1. Introduction

The introductory chapter will explain the motivation behind the thesis, as well as its goals and the research questions it will attempt to answer. Section 1.4 will list the main contributions of the thesis, and section 1.5 will give a high-level overview over the thesis.

1.1. Background and Motivation

The release of OpenAI's ChatGPT in November, 2022 (OpenAI, 2022) generated a hype within the general population and chat-based systems are flourishing. ChatGPT — or rather the underlying GPT-3 and GPT-4 models — is an example of how modern Large Language Models (LLMs) can provide a natural language interface between human and machine. Furthermore, significant advancements have been made within code generation, which essentially allows the LLM to execute computational tasks that can be defined within a snippet of code. Paired with the LLM's ability to often correctly interpret the user's intent regardless of the preciseness of their problem formulation, even individuals with no prior programming experience can carry out computational tasks that require the execution of code.

GIS analysis has traditionally been reserved for GIS *experts*. GIS professional are commonly required to know their way around one or more Geographic Information Systems, in addition to being proficient in programming languages suitable to data science task, such as Python or R. Extensive domain knowledge is often also necessary when tackling GIS tasks, like knowing which data to use for a particular tasks and where to get them. All of these points — and more — are barriers to entry for people that wish to make use of powerful GIS tools for their particular purposes, but lack the technical know-how required to use them correctly. This challenge serves as the overall motivation behind this master's thesis, which will mitigate these issues by utilizing the vast background knowledge and code generation abilities of modern LLMs.

1.2. Goals and Research Questions

Deriving from the motivation described in the section above, the overarching goal of this master's thesis becomes to investigate the possibilities of utilizing LLMs to have a natural language interface with a system that is capable of solving GIS-related tasks. The hypothesis is that modern LLMs are embedded with an understanding of common GIS workflows, and that their code generation abilities are of such a level that they can accomplish tasks in an autonomous manner.

Based on this overarching goal, three research questions have been constructed and are listed below:

1. Can an LLM-based system perform common GIS tasks?
2. What are core challenges when developing larger systems that rely on Large Language Models to control its logic flow?
3. Can an autonomous LLM-based GIS agent replace GIS professionals?

1.3. Research Method

Prior to this master's thesis, a specialization project on the same topic was conducted. The specialization project — as detailed in (Holm, 2023) — was of a theoretical character predominantly served as a literature study leading up to the master's thesis. The research questions listed in the above section, however, call for a different and more practical approach. Therefore, this master's thesis will revolve around a "proof of concept" and evaluating the usefulness of the system, as well as lessons learned from the development process.

1.4. Contributions

Below is a brief description of the contributions of this master's thesis:

1. A chat-based GIS named *GeoGPT* — powered by LLMs — that can solve common geospatial tasks.
2. Experimental results and examples that shows how GeoGPT can solve a range of common GIS tasks when provided only with a natural language prompt from the user.
3. Experimental results that highlight the importance of the accuracy of the initial prompt from the user, indicating that GIS experience will not become redundant in the foreseeable future.

1.5. Thesis Structure

The following bullet point list will provide an overview of the thesis structure:

- Chapter 2 introduces the theory and tools necessary for the reader in order to understand the rest of the work.
- Chapter 3 gives insight into the work that has been done in regard to autonomous systems powered by LLMs, in particular those within the field of geomatics.

- Chapter 4 presents the data provided for GeoGPT in the experiments.
- Chapter 5 will go into detail on the architecture of GeoGPT, providing both a high-level overview and details on important parts of the system.
- Chapter 6 is divided into one section describing how the experiments were conducted and a second section presenting and evaluating the results from the experiments.
- Chapter 7 will elaborate upon points of discussion that arise from the experimental results.
- Chapter 8 will conclude this master's thesis, describing the main contributions of the work and suggesting areas of improvement suitable for future research.

2. Background Theory

NB! Parts of the Background Theory chapter is reused material from the specialization project (Holm, 2023) preceding this master thesis. Below are the sections in question, together with a description of the extent to which, and how, the material is reused:

- *Subsection 2.1.2: Reused with some adjustments.*
- *Subsection 2.1.3: GPT part reused without modification.*

Chapter 2 will lay a theoretical basis for the work done in this master thesis, providing the user with the required understanding in order to understand the contributions of the work. Section 2.1 will explain the theoretical basis of the component which most modern Large Language Models (LLMs) are based upon — namely the Transformer — and the attention mechanism within it. Continuing, this section will present some of the leading LLMs as of 18th May 2024— both proprietary and open-source ones. Subsection 2.1.4 will present the concept of *prompt engineering* before subsection 2.1.5 concludes section 2.1. Section 2.2 will explain LangChain — a Python/JavaScript library that is used extensively throughout the code base of GeoGPT. Section 2.3 will conclude the Background Theory chapter, presenting the geospatial technologies that is used within GeoGPT.

2.1. Large Language Models

LLMs are a type of neural networks that excel at processing language. They can be developed for different Natural Language Processing (NLP) tasks, such a text classification, masked language modelling, and text generation. While they all have their use cases, only text generation will be relevant for this thesis. Generative LLMs are designed to take some input sequence and generate some output sequence, and are the types of models behind technologies like OpenAI's *ChatGPT*(OpenAI, 2022).

This section will lay the theoretical groundwork required to gain an overview of the inner workings of LLMs. subsection 2.1.1 will inform the reader on core concepts and terminology that will be used extensively throughout this thesis when discussing LLMs. subsection 2.1.2 will explain the attention mechanism and the related Transformer architecture, the latter of which serves as the core building block in most modern LLMs.

2.1.1. Tokens and the Context Window

While humans understand sentences as sequences of words, LLMs perceive them as sequences of **tokens**. An LLM possesses a fixed set of unique tokens in its vocabulary, from which it constructs words and sentences. Tokens can be entire words, short character sequences, or single characters. Figure 2.1 illustrates how the latest GPT models *tokenize* an English sentence. Notice how some words are deconstructed into more than one token. For example, the word "revolutionizing" is split into its root ("revolution") and its suffix ("izing"). Figure 2.2 shows how this process applies to other suffixes as well. Likely, the model has learned the meaning of "revolution" and elects to use different suffixes to modify its function within a sentence, as opposed to having an entirely new token for each version of the word.

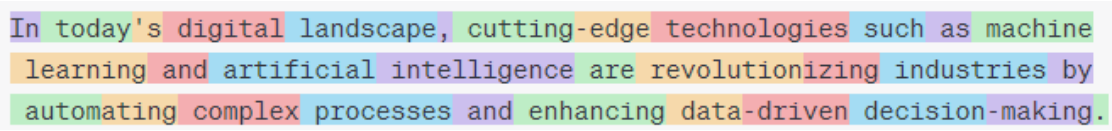


Figure 2.1.: Tokenization example for a sentence

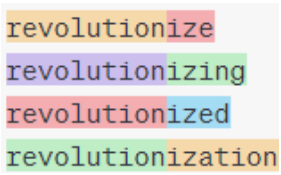


Figure 2.2.: Tokenization of "revolution" with different suffixes

The **context window** of an LLM is the range of tokens that an LLM is able to process. When an LLM generates text, it does so by generating a new token based on what it sees in the span of the context window. A larger context window will allow an LLM to take more and longer documents as context and answer questions generate answers based on these. Leveraging sophisticated techniques like Rotary Position Embedding (RoPE) (Su et al., 2024) and Positional Skip-wisE (PoSE) (Zhu et al., 2024), reserchers have able to efficiently extend the context window of open-source models like Llama.

2.1.2. Attention and the Transformer Architecture

Vaswani et al. (2017) managed to achieve new state-of-the-art results for machine translation tasks with their introduction of the Transformer architecture. The Transformer has later been proved effective for numerous downstream tasks, and for a variety of modalities. Titling their paper *Attention Is All You Need*, Vaswani et al. suggest that their attention-based architecture renders network architectures like Recurrent Neural Networks (RNNs) redundant, due to its superior parallelization abilities and the shorter

path between combinations of position input and output sequences, making it easier to learn long-range dependencies (Vaswani et al., 2017, p. 6).

The Transformer employs self-attention, which enables the model to draw connections between arbitrary parts of a given sequence, bypassing the long-range dependency issue commonly found with RNNs. An attention function maps a query and a set of key-value pairs to an output, calculating the compatibility between a query and a corresponding key (Vaswani et al., 2017, p. 3). Looking at Vaswani et al.'s proposed attention function (2.1), we observe that it takes the dot product between the query Q and the keys K, where Q is the token that we want to compare all the keys to. Keys similar to Q will get a higher score, i.e., be *more attended to*. These differences in attention are further emphasized by applying the softmax function. The final matrix multiplication with the values V (the initial embeddings of the input tokens) will yield a new embedding in which all individual tokens have some context from all other tokens. We improve the attention mechanism by multiplying queries, keys, and values with weight matrices that are learned through backpropagation. Self-attention is a special kind of attention in which queries, keys, and values are all the same sequence.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.1}$$

Attention blocks can be found in three places in the Transformer architecture (Vaswani et al., 2017, p. 5) (I will use machine translation from Norwegian to German as an example):

1. In the encoder block to perform self-attention on the input sequence (which is in Norwegian)
2. In the decoder block to perform self-attention on the output sequence (which is in German)
3. In the decoder block to perform cross-attention (also known as encoder-decoder attention) where each position in the decoder attends to all positions in the encoder

The Transformer represented a breakthrough in the field of NLP, and is the fundamental building block of modern LLMs, most famous of which are the GPT's.

2.1.3. State-of-the-Art Decoder-Only Large Language Models

While the work of Vaswani et al. is still considered perhaps the greatest breakthrough in NLP, most moderns LLM do not apply this encoder-decoder architecture. The subsequent evolution of LLMs has favoured generative decoder-only models, focusing entirely on the generative component of the Transformer, hoping to create models that can produce coherent and context-aware text. The first decoder-only model was OpenAI's GPT-1.

The GPT Family

Generative Pre-trained Transformer (GPT) is a type of LLM that was introduced by OpenAI in 2018 (Radford et al., 2018). Specifically designed for text generation, a GPT is essentially a stack of Transformer *decoders*. It demonstrates through its vast pre-training on unlabelled data that such unsupervised training can help a language model learn good representations, providing a significant performance boost while alleviating the dependence on supervised learning. While the original Transformer architecture as described by Vaswani et al. (2017) was intended for machine translation — thus having encoders to learn the representation of the origin language representation of a given input sequence and decoders to learn the representation in the target language and perform cross-attention between the two — the GPT is designed only to *imitate* language. This is why there are no encoders to be found in the GPT architecture, only decoders. The model employs masked multi-head attention (running the input sequence through multiple attention heads in parallel), and is restricted to only see the last k tokens — with k being the size of the context window — and tasked to predict the next one.

Training consists of two stages: unsupervised pre-training and supervised fine-tuning. The former is used to find a good initialization point, essentially teaching the model to imitate the corpora upon which it is trained. This results in a model that will ramble on uncontrollably, just trying to elaborate upon the input sequence it's given to the best of its knowledge. This will naturally produce undefined behaviour, and it is therefore necessary to fine-tune the model on target tasks in a supervised manner. Radford et al. (2018, p. 4) explain how the model can be fine-tuned directly on tasks like text classification, but how one for other tasks needs to convert structured inputs into ordered sequences because the pre-trained model was trained on contiguous sequences of text. In the case of ChatGPT, OpenAI used Reinforcement Learning from Human Feedback (RLHF) by employing a three-step strategy: first training using a supervised policy, then using trained reward models to rank alternative completions produced by ChatGPT models, before fine-tuning the model using Proximal Policy Optimization (PPO), which is a way of training AI policies. This pipeline is then performed for several iterations until the model produces the desired behaviour (OpenAI, 2022).

OpenAI's API currently features three flagship models: the GPT-3.5 Turbo model, which is their fast and inexpensive; the GPT-4 Turbo model, which as of 18th May 2024is described by OpenAI as their "previous high-intelligence model"; and lastly, GPT-4o, their "fastest and most affordable flagship model".[1] The latter model has unique multi-modal abilities and can reason across audio, vision, and text in real time, according to OpenAI.[2]

The Gemini Family

The suite of models known as Gemini is Google's latest response OpenAI's GPT models. The Gemini 1.0 suite (Gemini Team et al., 2024a), which is the first suite of Gemini models,

[1] https://openai.com/api/
[2] https://openai.com/index/hello-gpt-4o/

includes three different models: Ultra, Pro, and Nano. These are listed in descending order in terms of size (number of parameters). Like most other commercial LLMs the models of the Gemini 1.0 suite are multimodal, supporting text, image, audio, and video. Gemini 1.0 displayed new state-of-the-art performance on most major benchmarks, but performed significantly worse on the HellaSwag benchmark (which measures a model's common-sense understanding) compared to the newest GPT-4 model at the time. The models scored 87.8% and 95.3%, respectively. Supporting a context length up to 1M tokens, Gemini 1.0 Ultra surpassed the Claude 2.1's context window of 200k with a wide margin, and with the release of Gemini 1.5 Pro came also the possibility of utilizing a context window of up to 10M tokens, though in production this number is currently 1M (Gemini Team et al., 2024b; Pichai and Hassabis, 2024). Furthermore, Gemini 1.5 Pro outperforms Gemini 1.0 Ultra in some capabilities despite using significantly less training compute (Gemini Team et al., 2024b, p. 31).

The Claude Family

Developed at Anthropic, Claude is the third major, commercial LLM. Anthropic is one of the actors in the LLM market that has helped push in the direction of long-context LLMs, which their Claude 2 model being able to support up to 200k tokens (Anthropic, 2023, p. 9), being the best at the time of its release in November 2023. Latest in line is Claude 3, a family of LLMs of different sizes, largest of which is the Opus model which outperforms GPT-4 and Gemini 1.0 Ultra on most benchmarks (Anthropic, 2024, p. 6). Claude 3 is a family of language models, much like the Gemini family, featuring three main models: Opus, Sonnet, and Haiku. Again, these are listed in descending order in terms of number of parameters. Users can choose between Opus for the most advanced capabilities, Haiku for a fast and economical option, or Sonnet for a balance of both.

Open-Source Alternatives

OpenAI's GPT models, Google's Gemini models, and Anthropic's Claude models are all commercial and closed-source. This prevents developers from downloading these models and making custom improvements to them through fine-tuning. For these reasons, a number of open-source LLMs have joined the scene.

The **Llama** family of LLMs from Meta AI is perhaps the most famous open-source option to the commercial, closed-source LLMs. At the time of writing, the last in line is the Meta Llama 3 model (Meta AI, 2024) which comes in two size: 8B and 70B parameters. Both models display state-of-the-art performance on most major benchmarks compared to comparable open-source alternatives, and 70B model even surpasses closed-source models like Gemini 1.5 Pro and Claude 3 Sonnet on certain benchmarks.

