Wish an instant map!

Term Project

Department of Computer Science

OST - University of Applied Sciences

Campus St. Gallen

Autumn Term 2024

Authors:	Aziz Hazeraj, Thashvar Uthayakumar
Advisor:	Prof. Stefan F. Keller
Date:	20.12.2024

Abstract

The aim of this work is to develop a proof-of-concept (POC) for an open-source QGIS plugin capable of translating natural language queries into OverpassQL to perform spatial, temporal and attributive filtering of OpenStreetMap (OSM) data. The plugin visualises query results directly in QGIS. For example, a query like ”All cafés within 50 metres of a fountain in St. Gallen” is processed and rendered as a map. Unlike existing solutions that either require knowledge of OverpassQL (e.g. QuickOSM) or are proprietary, this plugin aims to make such functionality accessible to non-experts.

The query processing consists of three main steps: (1) geoname recognition and geocoding (e.g. resolving ”St. Gallen”), (2) recognition and semantic alignment of spatial entity sets (SES) with OSM attributes (e.g. mapping ”café” to ‘amenity=cafe‘), and (3) generation of OverpassQL queries. Early attempts to implement the plugin using open source models such as Llama yielded suboptimal results. Consequently, a fine-tuned OpenAI GPT-4o model was used, resulting in significant improvements in query generation. Geonames were resolved using Photon, an OSM-based geocoder, in addition to OpenAI's assistant, and SES were mapped using semantic similarity analysis with pre-embedded OSM tags.

The fine-tuned LLMs were evaluated using 100 natural language queries, with the best fine-tuned GPT-4o model achieving a BLEU score of 0.67, significantly outperforming base models and open source alternatives. The exact match rate was 0.09, indicating room for improvement in the generation of perfectly accurate queries. Most of the generated OverpassQL queries were functional within QGIS, with a high validity rate, although still lacking in semantic precision.

The resulting QGIS plugin, called *Wish an Instant Map! (WAIM)*, was implemented in Python using the two preprocessing steps, together with the fine-tuned OpenAI LLM. The graphical user interface includes text input for queries, support for current map extents, and an expert mode for editing OverpassQL. While the system has demonstrated feasibility through black-box testing with English language queries, challenges remain, including reliability of generated queries and reliance on proprietary LLMs. Future improvements can include the development of an OSM thesaurus to improve semantic matching, the integration of structured output for query validation, and the use of larger datasets or larger LLMs to fine-tune open-source models. Despite its limitations, WAIM illustrates the potential for combining AI with geospatial systems.

Management Summary

Initial situation and approach

The initial situation behind this project is the idea, to allow users with no expertise around Geographic Information Systems (GIS) to ”wish an instant map”.

The project aims to address the complexity of querying OpenStreetMap (OSM) data using OverpassQL, a language that requires technical expertise. The steep learning curve for OverpassQL presents a significant barrier for non-technical users who need to extract and visualize geospatial data.

To overcome this challenge, the primary goal was to create a proof-of-concept (POC) plugin for QGIS that enables users to generate OverpassQL queries using natural language inputs.

A key functionality of the project is the recognition of geonames within user prompts, allowing the system to identify geographical locations specified in natural language. Additionally, the system performs object entity recognition, detecting terms such as ”fountain” and categorizing them as object entities relevant to geospatial queries. These inputs are then processed to generate OverpassQL queries, by combining the recognized geonames and object entities and sending them to a fine-tuned LLM. The generated OverpassQL queries are integrated into QGIS, enabling users to view geospatial data directly within QGIS.

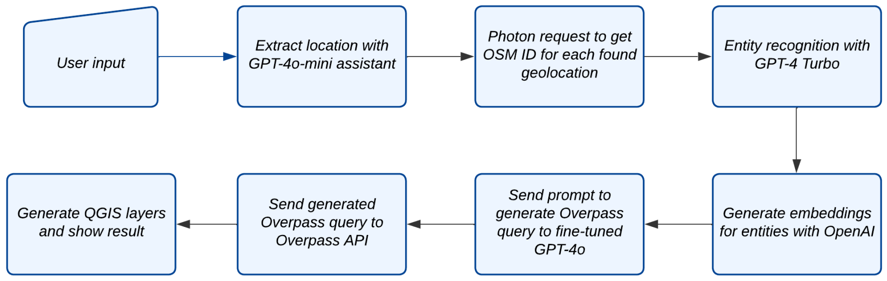

User input
Extract location with GPT-4o-mini assistant
Photon request to get OSM ID for each found geolocation
Entity recognition with GPT-4 Turbo
Generate embeddings for entities with OpenAI
Send prompt to generate Overpass query to fine-tuned GPT-4o
Send generated Overpass query to Overpass API
Generate QGIS layers and show result

Figure 1: Simplified Activity Diagram

The approach taken to achieve these functionalities involved several key components. First, a user-friendly graphical interface was developed within QGIS to allow users to input natural language queries and interact with generated results. Then, a few preprocessing steps are needed to generate an optimal Overpass query that yields the best result for the given user query. The first preprocessing step is geolocation processing, where the correct geolocation of the user query is extracted using a GPT Assistant and then processed with the geocoder Photon. In a second step, the object entities get extracted with GPT-4 Turbo and then embedded with OpenAI's embedding model. The resulting embedding is compared to a large number of precomputed embeddings of OSM tags using the cosine similarity of the embedding vectors. The tags with a high similarity are used to generate an extended query that is sent to a fine-tuned GPT-4o model. The resulting Overpass query is then sent to the Overpass API and the results get shown as point, line or polygon layers in QGIS. A simpler version of the steps is shown in the figure 1.

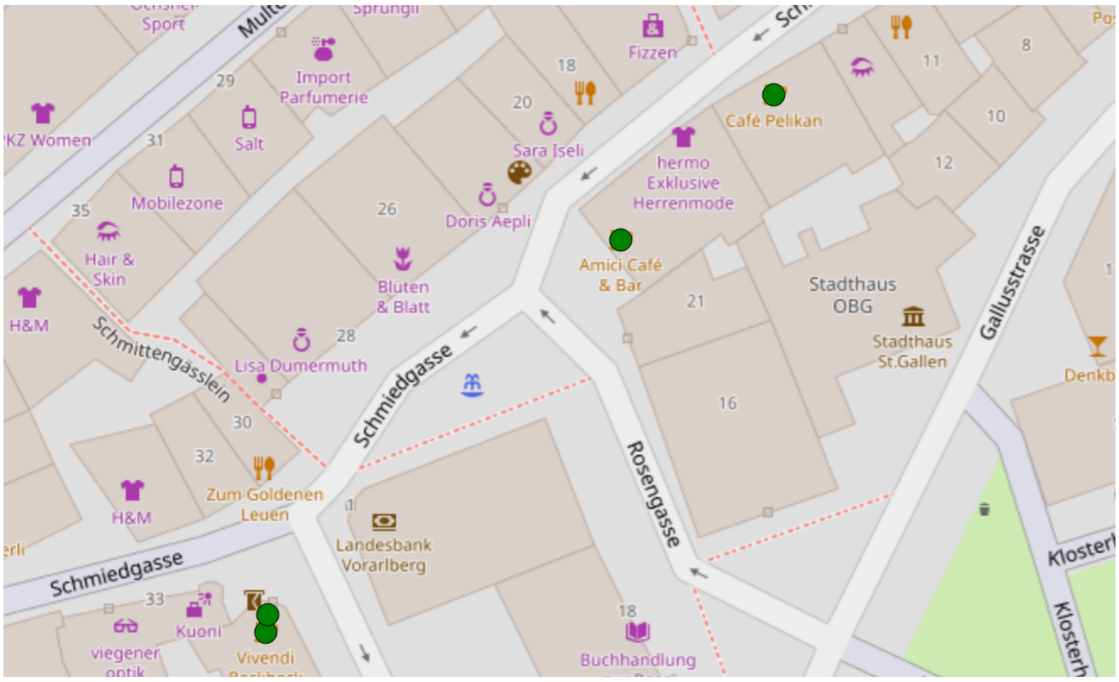


Figure 2: QGIS plugin, cafes near fountains

Result

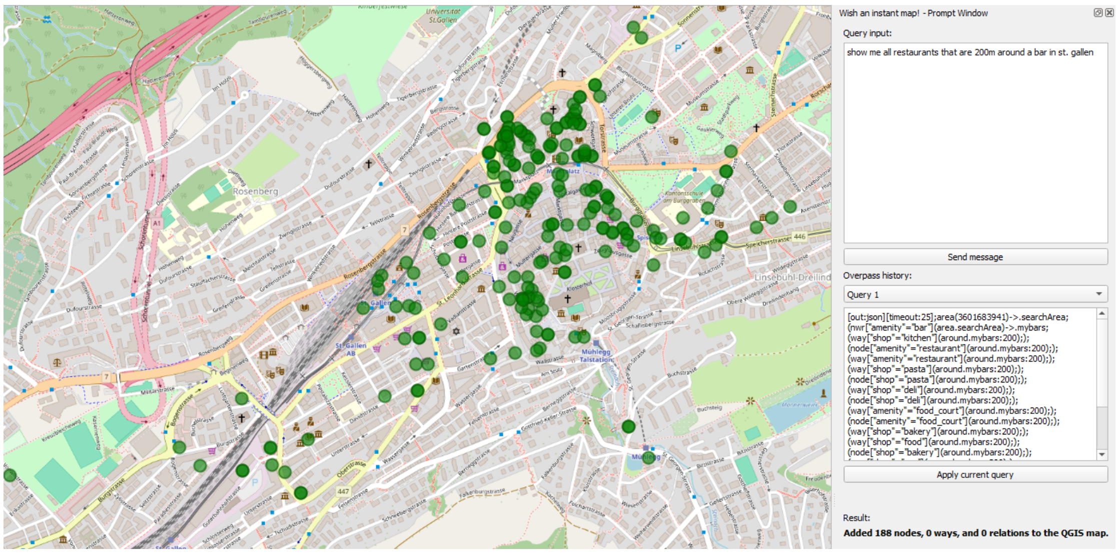
The resulting application demonstrates the feasibility of translating natural language queries into OverpassQL and visualizing the results effectively within QGIS. The plugin generates mostly syntactically correct Overpass queries, with fine-tuned GPT models averaging a BLEU score of 0.67. With the expert mode, experienced users are able to customize queries as needed and resend them directly to the Overpass API. By lowering the barrier for non-expert users to interact with complex geospatial data, WAIM provides a significant step forward in geospatial data accessibility.

While the project successfully demonstrated a proof-of-concept, several challenges and limitations were encountered. The inability to fully utilize open-source LLMs due to computational constraints forced us to diverge from our initial vision and use OpenAI's solutions instead. Additionally, the precomputed embeddings do not retain any hierarchical relationships between OSM tags, limiting the ability to find the most relevant tags. The reliance on proprietary APIs further highlights the need for future scalability and openness.

Outlook

Looking ahead, WAIM can be enhanced in several ways. Adopting structured outputs using JSON schemas would ensure consistency and machine-readability for generated OverpassQL queries. Direct integration of a thesaurus could improve the retention of relationships among OSM tags, leading to better semantic alignment. Future efforts could also focus on reducing dependency on proprietary tools by fine-tuning larger open-source models such as the new Llama3.3 70B. Improvements in query validation and accuracy, along with enriched visualization features in QGIS, could further expand the system's usability for both inexperienced and expert users.

In conclusion, the WAIM POC demonstrates the potential of combining natural language processing with geospatial systems to simplify OverpassQL query generation. By addressing current limitations and building on this foundation, WAIM has the potential to evolve into a robust open-source solution for geospatial data exploration.



show me all restaurants that are 200m around a bar in st. gallen

```
[out:json][timeout:25];area(3601683941)->.searchArea;
(nwr["amenity"="bar"](area.searchArea)->.mybars;
(way["shop"="kitchen"](around.mybars:200););
(node["amenity"="restaurant"](around.mybars:200););
(way["amenity"="restaurant"](around.mybars:200););
(way["shop"="pasta"](around.mybars:200););
(node["shop"="pasta"](around.mybars:200););
(way["shop"="deli"](around.mybars:200););
(node["shop"="deli"](around.mybars:200););
(way["amenity"="food_court"](around.mybars:200););
(node["amenity"="food_court"](around.mybars:200););
(way["shop"="bakery"](around.mybars:200););
(way["shop"="food"](around.mybars:200););
(node["shop"="bakery"](around.mybars:200););
```

Added 188 nodes, 0 ways, and 0 relations to the QGIS map.

Figure 3: Wish an instant map - Expert mode

Contents

// Chapter 1

Introduction

1.1 Background

The story behind this software project is simple, yet ingenious: ”Wish for a map with POIs - and the app creates it instantly”. The idea is to allow users to intuitively generate maps based on natural language queries, bringing the power of Geographic Information Systems (GIS) to everyone.

1.2 Problem Statement

Creating customized maps with points of interest (POIs) has traditionally been reserved for specialists with technical expertise. For example, generating a map based on a query like ”Show all fountains in Rapperswil” requires knowledge of advanced query languages and GIS tools. Additionally, POIs are not just points but can represent object classes (referred to as spatial entity sets) of various kinds, which adds complexity.

Currently, no comparable plugin exists that allows such natural language-based interaction. Existing tools like QuickOSM require prior knowledge of the Overpass Query Language (OverpassQL), while other solutions are proprietary and costly. This creates a significant barrier for non-specialists who wish to use OpenStreetMap (OSM) data within QGIS.

1.3 Objective

The objective of this work is to develop a proof-of-concept plugin for QGIS that translates natural language queries into Overpass API calls for spatial, temporal, and attributive filtering of OpenStreetMap data. The plugin will then return the query results directly visualised within QGIS.

The process involves three key steps:

- Geoname recognition (Geocoding): Identifying and resolving geographical names in the query (e.g., ”Rapperswil (SG)”).
- Object entity recognition: Extracting the object entities referenced in the query (e.g., ”fountains”).
- Query translation: Converting the user query into OverpassQL, the specialized query language for OpenStreetMap data.

The approach builds on a solution that has already proven successful with the QGIS plugin AIAMAS. Initially, the query searches for the most similar geographical or object name in a local configuration (e.g., ”fountain” for ”Brunnen”). If no match is found, a backend powered by a large language model (LLM) is queried for synonyms, and an Overpass API query is generated if possible.

1.4 System Context

As for the system context, we have one existing system. That system is QGIS. It has a long history and is an established open-source GIS application. Our plan with this project is to create a plugin for QGIS. To develop Plugins, we have the nice possibility of using python, which our group has already had some experience with.

Chapter 2

Project Design

2.1 Requirements Analysis

In this chapter, we will describe the vision of this project and go deeper into functional and non-functional requirements.

Our vision of this assignment is the ability to create a plugin for QGIS which has the ability to wish an instant map for a user. This means that the user only has to type in a prompt and the plugin automatically gives the desired output in QGIS.

2.2 Functional Requirements

2.2.1 Epics

Epics are identified with ”E” followed by a number. Each epic represents a broader category of functionality, with associated user stories to define specific requirements.

ID	E1
Subject	Configuration
Description	This epic focuses on the installation and configuration of the plugin.

Table 2.1: E1: Configuration

ID	E2
Subject	Prompts
Description	This epic focuses on the user's prompts and splits apart between our processing steps and what the user can expect in the normal-mode of the plugin.

Table 2.2: E2: Prompts

ID	E3
Subject	Expert-mode
Description	This epic introduces advanced features for more experienced users with OverpassQL.

Table 2.3: E3: Prompts

2.2.2 User Stories

In this section we will create User Stories for each category of our plugin. The only role we have, is the user and each user story has a priority between: low, medium and high.

E1: Configuration

ID	US1.1
Subject	Installation Process
Priority	High
Description	As a user, I want to be able to easily install the plugin per ZIP-file.

Table 2.4: User Story US1.1: Installation Process

ID	US1.2
Subject	Dependency installation
Priority	High
Description	As a user, I want that the plugin automatically installs all dependencies, which are necessary.

Table 2.5: User Story US1.2: Dependency Process

ID	US1.3
Subject	Settings
Priority	High
Description	As a user, I want to be able to give QGIS my tokens to authenticate with the correct user in OpenAI.

Table 2.6: User Story US1.3: Settings

E2: Normal-mode

ID	US2.1
Subject	Simple Usage
Priority	Medium
Description	As a user, I want to input my prompt and receive a response.

Table 2.7: User Story US2.1: Simple Usage

ID	US2.2
Subject	Local Prompt
Priority	High
Description	As a user, I want the locality ”Rapperswil” to be recognized in the prompt ”All Flower stores in Rapperswil,” and only points within this region to be displayed.

