Pipeline Documentation for the MScProject

A structured description of the route-matching pipeline

Author: MScProject Team

August 5, 2025

Scope and purpose

This document describes the end-to-end data processing pipeline contained in `pipeline` directory. The primary goal of the pipeline is to take a GPS[1] track (stored in the GPX file format) 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 ("Systems and software engineering–requirements for designers and developers of user documentation"). The pipeline is a component of the greater system, and will be eventually ported to a web application backend. This port will use the same code, but some changes will need to be made, such as containerizing more of the pipeline, and creating endpoints that interact with different pipeline stages. While the current iteration of the pipeline does not support multiple files, this will be added as a feature.

1 Overview of the pipeline

The pipeline is orchestrated by the Bash script `run_pipeline.sh`, which reads settings from a YAML configuration file and launches the various components. The steps outlined below summarise the high-level flow; see Figure **??** for a visual depiction. Each step logs its progress to the console, and the resulting artefacts are stored in the `data/` directory.

1. **Configuration loading**—The script reads `settings.yml` to determine the input GPX file, the desired output file names and operational parameters (e.g., chunk size and search radius).

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 ensures reproducibility.

3. **Data extraction and map download**—The shell script invokes[2] the Python module `prepare_map.py`. This module parses the GPX file, calculates a bounding box around the track and downloads the corresponding OpenStreetMap data from the Overpass API [1]. It then converts the sequence of track points into multiple `.json` files, in GeoJSON format. Each file is defined as a "chunk", and the size of each chunk can be defined

[1] GPS: Global Positioning System.

[2] 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

in `settings.yml`. The GPX points are split into multiple files to avoid exceeding URL length limits when communicating with the OSRM server, since the engine only supports **GET** requests, and not **POST** with a JSON body.

4. **Conversion of OpenStreetMap data to Protocolbuffer Binary Format (.pbf)**—Within a Docker container, the `osmium` tool converts the downloaded `.osm` file into the more compact `.pbf` format. If multiple `.osm` files exist, they are merged. The resulting `map.pbf` is stored in the volume `data/osrm_map/`.

5. **Graph preparation**—A second docker container runs the OSRM[3] backend. It performs preprocessing actions on the `.pbf` map to build the necessary routing graph, using the cycling profile in `docker/osrm/profile.lua`. This produces a collection of `.osrm` files that support fast, accurate map matching.

6. **Route matching**—The script then calls the Python module `batch_route_calc.py`, which performs threaded requests to the OSRM `/match` API. Each JSON chunk is submitted to the server with an adaptive search radius that grows as route heading variability increases. This is because turning off a twistier road would need a larger window of observation to verify that a turn is occuring because of a road change and not just inherent twistiness. The results are saved as `result_*.json` files. Any chunks that were not able to be processed for any reason are saved as empty JSON files in a separate directory. Later these files will be filled with the /route[4] endpoint, as a fallback.

7. **Merging and visualisation**—Once all chunks have been matched, `merge_routes.py` recursively combines the individual matchings into a single route. Each recursion iteration is multi-threaded to improve performance. The merged result is stored as `merge.json`.

8. **Visualisation**—A verification step, `plot_merged.py` takes the resulting JSON file, and visualises it using `folium`. The module creates an interactive HTML map illustrating the matched route and (optionally) the raw points. This map can be opened in a web browser. Visualisation is to make sure that the script successfully maps the route and doesn't have any issues.

9. **Clean-up**—Temporary files (intermediate JSON chunks, result files, Docker volumes) and the virtual environment are deleted. The script stops and removes the OSRM containers and resets the working directory to a clean state.

2 Component summary

The pipeline comprises several scripts and modules. Table 1 summarises their roles and highlights their main functions.

[3]OSRM: Open Source Routing Machine.

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

Table 1: Summary of key modules in the pipeline. The descriptions focus on the primary purpose of each module and list notable functions exposed for reuse.

