BÄRNHÄCKT

Team MapUnfold
SBB Map to Text

Technische Informationen für die Jury

Technische Informationen für die Jury

Aktueller Stand des Sourcecodes

- Link zu Github Repository

Ausgangslage

- Worauf habt ihr euch fokussiert?
- Welche technischen Grundsatzentscheide habt ihr gefällt?

Technischer Aufbau

- Welche Komponenten und Frameworks habt ihr verwendet?
- Wozu und wie werden diese eingesetzt?

Implementation

- Gibt es etwas Spezielles, was ihr zur Implementation erwähnen wollt?
- Was ist aus technischer Sicht besonders cool an eurer Lösung?

Abgrenzung / Offene Punkte

- Welche Abgrenzungen habt ihr bewusst vorgenommen und damit nicht implementiert? Weshalb?

Executive Summary

The challenge focuses on transforming spatial and infrastructure data into accessible, narrative text for public transport users. The key target group includes passengers with visual, cognitive, or mobility impairments who are unable to interpret static maps.

Our approach centered on:

Robust data extraction from heterogeneous sources (SBB Open Data, OpenStreetMap).

Text generation pipelines capable of producing multilingual (DE, EN, FR, IT) route and facility descriptions.

Designing a modular system architecture to support experimentation with different AI models and prompting strategies.

Key architectural decisions:

1. A command-line tool as the orchestration layer for reproducibility and automation.

2. Python as the core technology, ensuring compatibility with state-of-the-art LLM frameworks and libraries.

3. A lightweight database backend to persist generated descriptions and metadata.

4. A minimal frontend prototype for querying and visualizing results.
The solution architecture integrates heterogeneous data sources with AI-driven text generation:

Data Sources:

.parquet files were evaluated but discarded due to invalid coordinate formats.

.geojson datasets provided valid station and facility coordinates.

swiss.osm.pbf (OpenStreetMap extract) enabled access to nationwide routing and spatial context.

Processing Workflow:

1. Import and normalize station metadata from SBB DiDok datasets.
2. Cross-reference extracted entities (elevators, ramps, stairs, ticket machines, etc.) with OSM geometries.
3. Structure enriched data into a machine-readable schema.
4. Submit structured context to LLMs for multilingual text synthesis.
5. Persist results in a relational database for frontend consumption.

Technology Stack:

- Python 3.x (ETL pipelines, LLM integration, orchestration).
- Geospatial libraries (GeoPandas, PyProj) for spatial analysis.
- LLaMA 2 and API-based LLMs for text generation.
- SQLite/PostgreSQL for structured storage.
- Simple HTML/JS frontend for visualization.

Implementation Highlights:

- LLMs struggled to generate both long, detailed descriptions and valid JSON simultaneously. Schema enforcement frequently broke when outputs were lengthened.
- Multilingual generation exposed inconsistencies: partial outputs mixed German and English. Generating consistent outputs across DE/EN/FR/IT remains unsolved.
- Prompting experiments revealed trade-offs between conciseness and fidelity. When forced to produce longer outputs, models often ignored structural constraints.
- We hardcoded logic for accessibility routing (e.g., preferring ramps over stairs for wheelchair users). This demonstrated feasibility but highlighted difficulties: instructions sometimes routed users inefficiently (e.g., via a ramp even when an elevator was closer).

Technically Notable Aspects:

- Integration of geospatial OSM data with AI-driven natural language generation.
- Attempted use of structured JSON prompts to enforce machine-readable output.
- Early evaluation of schema-enforcing protocols (e.g., Model Control Protocol, MCP) for future reliability.

Implementation Highlights:

- LLMs struggled to generate both long, detailed descriptions and valid JSON simultaneously. Schema enforcement frequently broke when outputs were lengthened.
- Multilingual generation exposed inconsistencies: partial outputs mixed German and English. Generating consistent outputs across DE/EN/FR/IT remains unsolved.
- Prompting experiments revealed trade-offs between conciseness and fidelity. When forced to produce longer outputs, models often ignored structural constraints.
- We hardcoded logic for accessibility routing (e.g., preferring ramps over stairs for wheelchair users). This demonstrated feasibility but highlighted difficulties: instructions sometimes routed users inefficiently (e.g., via a ramp even when an elevator was closer).