Mistral AI is one of the most prominent actors in the world of open-source LLMs. Their debut model, Mistral 7B, outperformed Llama 13B (which was the best open-source LLM at the time) across all the benchmarks they evaluated (Jiang et al., 2023). Mistral AI has also gained fame for their Sparse Mixture-of-Experts (SMoE) architecture, which was introduced with the Mixtral 8x7B model (Jiang et al., 2024). It shares architecture

with Mistral 7B, but is each layer of the model is composed of 8 feed-forward blocks. Using a router at each layer, it is able to use only 13B out of a total of 47B parameters during inference, keeping cost and latency low.

Along with their Gemini models, Google released a family of open-source models called **Gemma**, which are based on the same research conducted for their Gemini models (Gemma Team et al., 2024). Gemma comes in two sizes: 2B and 7B parameters. At its release, the Gemma 7B it surpassed Llama 2 13B and Mistral 7B LLMs in 11 out of 18 benchmarks. Note, however, that Llama 3 8B has improved upon its predecessor and now performs better than Gemma 7B overall.

2.1.4. Prompt Engineering

Prompt engineering refers to the process of constructing a query that will be used as input to an LLM. In a chat-based context like that of ChatGPT, the prompt consists of a series of messages that are defined as an array of JSON objects. These messages can hold one of three different roles[3], each listed below:

- **system**: Generally used to set the behaviour of the LLM assistant, giving the assistant a specific personality or information on how to answer the user.
- **user**: A message from the user for the assistant to respond to.
- **assistant**: A message from the AI/LLM.

White et al. (2023) stress the importance of prompt engineering to efficiently converse with LLMs. They provide a catalogue of prompt patterns that aim to help enforce certain qualities of the output generated by the LLM. These patterns are organized into six distinct categories (White et al., 2023, p. 4):

- **Input Semantics**: Clarifying what information is fed into the LLM and how this input should be used to generate responses.
- **Output Customization**: Strategies to guide how the LLM should format and structure its responses.
- **Error Identification**: Methods to identifying and resolving errors in the outputs produced by the LLM.
- **Prompt Improvement**: Techniques to improve the quality of both the input provided to the LLM and the output it generates.
- **Interaction**: Strategies for enhancing interactions between the user and the LLM.
- **Context Control**: Controlling the contextual domain within which the LLM operates.

[3] https://platform.openai.com/docs/guides/text-generation/chat-completions-api

Patterns that turned out useful to GeoGPT include the *Template* pattern (Output Customization), the *Reflection* pattern (Error Identification), and the *Infinite Generation* pattern (Interaction Strategy). The *Template* pattern allows the user or the system to define a template for the LLM to fill out. This is closely related to function calling, which is discussed in subsection 2.1.5. Also related to function calling is the *Reflection* pattern, which allows the LLM to inspect its own output in order to identify and correct errors. The *Infinite Generation* pattern lets the LLM generate output indefinitely without requiring the user to re-enter the conversation after each generated message. This pattern is important when developing agentic behaviours in the LLM.

2.1.5. Function Calling LLMs

Function calling — also known as *tool calling* — was first introduced by OpenAI (Eleti et al., 2023) in April 2023. Function calling allows developers to provide function definitions to an LLM and have said LLM output a JSON object containing the name of one or more of the functions provided, as well as suitable arguments to these. Code Snippet 2.1 shows a description of a function that takes a path to a GeoJSON file and a layer name and adds the GeoJSON data at that location to some client-side map with the layer name specified. Function calling is possible through *fine-tuning*, and these function calling models are able to detect when a certain function should be called based on the name, description, and parameters specified in such JSON objects. Function calling makes it possible to give an LLM *hooks* into the real world, and provides a more reliable way for developers to integrate LLMs into applications.

```
{
  "type": "function",
  "function": {
    "name": "add_geojson_to_map",
    "description": "Use this to add arbitrary geojson data to a client-side map visible to the user.\n",
    "parameters": {
      "type": "object",
      "properties": {
        "geojson_path": {
          "description": "Path to the .geojson file that will be added to the map",
          "type": "string"
        },
        "layer_name": {
          "description": "Name of the layer (should be snake_case)",
          "type": "string"
        }
      },
      "required": ["layer_name"]
    }
  }
```

```
21 }
```

Code Snippet 2.1: Example of a tool definition

Possible use cases include using functions provide correct and up-to-date information that would otherwise require extensive training and fine-tuning. Having the LLM use function calling for information retrieval also make them more transparent, making it possible to trace a claim back to its source, something that is normally a difficult feat with LLM. Another use case is code execution. One could imagine a rather simple function `execute_python_code(code: string) -> string` that takes Python code as a string and returns the standard output that results from executing that code. This is likely the principle behind products like OpenAI's Data Analysis mode (previously Code Interpreter), in which ChatGPT functions as a code executing agent that can generate, execute, and self-correct its own code. Similar functions could be constructed for SQL, making it possible for LLMs to work against relational databases. As Eleti et al. (2023) describes, function calling can also be used to extract structured data from text.

An initiative among researchers at Berkeley (Yan, Fanjia et al., n.d.) lead to the creation of a benchmark that aims to evaluate the LLM's ability to call functions and tools. The benchmark, named Berkeley Function-Calling Leaderboard (BFCL), includes four different test scenarios: *single function accuracy*, where the LLM is provided with a single function definition; *multiple function accuracy*, where 2 to 4 functions are passed and the model must select the appropriate function; *parallel function*, where the model needs to determine how many functions should be called; and *parallel multiple function* a combination of parallel function and multiple function. Some models support different levels of function calling natively while others have to be carefully prompted in to help accommodate function calling abilities. At the time of writing (18th May 2024), a prompted version of GPT-4-0125-Preview tops the leaderboard[4].

2.2. LangChain

LangChain (LangChain AI, 2022) is an open-source project that provides tooling which simplifies the way developers interface with LLMs. This tooling includes composable tools and integrations that can be used to build prompts for LLMs, as well as off-the-shelf chains that perform higher level tasks. Chains are Directed Acyclic Graphs (DAGs) — or sequences of runnables — that take an input and produces and output. A runnable can be a prompt template with template literals that are substituted with values that are passed into the runnable. The output is the template with the template literals filled in. This output can then be chained into an LLM runnable calls a language model using the prompt template. The output from the LLM runnable could then be passed into an output parser, e.g. a JSON parser, that ensures that the chain outputs a JSON object. Such chains are the buildings blocks that make up LangChain.

[4] https://gorilla.cs.berkeley.edu/leaderboard.html

Common use cases for LangChain are:

- Building chatbots for question answering that use semantic retrieval from document store
- Creating agents with access to external tools be leveraging function calling (see subsection 2.1.5)
- Creating code executing agents for Python, SQL, or other programming languages

In January 2024, LangChain AI rolled out a new framework called LangGraph which builds on top of the LangChain ecosystem. While the chains commonly found in LangChain are good for DAG workflow, they are not suited to creating cyclic graphs. LangGraph can be used to add cycles to LLM applications, which are important for agent-like behaviours (LangChain AI, 2024). A graph in LangGraph is a set of nodes that pass some state around, state that can be modified by each node. The nodes are connected together by edges that define what node can succeed another node. These edges can also be conditional, which routes execution to a given node based on the output from a function giving the current state. This allows for complex logic and simplifies implementation of advanced agent patterns, some of which are discussed in section 3.2.

2.3. Geospatial Databases and Data Catalogues

This section will discuss the geospatial technologies that were used or considered for use in this master's thesis.

2.3.1. PostGIS

PostGIS (*PostGIS* 2001) is an open-source extension for the PostgreSQL DBMS. By adding the PostGIS extension one adds support for storing, indexing, and querying geospatial data. Data can be stored in both two and three dimensions, and they can have types like points, lines, polygons. These types can be stored along with a spatial index which can significantly reduce search time for these geometries. GiST (Generalized Search Tree)[5] is commonly used in PostgreSQL/PostGIS to take advantage of various tree-based search algorithms that are developed to retrieve spatial features quickly.

PostGIS also comes with a plethora of spatial database functions that to analyse and process geospatial data. These are prefixed with `ST_`, and some examples are `ST_DWITHIN`, `ST_BUFFER`, and `ST_TRANSFORM`. Code Snippet 2.2 displays a typical query for retrieving building outlines within a bounding box specified in WGS 84 latitudes and longitudes.

```
1 SELECT *
2 FROM osm_buildings_polygons
3 WHERE type = 'house'
```

[5] https://en.wikipedia.org/wiki/GiST

```
4    AND ST_Intersects(geom, ST_MakeEnvelope(min_lon, min_lat, max_lon,
       max_lat, 4326));
```

Code Snippet 2.2: PostGIS example code for retrieving building outlines within a specified bounding box

2.3.2. OGC API Features

OGC API Features is an API specification that defines modular API building blocks for interacting with features, real-world objects (Open Geospatial Consortium, 2022). These building blocks include blocks for creating, modifying, and querying features on the Web. A typical implementation of OGC API Features implements these building blocks for HTML, GeoJSON, and GML. These are called *requirement classes*, though none of them are strictly required. The HTML requirement class gives the user of the API a visualization of the features, whereas the GeoJSON and GML requirements classes are typically meant for use in other applications.

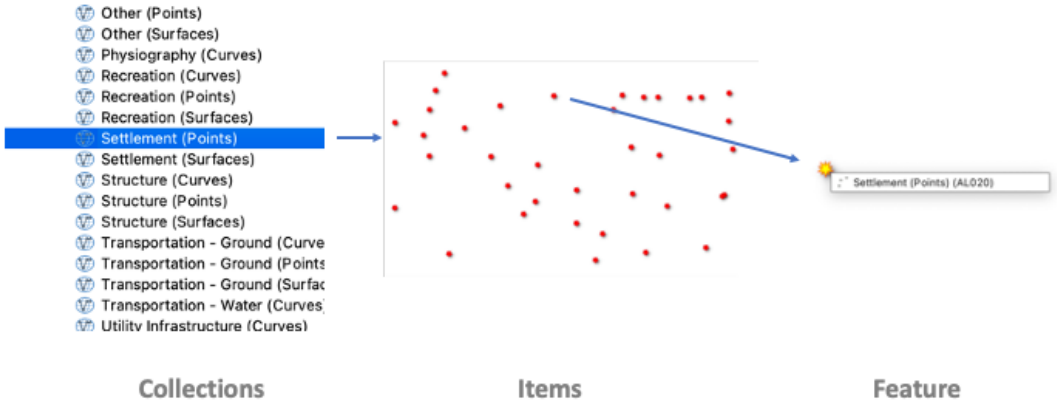


Figure 2.3.: Collections, items, and features in OGC API Features specification. Retrieved from https://features.developer.ogc.org/ on April 29, 2024.

The development of the Features standard is divided into several parts that are meant to build on top of one another. Here are the four parts that are listed on OGC's site about the standard [6]:

- Features - Part 1: Core[7]
- Features - Part 2: Coordinate Reference Systems by Reference [8]

[6] https://ogcapi.ogc.org/features/
[7] https://docs.opengeospatial.org/is/17-069r4/17-069r4.html
[8] https://docs.ogc.org/is/18-058r1/18-058r1.html

- Features - Part 3: Filtering[9]
- Features - Part 4: Create, Replace, Update and Delete[10]
- Features - Part 5: Schemas[11]

Part 1 specifies core capabilities that are described in the first paragraph of this section, while parts 2-4 spcify additional capabilities. Part 2 allows retrieval of features in Coordinate Reference Systems (CRSs) different to the default WGS 84 reference system. Part 3 enables filtering of features using Common Query Language (CQL). CQL is a language similar to SQL. This allows for filtering of collections, so that users of the API can retrieve only a subset of a given collection. Below are two examples of CQL queries:

```
1 \\ Example 1
2 county in ('Akershus', 'Buskerud', 'Ostfold')
3
4 \\ Example 2
5 DWITHIN(the\_geom, Point(63.4265, 10.3960), 1, kilometers)
```

Code Snippet 2.3: CQL examples

Part 4 defines how an API following the specification handle addition, replacement, modification, and/or removal for a collection. Part 5 describes how features can be described by a logical schema and how these are published. Furthermore, several proposed extensions, such as the Search extension which could allow for multi-collection queries, or the Geometry Simplification extension which proposes the use of simplification algorithms for retrieving simplified versions of a collection, have been created.[12]

2.3.3. SpatioTemporal Asset Catalog

The SpatioTemporal Asset Catalog (STAC) specification[13] is closely related to OGC API Features, and as Holmes — former board member of Open Geospatial Consortium — stated in a blog post that "STAC API implements and extends the OGC API — Features standard, and our shared goal is for STAC API to become a full OGC standard" (Holmes, 2021). The main difference to the OGC API Features specification is its requirement that all items/features should have a temporal component, thus making it *spatiotemporal*. The vision for the STAC specification is to minimize the need for developing new code with the release of each new dataset or API, by using a standard that does not change.

[9] https://docs.ogc.org/DRAFTS/19-079r1.html
[10] https://docs.ogc.org/DRAFTS/20-002.html
[11] https://docs.ogc.org/DRAFTS/23-058r1.html
[12] https://github.com/opengeospatial/ogcapi-features/tree/master/proposals
[13] https://stacspec.org/en

3. Related Work

NB! Parts of the Related Work chapter is reused material from the specialization project (Holm, 2023) preceding this master thesis. Below are the sections in question, together with a description of the extent to which, and how, the material is reused:

- *Subsection 3.1: Reused without modification.*

This chapter provides an overview of previous research that shares objectives similar to those of this thesis. The related work is divided into two main parts: section 3.1, which presents research investigating potential use cases of LLMs in the space of geospatial information technology; and section 3.2, which presents examples of three different types of patterns commonly used in LLM-based agents.

3.1. Using LLMs for Geospatial Purposes

Roberts et al. (2023) investigated the extent of GPT-4's geospatial awareness through a set of case studies with increasing difficulty, starting with general factual tasks and finishing with complex questions such as generating country outlines and travel networks. The authors found that GPT-4 is "skillful at solving a variety of application-centric tasks", almost having the ability to "see", despite being a language model and therefore only able to interface with the world through sequenced, textual input. Examples include its ability to serve as a travel assistant in providing itinerary suggestions for a trip when provided with requirements, and its ability to provide generally correct start and end locations of bird migration paths. While it quickly became obvious that a lot of geospatial context have been embedded within the model during the vast pre-training, the question of whether this is memorization or reasoning is a central one. The authors suggest that the variability of tasks in their experiments deems it unlikely that it is all memorization, but they say that some things appear to be memorized.

