Pipeline Documentation for Thesis

A structured description of the route-matching pipeline

Author: Jack Jibb

August 10, 2025

Scope and purpose

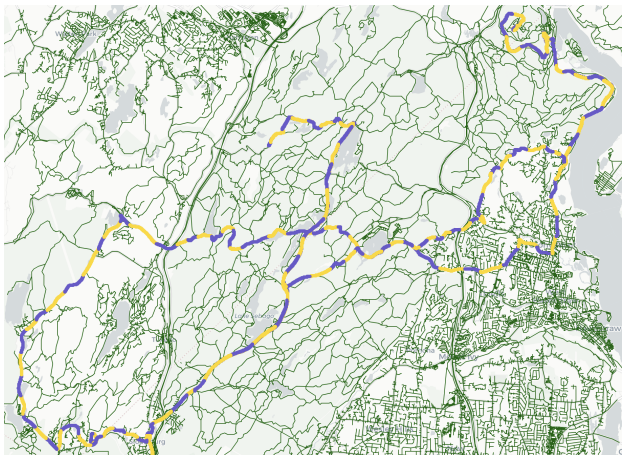
This document describes the end-to-end data processing pipeline contained in `pipeline` directory. The main purpose of the pipeline is to take a GPS[1] track stored in the GPS Exchange file format (GPX), and produce a matched representation of that track on an OpenStreetMap road network. The result is an ordered sequence of matched points and metadata that can be visualised and further analysed. The documentation follows established guidelines for software user documentation as described in ISO/IEC/IEEE 26514[2]. The pipeline is a component of the larger route segmentation system and is designed to be deployed as part of a web application backend. The code is already partly containerised using Docker, and the exposed scripts will eventually become HTTP endpoints when integrated into a web service. Recent development work has added native support for *batch processing*: instead of accepting only a single GPX file, the pipeline reads all files from a configurable source directory and processes them concurrently. This change means that multiple rides can be matched to the map in one run, and each input file produces its own matched output and visualisation. When ported to a web application, the same core code will be reused behind API endpoints that interact with the different pipeline stages. (Topografix 2023)

1 Overview of the pipeline

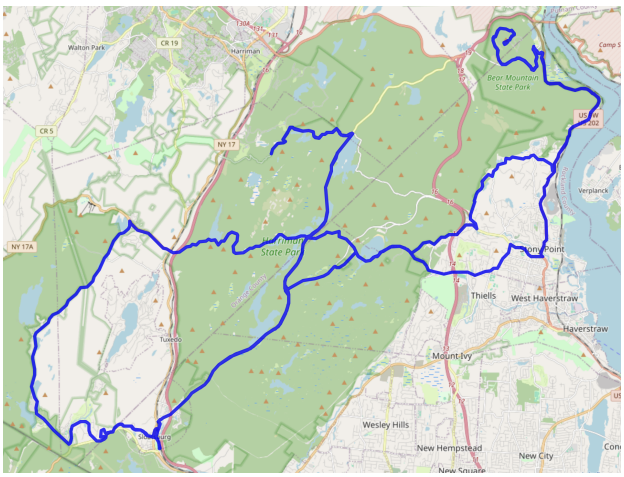
The pipeline is orchestrated by the Bash script `run_pipeline.sh`, contained in the `code/pipeline` directory. It reads settings from a YAML configuration file (`settings.yml`) and runs each component. The result of the pipeline is a matched GeoJSON route object that follows the path that the GPX file is suggesting. This removes GPS error from noise, and provides a structured network that can be used in analysis(Saki & Hagen 2022). Figures 1a to 1d show the comparison between mapped input GPX trackpoints and output GeoJSON route maps. The pipeline stages are listed below; see Figure 2 for a visual depiction. Each step logs its progress to the console, and the resulting artefacts are stored in the `data/` sub-directory.

[1] GPS: Global Positioning System.