Table 2.8: User Story US2.2: Local Prompt

ID	US2.3
Subject	Entity Prompt
Priority	High
Description	As a user, I want the point of interest ”fountain” to be recognized in the prompt ”All drinking fountains,” and also display other public drinking spots.

Table 2.9: User Story US2.3: Entity Prompt

ID	US2.4
Subject	Map View
Priority	High
Description	As a user, I want the prompt ”All restaurants in this area” to only return markers within my current map view.

Table 2.10: User Story US2.4: Map View

ID	US2.5
Subject	All OSM elements
Priority	High
Description	As a user, I want that every OSM element from a prompt is shown in the result.

Table 2.11: User Story US2.5: All OSM elements

E3: Expert-mode

ID	US3.1
Subject	View Overpass Query
Priority	Medium
Description	As a more experienced user, I want to be able to view the Overpass Query which was generated and previously generated queries.

Table 2.12: User Story US3.1: View Overpass Query

ID	US3.2
Subject	Edit Overpass Query
Priority	Low
Description	As a more experienced user, I want to be able to edit the generated Overpass Query and display the result in QGIS.

Table 2.13: User Story US3.2: Edit Overpass Query

ID	US3.3
Subject	Background processes
Priority	Low
Description	As a more experienced user, I want to be able to understand what the plugin is doing in the background with verbose logging.

Table 2.14: User Story US3.2: Background processes

2.3 Non-Functional Requirements

NFR1: Performance

ID	NFR1.1
Subject	Response Time
Priority	Medium
Description	The response time for user inputs (prompts) should be kept as low as possible to ensure a smooth user experience.
Acceptance Criteria	The Overpass query is generated within a maximum of 10 seconds, and the results are displayed.

Table 2.15: Non-Functional Requirement NFR1.1: Response Time

NFR2: Reliability and Availability

ID	NFR2.1
Subject	LLM Availability
Priority	High
Description	The plugin should ensure that the language model and all external APIs for prompts are always accessible, if not an Error is shown.
Acceptance Criteria	If the connection to the LLM fails, an appropriate error message and an alternative response are displayed.

Table 2.16: Non-Functional Requirement NFR2.1: LLM Availability

NFR3: Usability and User-Friendliness

ID	NFR3.1
Subject	Intuitive Usage
Priority	High
Description	The plugin's user interface must be intuitive and straightforward, enabling users to operate it without training.
Acceptance Criteria	Users can operate the plugin without much of an introduction.

Table 2.17: Non-Functional Requirement NFR3.1: Intuitive Usage

ID	NFR3.2
Subject	User Feedback
Priority	Medium
Description	Each query or processing step should be accompanied by a loading animation or a brief notification in the UI to inform the user of the progress.
Acceptance Criteria	The user receives a visual confirmation after entering a prompt that the query is being processed.

Table 2.18: Non-Functional Requirement NFR3.2: User Feedback

NFR4: Compatibility and Integration

ID	NFR4.1
Subject	Compatibility with QGIS Versions
Priority	High
Description	The plugin must be fully compatible with QGIS versions 3.34 and newer without causing errors or impairing functionality.
Acceptance Criteria	The plugin runs error-free on QGIS 3.34 and newer versions.

Table 2.19: Non-Functional Requirement NFR4.1: Compatibility with QGIS Versions

ID	NFR4.2
Subject	Support for Multiple Operating Systems
Priority	Medium
Description	The plugin should be operable on Windows, macOS, and Linux since QGIS is used cross-platform.
Acceptance Criteria	The plugin is installable and functional on the three specified operating systems.

Table 2.20: Non-Functional Requirement NFR4.2: Support for Multiple Operating Systems

NFR5: Maintainability and Extensibility

ID	NFR5.1
Subject	Modular Design
Priority	Medium
Description	The plugin's code should be modularly structured to facilitate future adjustments and extensions.
Acceptance Criteria	Individual components such as Photon and tag finder are implemented separately and modularly.

Table 2.21: Non-Functional Requirement NFR5.1: Modular Design

2.4 Market Analysis

As of right now, there are some comparable products out there, but not a single project is really similar to the Wish an instant map POC or is behind a payment wall.

2.4.1 QuickOSM for QGIS

QuickOSM is a user-friendly QGIS plugin designed for extracting and visualizing OpenStreetMap (OSM) data directly within QGIS. It streamlines the process of querying and importing OSM data, allowing users to generate Overpass API queries. These Overpass queries are built upon OSM tags, which the user has to know, and a location the user has to pass to QuickOSM. This means that it requires users to have prior knowledge of OSM's tagging schema and syntax, as the query-building process is entirely manual.

While QuickOSM is powerful for experienced users familiar with OSM data structures, it presents a significant barrier to entry for less experienced users who may not understand the complexities of OSM tagging.

2.4.2 Kue

Kue is another QGIS plugin that integrates AI functionalities into QGIS. It focuses on enhancing usability by providing intelligent assistance for symbology, dataset search, and the addition of basemaps Ashworth, 2024. Unlike QuickOSM, Kue employs AI to streamline these tasks, but its functionality does not extend to generating Overpass queries from natural language inputs.

Kue operates on a subscription-based model, making it a commercial solution with limited accessibility for users who cannot afford paid tools. Additionally, this means that very little information is available about how it works in the background or what AI solution it uses. While Kue demonstrates the growing demand for AI-enhanced tools within QGIS, it does not directly compete with Wish an Instant Map! in terms of capability expectations.

2.4.3 Text-to-OverpassQL

Text-to-OverpassQL Staniek et al., 2023 is a task that addresses the same problem as our QGIS plugin: Natural language querying within OSM, specifically targeting the Overpass Query Language. It allows novice users to generate OverpassQL queries without any prior knowledge of the language, lowering the barrier to entry for interacting with the OSM database.

This task provides a dataset of 8,352 queries paired with natural language input that can be used to train or fine-tune models for the Text-to-OverpassQL task.

Unlike the Wish an Instant Map! plugin, Text-to-OverpassQL operates as a research framework and does not integrate directly with QGIS or other geospatial software. Its primary focus is to advance semantic parsing for geospatial databases, making it a complementary rather than competitive solution.

The dataset of this task were very valuable for fine-tuning the models for our QGIS plugin, as there are not many labeled Overpass queries available.

2.4.4 AINO

AINO aino, 2024 is a relatively new AI-based tool that focuses on providing automated insights and data visualizations for geospatial datasets. It is marketed as an AI agent for consultants, data analysts, and spatial planners to perform geo spatial analysis. However, AINO does not specifically target OSM or Overpass API users, nor does it specialize in query generation.

AINO integrates with QGIS as there is a QGIS plugin available, but since it is not marketed towards OSM or Overpass users, we see it as a complementary tool rather than a direct competitor to Wish an instant map!.

2.5 Domain Analysis

The figure in 2.1 represents our domain model, it shows the problem domain in a very abstract context. In the following

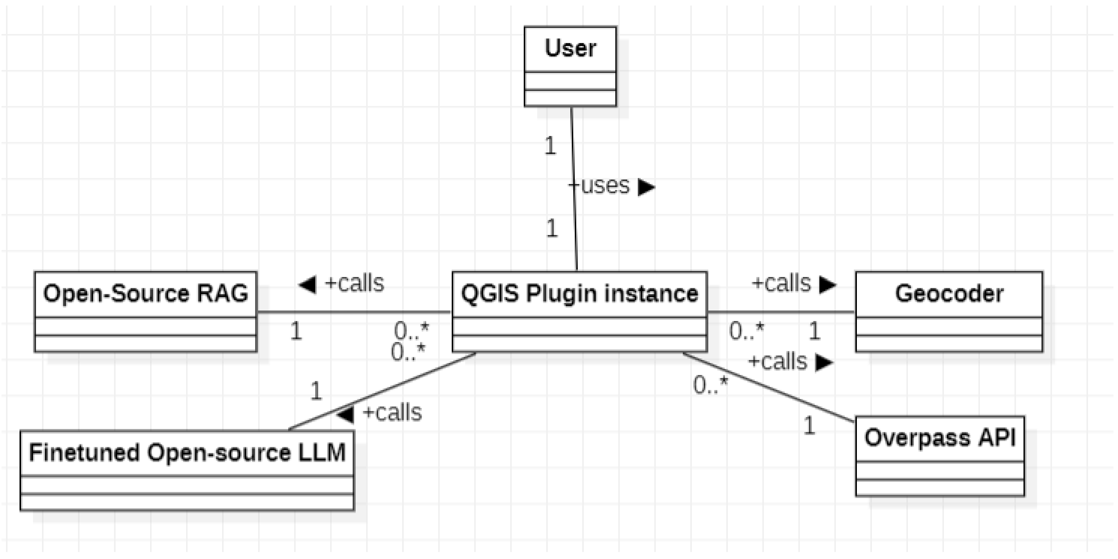


Figure 2.1: Domain model

User

- The entity representing the end-user interacting with the system.
- Responsibilities:
  - Provides prompts or commands to the QGIS plugin.
  - Views the visual output generated by the system in QGIS.

QGIS Plugin Instance

- The central component of the system, hosted within the QGIS application.
- Responsibilities:
  - Receives input from the User.
  - Preprocesses prompts by calling external services (e.g., geocoder, Open-Source RAG).
  - Generates OverpassQL queries using the fine-tuned LLM.
  - Visualizes query results within the QGIS interface.

Geocoder

- An external service used for resolving geonames mentioned in the User's prompt.
- Responsibilities:
  - Converts geonames into coordinates, if necessary.
  - Provides data necessary for OverpassQL query generation.

Open-Source RAG

- A semantic search mechanism used to improve query relevance.
- Responsibilities:
  - Retrieves contextually relevant data for input prompts.
  - Enhances OverpassQL query accuracy through similarity analysis.

Fine-Tuned Open-Source LLM

- A large language model fine-tuned for the task of generating OverpassQL queries.
- Responsibilities:

- Generates OverpassQL queries based on the User's prompt and preprocessed data.

Overpass API

- A service that executes OverpassQL queries to retrieve filtered geodata.
- Responsibilities:
  - Executes the queries generated by the QGIS Plugin.
  - Returns results to visualize within QGIS.

2.6 Use Case Diagram

The following use case diagram shows our use cases:

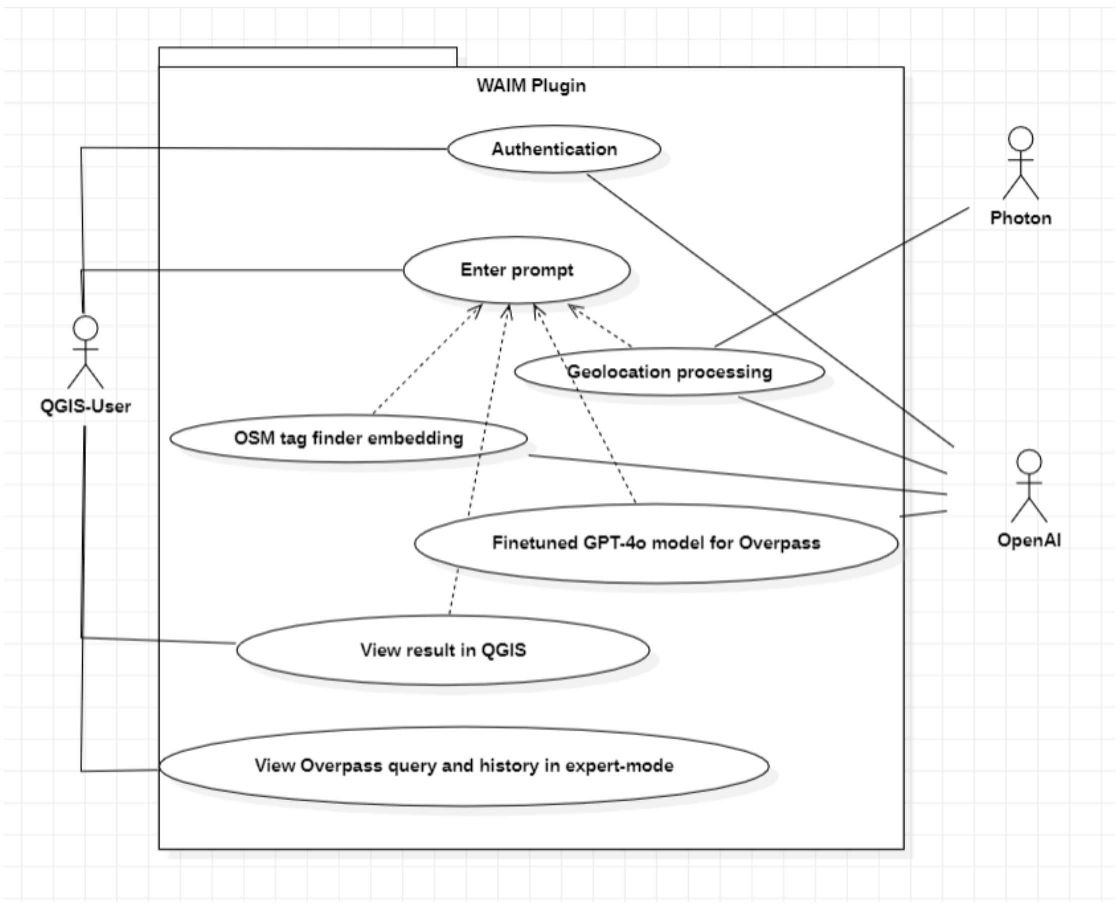


Figure 2.2: use case diagram

Authentication

The user has to enter the OpenAI API key and the OpenAI organizational token, to authenticate as a valid user, who is allowed to use OpenAI.

Enter Prompt

The user provides a text prompt that serves as input for the geolocation processing, the tag finding and the final query generation for the OverpassQL query.

Geolocation Processing

The plugin processes the user's prompt to determine geographic coordinates, utilizing services like Photon for geocoding.

OSM Tag Finder Embedding

The plugin analyses the prompt and identifies relevant OpenStreetMap tags similar to those used in the users's prompt.

Fine-tuned GPT-4o model

The plugin uses the entered prompt, with preprocessing steps, to enter the fine-tuned GPT-4o model.

View Result in QGIS

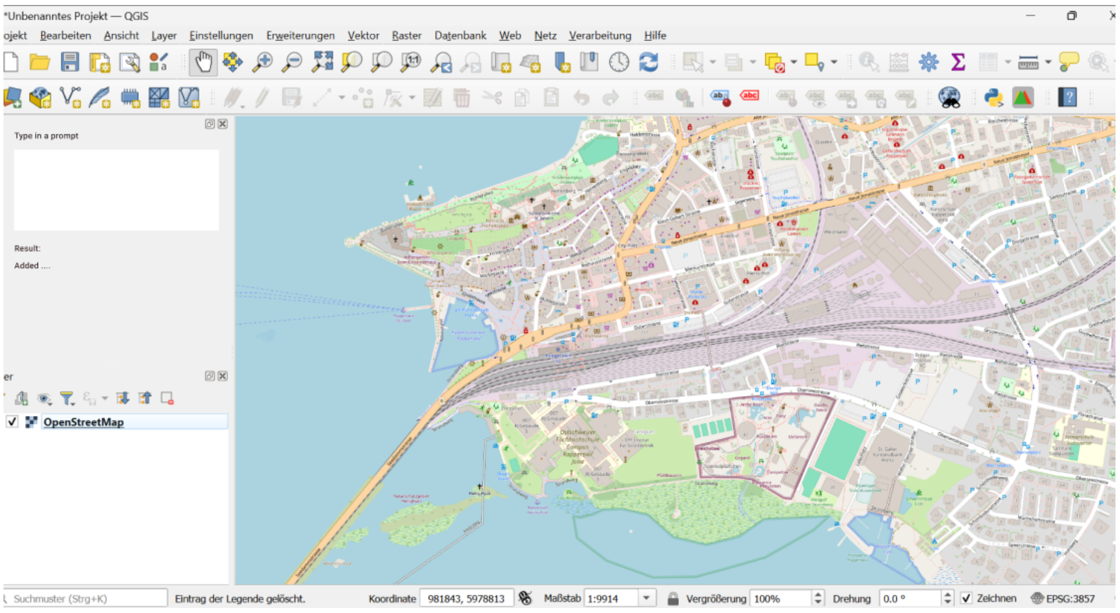
The user can view the processed results and rendered map layers, directly within the QGIS interface.

View Overpass Query and History in Expert Mode

Advanced users can switch to expert mode to inspect and manage the generated Overpass API queries, along with their query history.

2.7 Mockups

In this section we will show and explain our mockups. First of all, it was important for us that the user-interface (UI) was as simple as possible. Therefore, the only visible elements on there are a big text area field, where the user can type in his prompt. As our output is on the map, we do not need much text, that is the reason why there is just one line for text.