Mooney et al. (2023) examined the performance of ChatGPT in a Geographic Information System (GIS) exam, aiming to assess its ability to grasp various geospatial concepts, highlighting its capabilities and limitations. Experiments were conducted on GPT-3.5 and GPT-4, which delivered performances equivalent to grades of D and B+, respectively. Additional experiments were conducted for more specialized areas of GIS, including True/False questions about spatial analysis, and simple tasks in applied GIS workflows. Experiments on the latter showed that ChatGPT-4 was able to correctly answer a relatively complex GIS task involving seven different datasets, requiring seven steps in order

to obtain a perfect score. Generally, ChatGPT-4 outperformed ChatGPT-3.5 in all tasks. While clearly powerful, the authors highlight a range of limitations, among which the multimodal nature of GIS, which would hinder a straightforward application of existing models.

Li and Ning (2023) state that "autonomous GIS will need to achieve five autonomous goals: self-generating, self-organizing, self-verifying, self-executing, and self-growing.". They provide a "divide-and-conquer"-based method to address some of these goals. Furthermore, they propose a simple trial-and-error approach to address the self-verifying goal. They also highlight the need for a memory system in a mature LLM-based GIS system, referring to the use of vector databases in autonomous agents made with AutoGPT (Richard, 2023). Even with its shortages, the solution that Li and Ning (2023) provide—called LLM-Geo—is able to produce good solutions in various case studies by providing executable assemblies in a Python environment when provided with URLs to relevant data sets, along with a user-specified query.

Zhang et al. (2023) use the LangChain framework in order to combine different GIS tools in a sequence to solve various sub-goals, focusing on using the semantic understanding and reasoning abilities of LLMs to call externally defined tools, employing the LLM as an agent or controller. The authors take great inspiration from the AutoGPT framework (Richard, 2023). The externally defined tools are described (manually) by their names and descriptions. These descriptions contain information about the input parameters and output types of the tools/functions. Tools are defined for geospatial data collection, data processing and analysis, and data visualization. The effectiveness of the system is showcased in four case studies.

3.2. Agent Patterns

LLM-based agents can be implemented in many ways, and researchers have developed a plethora of *agent patterns* that seek to improve upon areas where LLMs tend to be less effective. This includes patterns for retrieval of external data, as well as multi-agent patterns. Such patterns can improve the performance of LLM-based agents within any domain, and should also be considered when developing one for geospatial purposes. The following sections will shortly explain some of the patterns that were considered in the development of GeoGPT.

The **multi-agent** pattern that takes inspiration from human collaboration in that it is made up from multiple specialized agents that work together to achieve some objective. There have been several implementations of the pattern, with certain differences. MetaGPT (Hong et al., 2023) is a LLM-based multi-agent system consisting of agents with human-level domain expertise. Using an assembly line paradigm, where the overall goal is divided into subtasks, Hong et al. showed that MetaGPT could generate more coherent solutions compared to the previous state-of-the-art multi-agent systems. At the time of release, MetaGPT set a new state-of-the-art performance on the HumanEval and MBPP benchmarks (Hong et al., 2023, p. 7), demonstrating the potential of the multi-agent pattern.

Patterns that employ **self-reflection** are commonly used with autonomous LLM-based agents. *Reflextion* (Shinn et al., 2023) is a pattern/framework that reinforces agents through linguistic feedback, essentially allow the agent to reflect upon the consequence of its decisions. The framework utilizes three distinct models: an *Actor* model responsible for generating text and actions; an *Evaluator* model which assess the quality of the outputs from the Actor; and a *Self-Reflection* model that generates reinforcement cues for the Actor based on the output and quality assessment of the other two models. Together, these three models form a loop that will run until the Evaluator deems the output from the Actor as correct.

Step-by-step reasoning is another pattern that has proved to be efficient in helping LLMs produce correct responses. Wei et al. (2023) demonstrated that so-called *chain-of-thought prompting* can be used for enhancing reasoning in LLMs by providing it with examples of how to reason for *similar* to that which it is trying to solve. By helping the LLM with decomposing multi-step problems into intermediate steps in this way, Wei et al. managed to achieve state-of-the-art accuracy on the GSM8K benchmark of math word problems.

4. Datasets

With the interest of investigating the ability of a Large Language Model (LLM)-based system to perform geospatial analysis, relevant datasets should be accessible to said system. Section 4.1 provides a description of the datasets used in the experiments. Furthermore, it was decided to explore different access channels to this data. Section 4.2 elaborates on this.

4.1. Data Sources

A total of eighteen datasets were used in the experiments. The data was downloaded from Geofabrik's website[1]. Geofabrik — German for "geo factory" — is a company that "extract, select, and process free geodata". They have gathered data from OpenStreetMap and published them as a collection of shapefiles, dividing them into categories such as "places of worship", "points of interest", and "traffic". Data can be downloaded for different regions of the world, and for experiments conducted in this thesis, data for Norway was used. Table A.1 in Appendix A lists all datasets used, along with a short description of their contents. Common for all datasets are their *fclass* attribute, which is short for *feature class*. Some datasets have additional attributes, such as the *maxspeed* attribute in the road data and the *type* attribute in the building data. Figure 4.1 shows a plot containing four selected datasets, constrained to a bounding box of Trondheim. The plot was created by GeoGPT.

4.2. Data Access

While leading LLMs are trained on increasingly large corpora, they are still only as familiar with a topic as the extent to which the training data exposes it to said topic. For instance, many LLMs are trained specifically to generate Python code, and are therefore fed with a vast number of Python code examples during training in the hopes of improving its performance on benchmarks like the Mostly Basic Python Programming (MBPP) benchmark (Austin et al., 2021). As it is unlikely that the training data is evenly distributed among many different topics, it is useful to get familiarized with a model's capabilities in the areas of interest for a particular use case. In the case of an LLM-powered GIS agent that should be capable of performing geospatial analyses, it is useful to know what data formats such an agent is most comfortable to understand and

[1] https://download.geofabrik.de/europe/norway.html

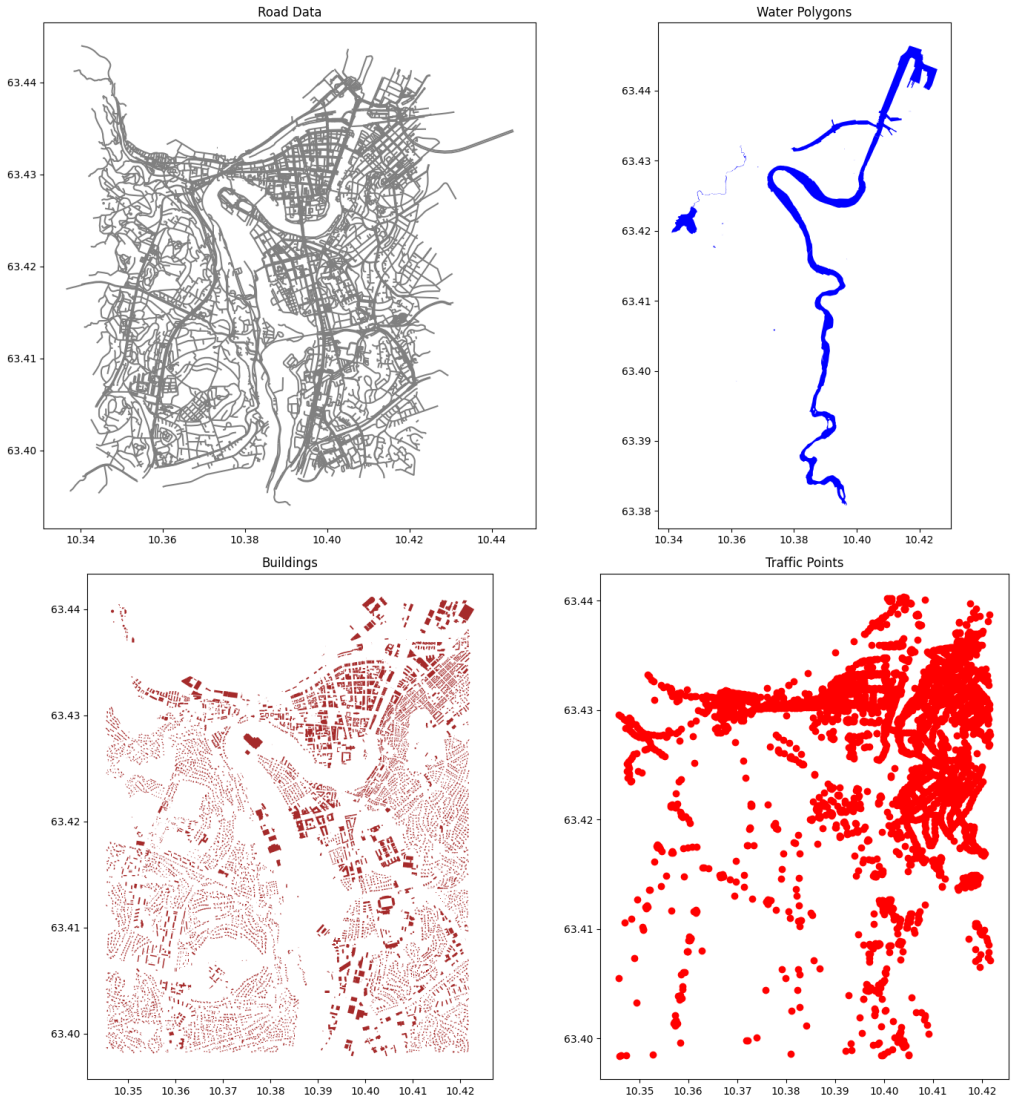


Figure 4.1.: A plot of four selected datasets constrained to a polygon of Trondheim

work with. The upcoming experiments therefore seek to benchmark model performance on three different data access methods.

The first method of presentation is to have the files from section 4.1 remain untouched. The files are stored locally on the computer on which the experiments are conducted and GeoGPT, which runs on the same computer, is able to interact with the data through the code that it generates. These files are stored on a location unknown to GeoGPT but are made available through a working directory — which location *is* known to GeoGPT — through symbolic links to the unknown location of the actual files. Code Snippet 6.3 in the chapter 6 shows how GeoGPT is able to use Python code it has generated to load road data, perform intersection with a bounding box, and writing the result to a new file in the working directory.

The second method used is to load the data into a spatial SQL database and provide the model with database schemas that can be used to generate queries. The datasets were uploaded to a Dockerized PostGIS database using QGIS's DB Manager plugin.

The third method for data access is to use the OGC API Features standard. In essense, this methods serves the data stored in the PostGIS database as a web server that allows consumers of the API to download the data over HTTP as GeoJSON.

5. Architecture

The Architecture chapter will give a detailed description of the inner workings of GeoGPT — the thesis' proposed solution to an LLM-based GIS. section 5.1 will present a high-level overview of the different modules that make up GeoGPT, and the way that they interact with each other. section 5.2 will delve into the architecture of the generic tool agent that is implemented using LangChain and LangGraph (see section 2.2), as well as the *three* different agent types that are implemented using this generic agent.

5.1. High-Level Application Architecture

A microservice architecture was employed in order to simplify development and separate concerns between the different microservices. The services are deployed as Docker Containers, and they are orchestrated using Docker Compose. Figure 5.1 shows how the application is divided into five distinct services, and the general direction of information flow between these. Figure 5.2 will go into greater detail, showing show the different services interact with each other when the user first loads GeoGPT and enters a question into the chat.

5.1.1. LangChain Server

The *LangChain Server* service is the heart of the application, and is where the Large Language Model (LLM)-related logic is situated. It is responsible for taking requests from the *Web UI* and returning suitable responses in what becomes a client-server architecture between the two services. Table 5.1 show the endpoints exposed by the server and how they can be used by a client.

5.1.2. Redis for Conversations

Redis (Sanfilippo, 2009) is a fast in-memory database that is often applied as a caching database that sits on top of some persistent database. It can also be used for vector-based storage and as a simple NoSQL database. The latter option is the way it is used in GeoGPT's architecture, and its only purpose is to store conversations. Whenever a user starts a conversation with GeoGPT an object with a unique session ID is stored to the Redis database. This object holds an array that represents the conversation. This array is written to every time either the human or GeoGPT produces a message.

Storing messages — either in memory as a simple array or in a database like Redis — is crucial to enable multi-message conversations. In order for a LLM to act as a

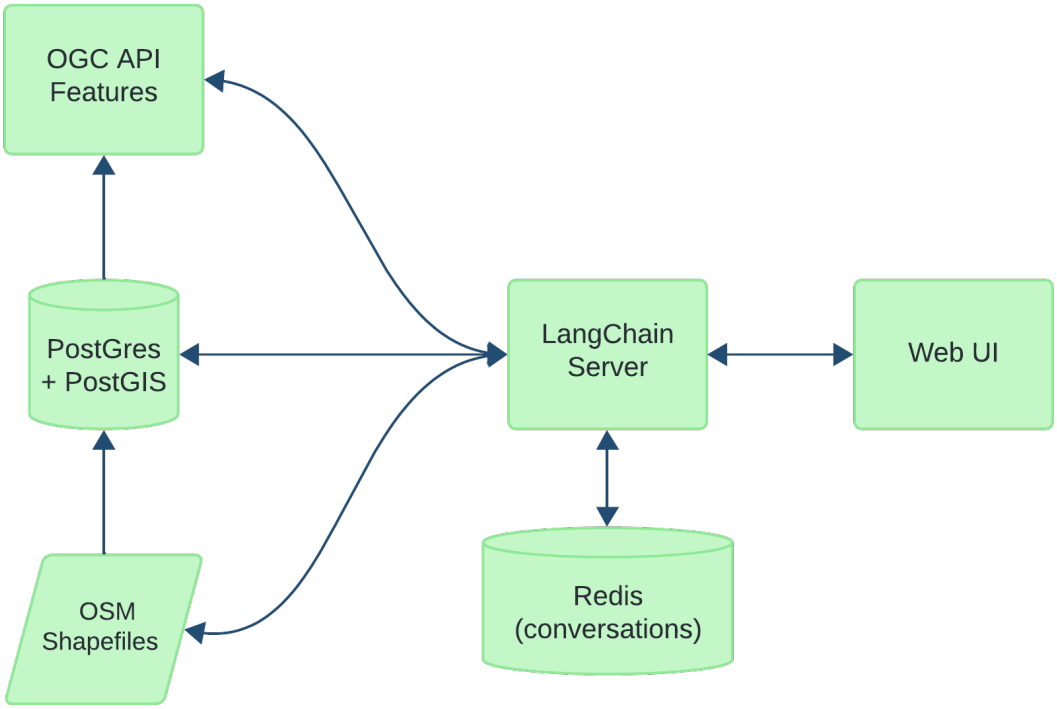


Figure 5.1.: Architecture overview

conversational agent, some sort of chat history needs to be prepended to the prompt. In the case of GeoGPT the entire chat history is prepended. This has the advantage of providing the LLM with the complete context of the chat history, but the disadvantage of potentially bloating the context window from which it is supposed to generate tokens. Therefore, as the chat becomes longer each new token will be both more expensive and take more time to get generated. A long chat history could also make the resulting prompt exceed the token limit of the LLM, or it could confuse model by providing it with messages no longer relevant to the conversation. These issue was not considered in great detail for this project, and are left for future work.