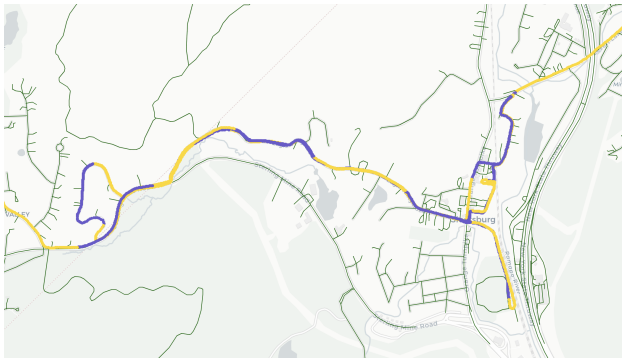
[2] `https://cdn.standards.iteh.ai/samples/77451/e161501288fd44f88c6fdd0f6e46b017/ISO-IEC-IEEE-26514-2022.pdf`



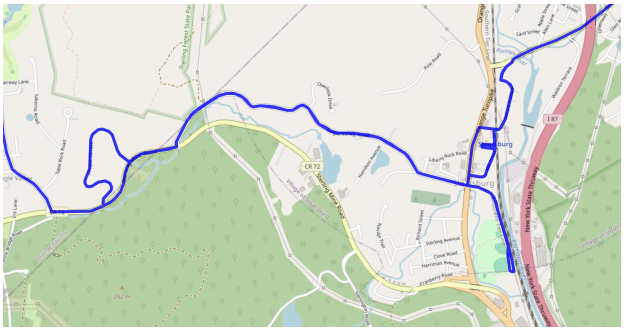
(a) Raw GPX track of whole file



(b) Matched Route to whole file



(c) A detailed subsection of GPX trackpoints



(d) The same area, in GeoJSON route form

Figure 1: Comparison of raw GPX tracks and OSRM-matched routes: (a) GPX Track 1, (b) Route 1, (c) GPX Track 2, (d) Route 2.

1.1 Description of pipeline processing steps

1. **Configuration loading**—The script reads `settings.yml` to load initial settings for the pipeline. This includes (`gpx_source`) where the pipeline looks for .gpx files, and (`osm_files`) the directory where downloaded OpenStreetMap `.osm` files are stored. It also loads some other options, such as the size of the "chunks" (defined later) that the program splits each GPX file into.

2. **Environment setup**—A Python virtual environment is created (if it does not already exist) and dependencies from `requirements.txt` are installed. This isolates the pipeline from the system environment, and any future upgrades or dependency updates can be managed through this step.

3. **Data extraction and map download**—The shell script[3] invokes the Python module `prepare_map.py`. In the current version the input to this module is a *directory* of GPX files. The script enumerates all `.gpx` files in the directory and, using a thread pool, performs the following steps for each file:

   - Parses the GPX track to obtain a sequence of latitude/longitude/timestamp points
   - Computes a bounding box around the full GPS track and downloads the OpenStreetMap .osm file for the bbox from the Overpass API, saving it as `<gpx_name>.osm` in the directory defined by `osm_files`. (Overpass API 2025)
   - Converts the track points into one or more OSRM GeoJSON objects and writes them to `data/temp/<gpx_name>`. Each JSON file represents a "chunk"[4] of at most `chunk_size`

[3]The call is performed using Python's `subprocess` module within `run_pipeline.sh`; for brevity the exact command is not reproduced here. See Appendix (Code Section) for the full shell script

[4]Chunking is necessary to avoid exceeding URL length limits when later calling the OSRM `/match` API, which accepts only HTTP GET requests.

points (a value of 0 in `settings.yml` disables chunking).

4. **Conversion of OpenStreetMap data to Protocolbuffer Binary Format (.pbf)**—Within a Docker container, the `osmium` tool converts the `.osm` files into the more compact `.pbf` format (Osmium Tool 2025). When multiple extracts have been downloaded (one per GPX file), they are merged into a single `merged.pbf` via the scripts located in `osmium/scripts`. The resulting file is renamed `map.pbf` and stored in `data/osrm_map/`, and the individual `.pbf` fragments are removed.

5. **Graph preparation with Docker**—Docker Compose is used to containerize the OSRM[5] backend. The `osrm-prep` service preprocesses the `map.pbf` using the custom cycling profile in `docker/osrm/profile.lua` to build the routing graph using contraction hierarchies(Project OSRM 2025). Once preprocessing completes, the `osrm-server` service is launched in detached mode, exposing the `/match` endpoint on port 5000, ready for route matching.

