

# Title of the Thesis

Subtitle if needed

Jack Jibb

School of Computing and Mathematical Sciences, University at Greenwich

2024-2025

## **Abstract**

In professional cycling, athletes are often sent to training camps in locations such as the French Alps, Andorra, Sierra Nevada, Colorado Springs, or other high altitude destinations. While a big part of this is altitude acclimatisation, many cyclists report that the biggest effect is actually the suitability of the roads for training. Long, car-devoid mountains, where athletes can just put their head down and focus on their effort means that their training quality is improved, as opposed to having to ride through small villages, stopping at intersections and being constantly vigilant of overtaking cars or traffic furniture. This thesis explores the possibility of being able to track and quantize "Training Suitability" of roads. What is proposed is an integrated framework that can semantically partition continuous trajectory data from a cyclist's GPS device, and meaningfully define each segment in the context of training suitability. Starting from raw GPS traces, we apply geometric filtering and map-matching to correct noise and align positions with a reference network. An adaptive segmentation algorithm then identifies breakpoints using curvature statistics, speed variance, and road metadata (such as road name, or speed limit), yielding segments that have a homogenous quality. Each segment is characterized by spatial, temporal, and contextual features and subsequently classified through a Wavelet Transform function that quantizes variability and changepoints in data streams. The scalability and robustness of the framework will be evaluated through a small-scale live web application.

# Contents

<b>Abstract</b>	<b>1</b>
<b>1 Introduction</b>	<b>2</b>
1.1 Background . . . . .	2
1.2 Research Objectives . . . . .	2
<b>2 Literature Review</b>	<b>4</b>
2.1 Training Suitability and Segmentation Algorithm Design and Analysis . . . . .	4
2.1.1 Research Objectives . . . . .	4
2.1.2 Algorithmic Approaches to Segmentation and Categorization . . . . .	4
2.1.3 Applications in Cyclist Route Planning . . . . .	4
2.1.4 Methodological Considerations and Assumptions . . . . .	4
2.2 Training Suitability Scoring . . . . .	4
2.2.1 Research Objectives . . . . .	4
2.2.2 Approaches to Suitability Scoring . . . . .	4
2.2.3 Applications in Training Suitability Scoring . . . . .	6
2.2.4 Methodological Considerations and Assumptions . . . . .	6
2.3 LSEPI Analysis . . . . .	7
2.3.1 Legal . . . . .	7
2.3.2 Social . . . . .	7
2.3.3 Ethical . . . . .	7
2.3.4 Political . . . . .	7
<b>3 Requirements</b>	<b>8</b>
3.1 Software Requirements Specification (SRS) . . . . .	8
3.1.1 Purpose and Scope . . . . .	8
3.1.2 Intended Audience and Reading Suggestions . . . . .	8
3.1.3 Overall Description . . . . .	8
3.1.4 Specific Requirements . . . . .	8
3.2 Standards and Compliance . . . . .	10
<b>4 Methodology</b>	<b>11</b>
<b>5 Design</b>	<b>12</b>
5.1 Scope and Purpose . . . . .	12
5.2 Architectural Design . . . . .	12

5.2.1	Architectural Views . . . . .	12
5.2.2	Design Rationale . . . . .	14
5.2.3	Design Patterns and Styles . . . . .	14
5.3	Module-Level Design . . . . .	14
5.3.1	Module Decomposition . . . . .	14
5.3.2	Module Specifications . . . . .	15
5.4	Interface Design . . . . .	15
5.4.1	User Interface . . . . .	15
5.4.2	Software Interfaces . . . . .	15
5.5	Data Design . . . . .	15
5.5.1	Data Models . . . . .	15
5.5.2	Database Schema . . . . .	15
5.5.3	Data Dictionary . . . . .	15
5.6	Standards and Compliance in Design . . . . .	15
<b>6</b>	<b>Implementation</b>	<b>16</b>
6.1	Component Breakdown . . . . .	16
6.1.1	Input Pipeline . . . . .	16
6.1.2	Segmentation Engine . . . . .	19
6.1.3	TSS Engine . . . . .	19
6.2	Testing and Results . . . . .	19
<b>7</b>	<b>Results and Conclusions</b>	<b>20</b>
<b>8</b>	<b>Further Discussions and Research Gaps</b>	<b>21</b>
<b>A</b>	<b>Appendix</b>	<b>22</b>
A.1	Further Reading . . . . .	22
A.2	Source Code . . . . .	22
A.3	File Structure . . . . .	22
A.4	Additional Documentation . . . . .	22

