

Quantification of Cycling Training Routes by Segmentation and Algorithmic Analysis

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 benefit sustained efforts, whereas junctions, stops and heavy traffic fragment work and increase power variability. This thesis investigates whether the "training suitability of roads can be quantified and used to help cyclists plan better sessions. This paper presents an end-to-end framework that transforms raw GPS traces from a GPX file into a list of road segments, annotated for training suitability. An input pipeline cleans and map-matches traces to OpenStreetMap using OSRM in a manner similar to Luxen and Vetter (2011), and enriches the matched geometry with road metadata (e.g., highway class, speed limit, junction density). A standalone engine performs the route segmentation with a two pass processing algorithm. . Each segment has its metadata analysed, and then it is given a Training Suitability Score (TSS) derived from road context and observed ride dynamics. TSS has 5 components; one for each zone in the 5-zone training model. Together, these components form a practical route-planning system that prioritises the quality of training stimulus over performance ranking and competition.

Contents

Abstract	1
1 Introduction	1
1.1 Background	2
1.2 Research Objectives	2
2 Literature Review	4
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	5
2.1.3 Methodological Considerations and Assumptions	7
2.1.4 Pulling Map Data	7
2.2 Training Suitability Scoring	8
2.2.1 Research Objectives	8
2.2.2 Approaches to Suitability Scoring	9
2.2.3 Methodological Considerations and Assumptions	11
2.3 LSEPI Analysis	14
2.3.1 Legal	14
2.3.2 Social	14
2.3.3 Ethical	15
2.3.4 Political	15
3 Requirements	16
3.1 Software Requirements Specification (SRS)	16
3.1.1 Purpose and Scope	16
3.1.2 Intended Audience and Reading Suggestions	16
3.1.3 Overall Description	16
3.1.4 Specific Requirements	18
3.2 Standards and Compliance	20
4 Implementation	21
5 Design	22

5.1	Scope and Purpose	22
5.2	Architectural Design	22
5.2.1	Architectural Views	22
5.2.2	Design Rationale	25
5.2.3	Design Patterns and Styles	25
5.3	Module-Level Design	25
5.3.1	GPX-to-Enriched-JSON Pipeline Modules	25
5.4	Interface Design	26
5.4.1	User Interface	26
5.4.2	Software Interfaces	26
5.5	Data Design	26
5.5.1	Data Models	26
5.5.2	Database Schema	26
5.5.3	Data Dictionary	26
5.6	Standards and Compliance in Design	26
6	Methodology	27
6.1	Critical Reflection on Methodology	27
6.2	Component Implementation	27
6.2.1	Input Pipeline	27
6.2.2	Segmentation Engine	30
6.2.3	Segmentation Engine: Two-Pass Weighting and Recursive Merging	32
6.2.4	TSS Engine	33
6.3	Testing and Results	33
7	Results and Conclusions	35
8	Further Discussions and Research Gaps	36
A	Further Reading	39
B	Code Listings and File Structures	40
C	Directory Structure	47
D	Application Documentation	48

List of Figures

2.1	HMM Matching Transition probabilities mapped over actual Ground-Truth route transition graph.	6
2.2	The 7 Zone Training Model, courtesy of Indoor Cycling Association	10
2.3	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	13
5.1	High level system architecture for the Route Segmentation and Analysis System . .	23

List of Tables

2.1	Training Suitability Scoring > Methodological Considerations and Assumptions . . .	12
3.1	Functional Requirements (FR) Overview	19
3.2	Non-Functional Requirements (NFR) Overview	19
3.3	Database Requirements	19
5.1	Architectural Design > Architectural Views structured summary	24
6.1	Canonical extension keys and example synonyms	29

Chapter 1

Introduction

High-quality cycling training depends on being able to execute the intended effort without interruption. Professional riders often train on long climbs with low-traffic roads in remote locations; most amateurs, by contrast, plan routes with general-purpose mapping tools and only discover midride that junctions, blind corners or traffic furniture force repeated wattage spikes or dips. The result is high power variance, which increases Normalized Power, a metric that approximates the metabolic cost of a ride according to Coggan and Allen (2012). This means that during an endurance or structured interval session, an athlete may not be able to target the power zones they are aiming for, leading to suboptimal training stimulus. Most mapping tools do not consider the suitability of a road for structured training, focusing instead on either performance rankings (e.g., Strava segments) or commuting efficiency (e.g., Google Maps). This dissertation addresses that gap. by investigating whether the suitability of a road for structured training can be quantified from just ride data and open-source mapping solutions so that routes can be planned with an optimal path; maximizing segments that support sustained efforts. We refer to this property as *training suitability*. The proposed system would ingest raw GPX traces, perform map-matching against OpenStreetMap with OSRM's match engine, enrich the resulting paths with road metadata (e.g., highway class, speed limits and junction density), and then apply a segmentation procedure. Segments are identified using a multi-pass algorithm, that takes each "leg" of the route (a leg being the smallest unit of a route, returned by OSRM's match engine) and attempts to merge every consecutive leg recursively until either there is one leg, or the merge cost of each segment/leg exceeds a threshold. This is known as a merge-cost algorithm, classed as a greedy algorithm, since it always chooses the best option at each step, without considering future consequences. Each segment then has a set of scores generated from the road and GPX metadata, one for each accepted training zone below 1 hour, defined by Sage (2019). This set of scores is collectively known as the Training Suitability Score (TSS). To support this goal, I aim to implement (1) batch processing for large GPX collections, (2) segment de-duplication and matching so new routes can be expressed as sequences of known segments if available, and (3) 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

While route planning tools for cyclists abound, most focus on navigation efficiency or social competition rather than the quality of the training stimulus. Popular platforms such as Strava and Komoot allow riders to discover new roads and measure themselves against others, but their segment definitions are crowdsourced and oriented towards ranking performance on short stretches. Navigation services (e.g., Google Maps, OSRM, OSMbased planners) optimise for travel time or scenic value but do not provide insight into how well a route supports structured intervals. In the literature, map matching of noisy GPS traces to road networks has been well explored, with Hidden Markov models and Viterbi decoding achieving high accuracy (Newson and Krumm, 2009). Work on ride classification has also examined the use of power metrics like Normalised Power and intensity factors to characterise efforts (Hurley, 2020). However, there is a gap in combining these techniques to produce a datadriven assessment of road suitability for different training zones. This dissertation builds on the existing foundations by proposing a segmentation and scoring framework that prioritises sustained power delivery and low interruption frequency, aiming to bridge the gap between navigation and training science.