*Unbenanntes Projekt — QGIS
ojekt Bearbeiten Ansicht Layer Einstellungen Erweiterungen Vektor Raster Datenbank Web Netz Verarbeitung Hilfe
OpenStreetMap
Suchmuster (Strg+K)

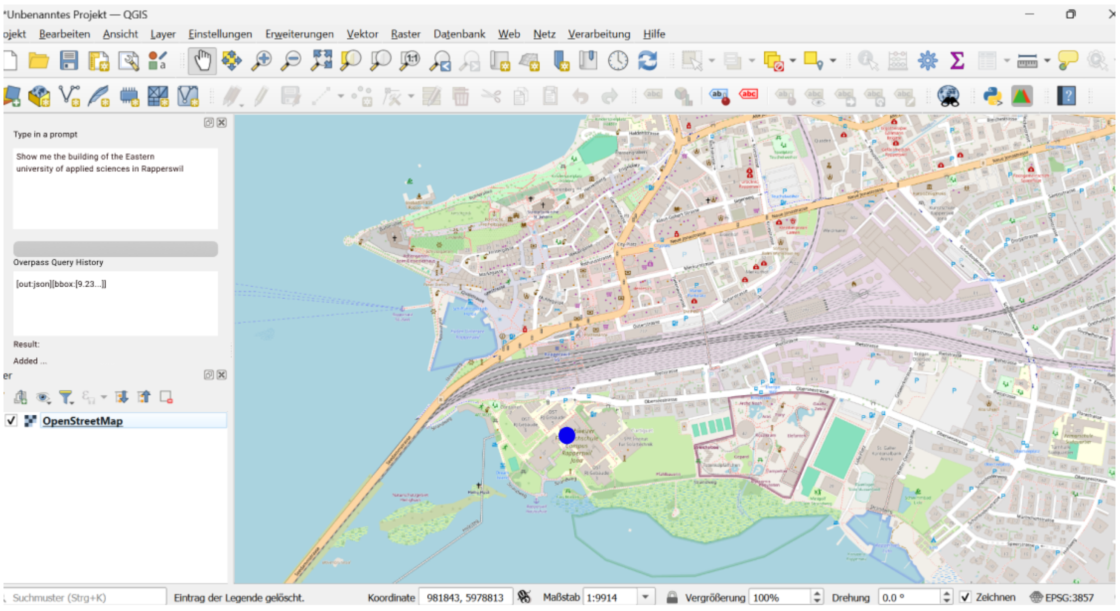
Type in a prompt

Result:
Added ...

Eintrag der Legende gelöscht. Koordinate 981843, 5978813 Maßstab 1:9914 Vergrößerung 100% Drehung 0.0 ° Zeichnen EPSG:3857

Figure 2.3: QGIS plugin UI normal-mode

In the text area, the user can type in a prompt, and it shows the result on the map and in the result text. The blue point on the map marks a result printed by the plugin.


*Unbenanntes Projekt — QGIS
ojekt Bearbeiten Ansicht Layer Einstellungen Erweiterungen Vektor Raster Datenbank Web Netz Verarbeitung Hilfe
OpenStreetMap
Suchmuster (Strg+K)

Type in a prompt

Show me the building of the Eastern university of applied sciences in Rapperswil

Overpass Query History

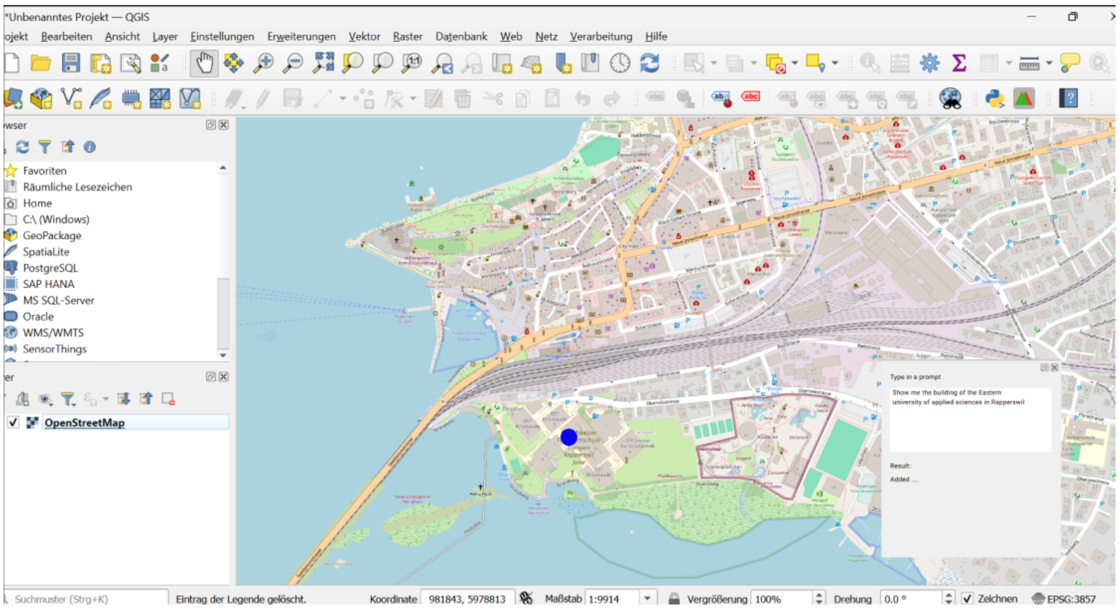
[out:json][bbox:[9.23...]]

Result:
Added ...

Eintrag der Legende gelöscht. Koordinate 981843, 5978813 Maßstab 1:9914 Vergrößerung 100% Drehung 0.0 ° Zeichnen EPSG:3857

Figure 2.4: QGIS plugin UI normal-mode with input

As QGIS uses PyQt in the background there are so-called dock widgets available. These dock widgets can be floating and are dockable to QGIS.



Type in a prompt

Show me the building of the Eastern university of applied sciences in Rapperswil

Result:

Added ...

Figure 2.5: QGIS plugin UI floating normal-mode

Next to the normal-mode the user can activate an expert-mode. This expert-mode is supposed to be for an expert, who has more knowledge about OverpassQL. Therefore he will be able to see an Overpass history, where he can view and edit previously generated queries.

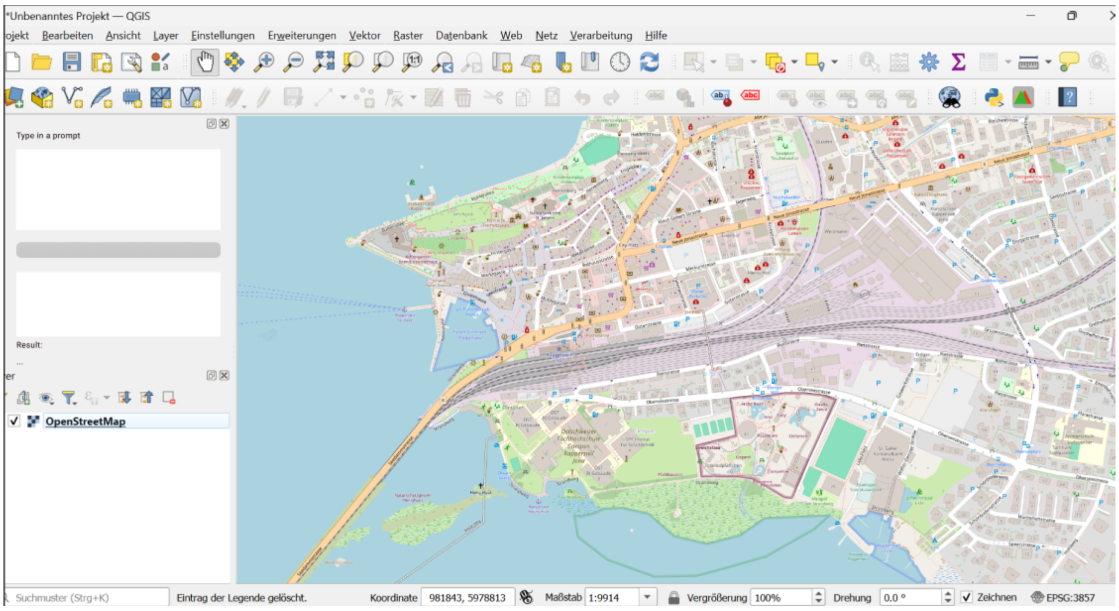
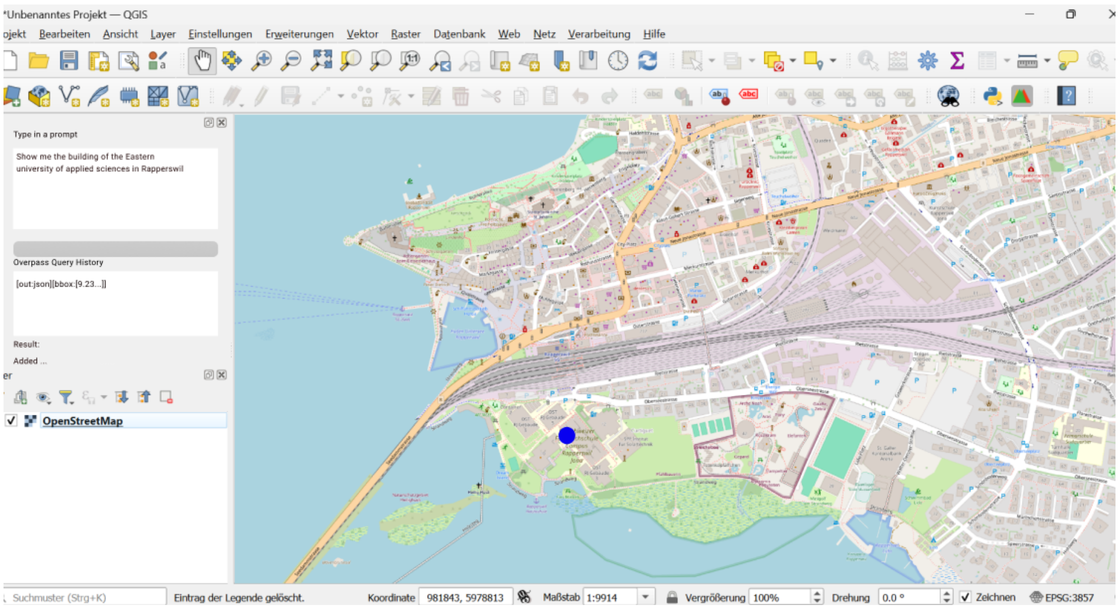

*Unbenanntes Projekt — QGIS
ojekt Bearbeiten Ansicht Layer Einstellungen Erweiterungen Vektor Raster Datenbank Web Netz Verarbeitung Hilfe
Type in a prompt
Result:
er
OpenStreetMap
Suchmuster (Strg+K)
Eintrag der Legende gelöscht.
Koordinate 981843, 5978813
Maßstab 1:9914
Vergrößerung 100%
Drehung 0.0 °
Zeichnen
EPSG:3857

Figure 2.6: QGIS plugin UI expert-mode

This is how the expert-mode will look like when there is a result for a prompt. The user will see the used OverpassQL query and he can edit and resend the OverpassQL query.


*Unbenanntes Projekt — QGIS
ojekt Bearbeiten Ansicht Layer Einstellungen Erweiterungen Vektor Raster Datenbank Web Netz Verarbeitung Hilfe
Type in a prompt
Overpass Query History
Result:
er
OpenStreetMap
Suchmuster (Strg+K)
Eintrag der Legende gelöscht.
Koordinate 981843, 5978813
Maßstab 1:9914
Vergrößerung 100%
Drehung 0.0 °
Zeichnen
EPSG:3857

Show me the building of the Eastern university of applied sciences in Rapperswil

```
[out:json][bbox:[9.23...]]
```

Added ...

Figure 2.7: QGIS plugin UI expert-mode with input

2.8 Architecture

In the following sections, we will explain and visualize our final architecture used in this project.

2.8.1 C4 Model

In the following section, we will explain our system context and the container diagram.

System Context

The system context in our project is very simple. We have a QGIS user, which uses the application QGIS.

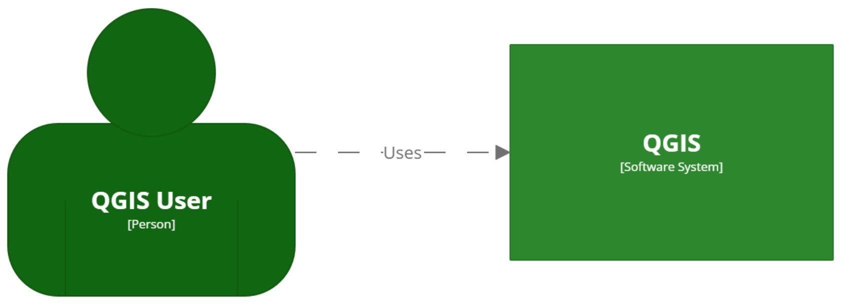


Figure 2.8: System context diagram

Component Diagram

The component diagram illustrates the integration of various external services used in our system. For geolocation processing, we use Komoot's Photon API (https://photon.komoot.io/) and the OpenAI Assistant.

For tag identification, the system uses the GPT-4 Turbo model along with OpenAI embeddings. To generate the final OverpassQL query, we use a fine-tuned OpenAI model that has been specifically trained for this purpose. Once the OverpassQL query is generated, it is sent to the Overpass API, which processes the query and returns the desired results.

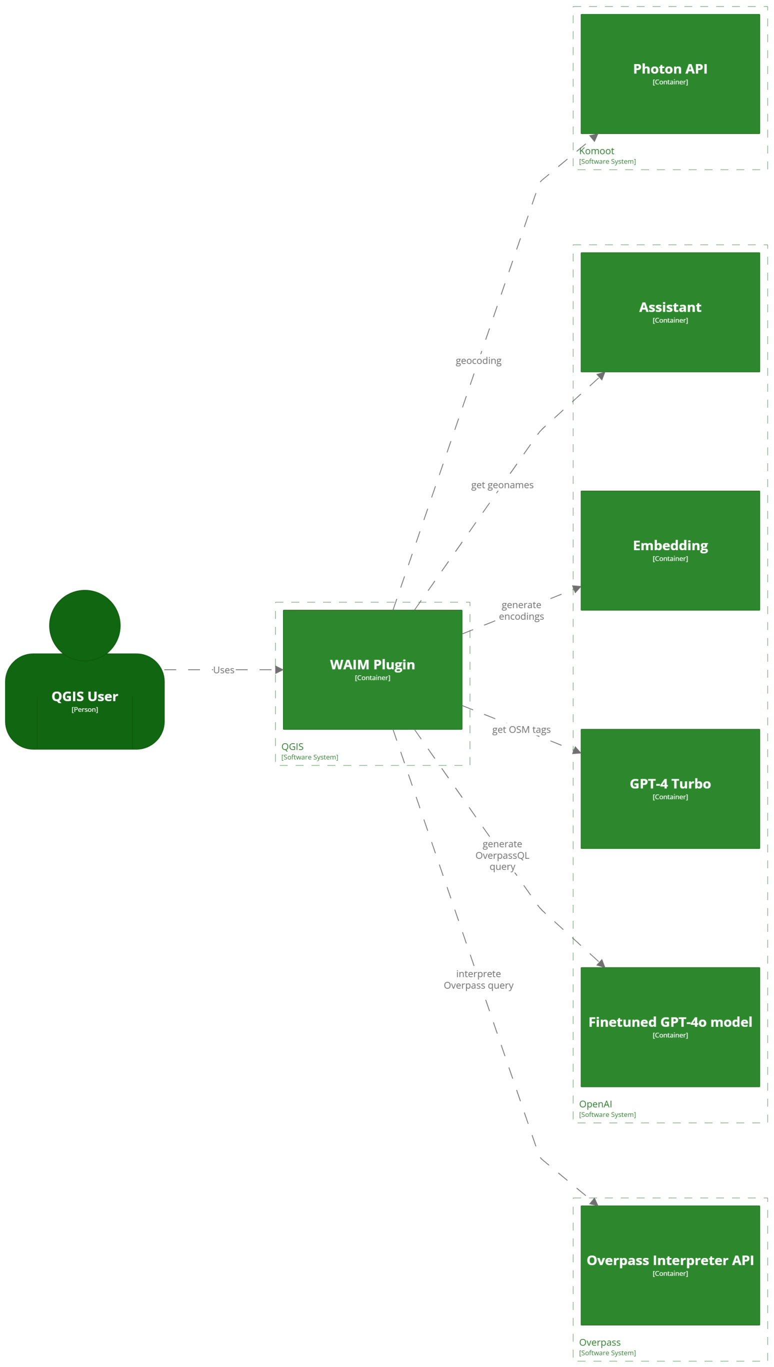


Figure 2.9: Component diagram

2.8.2 Activity Diagram

The figure 2.10 is an activity diagram to visually explain the flow of our project and architecture.

