

# Title of the Thesis

Subtitle if needed

Jack Jibb

School of Computing and Mathematical Sciences, University at Greenwich

2024-2025

## Abstract

Training quality depends not only on an athlete's physiology but also on the road environment: long, uninterrupted climbs or low-traffic corridors enable sustained efforts, whereas junctions, stops and heavy traffic fragment work and reduce effectiveness. This thesis investigates whether the training suitability of roads can be quantified and used to help cyclists plan better sessions. I present an end-to-end framework that transforms raw GPS (GPX) traces into a database of road segments annotated for training suitability. The pipeline (i) cleans and map-matches traces to OpenStreetMap using OSRM [1], (ii) enriches the matched geometry with road metadata (e.g., highway class, speed limit, junction density), and (iii) performs adaptive segmentation using curvature and speed-variance cues with topology-aware guards. Each segment is then described by spatial-temporal features and scored with a Training Suitability Score (TSS) derived from road context and observed ride dynamics. The score favours long, low-interruption, consistent-grade sections suited to threshold or aerobic work, and penalises frequent stops, sharp curvature, or dense junctions. The system supports batch ingestion of user files, segment deduplication and matching against an existing segment set, and a minimal web application to browse routes, visualise segments, and collect feedback. Together, these components demonstrate a practical route-planning aid that prioritises the quality of training stimulus, not just performance ranking.

# Contents

<b>Abstract</b>	<b>1</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Background . . . . .	1
1.2 Research Objectives . . . . .	1
1.2.1 Research Questions and Hypotheses . . . . .	3
1.2.2 Planned Evaluation (Datasets & Metrics) . . . . .	4
<b>2 Literature Review</b>	<b>5</b>
2.1 Training Suitability and Segmentation Algorithm Design and Analysis . . . . .	5
2.1.1 Research Objectives . . . . .	5
2.1.2 Algorithmic Approaches to Segmentation and Categorization . . . . .	5
2.1.3 Methodological Considerations and Assumptions . . . . .	5
2.1.4 Pulling Map Data . . . . .	6
2.2 Training Suitability Scoring . . . . .	7
2.2.1 Research Objectives . . . . .	7
2.2.2 Approaches to Suitability Scoring . . . . .	9
2.2.3 Methodological Considerations and Assumptions . . . . .	11
2.3 LSEPI Analysis . . . . .	12
2.3.1 Legal . . . . .	12
2.3.2 Social . . . . .	12
2.3.3 Ethical . . . . .	12
2.3.4 Political . . . . .	12
<b>3 Requirements</b>	<b>13</b>
3.1 Software Requirements Specification (SRS) . . . . .	13
3.1.1 Purpose and Scope . . . . .	13
3.1.2 Intended Audience and Reading Suggestions . . . . .	13
3.1.3 Overall Description . . . . .	13
3.1.4 Specific Requirements . . . . .	15
3.2 Standards and Compliance . . . . .	18
<b>4 Implementation</b>	<b>19</b>

<b>5</b>	<b>Design</b>	<b>20</b>
5.1	Scope and Purpose . . . . .	20
5.2	Architectural Design . . . . .	20
5.2.1	Architectural Views . . . . .	20
5.2.2	Design Rationale . . . . .	22
5.2.3	Design Patterns and Styles . . . . .	23
5.3	Module-Level Design . . . . .	23
5.3.1	GPX-to-Enriched-JSON Pipeline Modules . . . . .	23
5.4	Interface Design . . . . .	25
5.4.1	User Interface . . . . .	25
5.4.2	Software Interfaces . . . . .	25
5.5	Data Design . . . . .	25
5.5.1	Data Models . . . . .	25
5.5.2	Database Schema . . . . .	25
5.5.3	Data Dictionary . . . . .	25
5.6	Standards and Compliance in Design . . . . .	25
<b>6</b>	<b>Methodology</b>	<b>26</b>
6.1	Component Breakdown . . . . .	26
6.1.1	Input Pipeline . . . . .	26
6.1.2	Segmentation Engine . . . . .	29
6.1.3	TSS Engine . . . . .	31
6.2	Testing and Results . . . . .	31
<b>7</b>	<b>Results and Conclusions</b>	<b>32</b>
<b>8</b>	<b>Further Discussions and Research Gaps</b>	<b>33</b>
<b>A</b>	<b>Further Reading</b>	<b>34</b>
<b>B</b>	<b>Code Listings and File Structures</b>	<b>35</b>
<b>C</b>	<b>Directory Structure</b>	<b>39</b>
<b>D</b>	<b>Application Documentation</b>	<b>40</b>

# List of Figures

2.1	The 7 Zone Training Model, courtesy of Indoor Cycling Association . . . . .	10
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>	12
5.1	High level system architecture for the Route Segmentation and Analysis System . .	21

# List of Tables

6.1	Canonical extension keys and example synonyms . . . . .	28
-----	---------------------------------------------------------	----