5.1.3. PostGIS OGC API Features

A PostgreSQL database with the PostGIS extension containing OpenStreetMap (OSM) data was deployed using Docker. On top of this database is a RESTful geospatial feature server called *pg_featureserv* (CrunchyData, 2024). The web server is realized through the `pramsey/pg_featureserv` Docker image[1], which is simply passed a database connection string to the database one wishes to expose on the API. Any tables in that database which have a geometry column and a specified Coordinate Reference System (CRS) will

[1] https://hub.docker.com/r/pramsey/pg_featureserv

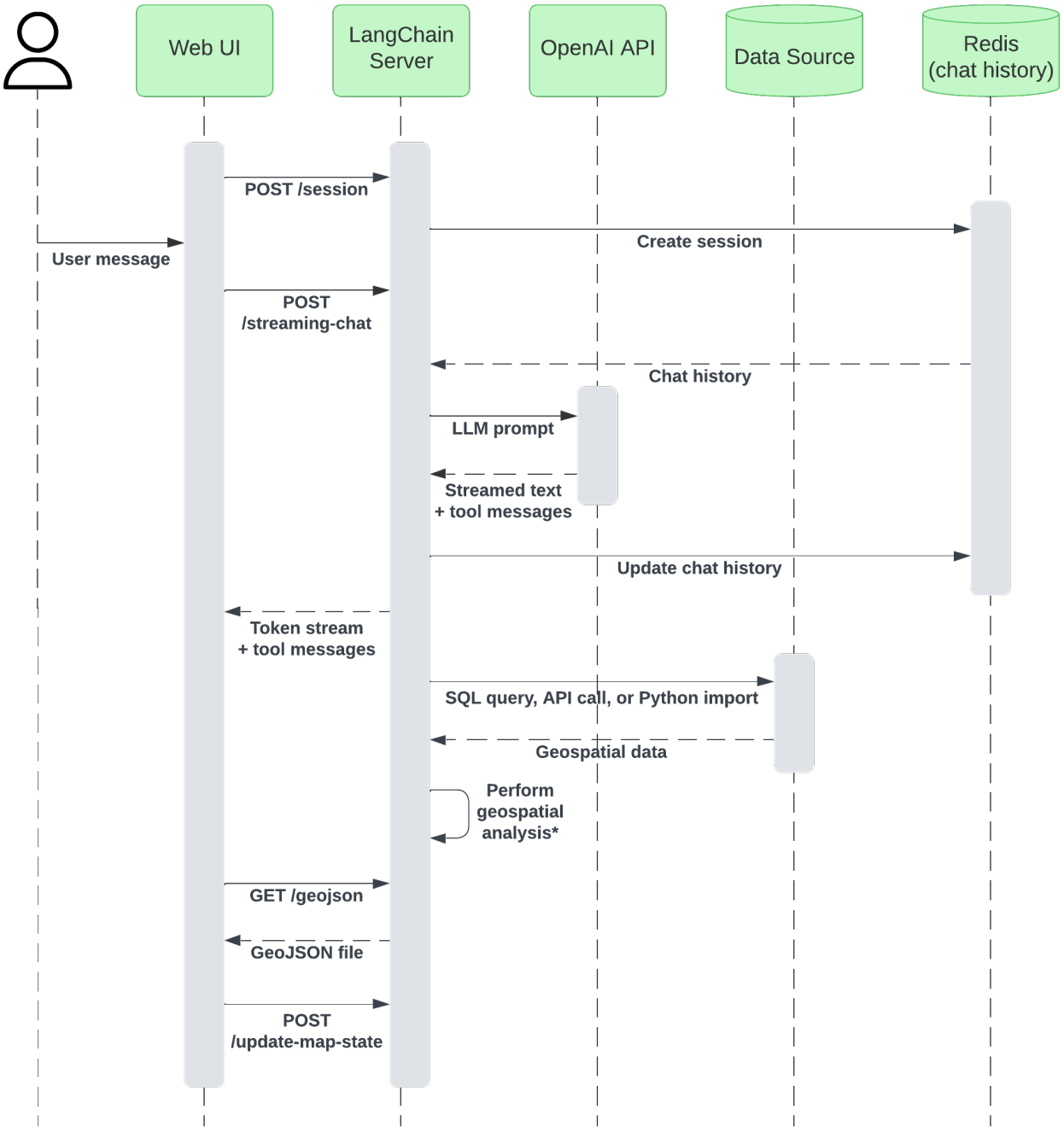


Figure 5.2.: Sequence diagram showing the information flow as the user loads and sends a message to GeoGPT

Table 5.1.: Summary of Server Endpoints

Endpoint	Method	Description
/session	GET	Takes a `session_id` as a query parameter, allowing the client to continue on a pre-existing session.
/session	POST	Creates a new session with an empty conversation.
/streaming-chat	GET	Endpoint for chatting the LLM. Takes a `message` as a query parameter and returns an event stream, allowing for token streaming from server to client.
/update-map-state	POST	Send the state of the client map to the server. Keeps the server updated on what layers are present in the map, their color, etc.
/geojson	GET	Takes a `geojson_path` as a query parameter. Allows the client to retrieve a given GeoJSON file that is stored in the working directory on the server.
/upload	POST	Allows the client to upload one or more files to the working directory on the server.

be exposed through the web server. pg_featuresserv allows for features like bounding box filtering, feature limiting, and CQL filtering. These are added as query parameters to the URL of the collection items (for instance: `.../collections/{collection_id}/items.json?limit=1000&filter=name IS NOT NULL`). In the internals of pg_featuresserv, this URL will be converted to an SQL query that will be run against the database. Results of the query will be returned as GeoJSON. Code Snippet 5.1 shows an example of how CQL code is converted into SQL code:

```
1 \\ CQL code passed through the `filter` query parameter
2 within(geom, POINT(0 0))
3
4 \\ SQL code that will be run agains the database
5 ST_Within("geom",'SRID=4326;POINT(0 0)'::geometry)
```

Code Snippet 5.1: Conversion from CQL to SQL

5.1.4. Web UI

The user interface is made with SolidJS. By design, it is very minimal. One of the goals of the project is to simplify the way we do GIS analysis. One of the key design goals was therefore to make the interface as familiar to the user as possible and lowering the chance of the user doing something wrong.

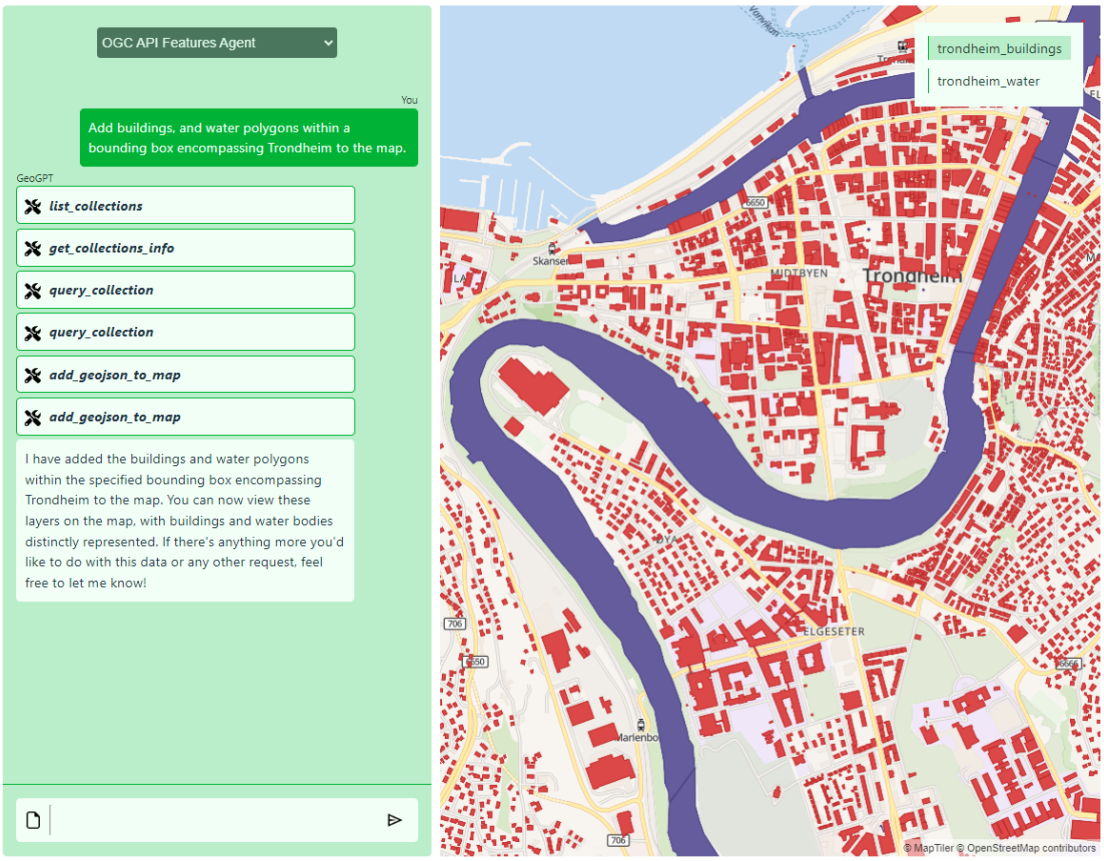


Figure 5.3.: Web UI

The chat interface was designed to imitate the interface of OpenAI's ChatGPT. Tokens and tool messages are streamed from the LangChain server, which helps the user follow the process that GeoGPT goes through when solving a problem. During this process it is possible for the user to cancel token generation if he/she sees that the system is heading off in the wrong direction. The text that is generated and streamed to the client is in Markdown format. A library called *showdown*[2] is used to convert Markdown into HTML, ensuring that tables, code blocks, lists, and other elements are properly rendered. Next to the input field on the bottom of the chat is a file upload button. Files that are uploaded here will be added to the working directory on the LangChain server, allowing

[2] https://github.com/showdownjs/showdown

GeoGPT to perform analyses on them and optionally adding them to the map.

The map is created using MapLibre[3], an open-source fork of Mapbox which in December 2020 changed to a non-open-source software licence. A base map from OSM is used, fetched through a website called MapTiler. GeoJSON files of any kind that are fetched from the server will be added to the map automatically with a random color. On the top-right of the map is an overlay listing all layers that are currently present in the map. Using the arrow keys on this list will change the z-index in the map of the selected layer.

5.2. Agent Architecture

Three different agent were implemented for GeoGPT. These are listed below:

- **OGC API Features Agent** – Utilizes OGC API Features to interact with geographic data through standardized web interfaces.
- **Python Agent** – Accesses and manipulates shapefiles using Python code, enabling detailed geographic data processing.
- **SQL Agent** – Interacts with data stored in an SQL database, allowing for complex queries and data management within GeoGPT.

Common to these agents is their agentic architecture, which is described in subsection 5.2.1. They way that they differ is through their assigned *tools*. These differences will be made apparent in Figure 5.4. Other slight differences are seen in the way that they are prompted. The prompting strategy used in GeoGPT and these minor differences will be discussed in subsection 5.2.3.

5.2.1. LangGraph Agent Implementation

The agentic behaviour of GeoGPT is implemented using LangGraph, an extension to LangChain intended to make implementation of cyclic behaviour easier. Figure 5.4 illustrates the flow between the various nodes that make up the agent. A state dictionary is passed between and updated by these nodes. Included in this state is the chat history (previous messages), the path to GeoGPT's working directory, a list of the current files in this directory, and other, less important state. The implementation is based upon a prebuilt implementation from LangGraph.[4]

The ___*start*___ node serves as the entry point of the agent. At this point, only one message is present in the state, namely the initial message from the user. The ___*start*___ node points to the *agent* node, where a response to the user is generated by an LLM. This response could contain a textual response, or it could contain instructions to execute some tool. For this reason, the current state containing both the user message and the AI message, is sent to the *conditional* node called *agent_should_continue*. This

[3] https://github.com/maplibre/maplibre-gl-js

[4] https://github.com/langchain-ai/langgraph/blob/main/langgraph/prebuilt/chat_agent_executor.py

node simply checks if the agent outcome (the last AI message) includes the "tool_calls" keyword argument. If this is the case then some tool should be executed, and we should "continue". If not, the agent has generated a final textual response to the user.

In the *action* node the values inside the "tool_calls" object are converted into `ToolInvocation` objects that are used to invoke predefined tools. "tool_calls" is a list of "tool_call" objects, each of which has attributes called *name* (the name of the tool) and *arguments* (arguments that should be passed to the tool). These tools are then executed, possibly having side effects on the server, and the resulting output of the tools are appended as messages to the chat history. Figure 5.5 illustrates this behaviour. This way the agent can inspect the results of the code it is "executing", as well as being notified about possible errors in the input parameters it provided. Through the cyclic behaviour of the agent graph the agent can repeatedly call tools to try and answer the request from the user. When the agent finds to reason to call any more tools it generates a textual response, making *agent_should_continue* return `False` so that the termination node (___*end*___) is reached.

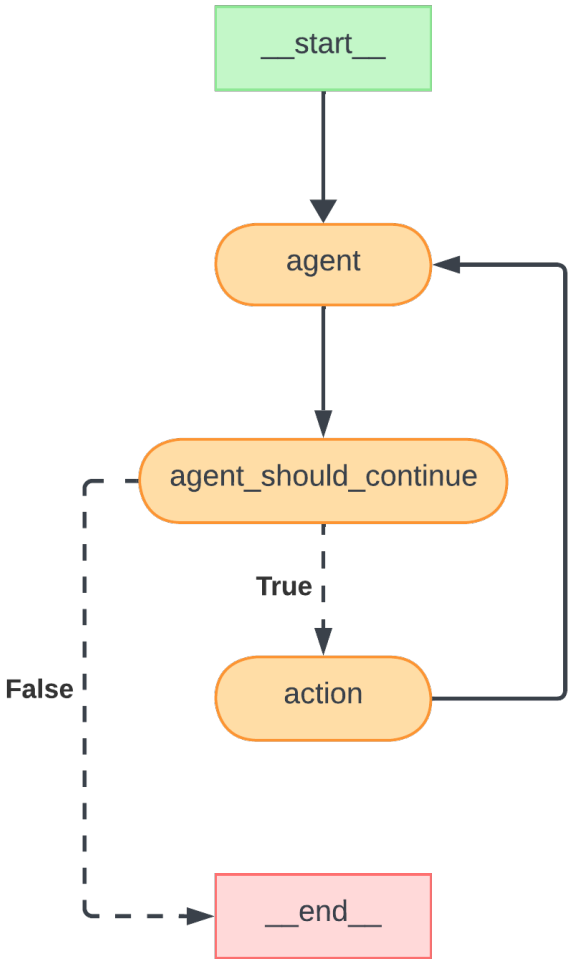


Figure 5.4.: Generic tool agent graph

5.2.2. Tools

Table 5.2 shows an overview of the tools that are available to each of the agents. As described in subsection 2.1.5, these are defined by a name, an overall description of the tool's functionality, and a description of the parameters the tool expects.