We start by getting the user input. With the user's input, we can extract the geonames and their corresponding OSM element. After extraction, we know if the user wants to use the current view of QGIS or whether there are geonames in the prompt. For each geoname found, we do a Photon request to find the OSM ID with the preferred OSM element. If the search for the preferred OSM element was unsuccessful, we just default to the first result of Photon, as it is the most relevant result. If the preferred OSM element was found, we will just generate the OverpassQL syntax for the found OSM element. For the case that it was not found and we work with the first result, we will fallback to the extent, which is a bounding box. Using this bounding box we can generate a prompt extension for the final prompt for the fine-tuned LLM. The last fallback case, if even the extent is not available, is the center coordinate of the first result. From that coordinate we apply a default-radius of 500 meters to calculate a bounding box around that coordinate. This concludes the geolocation processing.

The embedding functionality is the second step in the pre-processing workflow. To get all related OSM tags for a query, the plugin uses entity recognition, OpenAI's semantic embedding model and precomputed embeddings. First, the query is sent to GPT-4 Turbo to extract all entities from the user query. The extracted entities are sent to OpenAI's `text-embedding-3-text` to generate a vector embedding for each found entity. After doing a cosine similarity calculation with the precomputed embeddings, the most similar OSM tags are added as a prompt extension to the final prompt.

After processing the OSM tags, we create the final prompt by using the user's prompt extended by the geolocation prompt extension and OSM tags prompt extension. We can then send this to our fine-tuned LLM to create an OverpassQL query. The result of the fine-tuned LLM is then sent to the Overpass API to get the geodata from the desired request. Before visualizing the data in QGIS we first check whether the result is empty or not. If the result does end up being empty, we retry the request to the fine-tuned LLM, up to 3 total requests. The result is then shown inside of QGIS, which is the final step of a single prompt from the user.

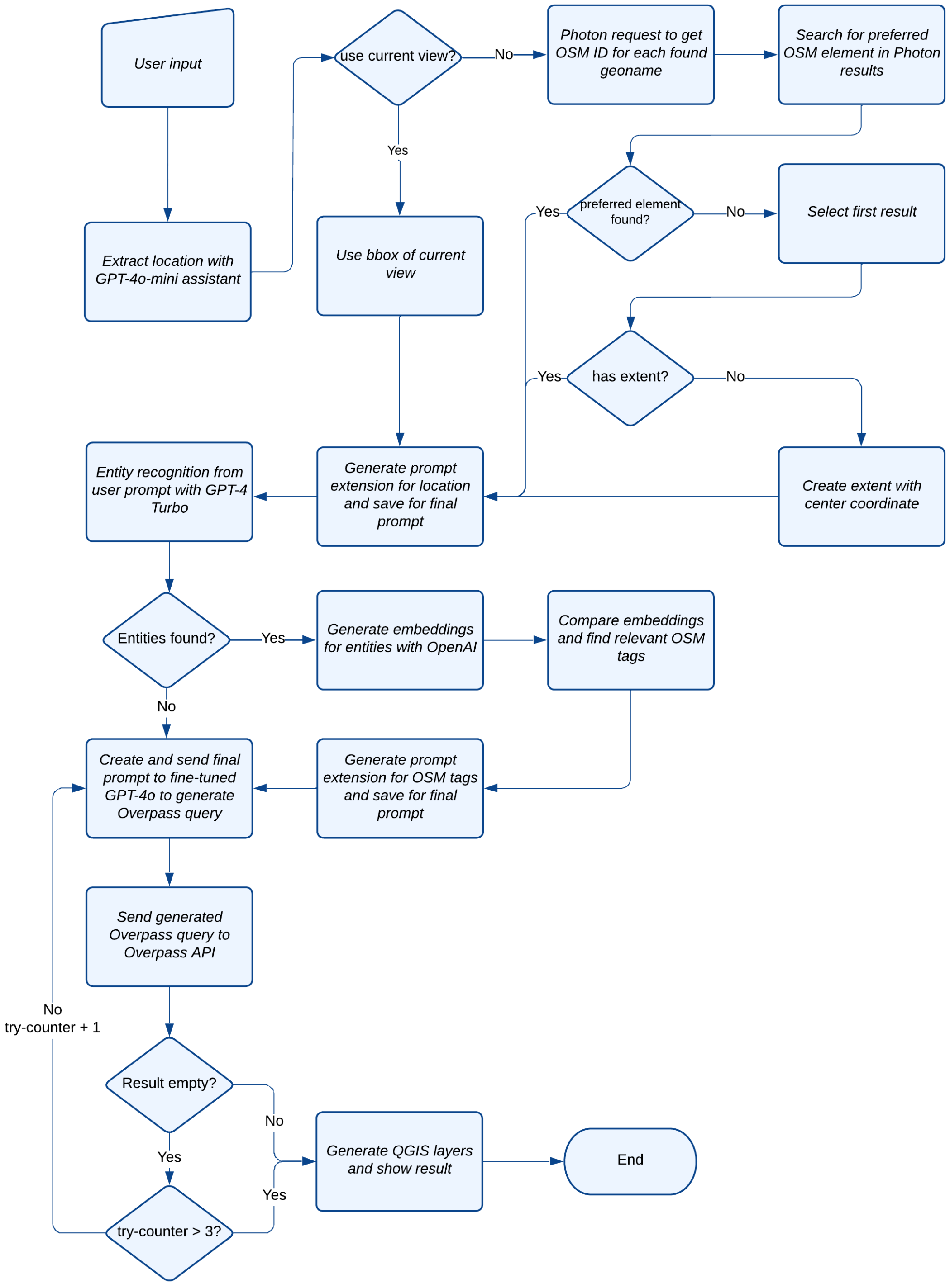

User input
use current view?
No
Photon request to get OSM ID for each found geoname
Search for preferred OSM element in Photon results
Yes
Extract location with GPT-4o-mini assistant
Use bbox of current view
Yes
preferred element found?
No
Select first result
Yes
has extent?
No
Entity recognition from user prompt with GPT-4 Turbo
Generate prompt extension for location and save for final prompt
Create extent with center coordinate
Entities found?
Yes
Generate embeddings for entities with OpenAI
Compare embeddings and find relevant OSM tags
No
Create and send final prompt to fine-tuned GPT-4o to generate Overpass query
Generate prompt extension for OSM tags and save for final prompt
Send generated Overpass query to Overpass API
No
try-counter + 1
Result empty?
No
Generate QGIS layers and show result
End
Yes
Yes
try-counter > 3?

Figure 2.10: Architecture activity diagram

Chapter 3

Implementation

In this chapter we will discuss our research and the decisions we made throughout the project.

3.1 Query Language

First of all, the goal of this project is to create a plugin for QGIS that can display the desired output for any prompt given by a user.

The core of this project is a natural language to geospatial data interpretation system, which requires a query language capable of retrieving relevant geospatial data from OpenStreetMap (OSM). Two prominent query languages can meet this requirement: SQL and Overpass query language.

3.1.1 SQL

SQL (Structured Query Language) is a powerful and widely used language for managing and querying relational databases. It excels at handling structured data, performing complex operations. Initially SQL does not support querying of geospatial data from OSM, but there is a project called OSM SQL Terminal Stefan, n.d., which has the ability to extract geospatial data from OSM through SQL queries. We decided against using SQL in favour of Overpass because it is specifically optimized for OSM geodata, allowing for more efficient spatial queries.

3.1.2 Overpass

Overpass is a query language specifically designed for extracting and filtering OSM geospatial data. It efficiently handles OSM's unique hierarchical structure (nodes, ways and relations), allowing for flexible tag-based and geospatial queries. Unlike SQL, Overpass is inherently optimised for OSM itself, making it faster and easier to use for tasks such as filtering by tags, performing spatial searches, or working within bounding boxes, without the need for additional database setups or extensions.This specialization makes it more efficient than SQL for OSM data extraction and this is why we chose Overpass over SQL.

3.2 Overpass Generation

Now that we know that we will be using Overpass, the next step is the process of the query generation. There are several approaches to this problem. In fact, there are 2 options that we have considered: Fuzzy string matching and an LLM.

3.2.1 Fuzzy string matching

Fuzzy string matching is a technique for determining how similar two strings are. There are different algorithms to determine similarity in different ways. Using fuzzy string match, you could identify keywords in the user's prompt by recognizing each word and sorting them into categories. For example, 'around', 'within' and 'near'. Each word has the same meaning in the context of an overpass query, so this should lead to the around function being used in the final query. We decided to use another approach because we did not think it matched well with Overpass. The reason for that is, even though Overpass does have a structure, the language itself has a lot of unusual features which are not well documented.

3.2.2 LLM

Introduction to Large Language Models (LLMs)

Large Language Models (LLMs) are AI systems trained on large datasets to understand and generate human-like text. They are incredibly versatile, capable of performing a wide range of natural language processing tasks such as text summarization, translation, and even code generation. However, while these models excel in many general-purpose applications, their performance often degrades in specific scenarios without further customization.

For our QGIS plugin, which focuses on generating Overpass queries for OSM data, a standard "off-the-shelf" LLM proved inadequate. Overpass queries have a specific syntax and structure, and the language used in user queries is often not directly aligned with OverpassQL. This mismatch required fine-tuning an LLM on existing Overpass query data to improve its understanding and ability to generate accurate Overpass queries.

In this section, we discuss the role of LLMs in our project, the challenges we encountered, and the steps we took to fine-tune a model to meet our needs.

Fine-Tuning

Fine-tuning is the process of adapting a pre-trained LLM to a specific task by training it further on a smaller, task-specific dataset. While large models like Llama3.1 or OpenAI's

GPT are pre-trained on massive datasets that cover many diverse topics, they often lack the detailed understanding required for niche tasks, such as generating Overpass queries.

Fine-tuning involves feeding the model labelled examples from the target domain, enabling it to learn the unique syntax, semantics, and patterns necessary for the task. For example, Overpass queries require precise syntax and tags that correspond to OSM data. Without fine-tuning, the model might generate incorrect or incomplete queries, leading to errors or irrelevant results.

For our project, we fine-tuned Meta's open-source Llama3.1 8B model and several OpenAI GPTs using the OverpassNL dataset. This dataset, derived from the ”Text-to-OverpassQL” project by the University of Heidelberg, contains labelled examples of user queries paired with OverpassQL syntax and is a great resource to create a training set from.

Llama3.1 8B

Our initial objective was to keep the project as open source as possible. To achieve this, the first large language model (LLM) we evaluated was Meta's open-source Llama3.1. Given the constraints of our computational resources and time frame, we opted for the 8B model, as fine-tuning the much larger 70B model was not feasible within our limitations.

To fine-tune the model, we used UnslothAI, a framework specifically designed to accelerate and simplify the fine-tuning of large LLMs. It provided the tools to train and export a fine-tuned version of Llama3.1 efficiently, even on limited hardware. The fine-tuning process used the OverpassNL dataset described above.

The fine-tuning was conducted using Google Colab, which provided access to a Tesla T4 GPU. Despite its relatively modest specifications compared to enterprise-grade GPUs, the T4 was sufficient for fine-tuning the 8B model due to optimizations in UnslothAI.

Key Steps in the Process:

1. Dataset Preparation: The OverpassNL dataset was formatted into a JSONL file where each entry included a natural language query (instruction) and its corresponding Overpass query (output). This structured data allowed the model to learn both syntax and tag matching from labelled examples.
2. Training Configuration: Adjusted sequence lengths to 1024 tokens and set up parameters for LoRA and batch size to fit the T4 GPU.
3. Training: The model's weights were optimized iteratively, with loss values logged to monitor progress.

The result was an improved Llama3.1 8B model, that generated better Overpass queries than before. The syntax of the queries improved noticeably and it demonstrated a better knowledge of the available OSM tags. However, it exhibited a few limitations. The model struggled with complex prompts, especially those deviating from the training data. It lacked flexibility for new cases and could not refine queries interactively. Overall, it did not produce the expected results.

Llama3.1 70B

Given the limitations of the 8B model, we decided to explore fine-tuning the Llama3.1 70B model, which is significantly more powerful and capable of handling complex queries. However, this required substantially more computational resources. The 70B model requires 80GB of VRAM for effective fine-tuning, far exceeding the capabilities of a Google Colab instance with a Tesla T4 GPU. Despite upgrading to a Pro subscription, Colab's available GPUs were insufficient. We requested access to a high-performance server from OST, which provided ample RAM but fell short on GPU memory. Although it could handle inference for the 70B model, fine-tuning was not feasible.

At this point, we had already spent too much time exploring different approaches of fine-tuning open source LLMs, that we had to cut our losses and decided to choose the OpenAI approach for the LLM processing steps of the plugin.

GPT-4o-mini

Using OpenAI for our plugin was not our preferred option. The goal was to keep the plugin open source, but due to technical and time constraints, we decided to explore OpenAI's GPT models. We began with GPT-4o-mini, a smaller and less resource-intensive variant of the GPT-4 models. Unfortunately, the initial results were underwhelming. The model frequently produced incorrect OSM tags, misinterpreting the user's intent or applying unrelated tags. There were numerous syntax errors in the generated Overpass queries, rendering them unusable without significant manual corrections. These shortcomings meant that the model needed to be fine-tuned to the specific requirements of Overpass queries. Without fine-tuning, the output quality was insufficient for seamless integration into our plugin, as the queries could not be sent to the Overpass API reliably.

Fine-tuning in the OpenAI dashboard is pretty simple: You select the model you want to fine-tune and upload the training set in a specific format. Optionally, you can configure parameters such as batch size and the number of training epochs. By default, these parameters are automatically determined based on the dataset. For the fine-tuning process, we used the same training dataset that was utilized for the Llama3.1 model. The dataset was formatted to align with OpenAI's specifications.

The results demonstrated a slight improvement in syntax accuracy, with fewer structural errors in the generated queries. It showed better alignment with Overpass syntax, particularly for prompts resembling those in the training dataset. However, the overall output quality was still unsatisfactory. Many queries remained unusable due to semantic inaccuracies or incomplete tag matching, requiring manual intervention to correct.

These results suggested that while GPT-4o-mini could grasp the fundamentals of Overpass syntax after fine-tuning, its smaller size and limited capacity to generalize hindered its effectiveness for complex or varied prompts. Given the limitations of GPT-4o-mini, we decided to test the larger and more powerful GPT-4o model.

GPT-4o - Initial fine-tune

The cost of fine-tuning the bigger model GPT-4o was significantly higher than fine-tuning GPT-4o-mini. We estimated that if we fine-tuned GPT-4o with the full training dataset of slightly over 6'000 queries, it would cost around 30-50 USD, depending on the total amount of tokens in the dataset. We decided to train the model with a smaller dataset of around 100 selected queries instead. This smaller dataset was curated to include a variety of Overpass query structures and various spatial conditions to help the model generalize better.

Once the GPT-4o model was fine-tuned with the smaller dataset, the results showed significant improvement compared to the previous models:

- Improved syntax accuracy: Unlike GPT-4o-mini and Llama3.1 8B, the fine-tuned GPT-4o produced syntactically correct queries more consistently.
- Semantic understanding: GPT-4o exhibited a better understanding of natural language prompts, even when they contained ambiguous or complex phrasing.
- Reduction in manual corrections: The generated Overpass queries required less manual correction. For simple to moderately complex prompts, the queries were ready to be passed directly to the Overpass API.

Although the model is far from perfect and sometimes still produces questionable queries, it is definitely an improvement to the other fine-tuned models we tested.

GPT-4o - Fine-tune with more data

Although the model was good enough for simple to moderate queries, it still had a lot of potential for improvement. For the next fine-tuning attempt, a training set of 500 queries were selected and a validation set of another 100 queries. These were picked out of the OverpassNL dataset, but it was filtered beforehand to only include queries that use the Overpass Turbo extension `geocodeArea`. We need the model to use `geocodeArea` because we want to be able to reliably replace that part with our own geolocation processing. After training this model and doing a few black-box-tests, it seemed to perform better than the initial GPT-4o fine-tune. To evaluate the performances of the models, another dataset of 100 evaluation queries were randomly picked and sent to each model to measure their performance.

3.2.3 Model Evaluation

Each model was evaluated on the same dataset of 100 evaluation queries to ensure a fair comparison of their performance. These queries were randomly selected from the OverpassNL dataset to represent a wide range of scenarios. The BLEU score was used to measure the similarity between the outputs generated by the model and the expected reference outputs, with scores closer to 1.0 indicating a better match. The Exact Match Rate assesses the percentage of outputs that perfectly matched the reference, providing a stricter measure of accuracy.