# Chapter 1

## Introduction

Highquality cycling training depends on being able to execute the intended effort without interruption. Professional riders often train on long climbs and lowtraffic roads chosen for this purpose; most amateurs, by contrast, plan routes with generalpurpose mapping tools and only discover midride that junctions, blind corners or traffic furniture force repeated stops and surges. The result is fragmented work and reduced training effectiveness. This dissertation addresses that gap. I investigate whether the suitability of a road for structured training can be quantified from ride data and open mapping sources so that routes can be planned around segments that support sustained efforts. I refer to this property as *training suitability*. The proposed system ingests raw GPX traces, performs mapmatching against OpenStreetMap using OSRM, enriches the resulting paths with road metadata (e.g., highway class, speed limits and junction density), and then applies an adaptive segmentation procedure. Segments are identified using curvature and speedvariance cues with topologyaware guards to avoid splitting at spurious points. Each segment is described by spatial and temporal features and given a Training Suitability Score (TSS) that favours long, lowinterruption, consistentgrade sections and penalises frequent stops or sharp bends. On top of this pipeline, I implement (i) batch processing for large GPX collections, (ii) segment deduplication and matching so new routes can be expressed as sequences of known segments, and (iii) a minimal web application for exploring routes, visualising segment scores and gathering feedback. In summary, this work aims to move route planning beyond performance rankings towards the deliberate selection of roads that maximise training stimulus. The remainder of this chapter provides background and states the research objectives that guide the study.

### 1.1 Background

### 1.2 Research Objectives

This dissertation investigates whether the *training suitability* of roads can be quantified from ride data and open map sources to support route planning for structured efforts. The objectives are:

01. **Define “training suitability”.** Formalise a segment-level construct that captures the ability to execute sustained work with minimal interruptions and predictable load.
02. **Build a reproducible map-matching pipeline.** Establish a deterministic GPX  $\rightarrow$  OSM route pipeline (OSRM [1]) with monotonic streaming matching and topology-aware guard windows for leg boundaries.
03. **Develop segmentation methods.** Compare candidates for route segmentation:
  - O3a. curvature/heading and grade change-points,
  - O3b. speed-variance and stop-density cues,
  - O3c. multi-scale (wavelet [2]) change detection,
  - O3d. topology-aware guards (junction density, crossings),
  - O3e. greedy vs. dynamic-programming formulations.
04. **Engineer segment features.** Design a feature schema: length, elevation gain, mean/variance of grade, curvature, stop/junction density, speed-limit, highway class, surface/cycleway tags, and simple traffic proxies derived from OSM metadata.
05. **Specify a TSS model.** Create (i) a transparent rule-based baseline and (ii) a learned model (e.g., logistic/ordinal regression or learning-to-rank) mapping features to a scalar training suitability score.
06. **Ground-truth construction.** Derive labels from ride dynamics (e.g., proportion of time in target zone, pause frequency,
07. **Validate alignment with training intent.** Test whether TSS correlates with: (a) sustained power feasibility, (b) low interruption frequency, (c) subjective suitability ratings.
08. **Ablation and robustness.** Quantify sensitivity to GPS noise, map-matching errors, missing OSM tags, and urban vs. rural context; evaluate cross-region generalisation.
09. **Efficiency.** Measure throughput (GPX points/s) and memory footprint for streaming vs. batch; target near real-time feedback for typical rides.
010. **Segment library operations.** Deduplicate, index, and match new routes to an evolving segment set; assess match precision/recall and update policies.
011. **Human-in-the-loop feedback.** Prototype lightweight feedback (thumbs-up/down or pairwise choices) and test whether incorporating it improves ranking metrics.
012. **Reproducibility and privacy.** Ensure runs are reproducible (configs, seeds, versions) and only aggregated, per-segment stats persist (no user-identifiable ride data).

### 1.2.1 Research Questions and Hypotheses

**RQ1:**

Which segmentation criterion best aligns with actual changes in riding regime?

**H1:**

Multi-cue segmentation (curvature + stop density + grade) yields higher boundary F1 vs. single-cue methods.

**RQ2:**

Can a learned TSS outperform a rule-based baseline in ranking segments for sustained efforts?

**H2:**

A learning-to-rank model (using the feature schema) achieves higher NDCG@k and Kendalls  $\tau$  against expert/pairwise labels.

**RQ3:**

Do topology-aware guard windows reduce spurious splits near complex junctions?

**H3:**

Guarded segmentation reduces over-segmentation rate and improves segment continuity (median segment length, lower split-at-junction rate).

**RQ4:**

Does TSS correlate with feasibility of holding target power/HR on held-out rides?

**H4:**

Segment-level TSS shows significant positive correlation (Spearman  $\rho$ ) with % time-in-zone and low pause/surge counts.

**RQ5:**

How robust is the pipeline to GPS noise and incomplete OSM tags?

**H5:**

Performance degrades gracefully under noise injection (10% drop in NDCG@10 at realistic jitter) and missing-tag ablations.

**RQ6:**

Can the segment library generalise across regions?

**H6:**

Models trained in Region A retain ranking performance within 510% NDCG on Region B after simple reweighting or calibration.



### 1.2.2 Planned Evaluation (Datasets & Metrics)

- **Data:** Your GPX set (post map-matching); optional power/HR to derive time-in-zone; small expert labels or rider pairwise preferences.
- **Segmentation metrics:** Boundary precision/recall/F1 against regime-change heuristics; segment continuity (median length, split-at-junction rate).
- **Suitability metrics:** Spearman  $\rho$  with feasibility proxies; AUC/accuracy for suitable vs. unsuitable; ranking NDCG@k / MAP; calibration (ECE).
- **Robustness:** Noise sweeps (GPS jitter), missing-tag ablations, cross-region transfer; report deltas on main metrics.
- **Efficiency:** Throughput (pts/s), end-to-end latency per ride, memory peak.
- **Ablations:** Feature-group and cue ablations (remove curvature, remove stops, remove grade) to quantify contribution.