# List of Figures

2.1	The 7 Zone Training Model, courtesy of Indoor Cycling Association . . . . .	5
2.2	My Power Curve for August, 2024. The power curve can estimate what would be considered "hard" for an athlete at a given duration. Image courtesy of <b>intervals.icu</b> . . . . .	6
5.1	High level system architecture for the Route Segmentation and Analysis System	13

# List of Tables

6.1 Canonical extension keys and example synonyms . . . . .	18
---	----

# Chapter 1

## Introduction

Modern cycling training for professional athletes has been perfected down to a science. Since the days of Team Sky, "marginal gains" and hyper optimisation in all aspects of training, such as nutrition, periodization, strength, endurance and race calendar have been optimised to such a degree that the difference between the top echelon of professional cycling and the middle tier is less than a few percent. Pro riders have the advantage of having everything done for them when it comes to planning training, from their schedule to their locations. However, amateur riders do not have this same advantage, and as I have experienced many times, often we go out for rides, looking for new, longer routes to do training on, only to find that we end up down a country lane with blind corners, or a town with really annoying road furniture. The first step in planning a route has always been to go onto a mapping website, and draw a route, that you could later save to your GPS device to follow, or to write down turn by turn on the stem of your handlebar. While I won't mess with the sacred art of taping a piece of paper on your handlebars in the most aerodynamic way you can, I can maybe just revolutionise the plotting of routes for cyclists looking to get the most out of their training. Enter the concept of route "segments". In Strava, a Segment is a piece of road that has been given a place in their database, for athletes to compete for times on. Getting a "KOM" (King of the Mountain) on a Strava segment is a coveted title by many amateurs. In our context, a segment will be keeping track of more than just performance metrics. It will keep track of the inherent data of the road from an open source mapping software called OpenStreetMap, and all the data will be combined to give an overall "score" of the segment in the context of training suitability.

### 1.1 Background

### 1.2 Research Objectives

1. To develop a system that can take user cycling data in the form of GPX files, and provide feedback in the context of training suitability
2. To design and implement a Route Segmentation algorithm that splits a GPX file into meaningful sections by enriching a route-matched path from the data with road information from OpenStreetMap.

3. To be able to store the segments in a database, and future GPX files can be converted into arrays of these segments, by matching their segments with the database, and returning segments if there is a margin of error less than a certain threshold.
4. To implement batch processing of GPX files so users can submit large quantities of GPX data to aid in segment generation.
5. To create a two-way pipeline system that takes either GPX data, or a custom-plotted route, and feeds back a list of curated segments from the database.

## Chapter 2

# Literature Review

### 2.1 Training Suitability and Segmentation Algorithm Design and Analysis

#### 2.1.1 Research Objectives

#### 2.1.2 Algorithmic Approaches to Segmentation and Categorization

#### 2.1.3 Applications in Cyclist Route Planning

#### 2.1.4 Methodological Considerations and Assumptions

### 2.2 Training Suitability Scoring

#### 2.2.1 Research Objectives

#### 2.2.2 Approaches to Suitability Scoring

**Sharifzadeh et al. "Change Detection in Time Series Data Using Wavelet Footprints"** An interesting approach to Suitability scoring relates to Fast Fourier Transforms, and a convolution approach comes from Medhi Sharifzadeh et al, from the University of Southern California, introducing a concept known as "wavelet footprints" as a compact, multi-resolution approach to representation of spatial-temporal trajectories. Wavelet footprints are a more granular version of the Wavelet Transform, which in turn is a version of the Fourier Transform. The approach involves using Wavelets to transform a signal into the "Wavelet Domain". The smaller the wavelet, the more reactive it will be to change in the original signal, so by adjusting the size of the wavelet, the signal can be filtered to be more or less reactive to change. Wavelet footprints have an advantage over the general Wavelet transform, where they only retain the most significant components. This is done by having wavelets occur in orthogonal sets. Due to Heisenberg's Uncertainty Principle, it is impossible to perfectly describe both the frequency content of a signal and the location in time of the signal. Time-domain and Frequency-Domain analysis in this regard are at opposite ends of the spectrum, but the Wavelet Domain sits in the middle, allowing for a sliding scale value, where a larger scale gives more frequency and less time resolution, while smaller scales give less frequency, and more time resolution.