The following labels are used in the visualizations for the evaluated models:

1. **gpt-4o-mini**: The base model of OpenAI's GPT-4o-mini
2. **gpt-4o**: The base model of OpenAI's GPT-4o
3. **gpt-4o-mini-2024-07-18-AWrDJgaa**: Fine-tuned model of GPT-4o-mini, fine-tuned with 100 training queries.
4. **gpt-4o-2024-08-06-AWq0zhwn**: Fine-tuned model of GPT-4o, fine-tuned with 100 training queries.
5. **gpt-4o-2024-08-06-AfdKTtGF**: Fine-tuned model of GPT-4o, fine-tuned with 500 training queries and 100 validation queries.
6. **gpt-4o-2024-08-06-AfwdZ90e**: Fine-tuned model on top of gpt-4o-2024-08-06-AfdKTtGF, using a fine-tuning method called Direct Preference Optimization (DPO).
7. **LLaMA**: Fine-tuned model of Meta's Llama3.1 8B, fine-tuned with 6000 training queries.

BLEU Score

The Bilingual Evaluation Understudy (BLEU) score is a metric used to evaluate the quality of text generated by a model by comparing it to a reference text. It is widely used in natural language processing for tasks such as machine translation and text generation. BLEU measures the degree of overlap between n-grams (sequences of words) in the generated text and the reference text. Once the BLEU score for every model was calculated, the metric was visualized in the following figures.

Average BLEU Score

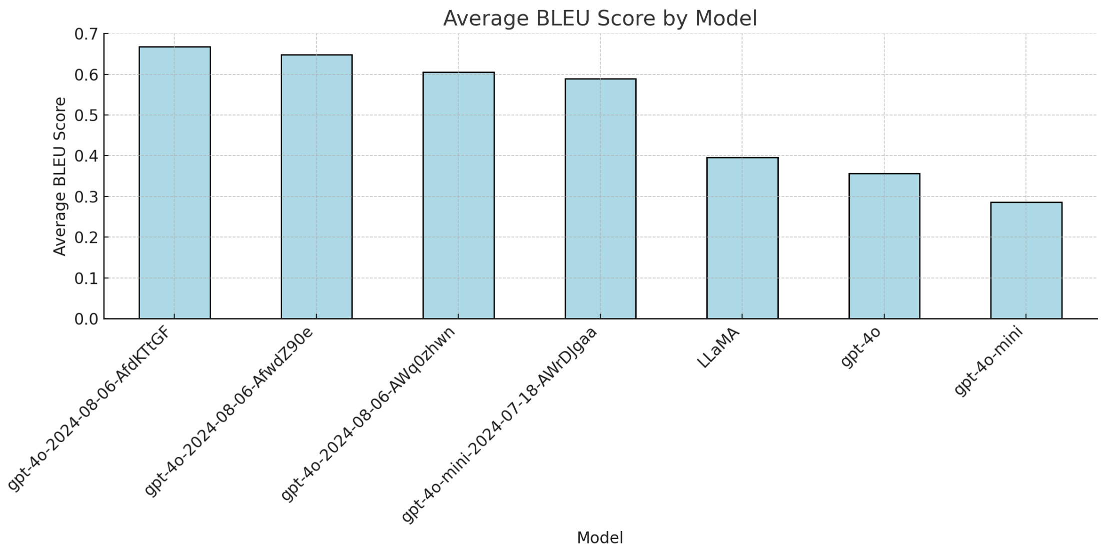


Figure 3.1: Average BLEU Score by Model

Figure 3.1 shows the average BLEU score achieved by each model across the evaluation dataset. The fine-tuned models generally outperformed their base counterparts, indicating that fine-tuning improved alignment with the reference outputs. The best-performing model was gpt-4o-2024-08-06-AfdKTtGF, which achieved a slightly higher average BLEU score, showing its superior ability to generate syntactically and semantically accurate Overpass queries. Although the Llama3.1 8B model performed better than the base GPTs, its score was still significantly lower than the fine-tuned GPTs.

BLEU Score Distribution

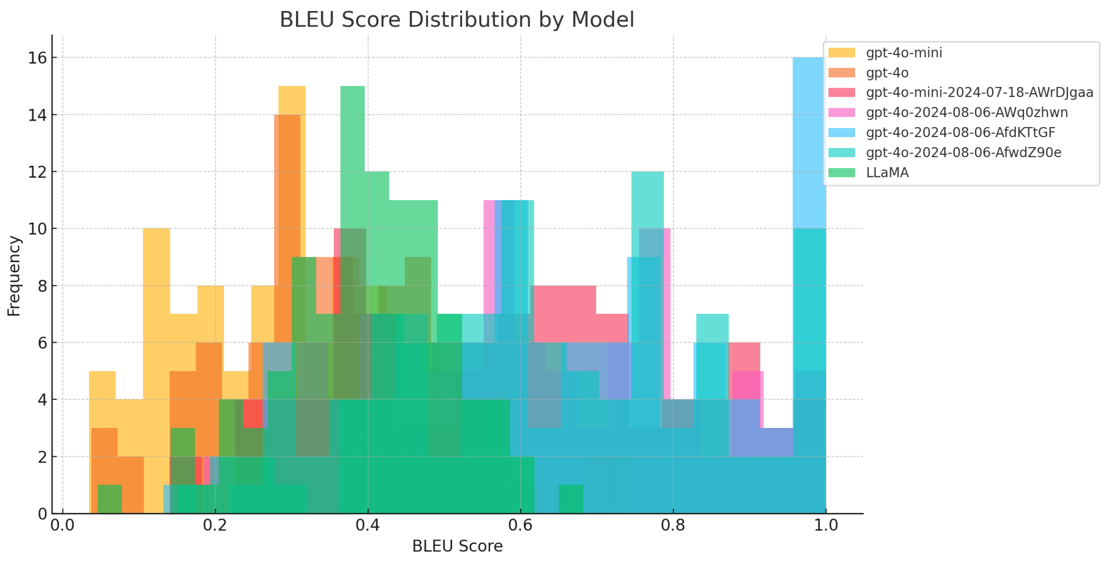


Figure 3.2: BLEU Score Distribution by Model

Figure 3.2 presents the distribution of BLEU scores for each model, providing insights and variability of their performance. Fine-tuned GPT models display narrower distributions with higher concentrations near 1.0, indicating greater consistency and accuracy across varied natural language queries. The presence of BLEU scores at 1.0 for several models reflects their ability to perfectly translate a significant number of natural language queries into correct Overpass queries.

Llama3.1 8B vs Best GPT Model

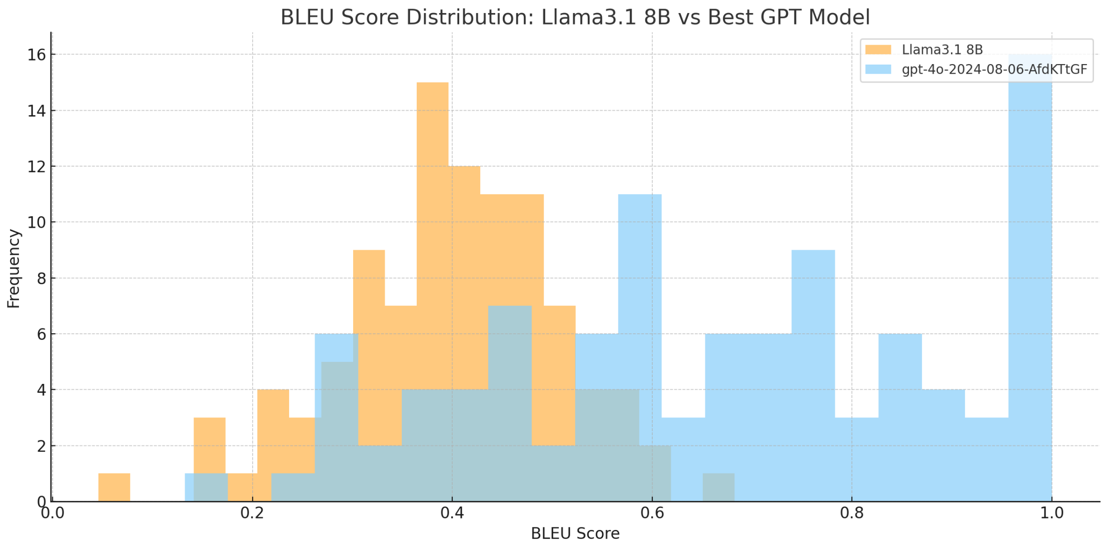


Figure 3.3: BLEU Score Distribution: Llama3.1 8B vs Best GPT Model

Figure 3.3 compares the BLEU score distributions of Llama3.1 8B and the best-performing fine-tuned GPT model. These results indicate that the fine-tuned GPT model outperforms the Llama3.1 8B model, even though the GPT model was trained with 10-times less queries.

Exact Match Rate

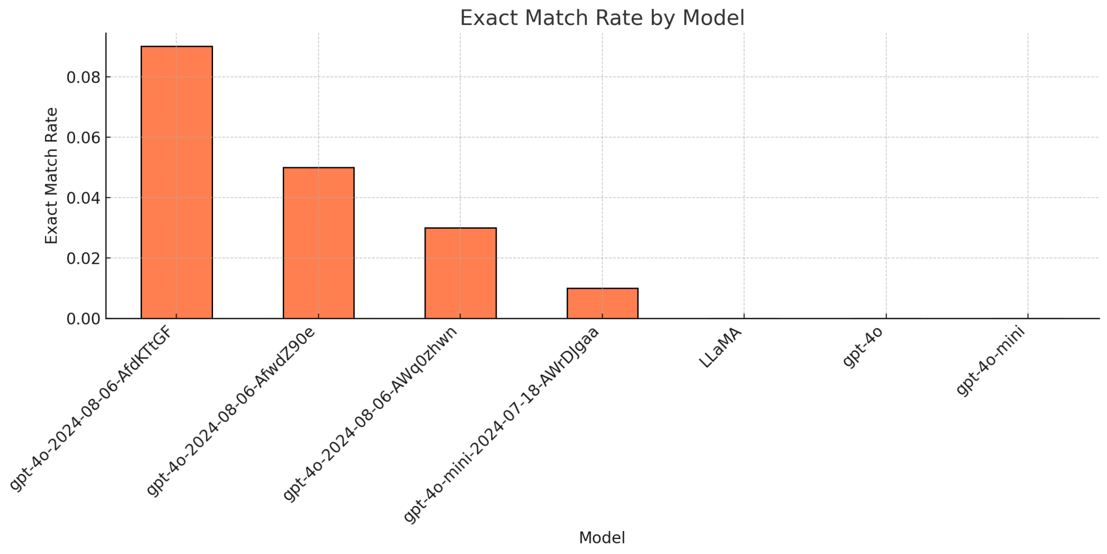


Figure 3.4: Exact Match Rate by Model

Figure 3.4 illustrates the percentage of exact matches achieved by each model. Again, the gpt-4o-2024-08-06-AfdKTtGF model outperformed the other models. A lower exact match rate for some models, despite a reasonable BLEU score, suggests that these models may generate outputs that are close but not identical to the reference.

3.3 Geolocation Processing

Geolocation processing is a key pre-processing step in our prompt engineering workflow. This process involves several steps to generate a usable prompt extension from the geoname provided in the input prompt.

3.3.1 Geocoder Evaluation

A geocoder is essential to this task as it allows us to extract relevant data about the geoname directly from OpenStreetMap (OSM). This step ensures that we do not rely solely on our fine-tuned LLM for accurate location information.

Using a geoname, we extract valuable details from OSM, including the OpenStreetMap ID, relevant coordinates and administrative information such as state, city and town names. The relevant coordinates include the administrative centre of the location, a bounding box formed by four points that define the extent of the area, and any OSM objects that delineate the boundary of the location.

To make the usage of the geocoder easier, we decided to use geopy.

We evaluated three geocoders for this task: Nominatim, Pelias and Photon.

Nominatim

Nominatim is a widely used geocoder for OpenStreetMap data, known for its simplicity and ease of use. It provides straightforward geocoding and reverse geocoding capabilities. However, it lacks support for elastic string matching, which makes it less flexible when handling variations or misspellings in geonames. While Nominatim is a solid choice for basic geocoding needs, its limitations in handling complex queries and dynamic string matching made it less suitable for our requirements.

Pelias

Pelias is a powerful geocoding engine designed for high customization and scalability. It offers advanced features like multi-language support and customizable data sources, making it a robust solution for complex geocoding tasks. However, Pelias does not provide an out-of-the-box API, which adds significant complexity to its setup and integration. The additional effort required to self-host and configure Pelias was deemed unnecessary for our proof-of-concept project, especially given the availability of simpler alternatives.

Photon

Photon emerged as the most suitable choice for our project due to its support for elastic string matching and its existing API, which simplified integration. As an open-source geocoder, Photon provides flexibility and can be self-hosted if needed, making it a scalable option for future deployment scenarios. For our proof-of-concept, the demo API from komoot and string-matching capabilities were more than sufficient to meet our needs.

3.3.2 Area Methods

The goal of the processing is to get the desired location of the user and preprocess it to make it understandable for our LLM. In every case we want to have a geographical area / area-of-interest according to what the user requests.

With these informations we concluded, that there are 4 different area methods we have to cover:

- Polygon from Openstreetmap
- bounding box using an extent from an Openstreetmap object
- bounding box using a coordinate from the center of an Openstreetmap object
- bounding box from QGIS

In Overpass, a bounding box (bbox) is used to specify a rectangular area of interest.

Each coordinate represents one side of rectangular box, to get a corner of a bounding box, you would need the two sides connected to the corner.

Polygons from OSM

Polygon from OSM in our context refers to delegating the computation of spatial data to Overpass, as Overpass will automatically compute the correct Polygon. If a geoname has an associated OSM element and ID, we can use OverpassQL by applying the appropriate code and extending the prompt provided to our LLM with the relevant details.

BBox with extent from OSM

If the Openstreetmap type and ID are not available, we check whether the object has an extent. The extent refers to a bounding box defined by four coordinates.

BBox with center coordinate from OSM

In the final phase of the geocoding process, if no Openstreetmap type and ID are available and the object has no extent, we will revert to the administrative center of the location and generate a bounding box around it using a default radius.

BBox from QGIS

If the user wants to set the current view as the bounding box, we will need to give that information to Overpass.

3.3.3 Initial Implementation

The initial implementation of the geolocation processing was very simple. We created a combobox where the user can select which of the 4 area methods he wants to use. For the area methods reliant on geocoding, we started by simply extracting the location using the GPT-4o-mini model, we only want to have the geoname, which was mentioned in the user's prompt. Using the result of the GPT-4o-mini model, we did a Photon request. From the Photon request, we used the first result, as it usually is the most relevant result.

Polygon from OSM

After extracting the geoname from the result, we used the OSM ID and created a simple prompt extension like the following:

For the city St. Gallen that would look like this:

”Use relation(1683941) for location St. Gallen”

This notation can be used inside Overpass to outsource the geocoding to the Overpass interpreter for a certain known OSM geolocation.

BBox with extent from OSM

If there is no OSM ID available, we fall back to the extent. The extent is just a list of 4 coordinates building a bbox, that can be available in some results of photon. Using this bbox, we will extend the final prompt for the fine-tuned LLM.

BBox with center coordinate from OSM

If there is neither an extent nor an OSM ID in the first result of the Photon request, we will fall back to the center coordinate, which is always a part of a Photon object. By applying a default radius of 500 to the center coordinate, we create a bbox around it. The code for this looks like this

We first extract the longitude and latitude from our coordinate. Then we calculate an offset for longitude and latitude. The value 111’320 meters is 1 degree of latitude, by dividing it through the radius we can get the offset. For the longitude varies with latitude. At the equator, it is also about 111’320 meters, but it decreases as you move toward the poles. The mathematical functions for the longitude offset give you how many degrees of longitude correspond to the given radius at the specific latitude. We use this bbox to extend the final prompt for the fine-tuned LLM.

BBox from QGIS

There is always the possibility that the user wants to use the current view inside QGIS for his query. To extract the bbox in QGIS we can use standard functions from QGIS to get the map view, form there on, we will have to extract each point from every side to build the bbox.

In our project, we use the coordinate reference system (CRS) EPSG:3857, but OverpassQL works with the CRS EPSG:4326. These are different frameworks used to define how geographic or spatial data is mapped to real-world locations. It provides the rules for describing geographic coordinates (latitude and longitude) or projected coordinates (e.g., meters or feet on a flat surface) and how they relate to the Earth’s surface.