## Chapter 2

# Literature Review

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

#### 2.1.1 Research Objectives

The purpose of creating an algorithmic approach to categorizing and segmenting a GPX route is to be able to rapidly and systematically construct new routes based on the requirements of the rider's training plan. It makes sense intuitively to segment a route, so riders does not have to analyse every part of a map to determine where to send their route. Riders are specifically interested in sections of the route such as climbs, descents, busy roads, and gravel. The objective of research should be to come up with a suitable method of segmentation, and subsequent categorization of the resulting segments so riders can select ones that suit their training needs.

Intuitively, we can segment a route by the roads that it follows, when a route changes roads, we add a segment. However this can prove more difficult to formalize, as a road can have many intersections. How can we categorize a segment by road name but also have node values that connect to other road names that may or may not be part of the same segment? The following research applies to the problem of Segmentation, and how we may approach defining a route segment formally.

#### 2.1.2 Algorithmic Approaches to Segmentation and Categorization

#### 2.1.3 Methodological Considerations and Assumptions

Firstly we must obtain a GPX file. This is give from either a new or existing activity. It can be downloaded in many ways, either from common fitness websites (Strava, Garmin, MapMyRide, etc.), or directly downloaded from a GPS device. Either way, there are some standards we can take advantage of. Each GPX is an XML-style file. Here is a snipped from a GPX file used in testing the system:

```
<trk>
  <trkseg>
```

```

<trkpt lat="38.82032" lon="-104.861694">
  <time>2024-08-06T13:12:58.000Z</time>
  <ele>1884.8</ele>
  <extensions>
    <power>168</power>
    <tpx1:TrackPointExtension>
      <tpx1:atemp>20</tpx1:atemp>
      <tpx1:hr>123</tpx1:hr>
      <tpx1:cad>65</tpx1:cad>
    </tpx1:TrackPointExtension>
  </extensions>
</trkpt>
...
</trkseg>
</trk>

```

It's clear to see that it is a hierarchical tree structure, with various metadata. Not all of the data is named constantly, so it is important to consider variations in naming. This can be normalized along with any other normalization processes.

Once downloaded, they must be normalized and processed. This can be done by converting to a standardized JSON structure, where each object contains at least Latitude, Longitude, Elevation, and timing data. After this, some processing should be done to remove any erroneous values. Simply detecting 0s or NaN values in the data stream is sufficient, and the missing data can be interpolated with the average of the surrounding present values. Next, derived metadata, such as speed or heading, should be calculated mathematically using the Haversine formula and trigonometry, and added as an additional extension in the JSON file.

### 2.1.4 Pulling Map Data

A useful library for python called OSMnx exists, which allows easy and efficient pulling of API data from the OpenStreetMap database. This is what we will use to correct the GPS data, and to enrich the data for segmentation and categorization. To do this, we use the `os.graph_from_bbox()` function to pull only nearby data (within the bounding box of the route). The approach is as follows:

1. Find the largest and smallest coordinates in the route
2. Generate a bounding box of the extreme coordinates
3. Send the bounding box to a function that pulls the data from OSM

## 2.2 Training Suitability Scoring

### 2.2.1 Research Objectives

#### Construct and Scale Definition

1. **Define the construct:** Precisely define what TSS measures (segment suitability for endurance, tempo/threshold,  $\text{VO}_2\text{max}$ , anaerobic/neuromuscular work).
2. **Score scale and calibration:** Design a bounded scale (e.g., 0–100) with interpretable anchors; ensure monotonicity with increasing segment challenge.
3. **Task taxonomy:** Map segments to training intents; specify how a single segment can be suitable for multiple intents with different scores.

#### Data and Feature Engineering

1. **Core geometry features:** Length, elevation gain/loss, grade distribution (percent time in grade bins), curvature/turn density, stop density, junction density.
2. **Dynamics features:** Speed variance, acceleration peaks per km, cadence/power variability (if available).
3. **Context features:** Surface type, traffic class (if derivable), wind exposure (proxy via openness), time-of-day/week effects (optional).
4. **Directionality:** Treat opposing directions as distinct; verify feature asymmetry (e.g., climb vs. descent).
5. **Sensor fallbacks:** Specify safety fallbacks when power/HR are absent (e.g., speedgrade model, elevation-derived effort proxies).

#### Personalisation vs. Population Baseline

1. **Dual scoring:** Produce both (a) population TSS (sensor-agnostic) and (b) optional rider-adjusted TSS using FTP/HR zones when available.
2. **Normalization:** Ensure comparability across riders and devices (e.g., z-scores by region/segment length, or duration-normalized metrics).

#### Algorithm Design

1. **Model family:** Compare transparent formula-based models (weighted feature sum with calibrated bins) vs. learned models (ordinal regression / pairwise ranking).
2. **Weight learning:** Learn weights from coach labels or pairwise A more suitable than B judgments; constrain for interpretability.

3. **Multi-objective scoring:** Support per-intent heads (Endurance/Threshold/Neuromuscular) sharing a common feature backbone.
4. **Uncertainty:** Output a confidence/uncertainty score (e.g., via bootstrap or ensembling) for downstream UI.