The OGC API Features agent has access to a total of five tools. `list_collections`, which takes no parameters, sends a `GET` request to the `/collections` endpoint and uses the response to construct a string listing the names of available collections along with their descriptions. The tool is designed to give the agent an overview of what kinds of data are available. Using the response from `list_collections` the agent can now invoke the `get_collections_info`. This tool takes a list of collection names and returns

AI

```
sql_db_query
{
  "query": "SELECT * FROM osm_places_polygons WHERE name =
'Nordland' AND fclass = 'county';",
  "layer_name": "nordland_county"
}
```

TOOL

Query returned 1 feature.Below are the first 1 feature:

	id	geom	osm_id	code	fclass	population	name
0	4790	MULTIPOLYGON (((10.57806 65.47018, 10.57911 65...	408105	1041	county	241084	Nordland

SYSTEM

Files written to the working directory (`/tmp/tmphhikab_f`):
- /tmp/tmphhikab_f/nordland_county.geojson

AI

```
add_geojson_to_map
{
  "geojson_path": "/tmp/tmphhikab_f/nordland_county.geojson",
  "layer_name": "nordland_county"
}
```

TOOL

"`/tmp/tmphhikab_f/nordland_county.geojson` sent to client..."

SYSTEM

State of map on client:

```
{
  "layers": [
    {
      "id": "nordland_county",
      "type": "fill",
      "source": "nordland_county",
      "paint": {
        "fill-color": "#893f23"
      }
    }
  ]
}
```

Figure 5.5.: Example of a chat trace

relevant information for each of these collections. This includes the JSON response from the "landing page" of the collection, presenting details such as the collection description, spatial extent, and available attributes. In addition to this information, there is a list of common values for certain high-cardinality attributes, along with the prevalence of each value in percentages. Code Snippet 5.2 shows an example of this. This is achieved by querying a large number of features from `/collections/{collection_id}` and calculating the prevalences between them.

```
1 Property: fclass
2     tree: 71.1%
3     peak: 27.2%
4     beach: 0.9%
5     cave_entrance: 0.5%
6     spring: 0.2%
7     cliff: 0.1%
8     volcano: 0.0%
```

Code Snippet 5.2: Prevalences of common values for the *fclass* attribute in the *osm_natural_points* collection

The **`query_collection`** tool is used to retrieve features from a collection. It takes a collection name, a CQL filter, a bounding box, and a layer name. The CQL filter and bounding box allow for retrieval of a subset of the features of the collection being queried. Based on the parameters an URL like this is constructed:

> https://localhost:9001/collections/{collection_id}/items.json?limit=10000&filter={cql_filter}&{bbox}

The features retrieved from this query is saved on the working directory on the LangChain server as "{layer_name}.geojson", as a side effect of the tool. The message returned from the tool reads something like this: "Query returned 5627 features." If the GeoJSON itself were returned as a tool message, this would quickly bloat the context window of the LLM, and therefore it is avoided.

Common for both the OGC API Features agent and the Python agent is the **`python_repl_ast`** tool. This tool accepts a string of Python code, executes said Python code, and returns whatever the code prints to the standard output in a so-called Read-Eval-Print Loop (REPL). In case the code errors, the error message is returned instead. This Python tool is the main way for these two agents perform geospatial analyses. An advantage of using REPLs is that the code can be executed in blocks, with variables from one block being shared with other blocks. This means that the first block may load large files into memory — an often time-consuming operation — while subsequent blocks can reuse this in-memory variable, even if that block should error. This allows the LLM to quickly retry whenever the code errors or if the outcome of the initial code wasn't as expected. The OGC API Features agent will have available the files it downloaded using

Table 5.2.: Overview of the agent types and their tools

Agent Type	**Tools**
OGC API Features	`list_collections`
	`get_collections_info`
	`query_collection`
	`python_repl_ast`
	`add_geojson_to_map`
Python	`python_repl_ast`
	`add_geojson_to_map`
SQL	`sql_db_list_tables`
	`sql_db_schema`
	`sql_db_query`
	`add_geojson_to_map`

the `query_collection` tool for analysis using Python, whereas the Python agent will have files corresponding to all collections available for analysis. These files are stored locally on the machine and they are added to the agent's working directory at system launch using symbolic links[5]. This is because a new working directory is created every time a new agent is created in order to give each agent a "clean canvas" to work from.

`add_geojson_to_map` is the only tool that is common for all three agent types. The tool's job is to add layers to the map on the client. It takes two parameters: the name/path of a GeoJSON file stored on the LangChain server's working directory and a layer name. Invoking the tool will send a message to the client including the full path to the file on the server. The client will then make a `GET` request to the server on the `/geojson` endpoint, asking for the contents of this file to be returned so that it can be added to the map.

The SQL agent has tools very similar to the OGC API Features agent. The SQL agent is connected directly to the same database that the Features API is served on top of. **`sql_db_list_tables`** is a tool that will list all database tables along with their description. **`sql_db_schema`** takes a list of table names and will return information about attributes, prevalence of different values in high-cardinality columns, etc., about these tables, much like `get_collections_info`. **`sql_db_query`** takes as parameter arbitrary SQL code executes this code against the database. The tool will make sure that any

[5] https://en.wikipedia.org/wiki/Symbolic_link

query result that has a geospatial component will be stored as GeoJSON on the server. This is required to make it possible to add the geometries to the client map.

5.2.3. Prompt Templating

Prompt templating is a way to produce prompts for an LLM that follows a predefined structure. Figure 5.6 shows an example of what the LLM that drives GeoGPT's SQL agent sees when the user asks which county is the largest by size.

The overall template consists of a collection of tools, a system message, and a chat history. Each time the LLM in the *agent* node (see Figure 5.4) is called this entire template is passed to it. LLMs have no inherent memory, so in order to have a chat conversation the entire chat history is passed with the prompt.[6]

The system message has a structure on its own. The first half of the system message in Figure 5.6 is intended to give the LLM contextual awareness and hinting to how it should work towards solving tasks. We start by telling it that is "helpful GIS agent/consultant that has access to an SQL database containing OpenStreetMap data". This is a common way of giving LLMs a persona that it can adopt, and results in responses as can be seen in Figure 5.7. Continuing, we give it some information of how to string tool calls together to solve a tasks. We tell it to first list available tables, then look up schemas of relevant tables, then using the information gathered to construct an SQL query to answer the user's request. We also remind it that it has to add the result of the analyses to the map itself (using the `add_geojson_to_map` tool).

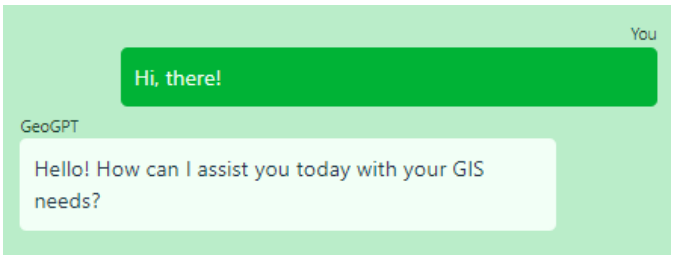


Figure 5.7.: Conversation showcasing the effect of giving an LLM a persona through the system message

The next part of the system message concerns the working directory that GeoGPT has to work with. First, we tell it what the path to the working is. This is especially important for the OGC API Features agent and the Python agent, as they need to manually read and write to the correct path. Code Snippet 6.1 shows the importance of reminding the LLM about the location of the working directory. Having a working directory is important to control which files the agent has available, and also to make

[6]Several strategies have been developed by researchers to avoid having to pass the entire chat history to the LLM each time, as this eventually will bloat the context window which could make generation slower and worse in quality. Strategies include picking only the n last messages in the chat history, passing a summary of the chat instead of entire messages, or utilizing knowledge graphs.

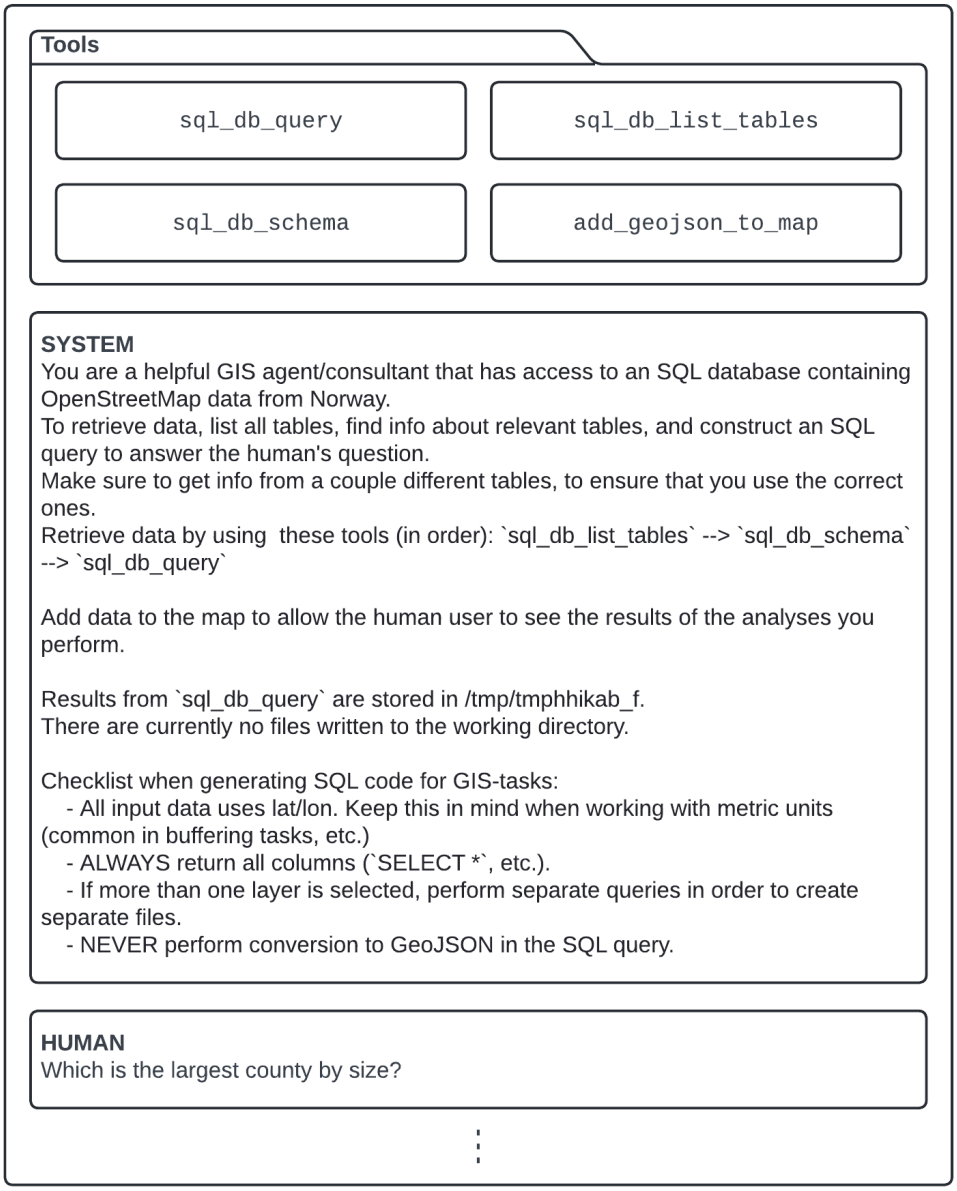


Figure 5.6.: Chat template

sure it doesn't save files to the folder on the server that GeoGPT is run from, as this would bloat the actual source code. Also, having a working directory lets us list the files it has available. Currently, in Figure 5.6, there "are currently no files written to the working directory", but if there were they would be listed in a bullet point list.

The final part of the system message is a checklist to remind the LLM about common pitfalls that it might run into when generating SQL code. A similar checklist for Python is provided for the other two agents, where we remind it that it needs to use metric CRSs when doing area calculations, etc.

Normally, there is only a single system message in LLM-based agents. GeoGPT, however, features a — to the author's knowledge — novel usage of system messages. As can be seen in Figure 5.5, system messages are appended mid-conversation, providing updates about state changes in the system. The first system message is added because a new file has been added to the working directory, as a result of invoking the `sql_db_query` tool. The system message helps GeoGPT stay up-to-date on what files it has available, and it uses this information immediately to add the only file available to the map using `add_geojson_to_map`. A second system message is eventually added to tell the GeoGPT that client map has been modified. The message helps ensure GeoGPT that the invocation of `add_geojson_to_map` was successful. If it wasn't, the system message would inform it about this.

6. Experiments

The Experiments is divided into two main sections: section 6.1, which describes the methodology and rationale behind the conducted experiments; and section 6.2, which presents the experimental results and offers interpretations of these findings. The latter section will also present several example results, including screenshots of the application and code that generated by GeoGPT.

6.1. Experimental Setup

The experiments conducted to evaluate the performance of GeoGPT on geospatial tasks are divided into two approaches. The first approach, elaborated upon in subsection 6.1.1, is intended to evaluate GeoGPT's ability to solve a variety of geospatial tasks. The second approach, presented in subsection 6.1.2, aims to uncover the importance of the initial user prompt in guiding GeoGPT to solve the task. Subsection 6.1.3 describes the hardware on which the experiments were executed, and the specific models that were used.

6.1.1. GIS Q&A Experiment

The first approach seeks to evaluate its ability to successfully answer questions that have a concrete answer. To do this, a Q&A dataset was constructed. This dataset consists of set of 12 GIS-related questions with corresponding correct answers. For each record in the dataset, a description of how a human would find it natural to approach the problem. This description is provided as a step-by-step path towards the solution, and is only included to guide any reader as to how the system would be expected to solve the system. The full Q&A dataset can be found in Table B.1. This set of experiments will allow for quantitative assessment of GeoGPT's GIS-abilities, and is a feasible way of benchmarking the system.

Another aspect that the benchmarking approach will try to evaluate is the consistency of the system, its ability to repeatedly provide an acceptable answer to the same user question. Each of the 12 questions are therefore asked three times per agent type.

With the implementation of three different agent types (see Table 5.2), the total number of test runs becomes the following:

$$12 \text{ questions} \cdot 3 \text{ agent types} \cdot 3 \text{ repetitions} = 108 \text{ tests}$$

Outcome Evaluation

Each test run's answer will be manually evaluated, and the outcome will be annotated as one of *success*, *partial success*, and *failure*. Table 6.1 shows the guidelines used when assigning test results.