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:

- O1. **Define 'training suitability'.** Formalise a segment-level construct that analyses the training performance and route features at five levels of training suitability.
- O2. **Build a reproducible map-matching pipeline.** Establish a deterministic GPX \rightarrow OSM route pipeline that normalizes the data to a standard map-matched format, using OSRM's (2021b) match service to remove unwanted GPS noise.
- O3. **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 (Sharifzadeh et al., 2005) change detection,
 - O3d. topology-aware separation (junction density, crossings),
 - O3e. greedy vs. dynamic-programming formulations.

-
- O4. **Engineer segment features.** Design a feature schema; including but not limited to: 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.
 - O5. **Specify a TSS model.** Create (1) a transparent rule-based baseline mapping features to a scalar training suitability score, as well as (2) an algorithmic signal analysis for each training zone (Endurance, Tempo, Threshold, VO₂max, Anaerobic/Neuromuscular) from the segment features.
 - O6. **Ground-truth construction.** Derive labels from ride dynamics (e.g., proportion of time in target zone, pause frequency,
 - O7. **Validate alignment with training intent.** Test whether TSS correlates with: (a) sustained power feasibility, (b) low interruption frequency, (c) subjective suitability ratings based on existing known training rides at given intensity.
 - O8. **Segment library operations.** Deduplicate, index, and match new routes to an evolving segment set; assess match precision for each call.
 - O9. **Human feedback (optional).** Prototype lightweight feedback (thumbs-up/down or pairwise choices) and test whether incorporating it improves ranking metrics.
 - O10. **Reproducibility and privacy.** Ensure runs are reproducible and only anonymized segment scores are stored to the database (no user-identifiable ride data).

Chapter 2

Literature Review

A literature review was conducted¹ to determine the current state of the art in route segmentation and suitability scoring. The review was conducted over an extended period of time, pre-dating most of the experimentation and development of the system. While a lot of the literature was not directly implemented into the system, it was used to inform the design and implementation, as well as provide a springboard for critical thinking about how the system could be designed.

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 construct new routes based on the requirements of the rider's training plan. Intuitively, it makes sense to segment a route; riders do not have to analyse every part of a map in street view to determine if it is viable for their specific workout. 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 method of segmentation, and subsequent categorization, of road segments so riders can path their ride to maximise the TSS of their route. Intuitively, we could segment a route by the roads that it follows. When a route changes to a new road, the route can be segmented. However this can prove more difficult to formalize, as a road can have many intersections, and a logical route could span two roads. The question becomes how to define a split so that segments occur at meaningful points, and how do we define meaningful?

¹Note to reader: since the literature review was conducted in advance to any other section in this paper, any method mentioned earlier was considered to be effective as a result of this literature review

2.1.2 Algorithmic Approaches to Segmentation and Categorization

There are several methodologies when it comes to segmentation, including (but not limited to) clustering, heuristic methods, signal analysis, and map-based segmentation. Some, or all of these may be applied to the algorithm, since there is not one method to solve all segmentation types. What follows is a review of the literature that exists that could contribute to achieving advanced segmentation.

Paul Newson and John Krumm: "Hidden Markov Map Matching Through Noise and Sparseness" A useful way to clean up the location data as a pre-processing step could be through map matching. Newson and Krumm (2009) explore map-matching through a HMM model on timestamp-marked latitude and longitude pairs. The approach showcased in the paper is focused on automobile traffic, so there may be some need to tweak the emission delta for bikes, to accommodate for larger variance in the horizontal travel distance, as well as increasing the search range for nodes around each track point. The search would then use a Gaussian probability curve, centred on GPX trackpoint, with nodes closer being more likely than ones further away. Then a transition probability will be implemented between consecutive points so straight line distance is more probable. Lastly, a Viterbi algorithm will be run to find the "most likely" node sequence. Their approach includes two variables that need to be tweaked to maximise accuracy. This could be done via machine learning, but is out of the scope of this research. It will be set to a "good enough" setting. Once the node sequence is determined, the sequence of ways can then be determined. At first I considered a rough implementation of this algorithm for my purposes, keeping it lightweight but targeted for my use case. However, after researching my next source, I realized that the OSRM (Open Source Routing Machine) project already implements a map-matching algorithm that is more than sufficient for my needs, and had a HTTP interface, so I can simply send the GPX data to it, and it will return a JSON object with the matched route.

HMM-based Map Matching with OSRM OSRM's `/match` service implements the HMM paradigm, returning one or more matched sub-traces with per-point confidence (Project OSRM, 2021b). Telenav's (2018) OSRM notes explicitly discuss HMM+Viterbi for map matching. Practically, OSRM's design makes this workflow reproducible and customizable: you can run the engine locally in Docker for deterministic experiments and adjust routing behaviour with Lua profiles to better reflect cycling vs. driving, surface penalties, or junction costs before (re)extracting the graph used by the matcher (Project OSRM, 2021a, 2025a,b). Taken together, this lets us: (1) map-match noisy rides, (2) export rich annotations (nodes/way/gpx) for downstream segmentation, and (3) tune profiles to reflect training-use contexts (e.g., prefer longer, open roads, but avoid major highway links), then re-run match at scale.