6. **Route matching**—For each GPX file, the script invokes `batch_route_calc.py` on its corresponding chunk directory. This module reads the OSRM JSON chunks in their directories and performs parallel requests to the OSRM `/match` API using a thread pool. Each request uses a *dynamic radius algorithm* that increases the search radius per track point when the heading (direction) of the track has a high variance. This gives more context for `/match`'s algorithm to decide whether to turn off a road or not(Newson & Krumm 2009). The results of successful matches are saved as `result_*.json` files in `data/results/<gpx_name>`, while failures are saved to a `gap/` subdirectory for later gap filling using the `/route`[6] endpoint.

7. **Merging**—Once all chunks for a GPX file have been processed, `merge_routes.py` recursively combines individual matching results into a single route. The merge logic decodes the geometries, concatenates all data arrays in the file, and computes aggregate distance and duration values. Merging is multi-threaded per recursion instance, and produces a single JSON file named after the input GPX file (e.g., `<gpx_name>.json`) saved directly in `data/results`. After merging, the per-file result chunks and their directories are removed.

8. **Visualisation**—Finally, `plot_merged.py` iterates over the merged JSON files in `data/results`, creating a Folium map for each route (Folium Developers 2025). The purpose of the map is mostly for debugging, as the main goal of the pipeline is to generate a continuous route that matches the GPX route as closely as possible. Each map can optionally overlay the raw GPS points for accuracy checking. The outputs are saved as `interactive_map_<gpx_name>.html` per input file, which can be opened in a web browser to inspect the matched route. This HTML can be served once the program is implemented into a web back-end, as a debug option.

9. **Clean up**—Temporary files (the intermediate JSON chunks, fragment `.pbf` files for individual GPX files and Docker volumes) are deleted. The virtual environment is deactivated and the OSRM containers are stopped and removed using Docker Compose. The script leaves the virtual environment directory intact so that subsequent runs reuse the existing dependencies.

[5] OSRM: Open Source Routing Machine.

[6] The `/route` endpoint calculates the path between two coordinates based on criteria such as distance or duration