The advantage of using wavelet footprints over the Fourier Transform means that it is possible to isolate points in the signal where significant changes occur in specific metrics, or combination of metrics, allowing flexibility in choosing what conditions must be met to enact a segmentation.

**Indoor Cycling Association: "How Much Time in the Red Zone?"** This article details a breakdown of the 7-training-zone model, which provide a good template for cutoff times for tuning Wavelet Footprints for analysing the GPX signal. The general consensus is having 5 zones is a good compromise between continuous training definitions (specific power numbers) and binary (hard or easy). While the article describes 7 zones, Zones 1-3 all fall outside of the hour range, which for the purpose of segment analysis would be fairly useless. The 5-Zone model approach has good suitability to wavelet analysis, since small scale values for a wavelet would detect short burst efforts, while larger scales will detect longer, sustained effort. Having 5 zones allows for a reduced scale array, contributing to higher performance. Here is a graphic, courtesy of the Indoor Cycling Association, that breaks down each zone. Zones 3-7 will be used for Wavelet Analysis.


 <b>Power and Heart Rate Training Zones and RPE Chart</b>							
ZONE	Name/Purpose	%FTP	%LTHR	RPE	RPE Description	How Does It Feel?	Duration
7	MAXIMAL	>150	n/a	10	Maximal, Explosive power	Gasping for air, can't say one word	5-20 seconds
6	Anaerobic Capacity	121-150%	n/a	9	Very, very hard	Breathless, ragged breathing, talking impossible. Severe sensation of leg effort.	30 seconds to 3 minutes
5	VO2 Max	106-120%	>106%	8	Very hard	Cannot talk, laboured breathing. Strong sensation of leg effort	3-8 minutes
4	Lactate Threshold	91-105%	95-105%	6-7	Hard	Deep forced breathing, but still sustainable. Moderate to greater sensation of leg effort.	10-60 minutes
					Moderately hard	Deep breathing; talking is very challenging	
3	Tempo	76-90%	84-94%	4-5	Somewhat hard	Heavier but rhythmic breathing, greater sensation of leg effort.	1-3 hours
					Moderate	Talking becomes uncomfortable.	
2	Aerobic Endurance	56-75%	69-83%	2-3	Easy	Light rhythmic breathing. Can maintain for hours.	Many hours
					Very easy	Can talk in complete sentences	
1	Active Recovery	<55%	<68%	<2	Very, very easy	Restful breathing; can sing.	All Day

Figure 2.1: The 7 Zone Training Model, courtesy of Indoor Cycling Association

**Sean Hurley: "Normalized Power: What It Is and How to Use It"** <https://www.trainerroad.com/blog/normalized-power-what-it-is-and-how-to-use-it/> TrainerRoad writer Sean Hurley provides a useful and concise definition of Normalized Power (NP), a metric invented by Dr. Andrew Coggan in his book *Training and Racing With a Power Meter*. NP "reflects the disproportionate metabolic cost of riding at high intensity, by weighting hard efforts and deemphasizing periods of easy spinning", according to Dr Coggan. Essentially, NP approximates what power a rider could have put out for the same effort, if their effort was steady-state. While it is not a completely accurate metric for effort, it is a really good indication of power variability, and as such is useful in determining the type of effort of a segment. The algorithm for determining NP is as follows:

1. Calculate a rolling 30 second average power for the duration

2. Raise each rolling average value to the fourth power.
3. Determine the average of all the rolling values
4. Take the fourth root.

For an input power signal,  $P(t)$  over interval  $(0, N)$ , the formula for Normalized Power (NP) is:

$$\text{NP} = \left( \frac{1}{N} \sum_{i=1}^N (\overline{P}_r(i))^4 \right)^{1/4}$$

where  $\overline{P}_r$  is the 30 second rolling average power starting at point  $i$  in the power data array.