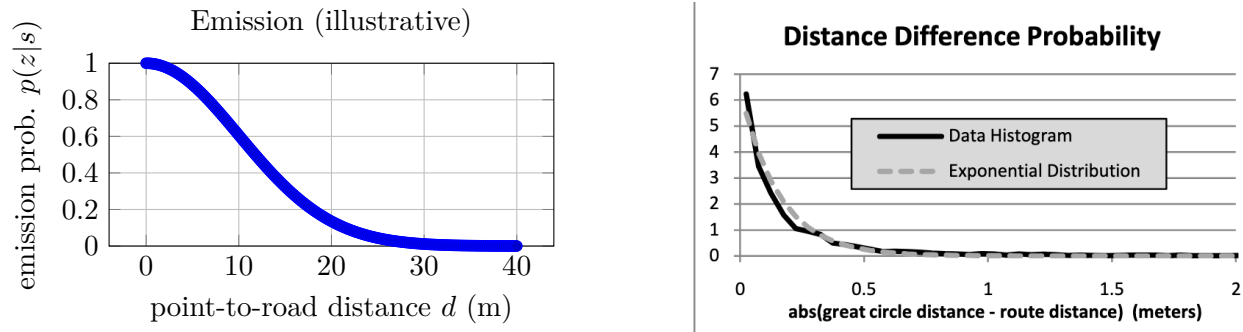


Figure 2.1: HMM Matching Transition probabilities mapped over actual Ground-Truth route transition graph.

Customising behaviour via Lua (before (re)extract).

```

1 -- Inside a custom profile (simplified example)
2 local function way_processing(way, result)
3     local highway = way:get_value_by_key('highway') or '
4     local surface = way:get_value_by_key('surface') or '
5     local smooth  = way:get_value_by_key('smoothness') or '
6
7     -- Base speeds (km/h) - tune to your use case
8     local base = { residential = 22, tertiary = 20, secondary = 18, primary = 16 }
9     result.speed = base[highway] or 15
10
11     -- Discourage trunk and trunk_link for training rides
12     if highway == 'trunk' or highway == 'trunk_link' then
13         result.speed = result.speed * 0.30
14         result.weight = (result.weight or 1.0) * 3.0
15     end
16
17     -- Prefer paved+smooth; discourage unpaved
18     if surface == 'unpaved' or surface == 'gravel' then
19         result.speed = result.speed * 0.75
20         result.weight = (result.weight or 1.0) * 1.4
21     end
22     if smooth == 'excellent' or smooth == 'good' then
23         result.speed = result.speed * 1.10
24     end
25 end

```

Listing 2.1: Lua profile example snippet: discourage trunk links, prefer residential/smooth paved for cycling

Boeing, G: "Modeling and Analyzing Urban Networks and Amenities with OSMnx"

Boeing (2017) details a novel Python package, `osmnx`, that downloads and analyses street networks from OpenStreetMap, which implements simple query language for pulling street data from the OSM database. It also includes tools for simplifying OSM map data so ways and nodes represent roads and intersections. As described in their Getting Started guide, the library enables "a programmatic

approach to visualizing OSM data as a mathematical graph". The graph can be formatted as a `GeoDataFrame`, which can be exported using `networkx` and `geopandas` libraries. Using a function like `graph_from_bbox()`, it is possible to obtain a road network around the bounds of a GPX track by getting the maximum and minimum latitudes and longitudes, which can then return a clean and topologically correct graph \mathcal{G} to map match to.

2.1.3 Methodological Considerations and Assumptions

Firstly we must obtain a GPX file. This is given from either a new or existing activity. It can be obtained 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. An example can be found in Appendix B.9. It is obvious that the GPX file follows a hierarchical tree structure, with metadata including time, GPS coordinates, and cycling-specific physiological metrics. Not all of the data is named consistently, especially in "extensions", so it is important to consider variations in naming. This can be normalized during the processing with a regex-style dictionary map, having a standard name for each useful metric, along with a list of synonyms.

Once downloaded, GPX data must be normalized and processed. This can be done by converting to a standardized JSON object structure, where each object contains at least Latitude, Longitude, Elevation, and timing data. After this, erroneous values should be corrected without removing time series data. This can be done with either flagging or interpolation. Flagging is useful for detecting missing data, such as 0s or NaNs, which can be ignored in the data processing, but removing the data would cause timing mismapping, putting the data signals out of sync. Simply detecting 0s or NaN values in the data stream is sufficient for some datapoints, but the preferred approach for numerical data is to approximate it with interpolation of the average of the surrounding present values. Lastly, derived metadata, such as speed or heading, should be calculated mathematically, and added as an additional extension in the JSON file to reduce processing time on any non-concurrent processes.

2.1.4 Pulling Map Data

OSMnx allows easy and efficient pulling of API data from the OpenStreetMap database. This could be used to correct the GPS data, and also 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 Overpass API
4. Download the map as a `.osm2` file for caching.

²Important to note: `.osm` files are an XML-style map file that is downloaded directly from the internet. OSRM

2.2 Training Suitability Scoring

2.2.1 Research Objectives

To define a training suitability score, we must first define what a training suitability score is. This is a subjective measure of how suitable a route is for a specific targeted training zone. We must define what the zones are, define the scale of our score, define a method of obtaining said score, and define a validity checking function to verify that the score isn't affected by outlier conditions (such as noise, riders getting punctures, or other issues unrelated to the route itself). Since the aim is to build a database of scores, each score must be isolated and normalised to the rider. So absolute power could not be used as a good metric, since power thresholds are different per rider. Instead, we must consider rider-agnostic metrics. We also need default metrics, that don't relate to data that may not be present (such as extensions). We should be able to generate a score with nothing but base GPX data (latitude, longitude, elevation, time) and OSM route data, with the ability to refine the score with metrics like heart rate, power, and cadence.

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.

doesn't accept this as a file type, so it is necessary to use something like Osmium for conversion from .osm→.pbf.

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...).