### Validation and Evaluation

1. **Reliability:** Test consistency on repeated rides (ICC, coefficient of variation) under GPS noise and sampling-rate changes.
2. **Criterion validity:** Correlate TSS with expert/coach ratings and with proxy physiology (normalized power,
3. **Discrimination:** AUC/average precision to separate segments intended for different training zones.
4. **Calibration:** Reliability plots (expected vs. observed difficulty bins).
5. **Generalisation:** Cross-validate across geographies (urban/rural, hilly/flat) and seasons; check domain shift.

### Robustness and Edge Cases

1. **GPS/Map-matching noise:** Stress-test with perturbed trajectories; target  $\leq 5\%$  score drift.
2. **Segment length effects:** Ensure fairness across short vs. long segments (length-aware normalization).
3. **Outliers:** Downweight extreme rides (e.g., device glitches, group sprints) to protect segment baselines. Can filter out data points above certain standard deviation value or absolute ceiling (2000+ watts/200rpm/250hr, etc...).

### Operational Objectives

1. **Versioning:** Define TSS schema & model versioning.
2. **Explainability:** Provide per-segment feature attributions (e.g., SHAP) for UI tooltips (score driven by 8% average grade and high curvature).
3. **Privacy:** Use only aggregated segment features for public stats; no storage of user-identifiable ride data.

## Datasets and Ground Truth


1. **Label collection:** Design a small expert-annotated set (coach ratings / pairwise preferences) covering common segment archetypes.
2. **Benchmark split:** Publish/train/val/test splits across regions and segment types; document acquisition and preprocessing.

### 2.2.2 Approaches to Suitability Scoring

**Sharifzadeh et al. "Change Detection in Time Series Data Using Wavelet [2] 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. 2.1, courtesy of the Indoor Cycling Association, 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

Modified from Andrew Coggan Power Training Levels

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

**Sean Hurley: "Normalized Power [3]: 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:

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

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

### 2.2.3 Methodological Considerations and Assumptions

In order to classify a segment, it is necessary to know what an athlete values in training. As a cyclist, I have used my own experience to compile this list:

- **Type:** What is the categorization of the road? (A, B, Service, Motorway, Bike path)
- **Surface:** What is the surface of the road made of? (Tarmac, concrete, gravel, dirt, wood?)
- **Behaviour:** How does the road act? (Straight, gradual bends, sharp curves, switchbacks)
- **Safety:** How safe is this route? Are there a lot of accidents, is there a lot of commercial traffic?
- **Elevation:** Is this route a consistent gradient, is it flat, uphill, downhill, rolling?
- **Length:** How long is the route in metres.

Implementing a summative score that represents a segment will also benefit from taking rider metrics, such as Power, HR, Cadence, and Speed. As a cyclist or group of cyclists ride a specific segment, the better picture we get of the route, as the average values will be a good representation of the training suitability of the route.

- **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 in 2.2.
- **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),



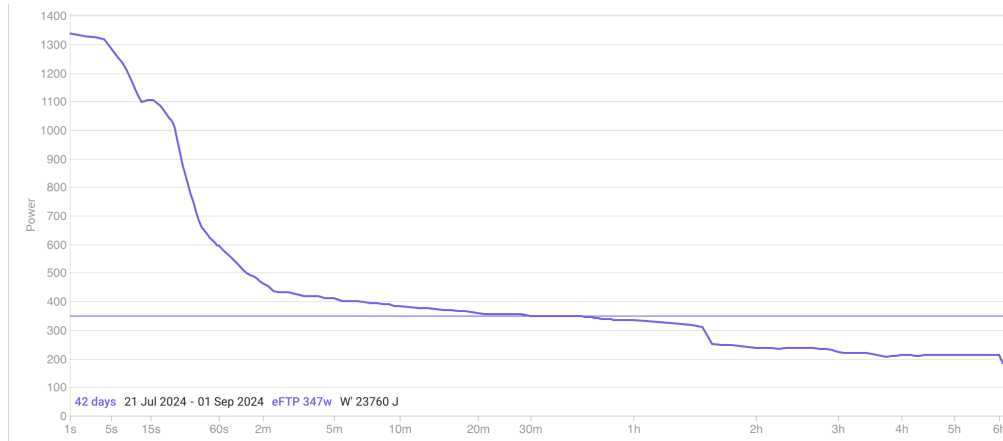


Figure 2.2: My Power Curve for August, 2024. The power curve can estimate what would be considered "hard" for an athlete at a given duration. Image courtesy of **intervals.icu**

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.

1. Safety (a composite metric defined by various factors, could also be gathered collectively from rider feedback)
2. Difference between Average and Normalized Power (in Watts/Kg)
3. Length (of segment)
4. Elevation Gain (or loss)
5. Road Quality
6. Absolute Normal Distance Change (aka how twisty the road is)

## 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

The purpose of this Software Requirements Specification is to define the functional and non-functional requirements for the GPX-Based Route Segmentation and Analysis System. This system enables end users (primarily cyclists and coaches) to upload recorded GPX ride data or draw routes on a web map, automatically segment the route into reusable segments, match them against a central database of known segments, and compute training-relevant statistics (Training Suitability Score, TSS). The SRS covers requirements for both *upload mode* (GPX ingestion, segmentation, database updates) and *query mode* (route matching without creation).

#### 3.1.2 Intended Audience and Reading Suggestions

- **Developers:** Read entire SRS to understand system scope, functional interfaces, and data models.
- **Testers:** Focus on Section 3.1.4 (Specific Requirements) for test case derivation.
- **Stakeholders (coaches, cyclists):** Sections 3.1.13.1.3 provide high-level understanding of capabilities.