Technically Notable Aspects:

- Integration of geospatial OSM data with AI-driven natural language generation.
- Attempted use of structured JSON prompts to enforce machine-readable output.
- Early evaluation of schema-enforcing protocols (e.g., Model Control Protocol, MCP) for future reliability.

Deliberate Scope Limitations:

- Did not implement full support for all accessibility profiles due to time constraints.
- Did not develop a production-grade frontend; visualization was limited to proof-of-concept.
- Did not integrate external translation APIs to improve multilingual consistency.

Open Issues & Future Work:

- Data Inconsistencies: SBB datasets differ in identifiers (BPS ID availability varies), requiring preprocessing.
- Structured Output: Current LLMs fail to reliably produce long-form, schema-conforming text.
- Multilingual Quality: R&D needed into hybrid translation workflows (e.g., enforcing keywords while allowing generative freedom).
- Accessibility Routing: Requires more sophisticated path optimization to dynamically choose between elevators, ramps, or stairs.
- Research into Model Control Protocol (MCP) and advanced prompting strategies could improve schema enforcement and multilingual reliability.

Given additional time, we would extend experimentation across multiple model families, implement translation quality control, and formalize schema enforcement mechanisms.

What currently is on Github

Overview

There is an experimental command line app as well as another experimental flask app. While the flask app includes a frontend that visualizes the generated experimental data, there is another frontend (visualize) that was created to display descriptions stored in a database.

Further experiments could use the command line tool including the tests related to the tool on one hand. On the other hand, the flaks app can be run and modified. Note that the approaches and models used in the command line tool or the flask app differ. The goal was to find a solution that generates descriptions for the people with disability and store these in a database. Because we never reached at a point where the descriptions were useful, the project remains in an inconsistent state that includes several unfinished approaches – our experimental playgrounds.

To use the command line tool or related tests, please follow the README in the project. Similarly, the frontend “visualize” includes its own README. The flask app can be installed and run as described in the next session.

Experimental flask app (app.py)

This application provides wheelchair-friendly route planning. Users input starting and ending coordinates (latitude and longitude), and the application generates step-by-step directions while considering accessibility features such as ramps and avoiding obstacles like stairs. The system leverages OpenStreetMap data (via the osmnx library) along with Points of Interest (POIs) from a dataset.

Setup & Installation

Follow these steps to set up and run the application:

1. Clone or download the project files.
2. Install dependencies:
   pip install flask osmnx geopandas shapely pandas networkx
3. Place your POI dataset file (e.g., datasets.geojson) in the project directory.

4. Run the Flask app:
   python app.py
5. Open your browser at http://127.0.0.1:5000

1 Configuration

The following configuration parameters are defined:
- POI_PATH: Path to the dataset containing Points of Interest.
- CATEGORY_FIELD: The field name for classifying POIs (e.g., ramp, stairs).
- BUFFER_METERS: Distance buffer around the route for detecting nearby POIs.
- RAMP_SEARCH_METERS: Search radius for finding ramps near stairs.
- NETWORK_TYPE: Type of network to use (e.g., 'walk').

2 Utility Functions

• load_pois(path, category_field): Loads Points of Interest from a CSV or GeoJSON file.
• shortest_path_route(G, start_latlon, end_latlon, weight): Computes the shortest path between two coordinates.
• points_along_route(pois, route_coords): Identifies POIs located within a buffer around the route.
• pair_stairs_with_ramps(pois_along): Matches stairs to the nearest ramp (if available).
• build_custom_instructions(route_len_m, pois_along, pairs): Generates human-readable step-by-step directions.

3 Flask Routes

@app.route('/'):
- GET: Renders the input form for coordinates.
- POST: Accepts user input, computes the route, identifies POIs, and returns wheelchair-friendly instructions.

4 Usage

1. Navigate to http://127.0.0.1:5000.
2. Enter the start and end latitude/longitude coordinates.
3. Submit the form.
4. The application will display step-by-step accessible route instructions.

5 Example Output

Example instructions generated by the system:

1. Start your journey. The total distance is about 1250 meters.
2. Around 300 meters ahead there are stairs. Use the nearby ramp instead (within 15 m).
3. At about 600 meters, you will find a ramp for easier access.
4. Continue until you reach your destination after about 1250 meters.