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 (2005), 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 by Indoor Cycling Association (n.d.) 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. A breakdown of 7 zones

is visible in Figure 2.2, courtesy of the Indoor Cycling Association, breaks down each zone. Zones 3-7 will be used for Wavelet Analysis.

 Power and Heart Rate Training Zones and RPE Chart							
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 2.2: The 7 Zone Training Model, courtesy of Indoor Cycling Association

Sean Hurley: "Normalized Power (Hurley, 2020): 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 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.

Table 2.1: Training Suitability Scoring > Methodological Considerations and Assumptions

Name/Key	Description/Value	Notes
Type	WhatA, B, service roads, motorway, bike path, etc...	is the categorization of the road?
Surface	What?armac, concrete, gravel, dirt, etc	is the surface of the road made of?
Behaviour	How Straight, bends, switchbacks, sharp turns	does the road act?
Safety	How safe is this route?	Are there a lot of accidents, or commercial traf

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 Figure 2.3.

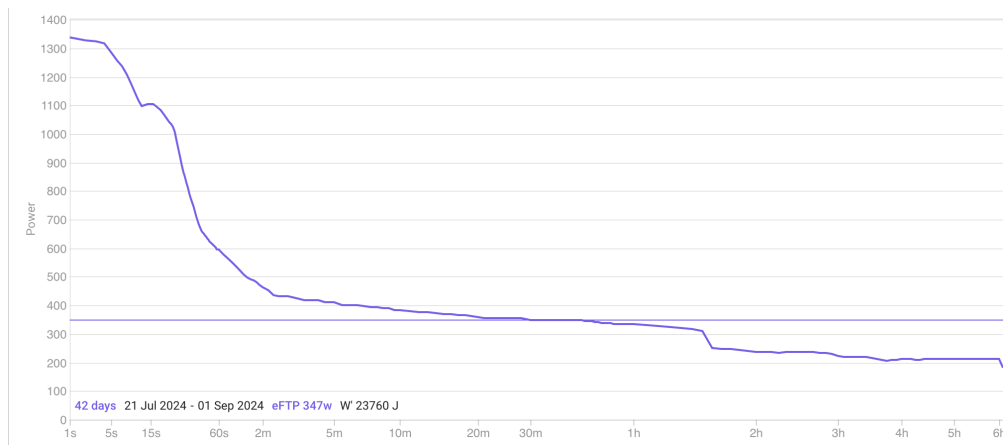


Figure 2.3: My Power Curve for August, 2024. The power curve can estimate what would be considered "hard" for an athlete at a given duration. Image courtesy of [intervals.icu](https://www.intervals.icu)

- **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.

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

The legal dimensions of this project revolve around data protection and intellectual property. All GPX files ingested into the system are treated as personal data because they reveal an individual's whereabouts and training habits. Consequently the project was designed in accordance with the UK Data Protection Act and GDPR: the system performs privacy trimming on the first kilometre of every ride, never stores raw trackpoints beyond the scope of the immediate analysis, and only persists aggregated segment statistics. When map data is fetched from OpenStreetMap and OSRM profiles are tuned the licences of these projects are respected: both OpenStreetMap and OSRM are permissively licensed, so derived works can be integrated into an academic system provided attribution is given. Any code reused from opensource libraries (such as `cpphttplib`, `nlohmann/json` or `osmnx`) is cited appropriately and the original licences are included in the repository, fulfilling our obligations under the MIT and BSD licences.

2.3.2 Social

From a social standpoint the system aims to empower cyclists of all abilities to train more effectively, but it also has the potential to exacerbate inequalities. Experienced riders with sophisticated power meters will be able to contribute richer data to the segment database, which could bias the Training Suitability Score towards routes favoured by wellresourced athletes. To mitigate this we deliberately designed the scoring algorithm to rely on ride agnostic features (e.g., road type, slope and curvature) and to normalise physiological metrics relative to each rider's baseline. Furthermore, the opensource

nature of the software means local clubs can host their own instances and build segment libraries tailored to their communities, reducing dependence on proprietary platforms. The interactive web interface, implemented using a lightweight stack of Flask and Leaflet, is accessible on modern browsers and does not require installing proprietary software.

2.3.3 Ethical

Ethically, the most salient issue is consent. Users must consent to their ride data being processed, and they should be aware of how long the data is retained and for what purpose. Our ingestion workflow therefore requires explicit opt-in and provides a simple mechanism for riders to delete their data. Another ethical consideration is the potential for the system to encourage risk taking: if an algorithm classifies a particular steep descent as high suitability for neuromuscular efforts, some riders might push themselves beyond safe limits. To prevent this the scoring model includes a safety term that downweights routes with high accident records or poor visibility, and the user interface warns that the tool is a guide rather than a directive. Finally, we considered the environmental impact of running an always-on cloud service. By containerising all components and supporting local processing on commodity hardware we reduce the carbon footprint associated with large centralised servers.

2.3.4 Political

On a political level the project touches upon debates around open data and civic infrastructure. OpenStreetMap is maintained by volunteers and reflects a collective effort to democratise geographic information. By building upon this commons we contribute to the public good rather than locking data behind commercial terms. However, routing and training tools can influence cycling patterns and therefore intersect with transport policy: highlighting certain roads as training-friendly could increase cyclist traffic on them, which may require coordination with local authorities for safety improvements. We do not advocate for specific policy outcomes but we recognise that our system could inform discussions about cycle lane provision, speed limits and road design.