Table 6.1.: Description of Success

Outcome	**Guideline**
Success	The question was answered correctly and little to no follow-up from the user was required to produce the desired outcome. No false assumptions were made by the system when answering the question.
Partial Success	Portions of the question were answered correctly or semi-correctly, and/or some follow-up from the user was required to guide the system toward the solution.
Failure	The question was answered incorrectly answered and/or false assumptions were made by the system while attempting to answer the question.

The annotated outcomes are then encoded using the ordinal encoding presented Table 6.2. A higher value indicates a better outcome. These encoded outcome values enable standard deviation calculations, which serve as a suitable measure for assessing repeatability. This approach also allows for comparisons across different agent types and configurations.

Table 6.2.: Encoding for Test Outcome

Outcome	**Encoded Value**
Success	2
Partial Success	1
Failure	0

Cost and Duration

The application is hooked up to LangChain AI's tracing system, *LangSmith*. Apart from being a useful tool for debugging purposes, it provides a simple way of obtaining detailed data for token and time usage for a particular run, as well as the total cost of the run. These are metrics that will be recorded and used in the evaluation of GeoGPT.

To summarize, the following metrics are recorded for a given test run:

- The outcome of the test (*success*, *partial success*, or *failure*)
- The total duration in seconds
- The total number of tokens used
- The total cost for the run in American dollar

6.1.2. Prompt Quality Experiment

The second set of experiments are constructed to evaluate the importance of the initial question/prompt from the human user. As stated in Background and Motivation, part of the motivation for developing an LLM-driven GIS like GeoGPT is to make GIS more accessible to non-experts. At the same time, it may be valuable to assess the extent to which a carefully constructed prompt by a GIS expert can enhance the system's output.

6.1.3. Hardware and Model Version

All experiments were executed locally on a Lenovo ThinkPad E490, which has an Intel® Core™ i7-8565U CPU @ 1.80GHz processor, 15.8 GB usable RAM, and 256 GB SSD storage. Everything but the LLM inference was executed locally. Text generation was done using OpenAI's API.

It is worth noting that two slightly different models were used during testing. The explanation of this is the release of the `gpt-4-turbo-2024-04-09` in mid-April. According to OpenAI, "this new model is better at math, logical reasoning, and coding" compared to `gpt-4-0125-preview`[1], which is the model that was used at the start of the experimentation phase of the master's thesis. At the new model's release, a decision was made to use this for the remaining experiments. The experiments that had already been conducted were not re-run due to time constraints and a belief that these slight model upgrades would not significantly change the outcome of the experiments.

6.2. Experimental Results

Subsection 6.2.1 and subsection 6.2.2 will present the outcome of the experiments presented in subsection 6.1.2 and subsection 6.1.2, respectively.

6.2.1. GIS Q&A Experiment — Results

Graphs created for this chapter are created using Matplotlib[2], a Python library suitable for creating visualizations like bar charts, box plots, etc.

[1] OpenAI has a GitHub repository containing the code they use to evaluate their Large Language Models (LLMs) and benchmark results for OpenAI models and reference models from other companies: https://github.com/openai/simple-evals.

[2] https://matplotlib.org/

Outcomes

As described in subsection 6.1.1, a total 108 tests were run using the 12 available Q&A samples, with the same question being repeated three times for each of the three agent types, resulting in 36 test runs per agent. Figure 6.1 displays a bar chart for the outcome distribution per agent.

Figure 6.1.: Outcome distribution between different agent types

From Figure 6.1, we can read that the OGC API Features and Python agent have comparable results, and that the SQL-based agent performs significantly better compared to the other two in terms of producing the desired outcome.

Cost and Duration

This section features three different box plots: Figure 6.2, Figure 6.3a, and Figure 6.3b. The Matplotlib implementation of the box plot follows the description found on Wikipedia[3,4]. Box plots allow us to easily visualize where the 0th (Q_0), 25th (Q_1), 50th (Q_2), 75th (Q_3), and 100th (Q_4) percentiles of the datasets lie, as well as the dataset's outliers. Outliers are those data points fall outside 1.5 times interquartile range, that is, the distance between Q_3 and Q_1 in each direction.

[3] https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.boxplot.html
[4] https://en.wikipedia.org/wiki/Box_plot

Figure 6.2 displays a box plot with a logarithmic y-axis showing the relative durations for task completion across the different agent types. Here we can see that the SQL agent spends the least amount of time per task, and that the OGC API Features agent has a slightly higher median but with a few time-consuming outliers. The Python agent is the odd one out with a median of ~ 82 seconds, a $Q3 \sim 293$ seconds, and a $Q4 \sim 984$. The large gap to the other two agents it largely due to the Python agent's tendency to load large datasets into memory without filtering the data on load using a bounding box. For instance, when attempting the task of calculating the difference between the polygon outlining Oslo and water polygons, the Python agent used nearly 40 minutes on the entire task. 94% of the time was spent executing the code presented in Code Snippet 6.1. The main reason for the long execution time is line 8, where the whole `osm_landuse_polygons.shp` dataset is loaded into memory. This dataset has a size of 1,383 MB, and loading such amounts of data in this way is very time-consuming. The Python agent was the only agent with such issues because the OGC API Features agent is limited to 10,000 features per dataset and the SQL agent does not load the data into memory like the other agents do.

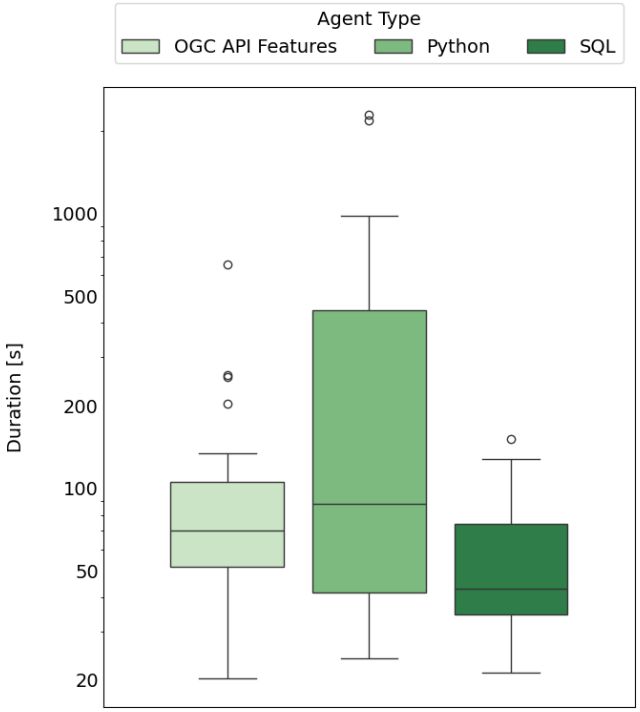


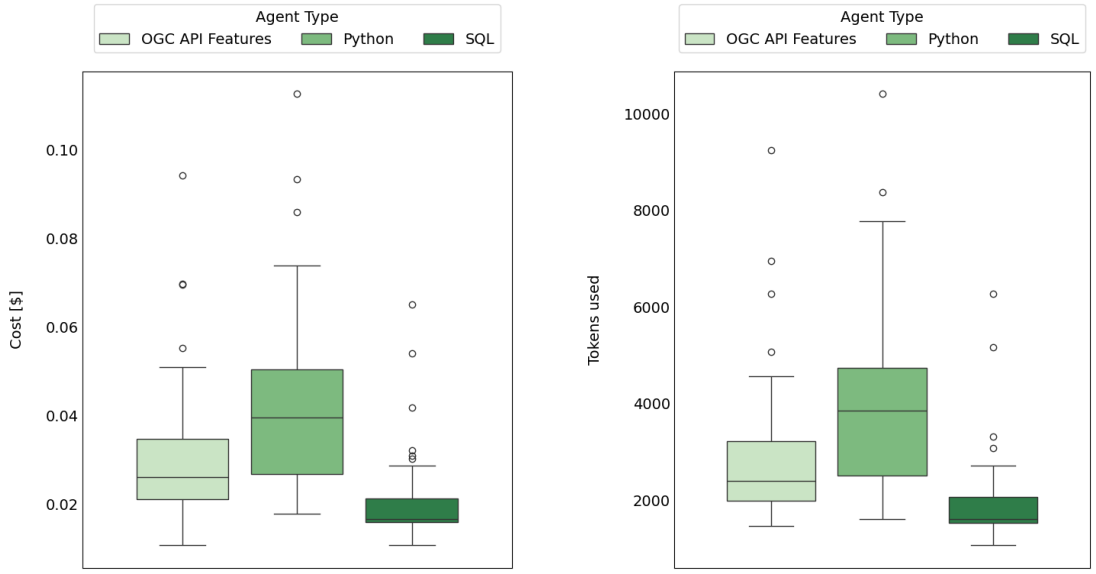
Figure 6.2.: Duration per Agent Type

Figure 6.3 shows box plots for the amount of tokens used per run and the total cost of these. Naturally, these figures appear very similar, as OpenAI sets fixed prices for the input tokens and generated output tokens for their GPT models. According to

```
import geopandas as gpd

# Paths to the shapefiles
landuse_path = '/tmp/tmpsutdy6it/osm_landuse_polygons.shp'
places_path = '/tmp/tmpsutdy6it/osm_places_polygons.shp'

# Load the data from shapefiles
landuse_gdf = gpd.read_file(landuse_path)
places_gdf = gpd.read_file(places_path)

# Filter out 'residential' areas from the landuse data
residential_gdf = landuse_gdf[landuse_gdf['fclass'] == 'residential']

# Compute the spatial difference to exclude residential areas from the places data
oslo_outline = gpd.overlay(places_gdf, residential_gdf, how='difference')

# Path for the output GeoJSON file
output_path = '/tmp/tmpsutdy6it/oslo_outline_no_residential.geojson'

# Save the resulting GeoDataFrame to a GeoJSON file
oslo_outline.to_file(output_path, driver='GeoJSON')

# Output the path to the saved file
print(output_path)
```

Code Snippet 6.1: GeoGPT-generated Python code aimed at computing the difference between the Oslo outline and water features within it

their websites[5], they charge $10 per million input tokens and $30 per million output tokens for their GPT-4 Turbo models. From the results of the experiments, a ratio of approximately 10.7 per million tokens — either input *or* output — was calculated, which is very close to the input token price. This aligns with the observation that the number of input/prompt tokens is significantly greater than the number of output/generated tokens for the experiments conducted. The correlation matrices in Figure 6.4 confirm that this ratio is consistent.



(a) Average cost per call for each agent type

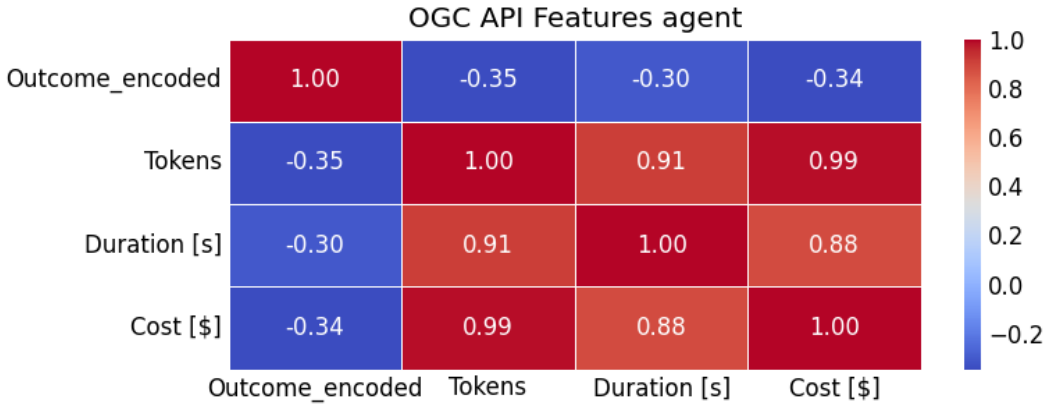
(b) Average token usage per agent type

Figure 6.3.: Cost and token usage

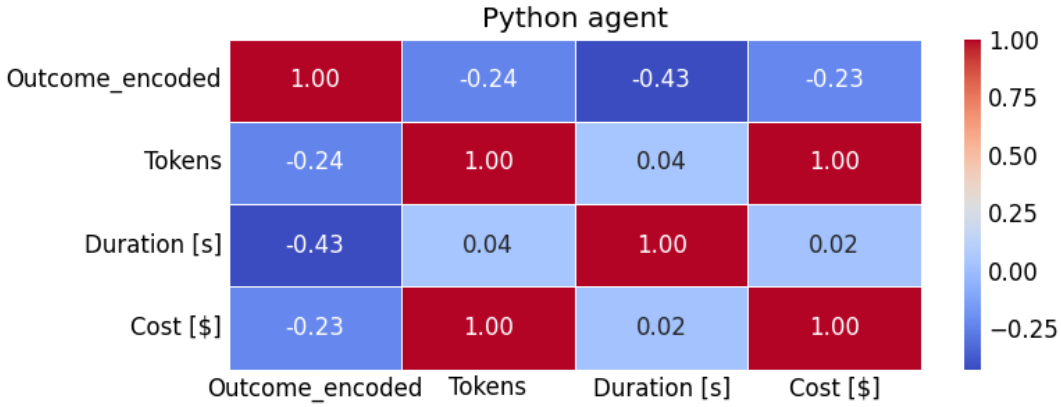
Another observation that can be made from Figure 6.4 correlation between duration and token usage is inconsistent across the agent types. The OGC API Features agent and SQL agent have strong correlation between these metrics, but the Python agent has next to none. This supports the observation that, for the Python agent, code execution time, particularly the time spent on expensive imports, is likely the most significant factor in determining the time required to complete tasks for the datasets and experiments used in this thesis.

A third observation that can be made from the Figure 6.4 is the slight negative correlation between the encoded outcome and each of the following variables: token usage, duration, and total cost. This suggests that a task that takes longer to complete, and thus is likely to be more expensive in terms of token usage, is more likely to produce an undesirable outcome. Possible reasons as to why this is the case will be explored in subsection 7.3.1 the Discussion.