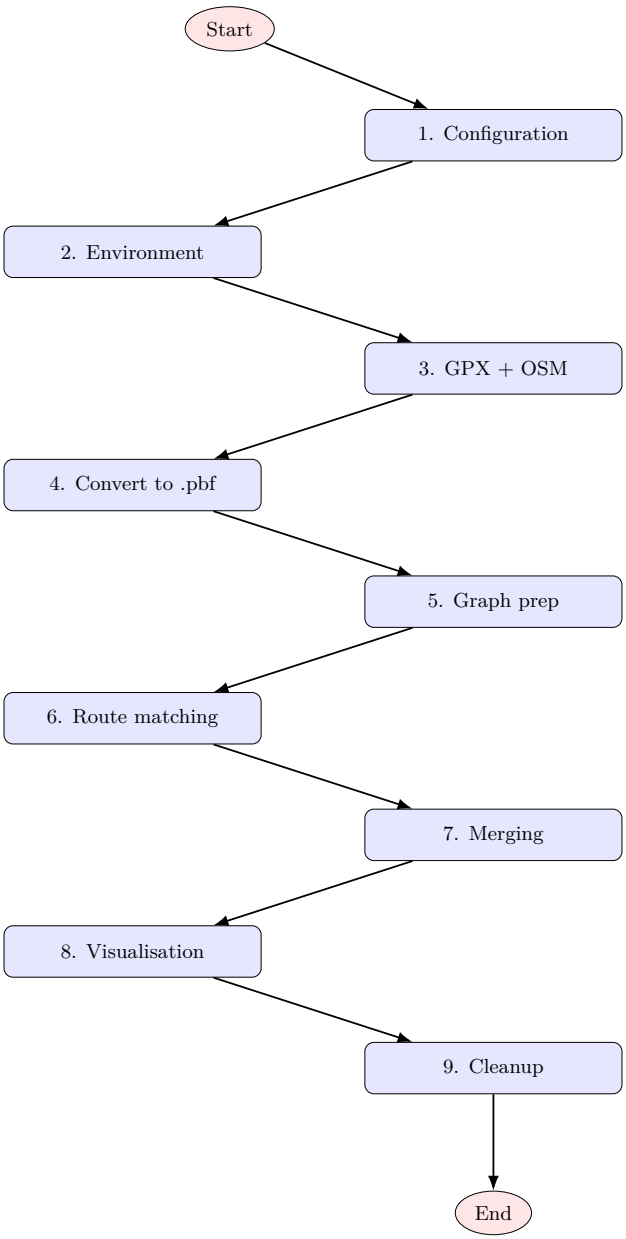


Figure 2: Pipeline flowchart for the GPX to OSRM-matched route processing system.

2 Python Module Breakdown

2.1 GPX Utilities (`gpx_utils.py`)

This module provides low-level helpers for GPX parsing, bounding-box computation, JSON I/O, and OSM downloads.

load_gpx(path) Opens and parses a GPX file using `gpxpy.parse()`, returning a `GPX` object with tracks, segments, and points. *Citation:* (Krajina 2023)

extract_data_from_gpx(gpx) Iterates through `gpx.tracks` → `.segments` → `.points`, building a list of dictionaries with:

- `lat`, `lon`: coordinates
- `elv`: elevation (or `None`)
- `time`: ISO-formatted timestamp

- any GPX `pt.extensions`, flattened into key–value pairs. Including but not limited to: Power, Cadence, Heartrate, and Temperature.

This list will be the heart of the categorization section of the program, as it contains all rider-specific metadata.

bounding_box_from_data(data) Given a list of point dicts, extracts all latitudes and longitudes, computes:

$$\text{south} = \min(\text{lats}), \quad \text{north} = \max(\text{lats}), \quad \text{west} = \min(\text{lons}), \quad \text{east} = \max(\text{lons}),$$

and returns the bbox tuple $(\text{south}, \text{west}, \text{north}, \text{east})$.

bbox_contains(outer, inner) Checks that the inner bbox lies fully within the outer by comparing numeric bounds:

$$\begin{aligned} \min(\text{lat}_{\text{inner}}) &\geq \min(\text{lat}_{\text{outer}}) \ \wedge \\ \min(\text{lon}_{\text{inner}}) &\geq \min(\text{lon}_{\text{outer}}) \ \wedge \\ \max(\text{lat}_{\text{inner}}) &\leq \max(\text{lat}_{\text{outer}}) \ \wedge \\ \max(\text{lon}_{\text{inner}}) &\leq \max(\text{lon}_{\text{outer}}) \end{aligned}$$

Returns `True`/`False`.

load_json(file_path) Reads and parses a JSON file via `json.load()`.

write_json(points, path) Ensures the target directory exists (`os.makedirs(..., exist_ok=True)`), then writes `points` with `json.dump(separators=(",", ":"), ensure_ascii=False)` for compact output.

download_osm_file(bbox, output_path) Constructs an Overpass "map" query URL of the form `https://overpass-api.de/api/map?bbox=west,south,east,north`, fetches via `requests.get()`, and writes the raw OSM XML to disk. *Citation:* (Overpass API 2025)

This module will also be used in the segmentation algorithm, but the relevant functions will be described in the Segmentation Algorithm documentation.

2.2 Data Extraction and Map Download (`prepare_map.py`)

The `prepare_map.py` module handles reading raw GPX tracks and fetching the corresponding OpenStreetMap extracts. It defines four primary functions:

parse_args() Uses Python's `argparse` library to define and parse the command-line interface (Python Software Foundation 2025*a*).

- `gpx_file_path`: Input directory containing one or more `.gpx` files.
- `osm_output_path`: Directory in which to save downloaded `.osm` files.
- `radius`: Search radius (in metres) used for each track point.
- `chunk_size`: Maximum number of GPS points per OSRM JSON chunk (0 = no chunking).

thread_processes(gpx_file, output_path, chunk_dir, radius, chunk_size) Executed concurrently for each GPX file via `ThreadPoolExecutor` (Python Software Foundation 2025*b*):