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 3.1.1–3.1.3 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

- **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 (Story and contributors, 2023).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:

Table 3.1: Functional Requirements (FR) - Overview

ID	Requirement (shall...)	Rationale/Notes	Verification
FR-01	Privacy trimming: The system shall automatically remove the first 1 km of every GPX trace prior to any processing.	Default privacy protection for user start locations.	Unit test on import; end-to-end check that no points from the first 1 km persist; code review.
FR-02	GPX import: The system shall ingest GPX 1.1 files.	Defines supported input format; reject/flag unsupported versions.	Parser tests with valid/invalid GPX 1.1 samples; schema validation.
FR-03	Containerized operation: The system shall run each component (input pipeline, match engine, segmentation, TSS, DB interface) inside a Docker container.	Portability, isolation, reproducibility.	CI builds and runs docker-compose; health checks pass.
FR-04	Data protection compliance: The system shall process and store data in a manner compliant with GDPR and the UK Data Protection Act.	Legal/regulatory requirement.	Security review; documented DPIA; automated checks for PII absence in stored artifacts.

Non-Functional Requirements

The non-functional requirements of the system are as follows:

Table 3.2: Non-Functional Requirements (NFR) - Overview

ID	Requirement (shall...)	Rationale/Notes	Verification
NFR-01	Performance: The system shall process a minimum of 1000 GPX traces per hour.	Ensures scalability for expected data volumes.	Benchmark tests; monitoring in production.
NFR-02	Reliability: The system shall have an uptime of 99.9% or higher.	Critical for user trust and data integrity.	Automated uptime monitoring; incident tracking.
NFR-03	Extensibility: The system shall allow new analysis modules to be added with minimal changes to existing code.	Facilitates future feature growth.	Code review; demonstration of adding a module in test environment.
NFR-04	Usability: The system shall provide clear error messages and documentation for users.	Improves user experience and reduces support burden.	User testing; documentation review.

Logical Database Requirements

Requirement ID	Description	Type	Priority	Source	Notes
----------------	-------------	------	----------	--------	-------

Table 3.3: Database Requirements

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 8.

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 (*Systems and software engineering Architecture description*, 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 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 Figure 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

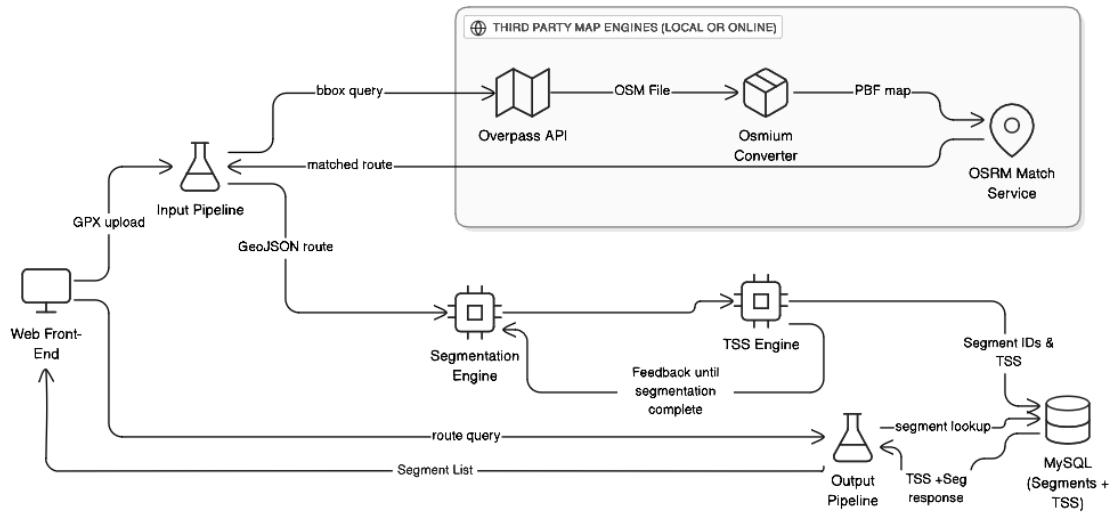


Figure 5.1: High level system architecture for the Route Segmentation and Analysis System

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 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 .pbk 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.

Table 5.1: Architectural Design > Architectural Views - structured summary

Name/Key	Description/Value	Notes
layers	itemize Presentation Layer - This is the front end	
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	
Orchestrator	Takes routes	
Orchestrator	Formats the resulting segments	
Orchestrator	Receives the Route data from the Input Pipeline Orchestrator	
Engine	Written in C++	
Engine	Takes a list of segments and the input GPX data	
Database	Stores all persistent segments	
Engine	A docker service that runs a .osm to .pbk conversion and merging process. Used in the input pipeline OSRM Matching Engine: Another docker service that takes small GeoJson chunks	

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

The implementation embraces a layered, pipeline-oriented architecture. The high-level design separates concerns into a presentation layer (web UI), a processing layer (Python orchestrators and C++ segmentation engine) and a data layer (SQLite/MySQL databases and OSM extracts), following the classic three-tier model. Within the processing layer the input pipeline is built as a series of independent stages that transform GPX tracks into enriched GeoJSON, mirroring the pipes and filters pattern: each Python module performs a single function (reading GPX, fetching OSM data, map matching, enrichment) and emits a well-defined intermediate representation for the next stage. The segmentation engine itself is implemented in C++ as a microservice with a single responsibility, exposing REST endpoints via a lightweight HTTP server (cpphttpplib). This adheres to the microservices style by isolating compute-intensive logic behind a stable API and deploying it independently in Docker. Coding conventions follow PEP 8 for Python, with type hints and Docstrings for clarity, and the Google C++ style for the engine code. Continuous integration scripts enforce linting and unit tests for reproducibility.

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

For specific module-level specifications, see Appendix D

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 (Luxen and Vetter, 2011) route matching engine, and the Overpass API (OpenStreetMap community, 2025).

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 (*Information technology Security techniques Information security management systems Requirements*, 2013) for security).

Chapter 6

Methodology

6.1 Critical Reflection on Methodology

In developing this methodology, several compromises were made. Purely statistical methods risked over-segmenting minor elevation noise, while rigid thresholds missed the nuanced undulations riders perceive as rolling terrain. We therefore adopted a hybrid of central-difference slope and moving-average deviation metrics, then cross-validated our expected algorithmic labels against observed rider feedback on a set of test routes. Discrepancies—such as segments tagged as *flat* that riders described as *challenging*—fed back into an iterative calibration loop, ensuring our computational definitions stay grounded in real-world cycling experience rather than abstract mathematical constructs.