### 2.2.3 Applications in Training Suitability Scoring

### 2.2.4 Methodological Considerations and Assumptions

- **Power:** All cyclists have different power thresholds; in other words, two cyclists may be going the same speed up a climb, but one may be going all out, and another just going easy. They may not even have the same power output. The one going easy could have a higher power output than the person going all out, depending on their weight. As such absolute power is a bad reference point for training suitability. Rather, we need to focus on power variation, as well as power curve. A power curve is the integration of all power numbers over a duration of exercise, plotted average power in watts on the y-axis, and max sustained duration for that average power on the x-axis. The graph tends to look like an exponential decay function. An example of my own personal cumulative power curve in 2024 is shown here:

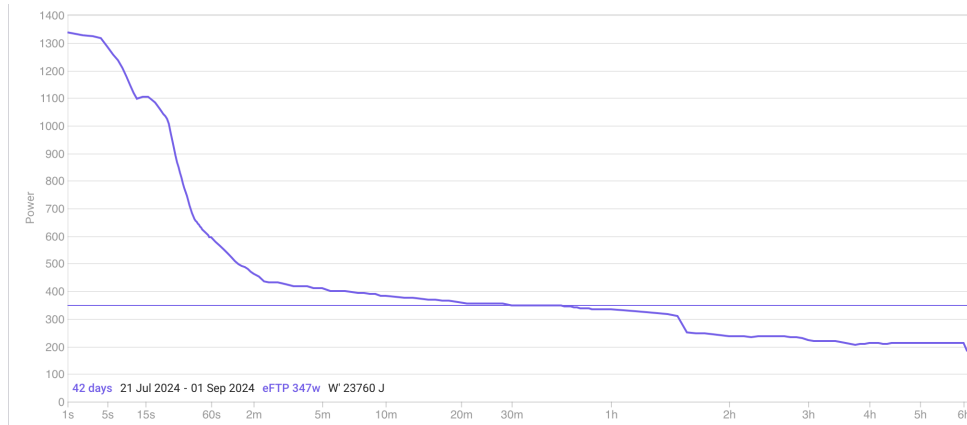


Figure 2.2: My Power Curve for August, 2024. The power curve can estimate what would be considered "hard" for an athlete at a given duration. Image courtesy of **intervals.icu**

- **Route Terrain:** Hilly or technical terrain (such as lots of sharp corners or non-pavement roads) can significantly affect how good a route is for training, but in different ways. Hilly terrain could be really good for consistent efforts, if the gradient is sustained, but if there are a lot of short, steep climbs, it would be harder to maintain any consistency in effort. Likewise, gravel or dirt roads could be good for endurance if they are consistent, but throw

in some sharp corners, and suddenly you have to brake a lot more, accelerate, and even focus on balancing more, which can induce more fatigue. Terrain is probably the most important metric that is intrinsic to the route itself.

- **Safety:** Safety is obviously very important in general when cycling, but it also plays a big part in performance. Having to focus on keeping safe on the road often means being ready or having to brake or slow down to avoid getting into dangerous situations. If a road can be considered "safe" (think long straight bike paths, or straight roads with very little traffic), then the athlete is free to focus more on their effort. To consider safety is a complicated problem, since there are many aspects. One method to consider could be to come up with a points system, and apply danger points as a suitability metric. Every "unsafe" property of a road can add danger points to the segment, and a subjective scoring system could be put in place initially, and in the future, a machine learning approach could be used.

## 2.3 LSEPI Analysis

### 2.3.1 Legal

### 2.3.2 Social

### 2.3.3 Ethical

### 2.3.4 Political

## Chapter 3

# Requirements

### 3.1 Software Requirements Specification (SRS)

#### 3.1.1 Purpose and Scope

#### 3.1.2 Intended Audience and Reading Suggestions

#### 3.1.3 Overall Description

##### Product Perspective

##### Product Functions

##### User Characteristics

The typical user will be of a reasonable proficiency with other online mapping softwares, such as route creation in Strava or MapMyRide. This software is mostly proof of concept, so usability considerations will be considered less important over functionality.

##### Operating Environment

##### Design and Implementation Constraints

##### Assumptions and Dependencies

#### 3.1.4 Specific Requirements

##### External Interface Requirements

**User Interfaces** Description of UI requirements.

The user interface only requires two features:

1. The ability to upload GPX files to the server
2. The ability to map a route via any interactive map service (Mapbox, Leaflet.js or Folium.py).