Before we can use the coordinates from QGIS, we will have to transform them to EPSG:4326. There is a QGIS standard command, which does that for us. After that we can just extend the final prompt with the bbox information.

3.3.4 Problems

Firstly, the current method of area selection is not user-friendly and requires users to have prior knowledge of geolocation, which is contrary to our goals.

Secondly, Photon's results often fail to meet user expectations. In some cases, when searching for a town, the first result returned is not the area of the town but a point within the town, leading to inaccuracies.

The third problem is, that the OverpassQL syntax for OSM elements with OSM IDs does not work consistently. OverpassQL does not always calculate polygons correctly using the "relation(OSM ID)" notation, and the same problem occurs with ways.

To fix these problems we need to re-implement the process. This new implementation should automatically select the most appropriate area method, accurately determine the correct location, and provide a more reliable approach to retrieving areas.

Our final solution combines new aspects with some features of the initial implementation.

3.3.5 OpenAI Assistant

Initially, we used the GPT-4o-mini model to extract geonames from user prompts. However, we decided to replace it with an OpenAI assistant to enhance functionality and integration.

OpenAI assistants are systems designed to interact with users using natural language, providing answers, generating content, solving problems, and performing tasks. They can also return structured outputs, such as JSON, enabling seamless application integration. In our case, we configured the assistant to deliver structured JSON outputs according to a JSON schema.

The assistant continues to extract geonames from user prompts but now includes an additional feature: evaluating the geonames to determine their corresponding OSM (OpenStreetMap) types. This evaluation is straightforward: the assistant analyses the user's prompt, identifies each geoname mentioned, and determines which of the following OSM elements is most appropriate:

- Node: Represents a point, such as a single building or a precise address.
- Way: Refers to linear features, like streets or roads.
- Relation: Represents complex entities, such as countries, counties, states, or cities.

Additionally, we introduced a flag that allows users to specify whether they want to use the bbox from QGIS. If this flag is set, we will use the bbox provided directly by QGIS.

3.3.6 Area Method Processing

Now that we have a preference of an OSM element for each location, we can select the result of Photon based on the OSM element.

We removed the limit of 1 result to up to 5 geolocations, from which we try to recognize the most suitable one, by comparing the preference of the OSM element generated by the assistant. Meaning that if we get a way as the first result, but the preference is the OSM element relation, that result is skipped. We save the first result that matches the requested OSM type.

If there was no result regarding the requested type, we will fallback on the first result of the Photon request.

Now that we have calculated the most suitable Photon result of the geonames, we can go on to do the selection of area methods.

As we now have a preferred OSM element, we will always go after that, but there is a difference, between the processing of nodes and the processing of ways and relations. The reason for that is most of the time the user will input ways and relations and for those we can use the OverpassQL syntax ”area(OSM ID)”. To use area-function with the OSM ID of ways and relations, we have to add 2’400’000’000 to the OSM ID of ways and 3’600’000’000 to the OSM ID of relations. The reason for that is, that OSM has created separate area objects for ways and relations starting at those numbers. As for nodes we will use ”node(id:OSM ID)”. This covers the calculation up to the preferred OSM element, and removes the faultiness of the previous OverpassQL syntax we have used.

The next step is the fallback to the extent. The calculation of the extent has not changed since the initial implementation, as well as the calculation of the bbox around the center coordinate, which is the last fallback if the extent is not available.

Through this fallback chain, we can omit the selection of the area method the user had to choose initially.

With those improvements, we have cleared up the faults in the Problems section of the initial geolocation processing.

3.3.7 Replacement of GeocodeArea

The final enhancement we made to the geolocation processing step was to replace geocodeArea. This was necessary because geocodeArea is not supported by the Overpass API interpreter, being a feature of the Overpass Turbo extension.

Our fine-tuned model is familiar with the syntax of geocodeArea because it is both simple and easy to learn. This simplicity allows us to seamlessly replace it with the standard area syntax supported by the Overpass API. To implement this, we used a regex to detect instances of geocodeArea in the input and replace them with the appropriate OSM ID with the area syntax for the specified location. The OSM ID required for the replacement is stored in the previous step of the geolocation processing, under the same name as the city. The whole regex matching method can be viewed below.

If no geocodeArea syntax is detected, it probably means that no preferred OSM feature was found. In such cases the fallback options - such as the QGIS bounding box - were used instead.

3.4 Embedding

The embedding functionality is another important step in the pre-processing workflow. To get all related OSM tags for a query, the plugin uses precomputed embeddings, OpenAI's semantic embedding model, and entity recognition. For instance, when a user searches for "Find cafes in St. Gallen", the system identifies "cafes" as the main entity, searches its vector database for similar tags, and returns relevant results such as amenity=cafe with descriptions like "Places serving coffee and snacks." These tags can then be used to generate Overpass queries and display the corresponding data in QGIS.

3.4.1 Introduction

Semantic embeddings are numerical representations of words, phrases or whole sentences that encode their semantic meaning in a high-dimensional vector space. These embeddings are generated using machine learning models trained on large amounts of text, where words with similar meanings or contexts are mapped to vectors that are close together in the vector space. For example, the words "cafe" and "coffee shop" might have very similar embeddings because they are often used interchangeably in similar contexts.

Semantic embeddings allow us to measure the similarity between pieces of text by calculating the distance or angle between their vectors. A common approach to measuring similarity is cosine similarity, which measures how close two vectors are to each other. The closer the vectors, the more similar the meaning of the text.

In the context of our QGIS plugin, semantic embeddings are particularly valuable for matching natural language queries to OSM tags and descriptions. OSM tags follow a structured, domain-specific vocabulary that can sometimes differ from the language used in user queries. For example, a user might search for "bicycle parking" while the corresponding OSM tag is `amenity=bicycle_parking`. By embedding both the user query and the OSM tags/descriptions in the same vector space, we can compute their semantic similarity and identify the most relevant tags, even if the wording is different.

3.4.2 Precomputed Embeddings

To ensure efficient and accurate tag matching, our implementation relies on precomputed embeddings for OSM tags and their descriptions. These embeddings were generated in advance and stored in a compressed format, enabling fast retrieval and similarity computation during runtime. This pre-computation process eliminates the need for real-time embedding generation, significantly improving the plugin's performance.

The tags and descriptions used for embedding generation were sourced from the TagFinder (Gwerder, 2024) thesaurus, which provides a structured dataset of OSM key-value pairs and their associated meanings. The extracted tags and descriptions from the thesaurus were saved into a CSV file for easier processing. Each tag-description pair from the CSV was combined into a single text input and processed using OpenAI's `text-embedding-3-small` model to create a high-dimensional vector representation. The embeddings, along with their corresponding tags and descriptions, were then saved locally in a compressed .npz file.

This approach ensures that when a user provides a query, the system can efficiently compare the query's entities embedding to the precomputed embeddings, performing similarity computations without any delays.

3.4.3 Entity Recognition

For entity recognition, we use the OpenAI GPT-4 Turbo model. This step extracts the main entities from user queries to ensure accurate OSM tag matching. For instance, in a query like "List all coffee shops and bakeries in St. Gallen", the system identifies "coffee shops" and "bakeries" as the entities of interest while ignoring location-specific phrases, such as "St. Gallen". These entities are then embedded semantically using OpenAI's `text-embedding-3-small` model.

System Prompt for Entity Recognition

The entity recognition process starts by defining a system prompt for the GPT-4 Turbo model. This prompt provides clear instructions to the model, specifying its role and the desired output format.

Querying the Model

When a user provides a natural language query, the `get_entity_from_query` method sends the input to the OpenAI GPT-4 Turbo model along with the predefined system prompt. The model processes the query and returns a structured JSON response containing the recognized entities.

Parsing the Response

The response from the model is expected in JSON format, containing an array of entities.

The method parses the JSON response and extracts the entities. If parsing fails, a fallback mechanism ensures that the raw response is treated as a single entity to avoid disrupting the workflow.

3.4.4 Embedding Entities and Similarity Computation

After extracting entities from the user query, the next step involves embedding these entities into a semantic vector space and computing their similarity with precomputed embeddings of OSM tags.

Embedding the Query Entities

The extracted entities from the user query are first transformed into embeddings using the same model that was used for pre-computing the tags: OpenAI's `text-embedding-3-small`.

Similarity Computation

The next step involves computing the semantic similarity between the query embeddings and the precomputed embeddings. This is achieved using cosine similarity, a metric that measures the cosine of the angle between two vectors in a high-dimensional space. Vectors with smaller angles (closer to 1 in cosine similarity) represent more semantically similar texts.

In the `find_similar_tags method`, cosine similarity is computed between the query embedding and all precomputed embeddings:

Filtering and Ranking

After computing similarity scores, the tags are filtered and ranked based on a predefined `similarity_threshold` and the `top_k` most relevant results are selected. These values are initialized with the following values: `similarity_threshold = 0.3` and `top_k = 10`.

The remaining tags are sorted by similarity score in descending order, and the `top_k` results are selected: Finally, the `find_similar_tags` method returns a structured dictionary where tags are grouped by entity. Each group contains the most relevant tags for the respective entity:

3.4.5 Potential for Improvement

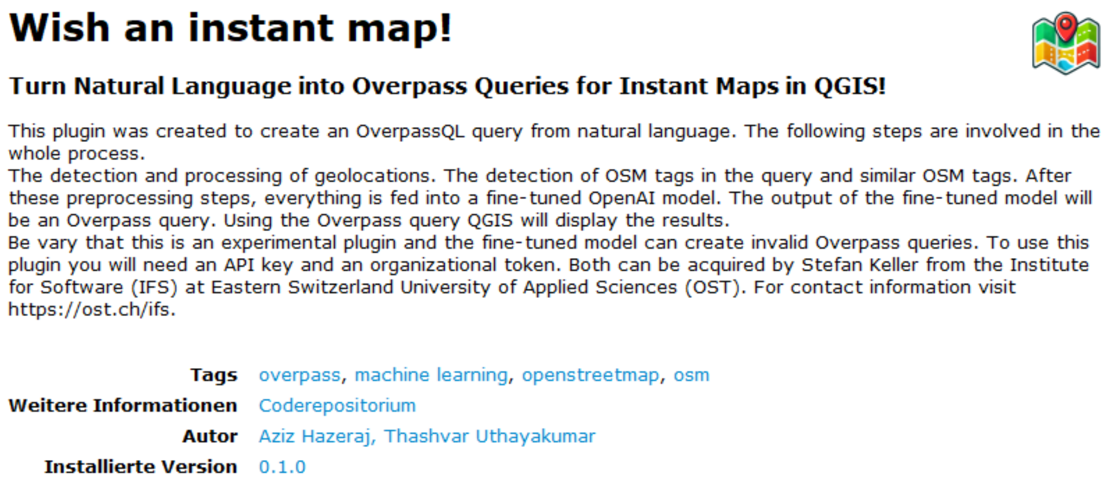
There are a few areas where this part of the pre-processing can be optimized. There is still potential for improvement in the quality and completeness of the precomputed embeddings. While these embeddings are a good foundation, they could benefit from refinement. Additionally, the implementation currently lacks a mechanism for understanding relationships between tags, such as parent-child relationships or hierarchical structures. Incorporating a thesaurus or ontology of OSM tags could address this limitation. For example, a thesaurus could define relationships like ”restaurant” as a parent tag of ”cafes,” enabling the function to provide broader matches when required. Finally, exploring open-source alternatives instead of using OpenAI's embedding model could improve both accessibility and cost-effectiveness.

3.5 QGIS Plugin

The QGIS Plugin is that part of the project, which will serve as the input and output of our tool. The development of the UI in QGIS works with the PyQt framework, which is a cross-binding from the C++ framework Qt. But there are several different parts to a QGIS plugin.

3.5.1 metadata.txt

The metadata.txt file is a part of every QGIS plugin out there, it is the first thing someone will look for if they want to know more about the plugin. Our metadata can be seen in the figure 3.5. It contains the title, a subtitle and a careful description of our plugin, and how you can start to use it. There are QGIS plugin tags, which are relevant to our plugin and a link to the code repository. Beneath it are the authors, and the installed version of the plugin.



Wish an instant map!

Turn Natural Language into Overpass Queries for Instant Maps in QGIS!

This plugin was created to create an OverpassQL query from natural language. The following steps are involved in the whole process.
The detection and processing of geolocations. The detection of OSM tags in the query and similar OSM tags. After these preprocessing steps, everything is fed into a fine-tuned OpenAI model. The output of the fine-tuned model will be an Overpass query. Using the Overpass query QGIS will display the results.
Be vary that this is an experimental plugin and the fine-tuned model can create invalid Overpass queries. To use this plugin you will need an API key and an organizational token. Both can be acquired by Stefan Keller from the Institute for Software (IFS) at Eastern Switzerland University of Applied Sciences (OST). For contact information visit https://ost.ch/ifs.

Tags	overpass, machine learning, openstreetmap, osm
Weitere Informationen	Coderepositorium
Autor	Aziz Hazeraj, Thashvar Uthayakumar
Installierte Version	0.1.0

Figure 3.5: metadata of our plugin

3.5.2 Settings

As we are using OpenAI, we will need the OpenAI API key from the user. Additionally, we also need an organizational ID from the user to enforce the usage of a certain organization.

Therefore we want to add those fields on an options page. QGIS already has a standard options page, which we can extend with a page for our plugin.

For that we first need to create the UI. For the user-interface we saw and implemented a similar Options user-interface to that of the AIAMAS project, which was also created under our advisor. We modified and extended the UI so that it applies to our plugin. We changed it so that there is an OpenAI API key field. Then we added another line edit field, where the user can type in the organizational token and we added an expert-mode checkbox to toggle the expert-mode. At last there is a field verbose logging. In QGIS there is a python console, and that is where the verbose logging will be logged.

For both tokens in our application, there is a token check, where if one of them is not visible, there will be a warning message at the top of QGIS.

3.5.3 Dock Widget

The primary interface for our plugin is a dock widget. In the PyQt framework, dock widgets are versatile floating windows that can be docked within the QGIS interface. This allows users to attach the widget to a specific side of the QGIS window or use it as a standalone floating window. An example of a floating dock widget is shown in the figure 3.6.

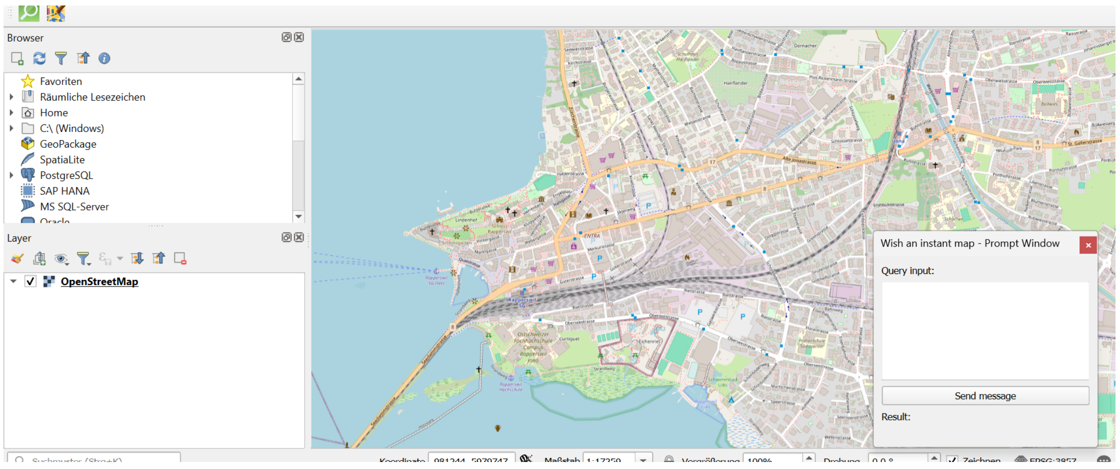


Figure 3.6: floating dock widget view

If the dock widget is docked inside of QGIS, it will look like in figure 3.7

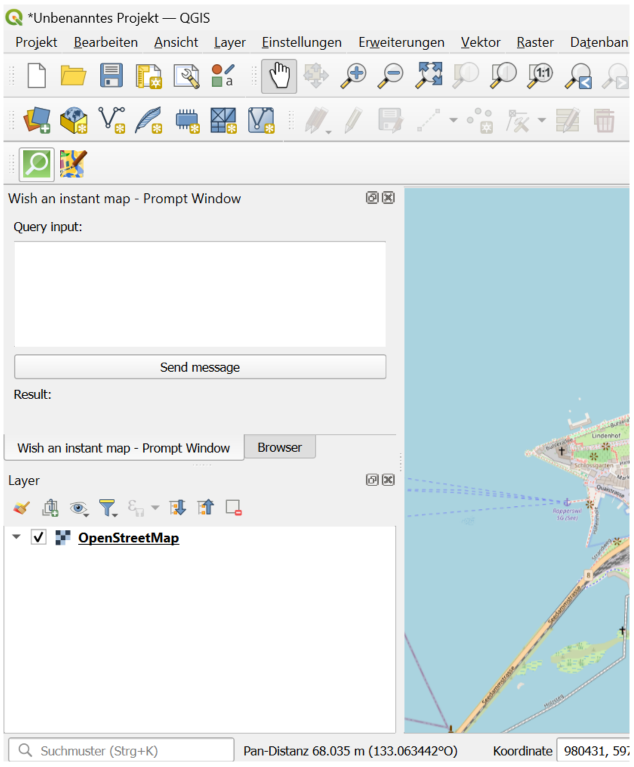

*Unbenanntes Projekt — QGIS
Projekt Bearbeiten Ansicht Layer Einstellungen Erweiterungen Vektor Raster Datenban
Wish an instant map - Prompt Window
Query input:
Send message
Result:
Wish an instant map - Prompt Window
Browser
Layer
OpenStreetMap
Suchmuster (Strg+K)
Pan-Distanz 68.035 m (133.063442°O)
Koordinate 980431, 59

Figure 3.7: docked widget view

The final user-interface consists of a ”QTextEdit”- PyQt Object, which is basically a big text area input field, where the user can type in his whole prompt. In addition, there is a button to start the processing of the prompt, and there is a result label to show errors or the amount of added nodes, ways and relations. We decided to keep the user-interface as simple as possible, as we are striving for a user-friendly plugin.