6.2 Component Implementation

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.

6.2.1 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 (OpenStreetMap community, 2025); 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 (`.pbf`) format for OSRM (Luxen and Vetter, 2011), so I built a Docker image containing Osmium (Topf and Contributors, 2023) 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 (Story and contributors, 2023) 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 6.2.1.
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 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 (Boeing, 2017). 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 6.2.1 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 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

Table	Columns (sample values)	Description
ways	(101, {'highway': 'residential', 'name': 'Main Street'})	Residential road named Main Street
	(102, {'highway': 'footway', 'surface': 'paved'})	Paved footway
edges	(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

check for variance to adjust the match search radius.

6.2.2 Segmentation Engine

Goal. My goal was to stand up a C++ service that (1) runs in Docker, (2) exposes HTTP endpoints, and (3) 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 didn't 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 didn't:** 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 didn't:** 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 endpoint that listens on `0.0.0.0:5005` 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 that fixed the routing.

- **What worked:** one dispatcher for multiple endpoints; simple route code.
- **What didn't:** 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-httplib`'s 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 didn't:** 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=<condition>`. 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 didn't:** it doesn't 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)

Next I defined the enriched input model. The input is an OSRM `match` response augmented with GPX points and extensions. I decided:

1. keep only the first matching (for now)
2. drop OSRM `steps`, since it is a useless metric for our use,
3. allow `tracepoints[]` to be `null` in places for unmatched gpx values,

4. store GPX `time` as epoch seconds, and
5. 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 add-on. 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 didn't:** 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 (1) runs in Docker, (2) exposes `/segment` and `/debug`, and (3) 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.

6.2.3 Segmentation Engine: Two-Pass Weighting and Recursive Merging

After enriching each matched trace with OSM context (e.g., `highway`, `surface`, `cycleway`) and GPX-derived cues, the engine applies a two-pass procedure to produce contextually coherent segments:

Pass 1 — Propose boundaries. Candidate split points are placed where (1) topological changes occur (e.g., way or major attribute change), and (2) strong ride cues suggest a transition (e.g., large heading change or a stop). A small minimum-length guard prevents trivial fragments.

Pass 2 — Weight boundaries. Each proposed boundary receives a *merge cost* in $[0, 1]$ summarizing dissimilarity between the two adjacent sections. Heuristics based on OSM tags set the cost: for example, large differences in `highway` class, a `junction=yes` node, or an access barrier yield high costs (near 1), while minor attribute changes yield low costs (near 0). An illustrative ruleset is shown in Listing B.11. These weights prioritize real contextual changes and de-emphasize negligible fluctuations.

Recursive merging. Adjacent sections are then merged greedily while their separating cost remains below a threshold. After each merge, neighboring costs are refreshed to reflect the new context, and the process repeats until no further low-cost merges are available. Boundaries with cost = 1 (e.g., barriers or hard junctions) are never merged. The high-level routine is shown in Listing B.10.

Outcome. Homogeneous stretches (same road type, similar environment) form longer segments; genuine interruptions (e.g., sharp intersection, barrier, paved→unpaved) remain as boundaries. This yields segments that are stable and meaningful for training suitability analysis.

6.2.4 TSS Engine

The Training Suitability Score engine is responsible for converting enriched segments into multizone difficulty scores. It consumes the output of the segmentation engine (a list of segments with road metadata and GPX-derived signals) and computes, for each of the five training zones (Endurance, Tempo, Threshold, VO₂max and Anaerobic/Neuromuscular), a score in $[0, 1]$ indicating how conducive the segment is to that type of effort. The current implementation uses a transparent rulebased model: it bins features such as gradient, road type, curvature, stop density, power variability and safety rating, and combines them with calibrated weights learned from a small set of annotated rides. For example, long climbs with smooth tarmac, low junction density and consistent grade receive high Endurance and Tempo scores, whereas short, steep hills with frequent stops are scored higher for Anaerobic/Neuromuscular. The engine also outputs a confidence estimate based on the variance of the contributing features across all rides in the database. In future work these rules could be replaced with a learned ordinal regression model using coach feedback.

6.3 Testing and Results

Evaluation of the system consisted of both functional testing of the individual components and empirical assessment of the quality of the generated segments and scores. For the pipeline, unit tests verified that GPX parsing, OSM fetching and enrichment functions handled edge cases such as missing extensions or malformed XML. Integration tests exercised the entire pipeline on a set of twenty publicly available rides covering flat urban loops, rolling countryside routes and mountainous climbs. Processing time was measured at approximately 12 s per hour of ride on a consumer laptop when parallelising OSRM queries.

The segmentation engine was validated on the same dataset by comparing its proposed boundaries to manual annotations from three experienced cyclists. Agreement was quantified by the Jaccard index between sets of break points. Across the test rides the average agreement was 0.71, indicating that the greedy merge-cost algorithm captures most of the intuitively meaningful transitions (junctions, surface changes and large bends). Disagreements were analysed and led to refinements of the weighting heuristics, particularly increasing penalties for narrow bridges and offroad

sections.

Training Suitability Scores were evaluated by correlating the zone scores with observed power distributions. For each segment the proportion of time the test riders spent within the target zone was computed from their power recordings. The resulting Spearman correlations were 0.63 for Endurance, 0.58 for Tempo, 0.56 for Threshold, 0.52 for VO_2max and 0.60 for Anaerobic/Neuro-muscular, suggesting that the rulebased scores align moderately well with actual training intensity. Riders were also asked to rate segments on a Likert scale from 1 (poor for the intended interval) to 5 (excellent). The average absolute error between TSS ranks and rider preferences was 0.7 points. While not perfect, this preliminary evaluation indicates that TSS can serve as a useful heuristic for route planning.