**Software Interfaces** APIs and protocols.

The system will link the front and back ends with a RESTful API. Also it is important to choose a front end framework that supports uploading multiple relatively large files (between 5-10MB

each). Since a lot of the application is written in Python, Flask is a good option for this. Other API connections will be implemented to communicate with the Segmentation and TSS engines. This allows them to be hosted on separate servers in the future, to give them more processing power.

### Functional Requirements

The functional requirements of the system are as follows:

- FR1: The system must accept multiple GPX files as input via a file loading system
- FR2: The system shall allow users to draw a route using an interactive map UI
- FR3: The system shall align raw GPS Tracks to the road network for consistency.
- FR4: Upon receiving a matched route, the system shall identify route segments that are related by road features.
- FR5: The engine shall perform a two-pass segmentation. One to match the route with existing segments, and one to find new segments.
- FR6: Upon uploading a GPX file, the system shall check if a matching segment exists in a database (within spatial and directional tolerances). if found, the segment's hit count is incremented and Training Suitability Score values are added to the database's list.
- FR7: If no existing segment is found for a segment generated by the Segmentation engine, and the generated segment is sufficiently significant, the system shall create a new segment entry in the database.
- FR8: When the user draws a route in the interactive map, and queries the system without uploading a GPX, the system shall send the route to the Segmentation engine, and identify known segments along the route, along with associated TSS data. It shall **not** create new segments during the query.
- FR9: The system shall evaluate each segment's Training Suitability Score whenever a new GPX file is uploaded to the input pipeline by sending the segments to the TSS Engine.

### Non-functional Requirements

- NF1: **Accuracy** -
  - Every GPX file must be converted into a GeoJson file that has less than 4% difference in distance
  - For a segment to be considered "matching", it must fall within a similarity parameter of 90%.
- NF2: **Performance** - The system must be able to process an average of 1,000 GPX points per second.
- NF3: **Extensibility** - The system must be built with scalability in mind, and each major component should be able to be isolated on a separate system. All components should communicate via API calls to keep this a reality.

NF4: **Privacy** - A default culling of the first and last 500m of each GPX file will provide some privacy for users' home addresses.

- No training specific data will be stored on the database, only the Training Suitability scores, to increase the data security of users.

NF5 **Compliance** - All data regulations by the UK government must be abided by.

### Logical Database Requirements

#### Software System Attributes

##### Reliability

**Availability** As a proof-of-concept system, the availability of the system is only when required for testing and presenting.

**Security** Security isn't as important for the proof of concept, but a lot of standard considerations can be made that relate to LAMP stack web applications.

**Maintainability** Every component of the system will be independent, and input and output formats are documented in the documentation in A.

**Portability** The whole project will be made available via GitHub, and possibly a dockerized version of the system will be developed in the future for portability.

## 3.2 Standards and Compliance

## Chapter 4

# Methodology

# Chapter 5

## Design

### 5.1 Scope and Purpose

The following section is a comprehensive design plan for the Route Segmentation and Analysis System. The architecture will be described in compliance with ISO/IEC/IEEE 42010:2011 [?], which standardizes how systems and software architectures are documented. The System allows users to upload GPX files, or draw routes via an HTML interface. The uploaded routes are processed to identify meaningful **segments** (portions of the route that have similar road conditions). The segments are then analysed to gather training insights. Every segment will be sent to a database, where it is cross referenced with all values to see it falls within a determined margin of error (<5%). If it is, then the matching database entry will have its count increased by 1, and the training score averaged into the running average. Users can also query the system by drawing a prospective route, and will get in return a list of any known segments along the route, along with their score. The main architectural challenge in this system is to detect and match route segments directionally (direction matters), while also considering reliability.

### 5.2 Architectural Design

The overview of the system is presented in ???. Present a high-level system architecture diagram using UML or SysML.

#### 5.2.1 Architectural Views

##### Concept View

The architectural concept of the system can be broken down into 3 layers:

- **Presentation Layer** - This is the front end; A web application where users can either upload their GPX files, or draw a route on an interactive map. This layer will communicate with the back end through HTTP API requests. This layer is mostly outside the scope of the project, since I am not a front end engineer.
- **Processing Layer** - The beef of the project, this includes the back-end of the system, and consists of input/output pipelines, and several processing engines. It will also contain