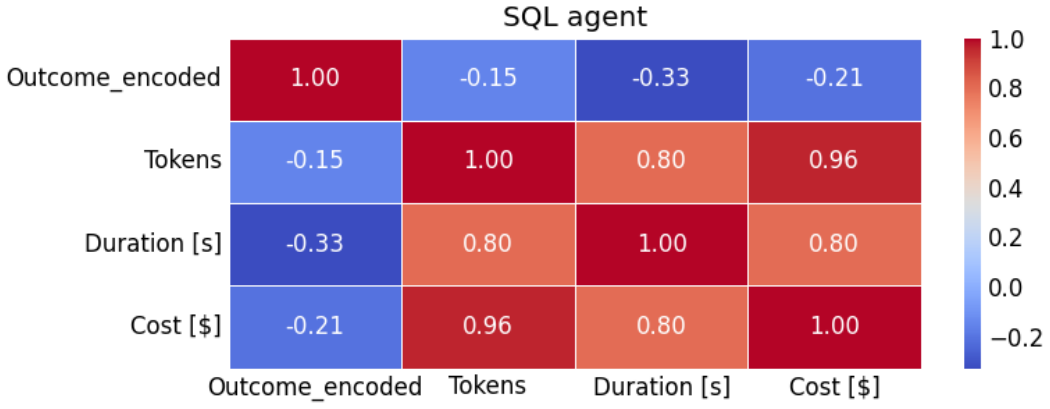
[5] https://openai.com/pricing



(a) Correlation matrix for the OGC API Features agent



(b) Correlation matrix for the Python agent



(c) Correlation matrix for the SQL agent

Figure 6.4.: Correlation matrices for metrics for the three agent types

Repeatability

Table 6.3 shows the average standard deviation for each agent type, as well as the mean of these three standard deviations. The latter serves as an overall measure of GeoGPT's ability to repeatedly produce the desired outcome from a given query. Standard deviations were calculated for each triplet of identical test samples, in which both the question and the agent type remained the same. To produce a numerical value for the standard deviations, the encoded outcomes (see Table 6.2) were used. Taking the average of the standard deviations for all 12 triplets for each agent type produced the numbers found in Table 6.3.These numbers indicate that there is a notable amount of inconsistency in GeoGPT's ability to produce correct answers, on an average deviating with almost "half an outcome category" (0.376 for the encoded values) for the same task and agent type.

Table 6.3.: Standard Deviation by Agent Type

Agent Type	**Outcome Std. Deviation**
OGC API Features	0.552
Python	0.337
SQL	0.241
Mean	0.376

Successful Responses

Figure 6.5 shows a successful response from GeoGPT's SQL agent when asked how many counties the Glomma river runs through.[6] As a final answer to the initial prompt it lists all four correct counties in a numbered list. Code Snippet 6.2 shows the code generated by GeoGPT for the second invocation of `sql_db_query`. The second invocation is nearly identical to the first one, but in the first invocation the geometry column — named "geom" — was not included in the response, something the tool informs GeoGPT of in its tool message. GeoGPT therefore decided to run the query again, now making sure to include the geometry column in the query result, before adding the result to the map using `add_geojson_to_map`. A follow-up message instructed GeoGPT to "Add Glomma to the map", allowing for visual verification that the results were correct.

[6] This run was not part of the results used for evaluation due to a bug in the SQL agent that caused the `sql_db_query` to unnecessarily execute many queries twice. GeoGPT's response would not be different had the bug not been present for the run, but it would have taken a bit longer to complete.

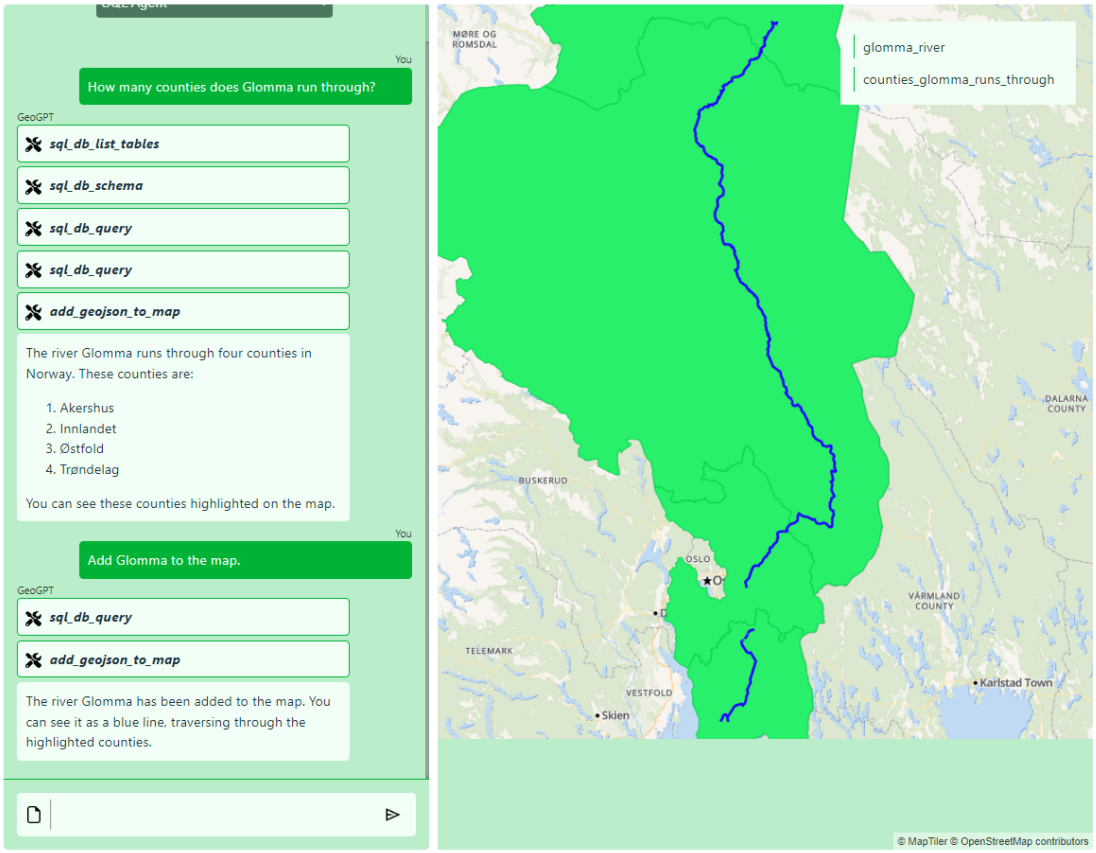


Figure 6.5.: Successful response from GeoGPT's SQL agent when asked how many counties the Glomma river runs through

```sql
WITH river AS (
    SELECT geom
    FROM osm_waterways_lines
    WHERE fclass = 'river' AND name ILIKE 'Glomma'
),

places AS (
    SELECT geom, name
    FROM osm_places_polygons
    WHERE fclass = 'county'
)

SELECT DISTINCT places.name AS county_name, places.geom AS geom
FROM river, places
WHERE ST_Intersects(river.geom, places.geom);
```

Code Snippet 6.2: SQL code generated by GeoGPT to retrieve polygons of the counties that the Glomma river runs through

Figure 6.6 shows a partially successful attempt from GeoGPT's Python agent to calculate the number of trees along Munkegata in Trondheim. While setting a definitive answer is difficult, the correct answer was set to about 100 trees when using a buffer of 20 meters around the line segments that make up the road. In Code Snippet 6.3 we can see how GeoGPT loads the road data using an approximate bounding box around Munkegata *in Trondheim* to avoid getting roads named "Munkegata" in other cities, before filtering and saving the loaded data to only for features with the appropriate value for the `name` attribute. In the following code block (see Code Snippet 6.4), GeoGPT gets the tree data by filtering the `osm_natural_points` dataset on the `fclass` attribute before loading the road data for Munkegata from obtained in the subsequent code block, converting both datasets into a metric CRS, creating a buffer around the road data, and finally performing an intersection between the buffer and the tree data and printing the number of trees found to the standard output.

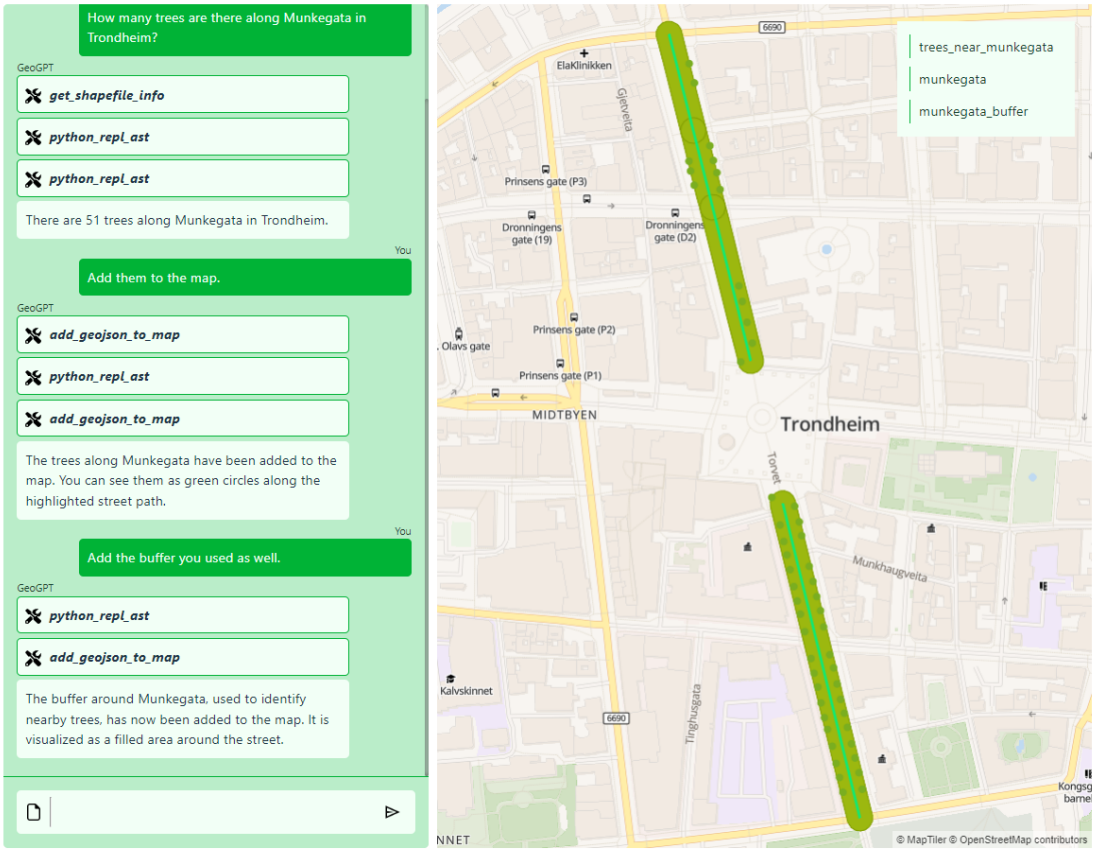


Figure 6.6.: Partially successful response from GeoGPT's Python agent when asked how many trees there are along Munkegata in Trondheim

```python
import geopandas as gpd

# Load roads data with a focus on Munkegata
roads_path = '/tmp/tmpl8zsdj6m/osm_roads_lines.shp'
munkegata_bbox = (10.3910, 63.4270, 10.4030, 63.4330) # Approximate bounding box around Munkegata
munkegata_roads = gpd.read_file(roads_path, bbox=munkegata_bbox)

# Filter for Munkegata
munkegata = munkegata_roads[munkegata_roads['name'] == 'Munkegata']

# Save filtered Munkegata road to GeoJSON
munkegata_path = '/tmp/tmpl8zsdj6m/munkegata.geojson'
munkegata.to_file(munkegata_path, driver='GeoJSON')

print(munkegata_path)
```

Code Snippet 6.3: Python code to exclude roads named Munkegata *outside* of Trondheim

```python
from shapely.geometry import box
import geopandas as gpd

# Load the trees points
natural_points_path = '/tmp/tmpl8zsdj6m/osm_natural_points.shp'
trees = gpd.read_file(natural_points_path)

# Filter for trees
trees = trees[trees['fclass'] == 'tree']

# Load Munkegata road
munkegata_path = '/tmp/tmpl8zsdj6m/munkegata.geojson'
munkegata_road = gpd.read_file(munkegata_path)

# Project to UTM for accurate distance measurement
munkegata_utm = munkegata_road.to_crs(epsg=32633)
trees_utm = trees.to_crs(epsg=32633)

# Buffer Munkegata by 10 meters to include nearby trees
tree_buffer = munkegata_utm.buffer(10)

# Spatial join trees within buffered Munkegata
trees_near_munkegata = gpd.sjoin(trees_utm, gpd.GeoDataFrame(geometry=tree_buffer), how='inner', op='intersects')

# Count the number of trees
tree_count = trees_near_munkegata.shape[0]
print(tree_count)

```

```
29 # --> 51
```

Code Snippet 6.4: Python code to calculate the number of trees along Munkegata in Trondheim

Unsuccessful Responses

While GeoGPT was able to get a lot of questions correct, it provided more or less unsuccessful answers at least once for all questions but one.

Figure 6.7 exemplifies a common mistake with GeoGPT's agents which is a central topic in the Discussion. Here, the Python agent has failed to recognize that the road datasets it has available is in fact for the whole of Norway. As Code Snippet 6.5 shows, the agent performs the correct analysis based on the assumptions it has already made, but the answer is far from what the user enquired.