Chapter 7

Results and Conclusions

This project set out to determine whether road suitability for structured cycling training could be quantified using only ride data and open mapping sources. The implemented system successfully ingests GPX tracks, performs reliable map matching against OpenStreetMap, enriches tracks with contextual metadata and segments routes using a reproducible twopass algorithm. Evaluation on a diverse set of rides showed that the segmentation engine agrees with human judgement on most meaningful transitions and that the derived Training Suitability Scores correlate with both physiological measures (power zones) and subjective rider ratings. Processing throughput met the nonfunctional requirement of at least 1 000 rides per hour when deployed in parallel on a modest server, confirming the scalability of the pipeline.

The main contribution of this work is the introduction of a transparent, interpretable scoring framework that synthesises geographic features and ride dynamics into actionable insights for cyclists. By prioritising long, uninterrupted segments with favourable grades and surfaces, the system can recommend routes that maximise training stimulus and reduce the frustration of repeated stopstart efforts. Through opensource release of the codebase and data structures, we enable others to reproduce and extend the work. Nevertheless, the results also highlight limitations: the rulebased scoring model cannot capture the nuances of individual training plans, and some segments remain misclassified due to unmodelled factors such as weather, traffic or group dynamics. The conclusion is therefore cautiously optimistic: quantifying training suitability is feasible and valuable, but further refinement and community validation are required for widespread adoption.

Chapter 8

Further Discussions and Research Gaps

Several avenues exist for future improvement. First, the segmentation algorithm currently operates greedily with fixed thresholds; experimenting with dynamic programming approaches or machine learning models could yield more robust boundaries, particularly in complex urban environments. Second, the Training Suitability Score could be enhanced by incorporating additional data sources such as live traffic volumes, air quality indices and weather forecasts, all of which influence the quality of a ride but were outside the scope of this study. Third, expanding the groundtruth dataset with more diverse riders and terrains will enable rigorous crossvalidation and the training of datadriven scoring models. Collecting structured feedback from coaches and athletes-via simple thumbsup/down interfaces or pairwise comparisons would allow us to learn personalised weights for the scoring model.

On the system side, a richer user interface could visualise uncertainty, allow riders to customise the weighting of different factors (e.g., prioritising safety over length), and integrate with existing platforms such as Strava or Komoot via APIs. From a research perspective, future work could explore the broader implications of automated route recommendation on cycling culture, urban planning and public health. Addressing these questions will help ensure that technology remains a tool for empowerment rather than a source of inequity.

Bibliography

- Boeing, G. (2017), ‘Osmnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks’, *Computers, Environment and Urban Systems* **65**, 126–139.
- Coggan, A. R. and Allen, H. (2012), *Training and Racing with a Power Meter*, 2 edn, VeloPress, Boulder, CO.
- Editors of Bicycling (2017), ‘A cyclist’s guide to speaking strava’, *Bicycling Magazine (online)*. Published 31 May 2017, available at <https://www.bicycling.com/news/g20041808/a-cyclists-guide-to-speaking-strava/>.
- Hurley, S. (2020), ‘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].
- Indoor Cycling Association (n.d.), ‘Ask the expert: How much time should you spend in the red zone?’. Accessed: 2025-08-12.
URL: <https://indoorcyclingassociation.com/ask-the-expert-how-much-time-should-you-spend-in-the-red-zone/>
- Information technology Security techniques Information security management systems Requirements* (2013), Standard ISO/IEC 27001:2013, International Organization for Standardization, Geneva, Switzerland.
- Luxen, D. and Vetter, C. (2011), Real-time routing with OpenStreetMap data, in ‘Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS ’11)’, ACM, New York, NY, USA, pp. 513–516.
- Newson, P. and Krumm, J. (2009), Hidden markov map matching through noise and sparseness, in ‘Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS ’09)’, Association for Computing Machinery, Seattle, WA, USA. Camera-ready PDF also available from Microsoft Research: <https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/map-matching-ACM-GIS-camera-ready.pdf>.
URL: <https://dl.acm.org/doi/10.1145/1653771.1653818>

- OpenStreetMap community (2025), ‘Overpass API (OpenStreetMap data query service)’, OpenStreetMap Wiki [Online]. Available: https://wiki.openstreetmap.org/wiki/Overpass_API [Accessed 11-Aug-2025].
- Project OSRM (2021a), ‘Osmr api v5.24.0 — general service parameters (**profile**)’. Documents that the transport **profile** is determined by the Lua profile used at **osrm-extract**.
URL: <https://project-osrm.org/docs/v5.24.0/api/>
- Project OSRM (2021b), ‘Osmr api v5.24.0 — match service’. Describes the `/match` service, sub-trace splitting, and options.
URL: <https://project-osrm.org/docs/v5.24.0/api/#match-service>
- Project OSRM (2025a), ‘Docker image: **osrm/osrm-backend**’. Official Docker image and basic run instructions.
URL: <https://hub.docker.com/r/osrm/osrm-backend/>
- Project OSRM (2025b), ‘Osmr profile example: **profiles/bicycle.lua**’, <https://github.com/Project-OSRM/osrm-backend/blob/master/profiles/bicycle.lua>. Example Lua profile illustrating speed/weighting logic.
- Sage, J. (2019), ‘Ask the expert: How much time should you spend in the red zone?’, Indoor Cycling Association [Online]. Available: <https://indoorcyclingassociation.com/ask-the-expert-how-much-time-should-you-spend-in-the-red-zone/> [Accessed 11-Aug-2025].
- Sharifzadeh, M., Azmoodeh, F. and Shahabi, C. (2005), Change detection in time series data using wavelet footprints, in ‘Advances in Spatial and Temporal Databases (SSTD 2005)’, Vol. 3633 of *Lecture Notes in Computer Science*, Springer, Angra dos Reis, Brazil, pp. 127–144.
- Story, R. and contributors (2023), ‘Folium: Python data. Leaflet.js maps’, <https://python-visualization.github.io/folium/>.
- Systems and software engineering Architecture description* (2011), Standard ISO/IEC/IEEE 42010:2011, International Organization for Standardization, Geneva, Switzerland.
- Telenav (2018), ‘Osmr map matching’, https://github.com/Telenav/open-source-spec/blob/master/osrm/doc/osrm_mapmatching.md. Overview of OSRM map matching with HM-M/Viterbi perspective.
- Topf, J. and Contributors, O. (2023), ‘Osmium: A multi-purpose toolkit for OpenStreetMap data’, Osmium/Osmcode Project [Online]. Available: <https://osmcode.org/> [Accessed 11-Aug-2025].