Expert-mode

As the standard user does not see a lot of the processing and the OverpassQL query in the background, we have decided to add an expert-mode. The expert-mode has a history of queries executed in the current QGIS session. Each of the previous OverpassQL queries can be viewed, modified and executed. For example, there could be the case that the OverpassQL has a small error, which the expert-user detects and fixes, afterwards he can just run the query to see the result.

Verbose Logging

Verbose logging is also not thought for the standard user of our plugin. It is more useful for developing, but also shows what is happening in the background of our plugin, as every step will be printed in the python console of QGIS.

3.5.4 View Layers

The plugin processes Overpass query results to generate QGIS-compatible layers. The following describes the workflow and necessary transformations to convert the Overpass API JSON response into effective QGIS layers to display to the user.
The process can be broken down into the following processing steps:

- Query Execution and Response Parsing
- GeoJSON Feature Classification
- Creating QGIS Layers
- Filtering and Enhancing Points
- Styling Layers

Query Execution and Response Parsing

First, the Overpass query is sent to the Overpass API via the `call_overpass_api` method. The response is received in JSON format, which is then converted to GeoJSON using the `osm2geojson` library. We initially tried to manually create the points, lines and polygon layers directly from the Overpass JSON, so that we would not need the `osm2geojson` library. This turned out to be the wrong decision, as the raw JSON from Overpass would not return in a uniform format, where the nodes could be extracted easily. GeoJSON, on the other hand, is a standard format supported by QGIS, so it was much easier to implement the different layers.

GeoJSON Feature Classification

To accommodate QGIS's layer structure, the plugin organizes the GeoJSON features into separate collections based on geometry type:

- Points: Features with `geometry.type` set to `"Point"`.
- Lines: Features with `geometry.type` set to `"LineString"`.
- Polygons: Features with `geometry.type` set to `"Polygon"` or `"MultiPolygon"`.

Creating QGIS Layers

Each classified feature collection is used to create corresponding QGIS vector layers. A helper function, `add_geojson_layer`, converts the GeoJSON strings into QGIS layers using the `QgsVectorLayer` API. Then the layers are validated for correctnesss and added to the QGIS map.

Filtering and Enhancing Points

When both polygon and point layers exist, the plugin removes points that overlap with polygon geometries. Otherwise the polygon edges would also be shown as a collection of points. This is achieved using the `remove_used_points` function. The function identifies points overlapping polygon geometries and removes them using the geometry operation `difference`.

Styling Layers

To ensure clear visualization, styles are applied automatically to each layer:

- Points: Styled with green markers with outlined edges.
- Lines: Styled with blue lines
- Polygons: Styled with semi-transparent orange fills with outlined borders.

3.5.5 Dependencies

Installing dependencies in QGIS is a common problem for QGIS plugin developers. Moreover, there are some general known problems with dependency handling in QGIS:

- we need to have the dependency available at runtime, meaning we need to have the dependency installed before starting our plugin
- certain plugins use specific versions of libraries
- libraries are not always cross-platform compatible

There are several different approaches to this problem, which we have tested, including:

- The use of binary distributions
- installing dependencies in the plugin scope
- letting the user install the dependencies
- installing the dependencies in the normal python library scope

Binary distributions

Wheel packages are a binary distribution format for Python libraries, identified by the .whl extension. They contain precompiled code and metadata, allowing for faster and more efficient installations by eliminating the need for compilation during setup. Although wheels are the preferred format for distributing Python packages, we ultimately decided against this approach for the following reasons:

- It requires a separate binary for each minor Python version.

- Changing the Python interpreter's import path in QGIS could potentially interfere with other plugins, which we aim to avoid at all costs.

Although technically feasible, this approach is also incompatible with the QGIS plugin repository policy, which does not allow plugins that use binaries to be uploaded.

Plugin Scope Installation

We also explored the approach of installing dependencies within the scope of our plugin directory. The main advantage of this method is that it prevents other plugins from overwriting the versions of our dependencies. However, this approach requires manipulating the import path in QGIS, which could cause conflicts with other plugins using the same dependencies. To avoid such potential interference, we decided against this method.

Installation by user

We considered the approach of providing users with instructions to install the required dependencies themselves, as this method has reportedly worked in some cases. However, we found this solution unsatisfactory, as it does not align with our goal of creating a user-friendly plugin. Requiring users to manually install dependencies adds unnecessary complexity.

While this method installs libraries within the standard Python library scope, we also explored automating the process by installing the dependencies programmatically within the plugin. Therefore, the approach of having the user install the library was also ultimately abandoned as well, as it did not meet our standards for simplicity and user experience.

Installing libraries in library scope

The final approach we adopted was to automatically install the required libraries into the standard Python library scope via code. This method minimises interference with other plugins while maintaining usability, making it the ideal solution for the final version of our plugin.

When the plugin starts, it checks for the availability of the required dependencies by attempting to import them. If the dependencies are not already installed, the plugin will install them automatically.

Our plugin relies on three key dependencies, which are automatically installed if not already present: OpenAI, Geopy and Osm2geojson.

OpenAI

The OpenAI Python library simplifies the integration of OpenAI's models into Python projects. It manages essential tasks such as sending prompts, receiving responses, and handling authentication. We rely on the OpenAI Python library for all our requests to OpenAI, including interactions with an OpenAI assistant, calls to the GPT-4-turbo model, OpenAI embeddings, and our fine-tuned GPT-4o model.

Geopy

The geopy library streamlines working with geographic data in Python by providing seamless integration with various geocoding services. We use geopy in particular because it supports the Photon geocoder, which is essential for our implementation.

Osm2geojson

This library facilitates the conversion of Overpass results into a format suitable for use in QGIS. The osm2geojson library parses Overpass results to allow easy import into QGIS. Directly importing results from Overpass can be challenging for QGIS as they are provided as OSM elements rather than the point, line or polygon formats natively supported by QGIS and GeoJSON.

3.5.6 Plugin installation

There are two ways to install a plugin in QGIS. The primary method is to use the QGIS plugin repository, while the alternative is to manually install a ZIP file. Since our plugin is a proof of concept (POC), we do not plan to upload it to the plugin repository. As a result, our plugin will only be installed using the ZIP file method.

To create the required ZIP file, simply compress the parent folder containing the plugin code. Make sure that the folder structure is preserved so that QGIS can recognise and load the plugin correctly.

Chapter 4

Conclusion

4.1 Results

The result of this work is a proof of concept (POC) called ”Wish an Instant Map! (WAIM). The graphical user interface (GUI) includes a text input field at the top where users can enter their query, and a result label at the bottom where text-based results are displayed. The primary output is the visual representation of the point layer, line layer and polygon layer within QGIS. Users can specify a geoname in their prompts and automatically include the current map extent from QGIS. In addition, an expert mode is available for advanced users to view and edit previously generated OverpassQL queries.

4.1.1 Preprocessing Steps

The query generation process involves two primary pre-processing steps, followed by using the outputs of these steps as input to a fine-tuned GPT-4o model to generate OverpassQL queries.

- Geocoding: All geonames mentioned in the user query are processed using the Photon geocoder to ensure accurate geographic references.
- Semantic similarity check: A semantic similarity analysis is performed using an OpenAI embedding model. This step enables the use of RAG to retrieve similar OSM tags to those used in the user’s prompt.

By combining these pre-processing steps, the fine-tuned LLM generates OverpassQL queries with a reasonable degree of confidence.

4.1.2 Performance Metrics

The system was evaluated using a validation dataset of 100 natural language queries. Key metrics include:

- BLEU Score: Fine-tuned models achieved BLEU scores averaging 0.67, significantly outperforming base models and open-source alternatives.
- Exact Match Rate: The best-performing fine-tuned GPT model achieved an exact match rate of 0.09, highlighting the system's ability to produce perfectly accurate queries in specific cases.
- Validity Rate: Most of the generated OverpassQL queries were valid and functional when executed within QGIS, although sometimes the queries were semantically not entirely correct.

Testing Environment: The system was tested both directly on fine-tuned models and through black-box testing within the plugin environment, which included preprocessing steps. This holistic testing approach ensured reliability and functionality.

4.1.3 Challenges and Limitations

Due to time constraints, the project relied on OpenAI for all LLM calls. While attempts were made to fine-tune the Llama 3.1 8B model, the results were below expectations. Training larger open source models, such as Llama 3.1 70B, was not feasible due to infrastructure limitations, as such models require at least 80 GB of VRAM for fine-tuning. These limitations made the project dependent on OpenAI's infrastructure and prevented the realization of a fully open plugin. The fine-tuned GPT models outperformed LLaMA 3.1 8B, achieving better BLEU scores and exact match rates, despite requiring fewer training queries.

4.2 Improvements

As this is a proof-of-concept (POC), there are numerous potential improvements that could significantly enhance the final plugin.

4.2.1 Structured outputs / JSON Schemas

Structured outputs in OpenAI, guided by JSON Schema, enable the generation of responses in a predictable and machine-readable format. By defining a schema, developers can specify the structure, data types, and constraints of the desired output, ensuring consistency and reliability. This approach is especially valuable for applications requiring precise data formats, such as API responses, configuration files, or structured data extraction. JSON Schema reduces the need for extensive post-processing, prevents errors by validating outputs, and enhances integration with downstream systems, making OpenAI models highly effective for tasks demanding structured and consistent outputs. Especially for the generation of OverpassQL this could have added a lot of value. We were not able to implement such structured outputs for our LLM as we did not have enough time for this task.

4.2.2 Thesaurus

The use of a thesaurus could greatly enhance the semantic similarity component of the plugin. By preserving parent-child relationships and other hierarchical structures within the thesaurus, the system could better align user queries with related OSM tags. In the current implementation, tags were extracted from the thesaurus into a CSV format, embedded, and compared for semantic similarity using cosine similarity. This process resulted in the loss of structural relationships, which limited the accuracy and contextual alignment of the retrieved tags. Integrating the thesaurus directly into the query generation pipeline would significantly improve the overall semantic alignment.

4.2.3 Larger LLM Models

Leveraging larger open-source LLMs, such as the new Llama 3.3 70B, could improve the system's ability to handle more complex and diverse natural language queries. Larger models generally offer better generalization and performance due to their greater capacity to understand and generate nuanced outputs. However, fine-tuning such models was not feasible for this POC due to hardware limitations, as these models require significant computational resources, including over 80 GB of VRAM for training.

4.2.4 Open Source Embedding Models

Using open-source embedding models for semantic similarity tasks would reduce dependency on proprietary APIs and enhance the system's openness. Current embeddings rely on OpenAI's infrastructure, which, while effective, limits the plugin's scalability and cost-efficiency. Open-source embedding models, such as Sentence-BERT or similar alternatives, could provide comparable performance and allow for better integration into open-source pipelines, aligning with the original vision of creating a fully open-source plugin.

4.2.5 Improved Fine-Tuning on Open-Source LLMs

Fine-tuning open-source LLMs with more specialized datasets could further enhance their performance for OverpassQL generation. The current attempts to fine-tune the LLaMA 3.1 8B model yielded suboptimal results, likely due to insufficient quality of the fine-tuning data or inadequate training infrastructure. Future efforts should focus on optimizing the fine-tuning process for open-source LLMs, exploring larger datasets, better training methodologies, and leveraging techniques such as Direct Preference Optimization (DPO) to improve alignment with user expectations.

4.3 Conclusion

The development of ”Wish an Instant Map! (WAIM)” aimed to create an open-source solution for translating natural language queries into OverpassQL, leveraging advanced natural language processing and geospatial techniques. While the project achieved important milestones, there were notable gaps between the envisioned goals and the implemented system.

4.3.1 Achievements

The WAIM proof-of-concept successfully demonstrates the potential of integrating natural language interfaces with geospatial data exploration in QGIS. Key functionalities include:

- **Geolocation Capabilities**: The system effectively handles geolocation tasks, enabling bounding box generation for the current map extent in QGIS and incorporating fuzzy location searches using Geopy and the Photon API.
- **OSM Tag Retrieval**: Relevant OSM tags are identified through semantic similarity checks, using embeddings and cosine similarity to align user queries with OSM tags.
- **OverpassQL Generation**: The system enables users to generate and visualize OverpassQL queries with natural language prompts, providing accessibility for non-experts and customization options for advanced users.

4.3.2 Challenges and Limitations

Despite these achievements, several challenges limited the system's ability to fully realize its potential:

- **Dependence on Proprietary Models**: All LLM-related tasks were performed using OpenAI models, diverging from the original goal of employing open-source LLMs. Time constraints and limitations in computational resources and infrastructure prevented the fine-tuning of large open-source models.
- **OSM Tags Relationship Loss**: OSM tags and their hierarchical relationships could not be successfully extracted from a thesaurus. Tags were embedded and evaluated individually for semantic similarity, leading to the loss of structural relationships.
- **Query Reliability**: The reliability of generated OverpassQL queries remains suboptimal. While the system achieves a BLEU Score of 0.67, there is significant room for improving accuracy and robustness, particularly for complex queries.

4.3.3 Lessons Learned

The project highlights the importance of balancing ambitious goals with practical constraints, such as computational infrastructure and development timelines. While the reliance on proprietary tools enabled rapid prototyping, it also underscored the need for scalable and open-source solutions in future iterations.

In conclusion, while Wish an Instant Map! demonstrates the feasibility of natural language to OverpassQL translation, it also reveals key areas for growth. This work serves as a valuable foundation for further research, offering insights into the challenges and possibilities of integrating AI with geospatial systems.

Chapter 5

Project management

5.1 Students

Our team consists of the following 2 students:

- Aziz Hazeraj
- Thashvar Uthayakumar

5.2 Resources

In this section we will talk about the resources, which we have used in this project.

Both of us have knowledge using Python and have completed the AI modules AI applications and AI foundations. Additionally, Thashvar has completed the Data Analytics module.

5.2.1 Time

This module gives 8 credits per student, where 1 credit amounts to around 30 h of work. This results in a total time of 480 h. We have around 14 weeks of time in this project, totaling around 17 hours/person/week.

5.2.2 Cost

The cost of this project consists of the API calls to OpenAI. For a user query, there would regularly be 4 requests, with a maximum of 6 possible.