#### 3.1.3 Overall Description

##### Product Perspective

This system is a standalone web service that is built on a LAMP-like technology stack (Linux, Nginx, MySQL, Flask), augmented by Python and Dockerized C++ processing engines. It integrates with third party mapping libraries (OpenStreetMap, Overpass, Osmium and OSRM). Conceptually, it acts as a data processor, taking user's GPX files from their GPS-enabled devices, and processing them against a central database. The system acts as an ETL (Extract-transform-load) pipeline

for incoming GPX files, as well as a query engine for user-inputted drawn map routes from a map drawing system.

### Product Functions

- **GPX Ingestion:** Accepts GPX 1.1 uploads, applies privacy-filter trimming, and parses into coordinate streams.
- **Map Data Preparation:** Fetches OSM extracts via Overpass; converts to PBF with Osmium; loads map into OSRM engine.
- **Map Matching:** Uses OSRM engine to snap GPX/drawn points to the routable network given by the .pbf file, yielding a formatted JSON object.
- **Segmentation:** Runs a two-pass C++ algorithm that (1) proposes segment boundaries by topological features and (2) matches segments in the database within spatial tolerances.
- **Database Management:** Inserts new segments (UUID + Route JSON), updates hit counts, and appends TSS values. Prunes segments below frequency thresholds after given amount of time.
- **Route Querying:** Given a drawn route, returns only the existing segments that fall along it, along with aggregated statistics.
- **User Interface:** Web pages for upload and drawing; interactive map highlighting segments and showing hover tooltips with TSS and hit count.

### 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

- **Server:** Linux (Arch Linux), Docker 20+, Nginx 1.28, Python 3.8+, MySQL 8.0+, C++17 toolchain.
- **Client:** Modern web browser (Chrome, Firefox, Safari) with JavaScript enabled.
- **Network:** HTTPS/TLS for all external calls, and internal HTTP calls with GET and POST.

## Design and Implementation Constraints

- **Privacy Trimming:** First 1 km of every GPX trace to be removed before any processing by default, can be overridden by user.
- **Modularization:** All components of the system must be contained within Docker containers, for easy deployment to most servers.
- **Data Formats:** GPX 1.1, GeoJSON (LineString), MySQL spatial types.
- **Standards Compliance:** Must adhere to GDPR and UK Data Protection Act; architecture described per ISO 42010.

## Assumptions and Dependencies

- Overpass API availability or a local Overpass instance.
- OSM data completeness for user routes geographic boundaries (using union of bounding boxes)
- OSRM server pre-loaded with relevant PBF extracts before running matching service.
- Users supply valid GPX conforming to standard schema. Some variance is ok, and a file with metric synonyms is accessible to the user.

### 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 [4].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 [5] 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

- The 'Segments' table *shall* define a primary key on 'segment\_id', of the UUID form.
- A SPATIAL index *shall* be defined on the 'geometry' column to accelerate spatial lookups.
- The 'tss\_values' array *shall* be horizontally scalable (e.g. limit array growth or average older values).

### Software System Attributes

**Reliability** System components are individually testable, and hardened for failure with catch blocks for any edge conditions, with the default behaviour defined as "ignore" per section. The multi-chunk approach to processing also allows for error, without corrupting the entire dataset. If a chunk is corrupted, it can just be ignored.

**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:

- All HTTP endpoints *must* require HTTPS/TLS.
- Uploaded GPX files *shall* be scanned for XML schema compliance to prevent malicious payloads.
- System is interconnected on a private, secure network. Only accessible by the web host.
- Enforce TLS 1.2+, HSTS, OCSP Stapling, and complex ciphers.
- Use reverse proxy and rate limiting for connecting to the server.
- Disable directory listing, set file permissions.

- Set all security headers in Nginx.
- Input validation, normalisation, and output encoding to prevent SQL-injection and XSS attacks.
- Randomize file input names, sniff content and enforce size limits (<100Mb).
- Separate DB user per application with least privileges (no root user)
- Bind the database to localhost or internal private IP.
- Enforce strict SQL modes

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

**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

# Implementation



# 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 [6], 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 margine of error (<5%). If it is, then the matching database entry will have it's 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 5.1. 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.

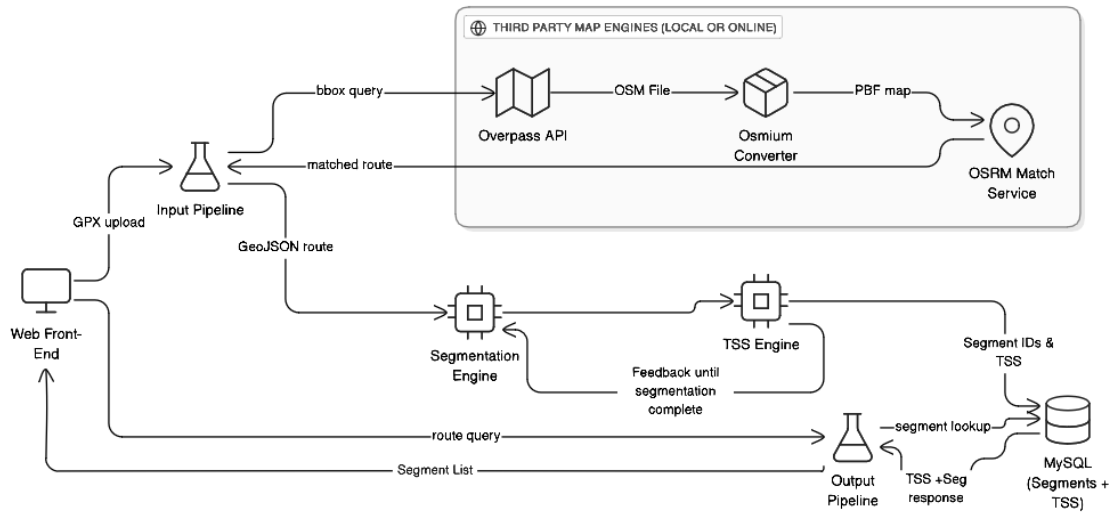


Figure 5.1: High level system architecture for the Route Segmentation and Analysis System