Module/file	Purpose	Notable functions/scripts
`run_pipeline.sh`	Orchestrates the entire pipeline, loading configuration values from `settings.yml`, creating a virtual environment, invoking the Python scripts, running Docker containers for `osmium` and OSRM, and performing clean-up.	Shell script; calls `python3 source/prepare_map.py`, `python3 source/batch_route_calc.py`, `python3 source/merge_routes.py` and `python3 source/plot_merged.py`.
`prepare_map.py`	Parses the GPX file, extracts track points, calculates a bounding box and downloads the corresponding OpenStreetMap extract from the Overpass API. Converts the track points into OSRM JSON files, supporting optional chunking.	`parse_args()`, `main()`; uses `utils.gpx_utils` and `utils.osrm_utils`.
`batch_route_calc.py`	Reads the OSRM JSON chunks and performs parallel map matching requests to a local OSRM server. Utilises a dynamic radius algorithm to adapt to heading changes, writes the matching results to `result_*.json` and handles failures gracefully.	`get_osrm_match()`, `format_list()`, `main()`
`merge_routes.py`	Merges multiple matched segments into a single route using a recursive "tree merge" algorithm. Combines distances, durations and geometries while maintaining order.	`merge_two()`, `tree_merge_routes()`, `main()`
`plot_merged.py`	Generates an interactive map (HTML) of the merged route using Folium. Alternates colours for different segments and optionally overlays raw points for comparison.	`main()`; uses `folium`, `itertools.cycle`
`utils/gpx_utils.py`	Provides helper functions for reading GPX files, computing bounding boxes, downloading OSM extracts and serialising data to JSON.	`load_gpx()`, `bounding_box_from_data()`, `download_osm_file()`, `extract_data_from_gpx()`
`utils/osrm_utils.py`	Defines functions to convert GPX points into OSRM JSON payloads, calculate adaptive radii based on heading changes and remove spurious segments. Also includes helpers for reading OSM bounds.	`points_to_osrm_json()`, `compute_dynamic_radius()`, `prune_spurs()`

Continued on next page

Table 1 continued from previous page

Module/file	Purpose	Notable functions/scripts
`docker/docker-compose.yml`	Describes the Docker services used for conversion and map matching. Defines an `osmium` service for converting `.osm` files to `.pbf`, an `osrm-prep` service for building the routing graph and an `osrm-server` service that exposes the `/match` endpoint on port 5000.	YAML file; run with `docker compose`

3 Configuration options

The pipeline reads configuration values from `settings.yml`. Table 2 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 2: Summary of configuration keys in `settings.yml`.

Key	Description
`gpx_file`	Path to the GPX file containing the raw GPS track. The file must be in standard GPX format with valid latitude, longitude and optional timestamp values. Example: `data/input/test.gpx`.
`osm_output`	Path where the downloaded OpenStreetMap extract is stored. The file extension should be `.osm`. Example: `data/osm_files/map.osm`.
`json_directory`	Directory in which temporary OSRM JSON chunks are written. This directory will be created if it does not exist and will be deleted during cleanup. Example: `data/temp`.
`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. Example: `100`.
`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 sliding window used when computing dynamic radii. Larger values smooth the radius changes over more points; smaller values react more quickly to heading changes. Example: `10`.

4 Running the pipeline

This section provides a step-by-step procedure for executing the pipeline. It assumes that 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 <repository_url> && cd MScProject/code/pipeline`

2. **Inspect and edit `settings.yml`** to point to your GPX file and adjust parameters as needed. Do not exceed a chunk size of 200 points. If necessary, increase the search radius for sparse tracks.

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.

4. **Monitor Docker containers**. While the OSRM services are running, you can check their status with `docker ps` and inspect logs with `docker compose logs`. The OSRM server listens on `localhost:5000` and provides a health check endpoint at `/health`. If the matching step fails, ensure that the server is healthy and that ports are not blocked by a firewall.

5. **View the results**. After the script completes, open `interactive_map_merged.html` in a web browser. The file is located in the `data/` directory. You can also inspect `merge.json` for a structured representation of the matched route.

6. **Clean up**. The script performs clean-up automatically. Temporary files are removed and containers are shut down. If you wish to rerun the pipeline, ensure that the working directory is clean and that the virtual environment has been removed.

5 Error handling and warnings

Several safeguards are built into the pipeline:

- 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.
- 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.
- If the Folium plotting script cannot find any match result files, it raises an explicit `FileNotFoundError`.

6 Suggested figures and photographs

Figures enhance understanding of the pipeline. The following photographs or screenshots can be inserted once you run the program. Save each image as a `.png` file in the `figures/` directory and reference it using `\includegraphics` in your LaTeX document.

- **Pipeline flowchart**—A block diagram showing the eight steps described in Section 2. Include the creation of the virtual environment, map download, OSRM preparation, matching and merging. Place this diagram here as Figure **??** and caption it "Overview of the route-matching pipeline."
- **Bounding box illustration**—A map snippet or screenshot from OpenStreetMap showing the bounding box used for extracting the map data. Highlight the GPS points within the box.
- **OSRM server status**—A screenshot of the terminal or browser showing the OSRM health endpoint returning a `200 OK` status. This confirms that the server is running correctly.
- **Sample matched route**—A screenshot of the interactive map generated by `plot_merged.py`, displaying both the merged route and the raw GPS points. Use different colours to distinguish matched segments.
- **Dynamic radius plot**—If you instrument the code to record dynamic radii, plot a simple chart showing how the radius varies along a track segment. This helps readers understand the purpose of the adaptive radius algorithm.

References

References

[1] Overpass API, *OpenStreetMap data extractor*, `https://overpass-api.de` (accessed August 2025).

[2] ISO/IEC/IEEE 26514:2008, *Systems and software engineering–Requirements for designers and developers of user documentation.*