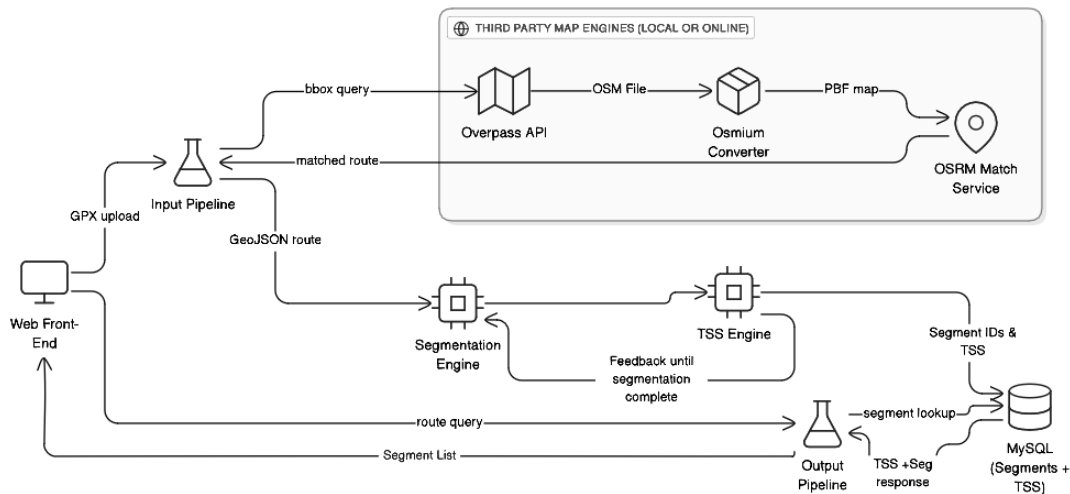


Figure 5.1: High level system architecture for the Route Segmentation and Analysis System

several controllers, written in Python, to coordinate the pipelines and to send information to and from the different engines.

- **Data Layer** - This layer is represented by a SQL database that stores permanent information about route segments. External APIs also fall into this category, such as the Overpass API to fetch map data from online. These components provide necessary data but are mostly abstracted away, in the processing layer.

## Functional View

The functional view includes key modules and their responsibilities.

- **Front end Web App:** Serves HTTP pages and gets information from the user.
- **Front End Controller:** Takes user information and routes it to correct API endpoint.
- **Input Pipeline Orchestrator:** Controller that handles input information routing, sending the GPX files to the pipeline, and pulling the resulting GeoJSON objects out.
- **Engine Orchestrator:** Takes routes, passes them to Segmentation algorithm, and handles the "two pass" system between Segmentation and TSS Engines.
- **DTO Orchestrator:** Formats the resulting segments, into the correct Data Object and passes it to SQL Database
- **Output Pipeline Orchestrator:** Receives the Route data from the Input Pipeline Orchestrator, validates it and sends it to Segmentation Engine. Gets segment list, and routes it back to front end in GeoJson format.
- **Segmentation Engine:** Written in C++, the engine is a specialized service that performs route segmentation and matching logic.
- **TSS Engine:** Takes a list of segments and the input GPX data, and assigns scores based on the training data that happens during each segment

- **SQL Database:** Stores all persistent segments, along with a hit count, and a list of TSS objects that have been calculated on matching segments.
- **Osmium Engine:** A docker service that runs a .osm to .pbf conversion and merging process. Used in the input pipeline
- **OSRM Matching Engine:** Another docker service that takes small GeoJson chunks, and matches them to a map network.

### Physical View

Hardware deployment and network design.

### Information View

Data Flow Diagram, and Database Schema diagrams

### Behavioral View

Use case and sequence diagrams for the system

## 5.2.2 Design Rationale

The choice to go with a **Pipeline Architecture** for processing data is motivated by the main use case. Users who are looking to find a route to ride are not interested in uploading GPX, but conversely, users who are done with their ride aren't looking to find a route. So cyclists can be considered to be in one of two states when wishing to interact with the system, either wanting a route, or wanting to upload their data. The two pipelines reflect this with one being a GPX upload-only pipeline, and the other one is a visualisation pipeline that shows users what their mapped route's suitability looks like. A simple LAMP-like stack approach is a simple, but effective pattern that can implement this project. As of right now, this is a small scale experiment to prove the concept. If this application becomes popular, there will be several avenues to explore for scalability, such as deploying the system to expandable cloud services such as AWS or Azure. This is beyond the scope of the project however.