- **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 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 [7] 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 .pbm conversion and merging process. Used in the input pipeline
- **OSRM [1] 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 GPX-to-Enriched-JSON Pipeline Modules

Python Modules for the input pipeline, including dependencies, and interfaces.

#### Module Dependency Diagram

#### Module Specifications

##### **gpx\_utils.py**

**Name:** Purpose and description.

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

**Behavior:** Algorithmic overview or pseudocode.

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

##### **osrm\_utils.py**

**Name:** Purpose and description.

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

**Behavior:** Algorithmic overview or pseudocode.

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

##### **osm\_utils.py**

**Name:** Purpose and description.

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

**Behavior:** Algorithmic overview or pseudocode.

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

##### **extension\_utils.py**

**Name:** Purpose and description.

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

**Behavior:** Algorithmic overview or pseudocode.

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

**prepare\_\_map.py**

**Name:** Purpose and description.

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

**Behavior:** Algorithmic overview or pseudocode.

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

**batch\_\_route\_\_calc.py**

**Name:** Purpose and description.

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

**Behavior:** Algorithmic overview or pseudocode.

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

**merge\_\_routes.py**

**Name:** Purpose and description.

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

**Behavior:** Algorithmic overview or pseudocode.

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

**build\_\_way\_\_index.py**

**Name:** Purpose and description.

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

**Behavior:** Algorithmic overview or pseudocode.

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

**gpx\_\_enrich.py**

**Name:** Purpose and description.

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

**Behavior:** Algorithmic overview or pseudocode.

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

**batch\_\_enrich.py**

**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 [1] route matching engine, and the Overpass API [7].

## 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 [8] for security).

# Chapter 6

## Methodology

### 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 [7]; 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 [1], so I built a Docker image containing Osmium [9] 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 [4] 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 `.pbf` 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 [10]. 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 6.1.1 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 the enrichment pipeline

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

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

**Goal.** My goal was to stand up a C++ service that (i) runs in Docker, (ii) exposes HTTP endpoints, and (iii) ingests large, enriched OSRM/GeoJSON inputs reliably as a foundation for segmentation.

#### Step 1 Scaffold & build

I started by sketching a minimal project layout (src/http, src/core, src/models, etc.) and wiring CMake. After a quick first compile, I hit the classic CMake ordering issue: I had placed `target_include_directories` before `add_executable`, so CMake complained the target didnt exist. I moved `add_executable` earlier and the build went through. See Listing B.2.

- **What worked:** a small, conventional layout kept includes and targets simple.
- **What didnt:** target ordering in CMake on the first attempt.
- **Changes made:** ensured `add_executable` appears before all `target_*` calls.

I then containerised the build with Ubuntu base and non-interactive `apt` to avoid tzdata prompts. See Listing B.3.

- **What worked:** reproducible Docker builds; a short `build_and_deploy.sh`.
- **What didnt:** I initially passed Docker flags with bad spacing and got `invalid containerPort: 5005--name`.
- **Changes made:** fixed the run script to pass `-p HOST:CONTAINER` and `--name` cleanly.

#### Step 2 HTTP server and dispatcher

Next I wrote a minimal server entrypoint that listens on `0.0.0.0:8080` and added a tiny dispatcher so routes call a single `callHandler(req,res, action)`. See Listing B.5. I also created `/segment` and `/debug` endpoints.

I briefly broke routing by registering endpoints in a loop with a lambda that captured the *action* string by reference. Every route then called the last handler. I changed it to capture by value, and routing behaved.

- **What worked:** one dispatcher for multiple endpoints; simple route code.
- **What didnt:** lambda capture by reference in route registration.
- **Changes made:** capture `action` by value (`[action, &handler]{...}`).

### Step 3 Early debug loop and large uploads

To debug end-to-end connection, I made `/debug` to count coordinates. That gave a nice smoke test before implementing segmentation.

Because my inputs can be large (>15 MB), I increased `cpp-httplibs` payload cap and timeouts in the server (see Listing B.5). After that, I ran `curl --data-binary @file.json` to send the file to the endpoint.

- **What worked:** a tiny point-counter confirmed HTTP + JSON wiring.
- **What didnt:** large bodies were rejected until I raised `set_payload_max_length`.
- **Changes made:** enabled larger payloads and longer read/write timeouts.

### Step 4 JSON validation with line/column

I added `/debug?validate=true`. If parsing fails, it returns the byte offset, line/column, and a short context snippet, which allowed diagnosing malformed uploads (Listing B.6). This only checks syntax, not schema.

- **What worked:** immediate pinpoint of syntax errors (exact line/column).
- **What didnt:** it doesnt catch type mismatches (that happens during deserialization).
- **Changes made:** kept this endpoint strictly for syntax; added defensive deserialization later.