1. **Load and parse GPX:** Calls `load_gpx(gpx_file)` and `extract_data_from_gpx(gpx)` from the `gpxpy` library to obtain a list of {lat, lon, timestamp} tuples (Krajina 2023).
2. **Prepare output paths:** Derives `gpx_name` via `os.path.splitext` and ensures `output_path` exists using Python's `os` module (Python Software Foundation 2025*c*).
3. **Chunk for OSRM:** Invokes `points_to_osrm_json(points, chunk_dir, gpx_name, radius, chunk_size)` (in `utils/osrm_utils.py`) to split the track into one or more JSON payloads ready for OSRM's `/match` endpoint.
4. **Fetch OSM if needed:** If `<gpx_name>.osm` does not already exist under `output_path`, calls `get_single_osm_from_gpx()`, otherwise logs a warning and skips.

get_single_osm_from_gpx(gpx_file, points, gpx_name, output_path) Retrieves OSM data via Overpass API:

- **Compute bounding box:** Uses `bounding_box_from_data(points)` (from `utils/gpx_utils.py`) to derive the min/max latitude and longitude.
- **(Re)load GPX points:** Calls `load_gpx()` and `extract_data_from_gpx()` again to ensure consistency.
- **Download OSM file:** Invokes `download_osm_file(bbox, full_output_path)`, which constructs an Overpass API query and writes the resulting `.osm` XML to disk.

main() Orchestrates the module workflow:

- Parses CLI arguments via `parse_args()`.
- Scans `gpx_file_path` for `.gpx` files using `os.listdir()` and `os.path.isfile()` (Python Software Foundation 2025*c*).
- Determines `MAX_THREADS` as min(32, number of GPX files).
- Creates a `ThreadPoolExecutor(max_workers=MAX_THREADS)` and submits `thread_processes()` tasks for each GPX file.
- Uses `as_completed()` to wait for all threads to finish before exiting.

This design parallelizes I/O and network operations, which are the most time-costly operations.

3 Configuration options

The pipeline reads configuration values from `settings.yml`. Table 1 summarises the available keys and their meanings. Ensure that file paths refer to existing locations and that chunk sizes remain within reasonable limits to avoid exceeding URL length limits when calling the OSRM API.

Table 1: Summary of configuration keys in `settings.yml`.

Key	Description
`gpx_source`	Directory containing one or more GPX files. All files with the `.gpx` extension in this directory will be processed. Example: `data/input`.

Continued on next page

Table 1 continued from previous page

Key	Description
`chunk_size`	Maximum number of points per OSRM JSON file. A value of 0 disables chunking. Large values may exceed the URL length limit of the OSRM API; the script enforces a maximum of 200.
`radius`	Initial search radius (in metres) used when matching points to the road network. The dynamic radius algorithm may adjust this value on a per-point basis. Example: `5`.
`dynamic_radius_window`	Size of the initial sliding window used when computing dynamic radii. Larger values smooth the radius changes over more points; smaller values react more quickly to heading changes.

4 Output JSON file structure

When the OSRM map–matching step completes successfully, each GPX track chunk is written to disk as a JSON file (`result_*.json` in `data/results/<gpx_name>`). After all chunks for a file have been matched, `merge_routes.py` concatenates the results into a single JSON object named after the input file (e.g. `<gpx_name>.json`). Understanding the structure of these JSON files is important for downstream analysis and for the enrichment step that augments them with sensor data. This section summarises the key top–level keys and nested objects you will encounter.

4.1 Top-level keys

- **code** – A short status string returned by OSRM. A value of `"Ok"` means that all points in the input chunk were matched successfully. If matching fails for any point, the code may contain error descriptions (e.g. `"NoSegment"`).
- **tracepoints** – An array with one entry per input point. Each element is either a JSON object describing the matched road coordinate or `null` if the point could not be matched.
- **matchings** – A list of matched routes. Although the pipeline normally produces one matching per chunk, OSRM may occasionally split the route into multiple matchings when long gaps exist. Each matching contains aggregated route metrics and the geometry of the matched path.