5.2.3 Tools

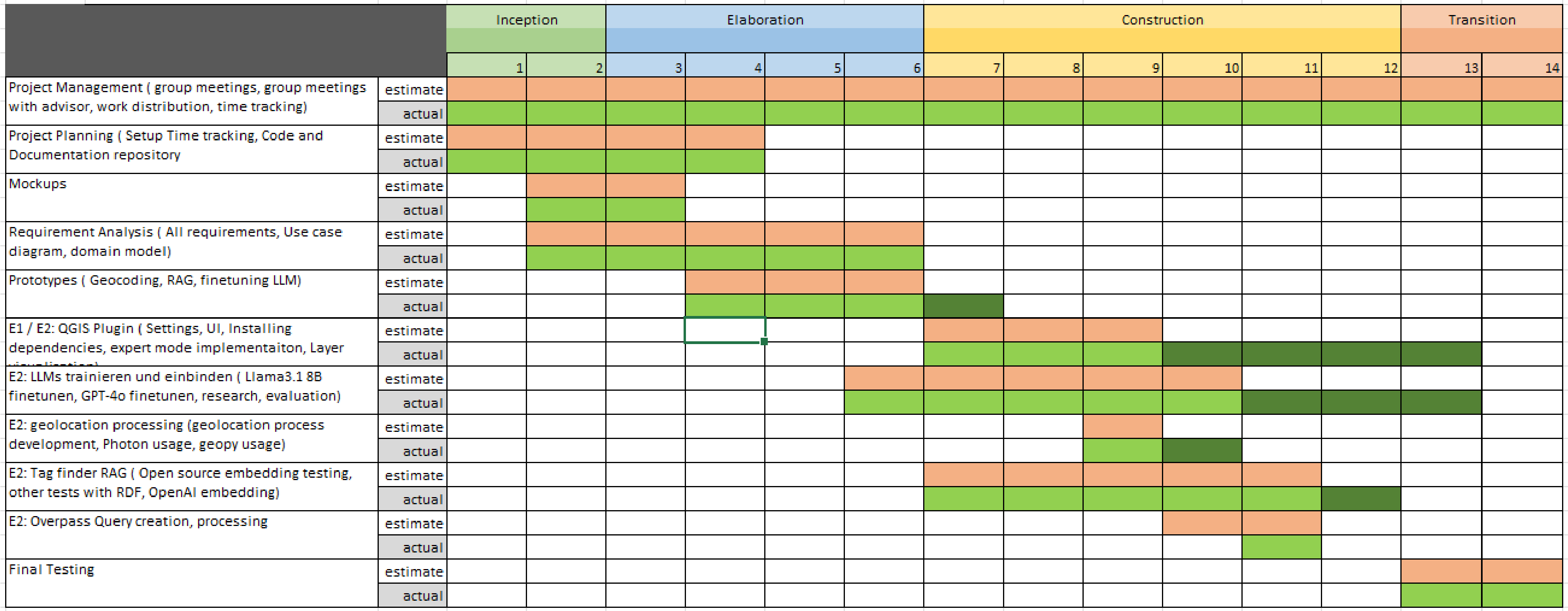
Here we describe the tools we used in this project:

Tools

Reason	**Tools**
Time management	Jira
Version control	git, gitlab
File sharing, communication	Teams
Documentation	LaTeX
Research	Google, Google scholar, ChatGPT, github
Mockups	Figma
Translation	DeepL, ChatGPT
Text improvement, text correction, text generation	ChatGPT, DeepL write
Idea generation	ChatGPT, Perplexity
Code help, code correction	ChatGPT

Table 5.1: Tools

5.2.4 Long Term Project Plan



		Inception		Elaboration				Construction						Transition	
		1	2	3	4	5	6	7	8	9	10	11	12	13	14
Project Management (group meetings, group meetings with advisor, work distribution, time tracking)	estimate	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧	🟧
	actual	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
Project Planning (Setup Time tracking, Code and Documentation repository	estimate	🟧	🟧	🟧	🟧										
	actual	🟩	🟩	🟩	🟩										
Mockups	estimate		🟧	🟧											
	actual		🟩	🟩											
Requirement Analysis (All requirements, Use case diagram, domain model)	estimate		🟧	🟧	🟧	🟧	🟧								
	actual		🟩	🟩	🟩	🟩	🟩								
Prototypes (Geocoding, RAG, finetuning LLM)	estimate				🟧	🟧	🟧								
	actual				🟩	🟩	🟩	■							
E1 / E2: QGIS Plugin (Settings, UI, Installing dependencies, expert mode implementaiton, Layer	estimate							🟧	🟧	🟧					
	actual							🟩	🟩	🟩	■	■	■	■	
E2: LLMs trainieren und einbinden (Llama3.1 8B finetunen, GPT-4o finetunen, research, evaluation)	estimate						🟧	🟧	🟧	🟧	🟧				
	actual						🟩	🟩	🟩	🟩	🟩	■	■	■	
E2: geolocation processing (geolocation process development, Photon usage, geopy usage)	estimate									🟧					
	actual									🟩	■				
E2: Tag finder RAG (Open source embedding testing, other tests with RDF, OpenAI embedding)	estimate							🟧	🟧	🟧	🟧	🟧			
	actual							🟩	🟩	🟩	🟩	🟩	■		
E2: Overpass Query creation, processing	estimate										🟧	🟧			
	actual											🟩			
Final Testing	estimate													🟧	🟧
	actual													🟩	🟩

Figure 5.1: Long Term Project Plan

5.2.5 Kickoff and Deadline

The kickoff of this project was the 12.09.2024. The submission deadline is the 20.12.2024 at 17:00.

5.3 Processes and meetings

To work in an agile project method, we decided to use a combination of Scrum and Kanban. The reason why we use Scrumban (Scrum and Kanban) as our agile project management method, is that scrum alone would be too much for just 2 people, therefore we use a combination of both We lay more focus on Kanban. The Kanban board is managed in Jira. We only use the sprint component of scrum and do sprint planning after each sprint. The addition of RUP is for long-term planning.

5.3.1 Sprint Meetings

Each sprint is 2 weeks long, starting and ending with the meetings held with our advisor. The meetings are held on Thursday every 2 weeks, the meeting time is defined in the previous meeting. Before each regular meeting, our team has sent our advisor an update answering the following questions:

- What has been achieved since the last meeting?
- What challenges are there?
- What is planned for the next sprint?

Attendants

The following people are always present at the regular advisor meetings:

- Advisor: Prof. Stefan F. Keller
- Team member: Aziz Hazeraj
- Team member: Thashvar Uthayakumar

Weekly meetings

Our team had a weekly meeting to talk about the distribution of the tasks, problems and what has been achieved.

5.4 Risk Management

The following risks will be monitored, adjusted, and revised by the team members throughout the project.

5.4.1 R1: Inadequate Project Scope Definition

Explanation

The project's boundaries and objectives are not clearly defined or understood at the beginning. An overly ambitious scope may result in the project being unachievable within the given time frame.

Initial Risk Level

High

Risk Level as of End of Elaboration Phase

Medium
The project's scope has been adequately defined and aligned with the available time frame and resources. There still is the risk of spending too much time in the exploration phase or finding the right LLM.

Impact

An undefined or excessively broad project scope can lead to scope creep, making it impossible to complete the project on time.

Mitigation

Weekly meetings and regular reviews with Prof. Stefan F. Keller ensure continuous alignment of project scope with objectives. These activities have helped address misunderstandings early and refine the requirements.

5.4.2 R2: Lack of Transparency

Explanation

Team members do not provide sufficient updates about the progress of their tasks, which could lead to missed deadlines and project delays.

Initial Risk Level

Medium

Risk Level as of End of Elaboration Phase

Low
Task tracking and weekly review meetings have provided sufficient transparency.

Impact

Without proper transparency, the backlog may not progress as expected, potentially resulting in an incomplete project by the end of the semester.

Mitigation

Tasks are tracked in Jira, and weekly synchronization meetings ensure that the team members are aware of the progress. These measures promote accountability and transparency.

5.4.3 R3: Sickness

Explanation

A team member may fall ill for 1-2 weeks or even longer, which could disrupt the progress of their tasks and overall project timelines.

Initial Risk Level

High

Risk Level as of End of Elaboration Phase

Medium
Tasks are well-documented, and contingency plans are in place to reduce the functional scope if prolonged sickness occurs. However, long-term illness would still introduce a disruptive element to the project.

Impact

Illness can result in a team member being unavailable for one or more sprints, increasing the workload for other team member and delaying project milestones.

Mitigation

The team members are expected to inform one another immediately in case of illness so schedules can be adjusted. If a member remains unavailable for an extended period, the project's functional scope will be reduced to ensure deliverables remain achievable.

5.5 Time Tracking Report

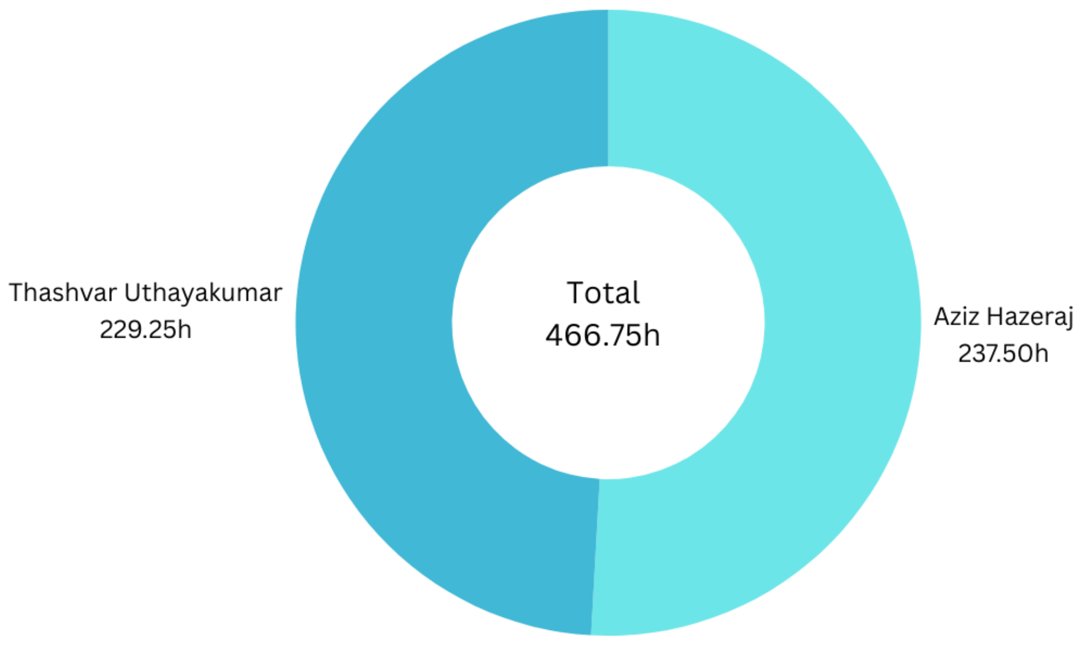


Figure 5.2: Total Time Tracked

Chapter 6

Appendix

6.1 Installation and configuration Manual

6.1.1 Installing the ZIP file

To install the plugin, use the zip-file located inside the code repository. Then inside of QGIS click on ”Plugin” then open the plugin manager. In the plugin manager you can click on the option ”Install from ZIP”. From there you will only have to load the zip to install the plugin.

6.1.2 Tokens

To obtain a OpenAI api key, you can go on the platform of the OST organization - dashboard (right upper corner) - API Keys (on the left side) - click on the button in the upper right corner ”Create new secret key”. From there on you can create a key and use it in the plugin. As for the organization ID, you can find it in your user settings, if you are a user inside of the OST organization. If you are inside of more than one organization, first confirm that you are in the right organizations tab in the upper left corner. Then open the settings page at the upper right corner. Go to the ”General” page and there you will see the organization id in the second field.

6.2 Acceptance protocol

6.2.1 Functional Requirements

ID	Subject	Description	Problem	Acceptance Status
US1.1	Installation Process	Enable users to easily install the plugin via a ZIP file.	-	Approved
US1.2	Dependency Installation	Automatically install all necessary dependencies for the plugin.	-	Approved
US1.3	Settings	Allow users to authenticate by providing OpenAI tokens.	-	Approved
US2.1	Simple Usage	Users can input prompts and receive responses easily.	-	Approved
US2.2	Local Prompt	Recognize locality like ”Rapperswil” in user prompts.	-	Approved
US2.3	Entity Prompt	Identify points of interest like ”fountain” in prompts and extend with similar OSM tags.	The entity is identified, but there is slight possibility that tags, which are not very similar are included.	Not Approved
US2.4	Map View	Restrict results to the current map view in QGIS if the user asks for it.	-	Approved
US2.5	All OSM elements	Show every relevant OSM element in the results.	When Points are part of or on a polygon, it can happen, that a point is not shown.	Not Approved
US3.1	View Overpass Query	Display generated Overpass queries for advanced users.	-	Approved
US3.2	Edit Overpass Query	Allow editing of generated Overpass queries.	-	Approved
US3.3	Background Processes	Provide verbose logging for background processes.	-	Approved

Table 6.1: Acceptance Protocol for Functional Requirements

6.2.2 Non-Functional Requirements

ID	Subject	Description	Test	Problem	Acceptance Status
NFR1.1	Response Time	Generate Overpass queries within 10 seconds.	Tested with several different queries of varying complexity.	Most of the queries take around 10 - 20 seconds.	Not Approved
NFR2.1	LLM Availability	If LLM or APIs, is not available, show an error.	Solved by showing errors, if it is not available.	-	Approved
NFR3.1	Intuitive Usage	Ensure the plugin UI is user-friendly and intuitive.	Tested with a new user.	-	Approved
NFR3.2	User Feedback	Provide visual feedback for all queries and processes.	Solved with a messagebar showing the progress	-	Approved
NFR4.1	Compatibility with QGIS Versions	Ensure compatibility with QGIS 3.34 and newer versions.	Tested for upto QGIS 3.34.3	-	Approved
NFR4.2	Support for Multiple Operating Systems	Operable on Windows, MacOS, and Linux.	Tested and worked for Windows and MacOs	Implemented for Linux but not tested for Linux	Not Approved
NFR5.1	Modular Design	Structure code modularly for easy maintenance.	-	-	Approved

Table 6.2: Acceptance Protocol for Non-Functional Requirements

Glossary

bounding box A bounding box (bbox) is a rectangular area defined by two coordinate pairs (usually the minimum and maximum latitude and longitude) that encloses a specific geographic region for spatial analysis or data extraction. . 19, 31, 32

geocoder A geocoder is a tool or service that converts addresses or place names into geographic coordinates (latitude and longitude) and vice versa, enabling location-based data processing and analysis. . 11

Openstreetmap OpenStreetMap (OSM) is a collaborative, open-source project and database that provides freely accessible geospatial data, created and maintained by a global community of contributors. . 31, 32

photon Photon uses Elasticsearch to provide fast and flexible geocoding and reverse geocoding with OpenStreetMap data. . 33

Plugin An extension of an existing software. In our case, we are talking about an extension of QGIS. . 39

Polygon In the context of this thesis, polygons describe the boundary of an area. . 31, 32

QGIS QGIS (Quantum geographic information system) is an open source program to view, edit, analyze and print geospatial data in various formats. . 2, 3, 31

RAG A machine learning framework that combines information retrieval with generative models to improve the quality of outputs. RAG retrieves relevant context or data from an external knowledge base or database and uses it as input to a language model to generate accurate and context-aware responses. . 11

Bibliography

aino. (2024). *Aino*. Retrieved December 15, 2024, from https://aino.world/

Ashworth, B. (2024). *Osm sql terminal*. Retrieved December 17, 2024, from https://plugins.qgis.org/plugins/kue-ai/

Dodds, L. (2024). *Osm query syntax reference*. Retrieved November 18, 2024, from https://osm-queries.ldodds.com/syntax-reference.html

Gautam, R. (2024). *Osm-gpt: Query openstreetmap with gpt*. Retrieved October 1, 2024, from https://osm-gpt.rohitgautam.com.np/

Gwerder, S. (2024). *Tagfinder*. Retrieved October 1, 2024, from https://tagfinder.osm.ch/apidoc

IFS, O. (2024). *Aiamas - ai-assisted map styler*. Retrieved October 1, 2024, from https://gitlab.com/geometalab/aiamas/aiamas-qgis-plugin/

Learn, Z. (2024). Graphrag explained: Enhancing retrieval-augmented generation (rag) with knowledge graphs. Retrieved October 1, 2024, from https://medium.com/@zilliz_learn/graphrag-explained-enhancing-rag-with-knowledge-graphs-3312065f99e1

naogify. (2024). *Ai-geocoder*. Retrieved October 1, 2024, from https://github.com/naogify/ai-geocorder

Olbrecht, R. (n.d.). Overpass user manual - bounding boxes. Retrieved October 1, 2024, from https://dev.overpass-api.de/overpass-doc/en/full_data/bbox.html

OpenAI. (2024). *Openai documentation*. Retrieved October 13, 2024, from https://platform.openai.com/docs/overview

Staniek, M., Schumann, R., Züfle, M., & Riezler, S. (2023). Text-to-overpassql: A natural language interface for complex geodata querying of openstreetmap.

Stefan, K. (n.d.). *Osm sql terminal*. Retrieved October 1, 2024, from https://terminal.osmdatapipeline.geoh.infs.ch/

Chapter 7

Listings

List of Tables

List of Figures