### Step 5 Enriched OSRM input model (no steps, null-safe tracepoints)

With the plumbing stable, I defined the enriched input model. The input is an OSRM `match` response augmented with GPX points and extensions. I explicitly decided: (i) keep only the first matching (if there are many), (ii) drop OSRM `steps` for now, (iii) allow `tracepoints[]` to be `null` in places (unmatched samples), (iv) store GPX `time` as epoch seconds, and (v) accept GPX `extensions` as either an array of `{key,value}` or an object.

I implemented non-intrusive `from_json(...)` converters (found by ADL), and I made them defensive: every string/number read is guarded with `contains()+is_*` before access. For `tracepoints`, a null entry becomes a placeholder that preserves indices. The model is shown in Listing B.7; with examples of deserializers in Listing B.8.

During this pass I hit two runtime errors and fixed both:

1. *type\_error.302* (“type must be string, but is number”) whenever a field named **"type"** wasn't actually a string in some addon. I replaced naive `value("type", "")` uses with guarded string reads.
  2. *type\_error.306* (“cannot use value() with null”) whenever I called `value()` on an object that was actually `null`. I stopped using `value()` on uncertain nodes and switched to `contains()+is_*()` with safe defaults.
- **What worked:** non-intrusive converters; null-safe tracepoints; epoch timestamps.
  - **What didnt:** assuming types with `value()` caused 302/306 errors on real data.
  - **Changes made:** defensive guards around every field; placeholder tracepoints to keep index alignment; “use first matching for consistency.

At this point the service (i) runs in Docker, (ii) exposes `/segment` and `/debug`, and (iii) reliably ingests large, enriched OSRM inputs into strong C++ structures with defensive parsing. This is the foundation I use for the segmentation rules in the next section.

### 6.1.3 TSS Engine

## 6.2 Testing and Results

## Chapter 7

# Results and Conclusions

## Chapter 8

# Further Discussions and Research Gaps

## Appendix A

### Further Reading

# Appendix B

## Code Listings and File Structures

**Listing B.1:** Code directory for Segmentation Engine

```
1 segmentation-engine/  
2   src/ (http/, io/, core/, models/)  
3   include/      # single-header deps: httplib, nlohmann/json  
4   build/        # Build files and settings for the engine  
5   config/       # settings.json,  
6   scripts/      # build & run helpers  
7   CMakeLists.txt  
8   Dockerfile
```

**Listing B.2:** CMakeLists.txt CMake file for Segmentation Engine

```
1 cmake_minimum_required(VERSION 3.10)  
2 project(segmentation_engine)  
3 set(CMAKE_CXX_STANDARD 17)  
4 set(CMAKE_EXPORT_COMPILE_COMMANDS ON)  
5  
6 add_executable(segmentation_engine  
7     src/main.cpp  
8     src/http/http_handler.cpp  
9     src/io/geojson_parser.cpp  
10    # (+ later models/ and analysis/ files)  
11 )  
12  
13 target_include_directories(segmentation_engine PRIVATE include src)  
14 target_link_libraries(segmentation_engine PRIVATE pthread)
```

**Listing B.3:** Dockerfile: builds and deploys the Dockerized Segmentation Engine

```
1 FROM ubuntu:20.04  
2 ARG DEBIAN_FRONTEND=noninteractive  
3 ENV TZ=Etc/UTC  
4 WORKDIR /app
```



```

5
6 RUN apt-get update && \
7     apt-get install -y --no-install-recommends \
8         cmake build-essential git curl && \
9     rm -rf /var/lib/apt/lists/*
10
11 COPY . /app
12 RUN mkdir -p build && cd build && cmake .. && make -j
13
14 EXPOSE 5005
15 ENTRYPOINT ["/app/build/segmentation_engine"]

```

**Listing B.4:** build\_and\_deploy.sh: Runs the build process and launches the Engine

```

1 #!/usr/bin/env bash
2 set -e
3 IMG=segmentation_engine:latest
4 CTR=segmentation_container
5 HOST_PORT=${1:-8080}
6 CONTAINER_PORT=${2:-8080}
7
8 docker build -t "$IMG" .
9 docker rm -f "$CTR" 2>/dev/null || true
10 docker run -d -p "$HOST_PORT:$CONTAINER_PORT" --name "$CTR" "$IMG"
11 echo "http://localhost:$HOST_PORT"

```

**Listing B.5:** main.cpp: Main orchestrator of engine, sets up server and defines POST endpoints

```

1 /* main.cpp: register routes by capturing `action` BY VALUE */
2 httpplib::Server server;
3 server.set_payload_max_length(512ull * 1024ull * 1024ull); // large uploads
4 server.set_read_timeout(60,0); server.set_write_timeout(60,0);
5
6 HttpHandler handler;
7 server.Post("/segment", [action=std::string("segment"), &handler]
8     (const auto& req, auto& res){ handler.callHandler(req,res,action); });
9 server.Post("/debug", [action=std::string("debug"), &handler]
10     (const auto& req, auto& res){ handler.callHandler(req,res,action); });
11
12 server.listen("0.0.0.0", 8080);

```

**Listing B.6:** http\_handler.cpp: Checks for parsing error of JSON body

```

1 // On parse_error: compute byte->(line,col) and return a snippet with a caret.
2 auto [line, col] = calc_line_col(req.body, e.byte);
3 nlomann::json err = {{"ok",false}, {"line",line}, {"column",col},
4     {"context", context_snippet(req.body, e.byte)}};