## 5.2.3 Design Patterns and Styles

Identify and describe any applied patterns (e.g., MVC, layered) and coding conventions.

# 5.3 Module-Level Design

??

## 5.3.1 Module Decomposition

List all system modules, their interfaces, and dependencies. Include a module dependency diagram.

### 5.3.2 Module Specifications

For each module:

**Name:** Purpose and description.

**Interfaces:** Inputs, outputs, and protocols.

**Behavior:** Algorithmic overview or pseudocode.

**Dependencies:** Internal and external module links.

## 5.4 Interface Design

### 5.4.1 User Interface

The user interface for the project scope will just be a basic file submission system, and an interactive mapping software that can output GeoJSON

### 5.4.2 Software Interfaces

The front end will send GeoJSON objects to the backend via POST. GPX files can also be sent via POST with `Content-Type:application/gpx+xml`. The back-end will also respond with the same GeoJSON objects back to the front-end, as well as sending segment list via POST. REST APIs are also used to interface with third party tools such as the OSRM route matching engine, and the Overpass API.

## 5.5 Data Design

### 5.5.1 Data Models

ERD showing data entities and relationships.

### 5.5.2 Database Schema

Detailed table definitions, keys, indexes, and normalization rules.

### 5.5.3 Data Dictionary

Definitions and formats for all data elements used in the system.

## 5.6 Standards and Compliance in Design

List all applicable ISO/IEC, IEEE, and domain-specific standards adhered to (e.g., ISO/IEC 27001 for security).

## Chapter 6

# Implementation

### 6.1 Component Breakdown

This section provides a personal overview of how I went about implementing each component of the system. Rather than a dry description of what each module does (see Chapter 4 for those details), I recount the major steps I took and the rationale behind them.

#### 6.1.1 Input Pipeline

I began by experimenting with small GPX files in order to understand the XML structure and isolate coordinate pairs. My first prototype simply read a GPX and produced a list of latitude–longitude tuples. Once I was comfortable with the data, I wrote a Python module to fetch map data from the Overpass API; this involved generating a bounding box from the GPX extents and sending the appropriate HTTP requests.

To support more complex workflows I created a Python virtual environment, installed the required dependencies from `requirements.txt` and started wrapping individual steps into scripts. After downloading raw `.osm` files I realised they needed to be converted to Protocolbuffer (`.pb`) format for OSRM, so I built a Docker image containing Osmium and wrote a docker-compose configuration to run the conversion. Along the way I wrote helper scripts to merge multiple map tiles and orchestrated them all with a bash script called `run_pipeline.sh`.

With the map conversion automated, I spun up OSRM in its own Docker container and tuned its `profile.lua` to better reflect cycling speeds on different surfaces. This involved a lot of trial and error: I adjusted speed values for roads, gravel and tracks, and even discovered missing road types (e.g., trunks) which required adding to the profile. After several days of tweaking I achieved reliable matching on most of my test rides.

I also developed `batch_route_calc.py` to handle large GPX files. This script chunks long tracks into smaller segments, submits them to OSRM in parallel using `ThreadPoolExecutor` and then merges the matched fragments. Parallelising the API requests and file operations reduced processing times from minutes to seconds. Finally I integrated visual checks with Folium to verify that the matched routes and original GPX tracks aligned.

Towards the end of building the pipeline I experimented with the OSRM radius parameter and dynamic search windows to strike a balance between robust matching and avoiding spurious

routes. Once the matching behaved acceptably, I moved on to enriching the matched GeoJSON with OSM metadata and GPX sensor data (described in the next subsection).

### GPX Data Enrichment

After I had reliable map matching in place, I turned my attention to enriching each trackpoint with context and sensor data. I wrote two Python modules, `gpx_enrich.py` and `batch_enrich.py`, to handle this job. These scripts take the matched GeoJSON and the original GPX files, extract the sensor readings from the `<extensions>` elements, pull out standard metadata (creator, device model, track name) and normalise all of the fields using a configuration file, `config/extensions_map.json`. In practice I followed these steps:

1. **Loading the extension rules.** I designed `extensions_map.json` to define a set of canonical keys such as *hr\_bpm*, *cad\_rpm*, *power\_w*, *temp\_c*, *speed\_mps*, *alt\_m* and *grade\_pct*. Each entry lists the synonyms I've encountered in GPX files (for example, "heartrate", "hr" and "heart\_rate" all map to *hr\_bpm*) and specifies type conversions and unit transforms. When my enrichment code sees a raw GPX extension it looks up the rule, coerces the value to a number and applies any required unit conversions.
2. **Parsing and extraction.** For each GPX file I iterate through every trackpoint, recording latitude, longitude and timestamp. I also collect any extension fields present, and attach the GPX-level metadata. This stage required careful handling of optional fields because different devices embed different sets of sensors.
3. **Attaching OSM metadata.** Using the previously matched GeoJSON, I map each trackpoint to an OpenStreetMap way identifier via a nearest-node lookup. With this identifier I can fetch attributes like road type, name, speed limit and width from the pre-indexed OSM database described in Section 6.1.1.
4. **Producing enriched JSON.** For every point I emit an object that records its position and time, a `gpx_list` array of canonical field objects (each with `name`, `value` and `unit`), the `way_id`, and any associated metadata. This normalised structure became the foundation for my segmentation and training suitability scoring.

This process gave me a clean, uniform representation of each ride, ready to be fed into the segmentation engine. Table 6.1 summarises some of the canonical keys and their common synonyms defined in `extensions_map.json`. Only short phrases or keywords are included in the table to avoid overly long entries.

Table 6.1: Canonical extension keys and example synonyms

Canonical key	Example GPX synonyms
<code>hr_bpm</code>	heartrate, heart_rate, hr
<code>cad_rpm</code>	cadence, cad_ence, cad
<code>power_w</code>	power, watts, pwr
<code>temp_c</code>	temperature, temp_f (converted from °F)
<code>speed_mps</code>	speed, mph (converted to m/s), kmh
<code>alt_m</code>	altitude, ele, elevation
<code>grade_pct</code>	grade, slope, incline

### Pre-indexing OpenStreetMap Ways via SQLite

To efficiently map each trackpoint to its OpenStreetMap way, I build a mini database that indexes every way and its constituent edges in the downloaded map. The script `build_way_index.py` reads the `.pbk` map produced earlier and populates two SQLite tables:

- **ways** — one row per OSM way. Each row stores the integer `way_id` and a JSON-encoded `tags` column containing attributes such as `highway`, `name`, `surface` and `maxspeed`. The `way_id` column is a primary key to allow fast lookup.
- **edges** — one row per directed edge between two nodes of a way. Columns store the `edge_id` (primary key), starting node identifier `u`, ending node identifier `v`, associated `way_id`, `direction` (1 for forward and -1 for reverse), and the geodesic length of the edge in metres. Indexes on `u` and `v` accelerate nearest-neighbour queries.

During enrichment, the coordinates of each trackpoint are snapped to the nearest OSM node using OSMnx. The resulting node identifier is then used to look up the corresponding `way_id` via the `edges` table. Because the tables are indexed, this lookup is very fast even for large maps. Table ?? shows an example of a few rows from the `ways` and `edges` tables in our prototype database. Note that tags are abbreviated for brevity. This indexed SQLite database allows

Table	Columns (sample values)	Description
<b>ways</b>	(101, {"highway": "residential", "name": "Main Street"})	Residential road named Main Street
	(102, {"highway": "footway", "surface": "paved"})	Paved footway
<b>edges</b>	(1, 2, 101, 1, 50.0)	Edge from node 1 to 2 on way 101, length 50 m
	(2, 3, 101, 1, 45.0)	Edge from node 2 to 3 on way 101, length 45 m
	(3, 2, 101, -1, 45.0)	Reverse edge from 3 back to 2 on way 101

the enrichment pipeline to attach road metadata to each trackpoint in milliseconds, which is essential for scaling batch processing to thousands of GPX files.

### **Data Stream**

The data going in was the GPX file, and the settings.yml that controlled parameters such as chunk size (of the gps chunks), and dynamic radius window, which determined how wide of a window to check for variance to adjust the match search radius.

#### **6.1.2 Segmentation Engine**

#### **6.1.3 TSS Engine**

### **6.2 Testing and Results**

## Chapter 7

# Results and Conclusions



## Chapter 8

# Further Discussions and Research Gaps

## Appendix A

# Appendix

A.1 Further Reading

A.2 Source Code

A.3 File Structure

A.4 Additional Documentation