4.2 Tracepoints and GPX enrichment

The `tracepoints` array aligns one–for–one with the original coordinates sent to OSRM. For a matched point the tracepoint object has the form

```
{
  "location": [lon, lat],
  "name": "",                // optional road name or empty string
  "matchings_index": i,      // index of the matching this point belongs to
  "waypoint_index": j,       // position of this point in the matching
  "alternatives_count": k,   // number of alternative candidates considered
  "classes": [ … ],          // optional OSM road classes

  "gpx_points": [{
    "time": "YYYY-MM-DDTHH:MM:SSZ",
    "elv": h,                // elevation in metres or null
    "extensions": {
```

```
      "hr_bpm": <heart rate>,
      "cad_rpm": <cadence>,
      "power_w": <power>,
      "speed_mps": <speed>,
      "temp_c": <temperature>,
      "alt_m": <altitude>,
      "grade_pct": <road grade>,
      ...
    }
  }, ... // multiple gpx points per tracepoint. contains points up until the next tracepoint.
  ]
}
```

The nested `gpx` object is an addition introduced in this project. It stores metadata extracted from the original GPX file:

- `time` and `elv` carry the timestamp and elevation of the tracepoint.
- `extensions` is a dictionary of canonical sensor values normalised according to `config/extensions_map.json` The canonical keys (`hr_bpm`, `cad_rpm`, `power_w`, `speed_mps`, `temp_c`, `alt_m` and `grade_pct`) are derived from the raw GPX extension tags via unit conversion and type coercion. Missing values are omitted from this dictionary.

Null entries in the `tracepoints` array indicate gaps where OSRM could not match a GPS coordinate to the network. These are carried through to the merged file so that you can identify portions of the track that require gap-filling via the `/route` API.

4.3 Matchings and their legs

Each entry in the `matchings` list represents a continuous segment of the route. A matching has the following important members:

- **confidence** – A floating-point value in $[0, 1]$ that quantifies OSRM's certainty that the matching is correct. Values close to 1 indicate that the GPS data fitted the road network well.
- **distance** and **duration** – Aggregate length (in metres) and travel time (in seconds) of the matching. These are sums over all legs.
- **geometry** – An encoded polyline string representing the path of the matching. It can be decoded into a list of coordinates using standard polyline libraries.
- **legs** – A list of legs; one leg for each pair of consecutive waypoints. Legs contain fine-grained information such as the line geometry between two matched points, a summary with per-leg distance and duration, and optionally annotations describing speed and elevation. When the pipeline merges results the leg lists from all chunks are concatenated.

Each leg has the optional datapoint "annotation", so that each node ID can be pulled, and matched against the SQLite indexed database(SQLite 2025) to get the respective way ID. The way IDs are then implemented into the leg object in the form of "runs". Each run is a semi-hashed list of tuples: (way_id, direction, edge_count), where the direction is either 1 or -1, depending on if the match in the database is forward or backwards. The edge_count variable counts the number of consecutive nodes that fall along the way.

Construction

1. Read `leg.annotation.nodes` and form consecutive pairs.
2. For each pair, look up the corresponding edge in the SQLite way index(SQLite 2025) to obtain `(way_id, dir, length)`.

3. Group adjacent pairs with the same `(way_id, dir)` and emit one object per group.

Fields in each run

- `way_id` — OpenStreetMap way identifier.
- `dir` — travel direction relative to the way's stored orientation (e.g., `1` or `-1`).
- `edge_count` — number of node–to–node edges collapsed into the run.
- `length_m` (optional) — total length of the run in metres (present when run with `--with-metrics`).

```
{
  "annotation":{
    "nodes":[             // ordered list of node IDs the leg travels through
      0123456789,
      1234567890,
      ...
    ],
    "distance":[          // Per-edge length (in metres) between consecutive nodes
      15.385,
      8.0151234,
      ...
    ]
  },
  "distance":29.8,        // total leg length in metres
  "duration":2.3,         // OSRM-calculated travel time from weightings in profile.lua
  "weight":2.3,           // OSRM cost for the leg. Is equal to duration in our case
  "runs":[                // the pipeline compression of consecutive edges
    {
      "way_id": 98765432, // Way that the path is on
      "dir": 1,           // direction of travel, 1 is forwards, -1 is backwards
      "edge_count":5      // the run covers 5 edges in the way
    }, ...
  ],
  "total_edges":17        // Number of node-to-node edges inside the leg..
}, ...
```

Downstream modules (e.g. the segmentation and analysis code) can iterate over `matchings` and their `legs` to compute cumulative distance, extract elevation profiles, or associate sensor readings with segments of the road. The enrichment process described earlier ensures that each tracepoint contains not only its matched map location but also the underlying GPX metadata for comprehensive analysis.