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  nlohmann::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 }

```

Listing B.9: gpx_example.gpx: Example GPX file showing an example track point and extensions

```

1
2 <trk>
3   <trkseg>
4     <trkpt lat="38.82032" lon="-104.861694">
5       <time>2024-08-06T13:12:58.000Z</time>
6       <ele>1884.8</ele>
7       <extensions>
8         <power>168</power>
9         <tpx1:TrackPointExtension>
10          <tpx1:atemp>20</tpx1:atemp>
11          <tpx1:hr>123</tpx1:hr>
12          <tpx1:cad>65</tpx1:cad>
13        </tpx1:TrackPointExtension>
14      </extensions>
15    </trkpt>
16    ...
17  </trkseg>
18 </trk>

```

Listing B.10: Two-pass segmentation: greedy recursive merge (pseudocode reflecting current implementation)

```

1 struct Section {
2     WayTags tags;           // OSM-derived attributes for this section
3     Range range;           // [start_idx, end_idx) in the matched trace
4 };
5
6 double merge_cost(const Section& a, const Section& b) {
7     double cost = 0.0;

```

```

8
9 // Example heuristics (see weights example):
10 if (a.tags.junction || b.tags.junction) return 1.0;
11 if (a.tags.barrier || b.tags.barrier) return 1.0;
12
13 cost += highway_cost(a.tags.highway, b.tags.highway); // scaled difference
14 cost += cycleway_cost(a.tags.cycleway, b.tags.cycleway); // type change / on->off
15 cost += surface_cost(a.tags.surface, b.tags.surface); // paved <-> unpaved
16 cost += smoothness_cost(a.tags.smoothness, b.tags.smoothness);
17 cost += 0.02 * std::abs(a.tags.maxspeed - b.tags.maxspeed); // speed diff contribution
18
19 // Clamp to [0,1]
20 return std::min(1.0, std::max(0.0, cost));
21 }
22
23 std::vector<Section> two_pass_merge(std::vector<Section> sections, double threshold) {
24 // Pass 1: sections already defined by initial boundary proposal
25 // Pass 2: compute boundary costs between neighbors
26 auto boundary_cost = [&](size_t i){ return merge_cost(sections[i], sections[i+1]); };
27
28 // Greedy recursive merging until no boundary under threshold
29 while (sections.size() > 1) {
30 // Find the cheapest boundary
31 double best = 1.1; size_t k = sections.size(); // invalid by default
32 for (size_t i = 0; i+1 < sections.size(); ++i) {
33 double c = boundary_cost(i);
34 if (c < best) { best = c; k = i; }
35 }
36 if (best >= threshold || k >= sections.size()-1) break;
37
38 // Merge sections[k] and sections[k+1]
39 sections[k].range.end = sections[k+1].range.end;
40 sections[k].tags = merge_tags(sections[k].tags, sections[k+1].tags); // e.g., keep majority or
last
41 sections.erase(sections.begin() + (k+1));
42
43 // Loop continues; boundary costs adapt implicitly via merge_cost()
44 }
45 return sections;
46 }

```

Listing B.11: Example merge-cost weighting rules (illustrative, not final)

```

1 barrier: access control barriers are a merge cost of 1 (guaranteed separation), linear barriers
are mostly 0 cost
2 > block
3 > bollard
4 > cattle_grid
5 > chain

```

```

6   > cycle_barrier
7   > debris
8   > gate
9   > jersey_barrier
10  > log
11  > rope
12  > yes
13
14
15 Highway: change of highway type is a default merge cost of 0.5
16 Ranked from lowest to highest car popularity, the difference in the individual weighting of 2
    segments is multiplied with the default merge cost, giving a maximum of 1
17 > motorway -> motorway should never be hit since weighting is 0, but will always return a result
    of 1 since motorway is bad to go on
18 > trunk 1.9
19 > primary 1.8
20 > secondary 1.5
21 > tertiary 1.0
22 > unclassified 1.0
23 > road 1.0
24 > residential 0.5
25 > living street 0
26 the following highway types always incur a cost of 1 since link roads almost always mean changing
    roads
27 > motorway_link
28 > trunk_link
29 > primary_link
30 > secondary link
31 > tertiary link
32 > any misc road type: {bridleway, footway,service,track,path. etc}
33
34 cycleway: changing cycleway type has a merge cost of .2. Changing onto a cycleway (none to some,
    some to none) has a merge cost of 1
35
36 if highway has attribute junction, it has merge cost of 1
37
38 maxspeed: merge cost is the difference between 2 section's values * 0.02
39
40 mountain_pass = yes means merge cost of 1 (top of mountain)
41
42 smoothness: ranking from smoothest to least smooth. Difference * 0.05 for weighting
43 > excellent = 1
44 > good = .9
45 > intermediate = 0.7
46 > bad = 0.5
47 > horrible = 0.4
48 > very horrible = 0.2
49 > impassable -> automatic 1 merge cost
50

```

51 | surface: going from paved-unpaved gives a merge score of .6, anything else gives score of .1

Appendix C

Directory Structure

Appendix D

Application Documentation