VW-N1.6	✗	Provide an option to visualize the functioning of the RAPTOR algorithm by visualizing the RAPTOR rounds until the target stop is reached.

Table C.3: Nice-to-have requirements.

D Deployment

This appendix provides an overview of the deployment process for the Naviqore system, including the public transit service and viewer.

D.1 Configuration

The public transit service is configured using environment variables for flexibility across development and production environments. Key variables are listed below:

Environment Variable	Description
GTFS_STATIC_URI	URL or file path for the static GTFS feed.
GTFS_STATIC_UPDATE_CRON	Cron expression for GTFS feed updates. Default: daily at 4 AM.
TRANSFER_TIME_SAME_STOP_DEFAULT	Default transfer time between same stops in seconds.
TRANSFER_TIME_BETWEEN_STOPS_MINIMUM	Minimum transfer time between different stops in seconds.
TRANSFER_TIME_ACCESS_EGRESS	Time for accessing or leaving a public transit trip in seconds.
WALKING_SEARCH_RADIUS	Walking search radius in meters for connections and transfers.
WALKING_SPEED	Average walking speed in meters per second.
WALKING_DURATION_MINIMUM	Minimum walking duration in seconds.
RAPTOR_DAYS_TO_SCAN	Number of days to include in RAPTOR scan. Default: 3.
RAPTOR_RANGE	Maximum range in seconds for range RAPTOR. Default: -1 for standard RAPTOR.

Table D.1: Key environment variables for configuring the public transit service.

In the `application.properties` file of the service, further settings related to caching, logging, and application management for the service are available.

D.2 Helm Chart

The Helm chart available at `https://github.com/naviqore/deployment` defines the Kubernetes resources for cloud deployment. Environment variables can be overridden using Helm's values mechanism.

D.3 Docker Compose

The Docker Compose setup, accessible at `https://github.com/naviqore/deployment/blob/main/docker-compose.yml`, is an alternative for local deployment. Environment variables can be configured in the Docker Compose file directly.

Declaration of Authorship

We hereby declare that

- We have independently and without external assistance completed the present thesis, except for the support explicitly mentioned in the assignment or agreed upon in writing with the supervisor.
- We have correctly cited all sources used according to common academic citation standards.
- We have not used any copyrighted materials (such as images) in an unauthorized manner.

Rapperswil, September 26, 2024

Authors:

Merlin Unterfinger 

Michael Brunner 

Lukas Connolly