```

```
5 res.status = 400; res.set_content(err.dump(2), "application/json");
```

**Listing B.7:** osrm\_enriched.hpp: In the header, defines all structs for holding route data

```
1 // models (subset): no OSRM steps; tolerate null tracepoints; epoch times in GPX.
2 struct Geometry {
3     std::string type; // "LineString"
4     std::vector<std::array<double,2>> coordinates; // [lon,lat]
5 };
6
7 struct Leg {
8     nlohmann::json annotation;
9     std::string summary;
10    double weight=0, duration=0, distance=0;
11 };
12
13 struct Matching {
14     double confidence=0;
15     Geometry geometry;
16     std::vector<Leg> legs;
17     std::string weight_name;
18     double weight=0, duration=0, distance=0;
19 };
20
21 struct GpxPoint {
22     double lat=0, lon=0, ele=0;
23     long long time=0; // epoch seconds
24     nlohmann::json extensions; // array {key,value} OR object
25 };
26
27 struct Tracepoint {
28     bool matched=false; // supports null entries
29     int alternatives_count=0, waypoint_index=-1, matchings_index=-1;
30     std::array<double,2> location{0.0,0.0};
31     std::string name;
32     std::vector<GpxPoint> gpx_list;
33 };
34
35 struct OsrmmatchResponse {
36     std::string code;
37     std::vector<Matching> matchings; // rule: use first
38     std::vector<Tracepoint> tracepoints; // null-safe placeholders
39 };
```

**Listing B.8:** osrm\_enriched.hpp Also in header, define function overloads for from\_json(), which replaces same function in json.get()

```
1 // Example: robust Geometry::from_json (no throws on null/wrong types)
2 inline void from_json(const nlohmann::json& j, Geometry& g) {
```

```
3  if (j.contains("type") && j["type"].is_string()) g.type = j["type"].get<std::string>();
4  g.coordinates.clear();
5  if (j.contains("coordinates") && j["coordinates"].is_array()) {
6      g.coordinates.reserve(j["coordinates"].size());
7      for (const auto& pt : j["coordinates"]) {
8          if (pt.is_array() && pt.size()>=2 && pt[0].is_number() && pt[1].is_number())
9              g.coordinates.push_back({pt[0].get<double>(), pt[1].get<double>()});
10     }
11 }
12 }
13
14 // Tracepoint: allow null entries and keep index alignment
15 inline void from_json(const nlohmann::json& j, Tracepoint& t) {
16     if (j.is_null()) { t.matched=false; t.waypoint_index=-1; t.matchings_index=-1;
17         t.gpx_list.clear(); return; }
18     t.matched=true;
19     if (j.contains("waypoint_index") && j["waypoint_index"].is_number_integer())
20         t.waypoint_index = j["waypoint_index"].get<int>();
21     // ... (similar guards for other fields; parse gpx_list if array)
22 }
```

## Appendix C

# Directory Structure

## Appendix D

# Application Documentation

# Bibliography

- [1] Dennis Luxen and Christian Vetter. Real-time routing with OpenStreetMap data. In *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS '11)*, pages 513–516, New York, NY, USA, 2011. ACM. doi: 10.1145/2093973.2094062.
- [2] Mehdi Sharifzadeh, Farnaz Azmoodeh, and Cyrus Shahabi. Change detection in time series data using wavelet footprints. In *Advances in Spatial and Temporal Databases (SSTD 2005)*, volume 3633 of *Lecture Notes in Computer Science*, pages 127–144, Angra dos Reis, Brazil, 2005. Springer. doi: 10.1007/11535331\_8.
- [3] Sean Hurley. Normalized power®: What it is and how to use it. TrainerRoad Blog [Online]. Available: <https://www.trainerroad.com/blog/normalized-power-what-it-is-and-how-to-use-it/> [Accessed 11-Aug-2025], 2020.
- [4] Rob Story and contributors. Folium: Python data. Leaflet.js maps. <https://python-visualization.github.io/folium/>, 2023.
- [5] Editors of Bicycling. A cyclist’s guide to speaking strava. *Bicycling Magazine (online)*, 2017. Published 31 May 2017, available at <https://www.bicycling.com/news/g20041808/a-cyclists-guide-to-speaking-strava/>.
- [6] Systems and software engineering architecture description. Standard ISO/IEC/IEEE 42010:2011, International Organization for Standardization, Geneva, Switzerland, 2011.
- [7] OpenStreetMap community. Overpass API (OpenStreetMap data query service). OpenStreetMap Wiki [Online]. Available: [https://wiki.openstreetmap.org/wiki/Overpass\\_API](https://wiki.openstreetmap.org/wiki/Overpass_API) [Accessed 11-Aug-2025], 2025.
- [8] Information technology security techniques information security management systems requirements. Standard ISO/IEC 27001:2013, International Organization for Standardization, Geneva, Switzerland, 2013.
- [9] Jochen Topf and Osmium Contributors. Osmium: A multi-purpose toolkit for OpenStreetMap data. Osmium/Osmcode Project [Online]. Available: <https://osmcode.org/> [Accessed 11-Aug-2025], 2023.

- [10] Geoffrey Boeing. Osmnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks. *Computers, Environment and Urban Systems*, 65:126–139, 2017.