5 Running the pipeline

This section provides a step-by-step procedure for executing the pipeline. It assumes that both Docker and Python 3.10 or higher are installed on the host system.

1. **Clone the repository** if you have not already done so. In a terminal, run:
   `git clone https://github.com/jibb34/MScProject && cd MScProject/code/pipeline`
   This will put you into the main pipeline directory, and you can see all the files and subdirectories with: `ls -la .`

2. **Inspect and modify `settings.yml`** to set `gpx_source` to the directory containing your GPX files and adjust parameters as needed. Do not exceed a chunk size of 200 points. If necessary, increase the search radius if the file contains a lot of GPS noise

3. **Execute the pipeline** by running the shell script:

   ```
   bash ./run_pipeline.sh
   ```

   The script will create a Python virtual environment, install dependencies, download the map, build the OSRM graph and perform route matching. Progress messages are printed to the console. While the script is running, do not attempt to modify any files or paths in the main `pipeline` directory

4. **View the results**. After the script completes, open all `interactive_map_<gpx_name>.html` files (one file per input GPX) in a web browser. The file is located in the main directory. You can also inspect the merged JSON files for a structured representation of the matched route. They are located in `./data/results`

5. **Clean up**. The script performs clean-up automatically. Temporary files are removed and containers are shut down. If you wish to start a clean run, you may delete the `.venv` directory to remove the python environment, and you should delete all html files and GeoJSON files with this command:
   `rm -rf .venv ./data/results/* ./*.html`

6 Error handling and warnings

The pipeline contains the following safeguards:

- If `settings.yml` is missing or lacks required keys, the script exits immediately with an error message.
- The chunk size is validated to ensure it does not exceed 200 points. Exceeding this limit may produce URLs that are too long for the OSRM server to process; a fatal error is raised if the limit is broken.
- Requests to the OSRM server are wrapped in exception handling. If a request fails (e.g., network error), an empty result is produced and saved to a `gap/` directory for later gap filling. This will be implemented later for more robust behaviour if necessary. (So far, no empty requests have occurred outside of programmatic errors.)
- During merging, the script checks for missing geometries or mismatched code values and reports errors if encountered. Merging proceeds even when some segments cannot be combined, but is flagged as incomplete.
- If the Folium plotting script cannot find any match result files, it raises an explicit `FileNotFoundError`.

References

Folium Developers (2025), 'Folium: Python library for interactive maps', `https://folium.readthedocs.io`. Accessed August 2025.

Krajina, T. (2023), '*gpxpy* (version 1.6.2) [python library]', `https://pypi.org/project/gpxpy/`. Accessed August 2025.

Newson, P. & 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)', Seattle, WA, USA.

Osmium Tool (2025), 'Multi-purpose osm data manipulation toolkit', `https://osmcode.org`. Accessed August 2025.

Overpass API (2025), 'Openstreetmap data extractor', https://overpass-api.de. Accessed August 2025.

Project OSRM (2025), 'Open source routing machine (osrm)', http://project-osrm.org. Accessed August 2025.

Python Software Foundation (2025*a*), 'argparse — parser for command-line options (python documentation)', https://docs.python.org/3/library/argparse.html. Accessed August 2025.

Python Software Foundation (2025*b*), 'concurrent.futures — launching parallel tasks (python documentation)', https://docs.python.org/3/library/concurrent.futures.html. Accessed August 2025.

Python Software Foundation (2025*c*), 'os — miscellaneous operating system interfaces (python documentation)', https://docs.python.org/3/library/os.html. Accessed August 2025.

Saki, S. & Hagen, T. (2022), 'A practical guide to an open-source map-matching approach for big gps data', *SN Computer Science* **3**(415).

SQLite (2025), 'Sqlite home page', https://sqlite.org. Accessed August 2025.

Topografix (2023), 'Gpx: the gps exchange format (version 1.1)', http://www.topografix.com/gpx.asp. Accessed August 2025.