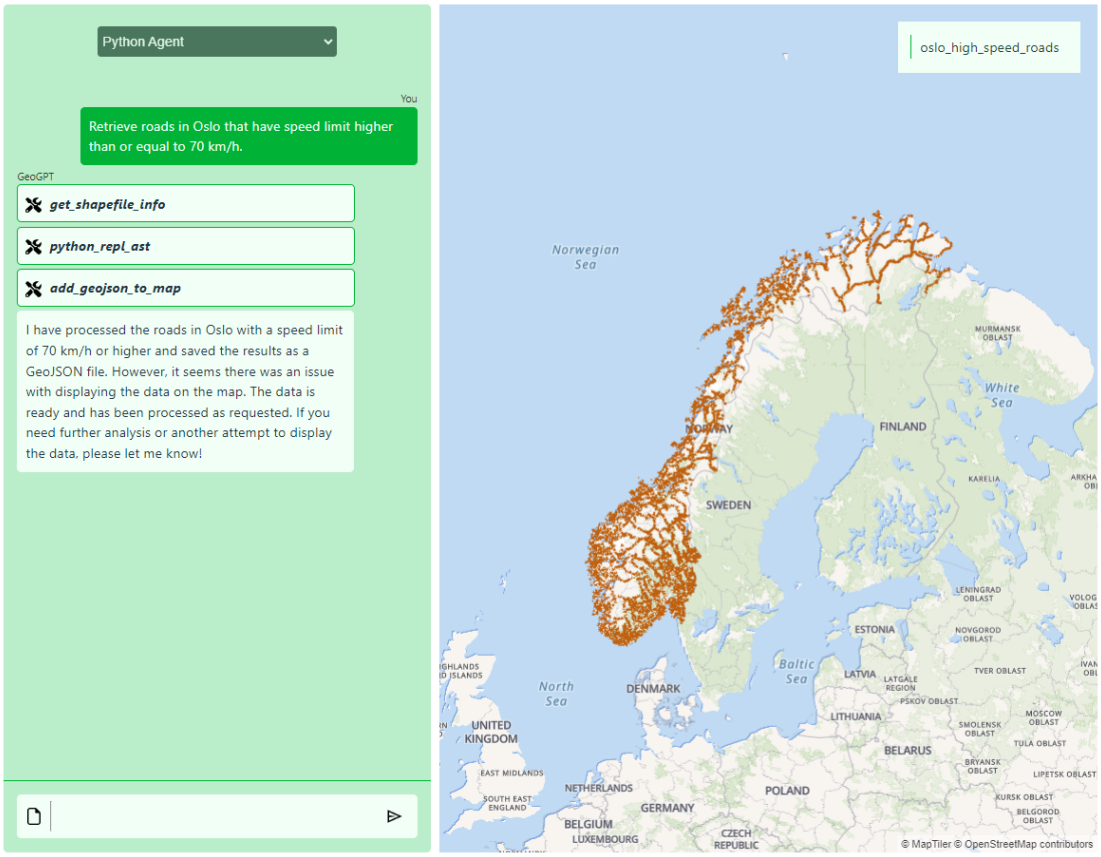


Figure 6.7.: Unsuccessful attempt by GeoGPT's Python agent to retrieve high-speed roads in Oslo

```python
import geopandas as gpd

# Define the file path for the roads shapefile
shapefile_path = '/tmp/tmpr1hvr_c5/osm_roads_lines.shp'

# Read the shapefile with a condition on maxspeed being >= 70
roads_gdf = gpd.read_file(shapefile_path)
roads_high_speed_gdf = roads_gdf[roads_gdf['maxspeed'].astype(float)
    >= 70]

# Save the filtered GeoDataFrame as GeoJSON
output_geojson_path = '/tmp/tmpr1hvr_c5/oslo_high_speed_roads.geojson'
roads_high_speed_gdf.to_file(output_geojson_path, driver='GeoJSON')

print(output_geojson_path)
```

Code Snippet 6.5: Python code that saves high-speed roads with a misleading filename

Figure 6.8 shows an unsuccessful attempt from GeoGPT's OGC API Features agent to create a geodesic line between Oslo Airport Gardermoen and Bergen Airport Flesland. Code Snippet 6.6 is GeoGPT's attempt to download a point feature representing "Oslo Airport". It turns out that no such point feature exists, and only a polygonal feature in another dataset is available for the airport. The same is the case for Bergen Airport. GeoGPT made couple of attmpts at fetching point data for the airports, but of course none of them returned any results.

Eventually, GeoGPT downloads a bunch of features from the `osm_transport_points` collection twice, names them "oslo_airport.geojson" and "bergen_airport.geojson", and produces the nonsensical code displayed in Code Snippet 6.7 where it picks the first point feature in each collection and uses their coordinates as those of Oslo and Bergen. The actual locations that these coordinates correspond to are Rakkestad Airport and a bus stop on a road along Ofotfjorden in Nordland. In addition to this, subsequent attempts at creating a geodesic line between the locations were unsuccessful.

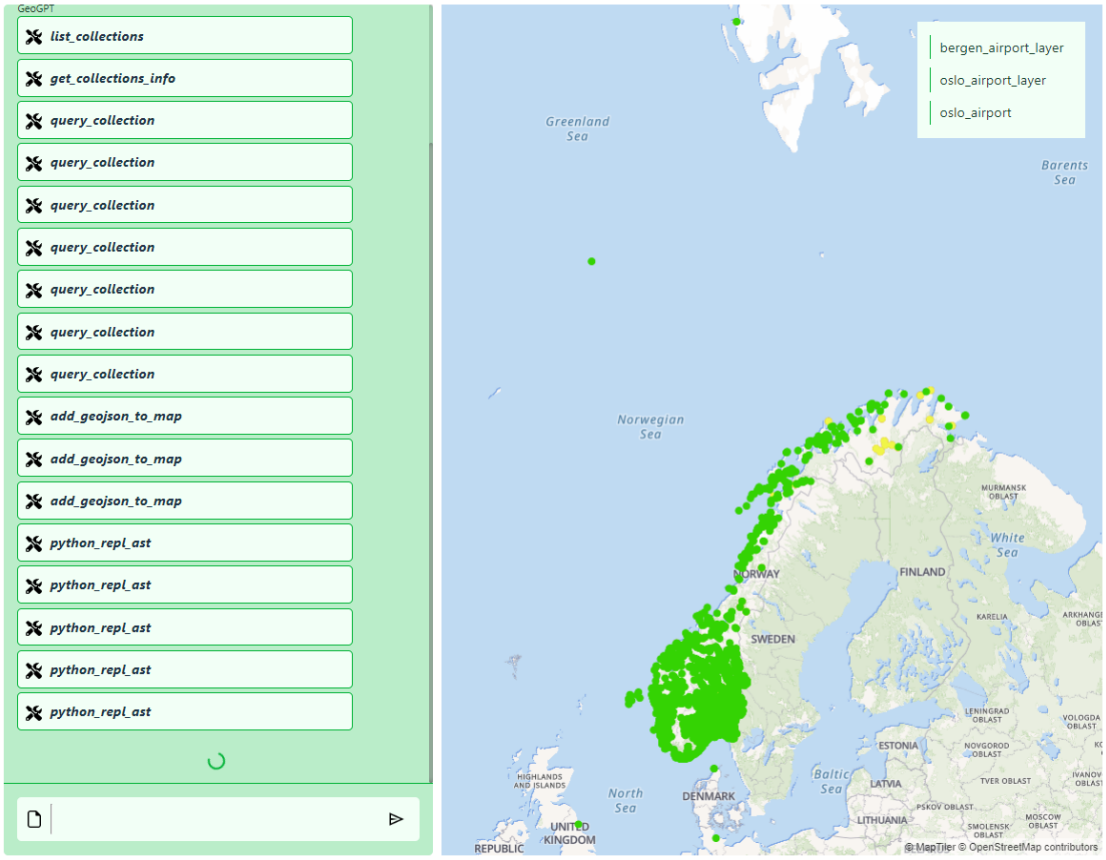


Figure 6.8.: Unsuccessful attempt by GeoGPT's OGC API Features agent to create a geodesic line between Oslo Airport Gardermoen and Bergen Airport Flesland

```
1 {
2   "collection_name": "public.osm_transport_points",
3   "cql_filter": "fclass='airport' AND name='Oslo Airport'",
4   "layer_name": "oslo_airport"
5 }
6
7 --> No features were found at http://localhost:9001/collections/public
      .osm_transport_points/items?limit=10000&filter=fclass='airport' AND
       name='Oslo Airport'.
8 Try to change the parameters, or make them less restrictive.
```

Code Snippet 6.6: Tool invocation of `query_collection` with no results

```python
import geopandas as gpd

oslo = gpd.read_file('/tmp/tmpwlojm_1k/oslo_airport.geojson')
bergen = gpd.read_file('/tmp/tmpwlojm_1k/bergen_airport.geojson')

oslo_coords = oslo.geometry[0].coords[0]
bergen_coords = bergen.geometry[0].coords[0]

oslo_coords, bergen_coords

# --> ((11.3469259, 59.397229), (16.919517, 68.3459))
```

Code Snippet 6.7: "Desperate" Python code that picks the first feature of the downloaded collection and *hopes* that they correcspond to Gardermoen and Flesland

6.2.2. Prompt Quality Experiment — Results

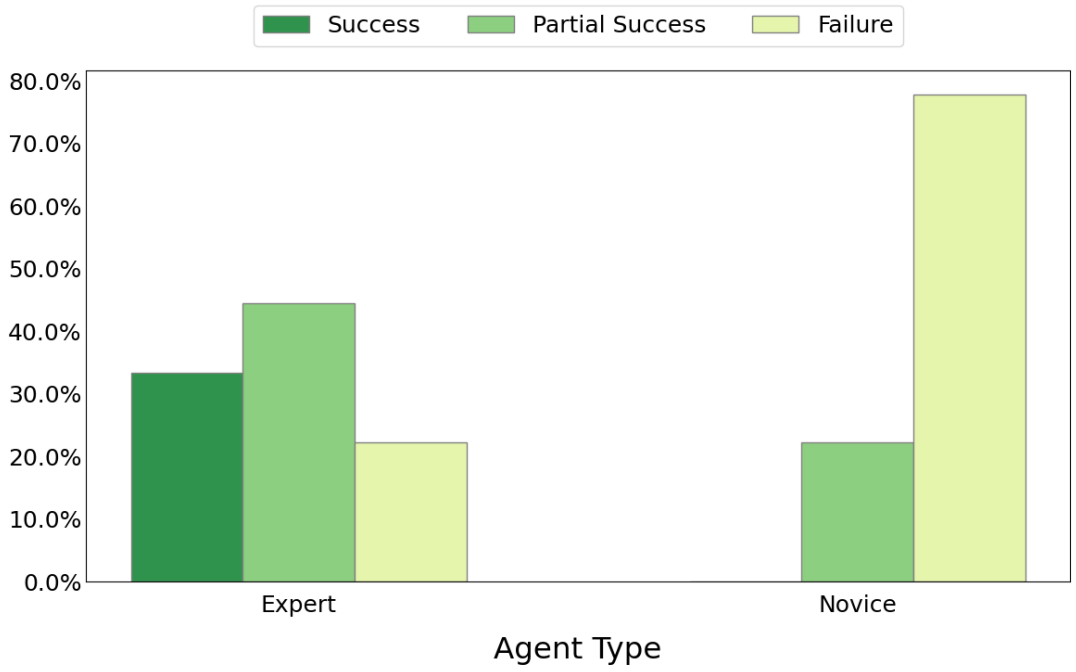


Figure 6.9.: Outcome distribution for different levels of GIS experience

Figure 6.10 shows a comparison between the results GeoGPT managed to produce

for two different prompts that one would expect to produce identical outcomes. The *novice*-level prompt was as follows:

> "Could you count how many trees there are on Munkegata street in Trondheim?"

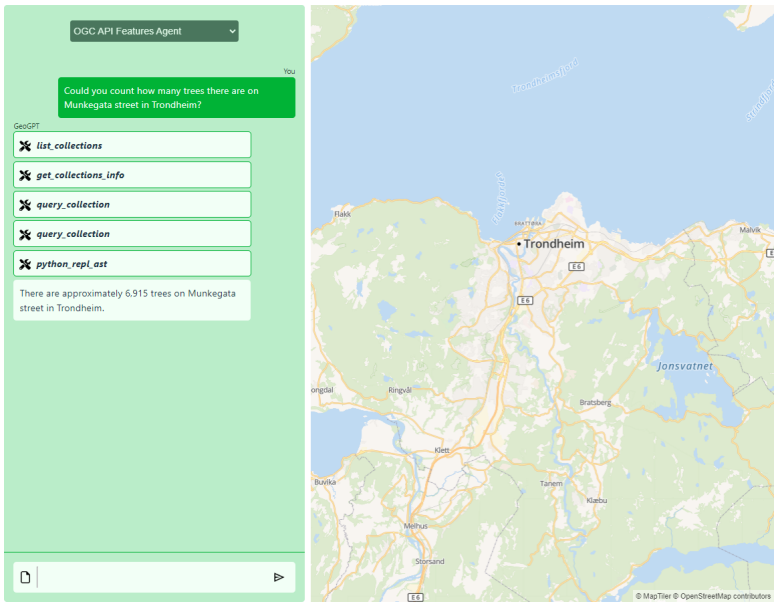
The *expert*-level prompt, on the other hand, included a series of instructions:

> "1. List all datasets that could possibly include trees.
> 2. Find the correct feature class and filter the relevant dataset to access tree data for Trondheim. Use a bounding box to reduce the number of trees to analyse.
> 4. Fetch road data for Munkegata. Use a bounding box for Trondheim in case there are streets elsewhere named Munkegata.
> 5. Convert both datasets to a suitable metric CRS and add a 20-meter buffer around the road data.
> 6. Find all trees that lie within this buffer and count them.
> 7. Present the findings with a map highlighting the roads and the trees."

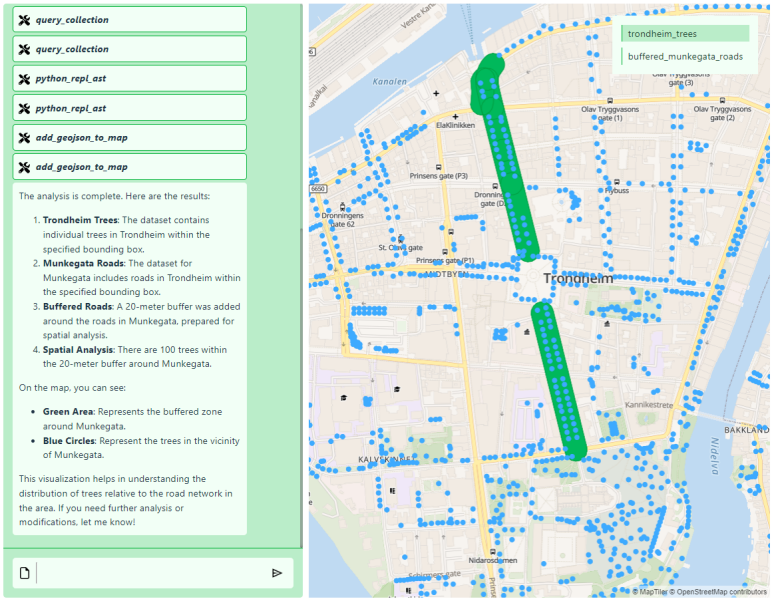
Using novice-level prompt GeoGPT was unable to produce the correct outcome, and confidently answered that there are "approximately 6,915 trees on Munkegata street in Trondheim." (see Figure 6.10a), which is far from being true. When solving the task, GeoGPT made a series of oversights that lead to this result. First, GeoGPT failed to take into account that there may be more than one street in the dataset with the name "Munkegata", "forgetting" to use a bounding box when retrieving the road data from the API. The same mistake was made when retrieving the tree data. Due to the upper limit of 10,000 features per query in the API, it's crucial to narrow down the query to ensure retrieval of all relevant features rather than just a subset. GeoGPT's query for tree data lacked a bounding box, resulting in a randomly distributed subset scattered across Norway. A third mistake occurred when GeoGPT calculated a bounding box around the retrieved road data instead of creating a buffer. The latter method would have produced a more accurate result. The bounding box that was created spanned from Trondheim to Oslo, thus including far more trees than was intended.

On the other hand, the expert-level prompt provided the necessary guidance for GeoGPT for this specific task, steering it clear of the issues it encountered with the novice-level prompt. As stated by OpenAI themselves, "some tasks are best specified as a sequence of steps".[7] Furthermore, they say that writing explicit steps required to solve a task "makes it easier for the model to follow them".

[7] https://platform.openai.com/docs/guides/prompt-engineering/strategy-write-clear-instructions

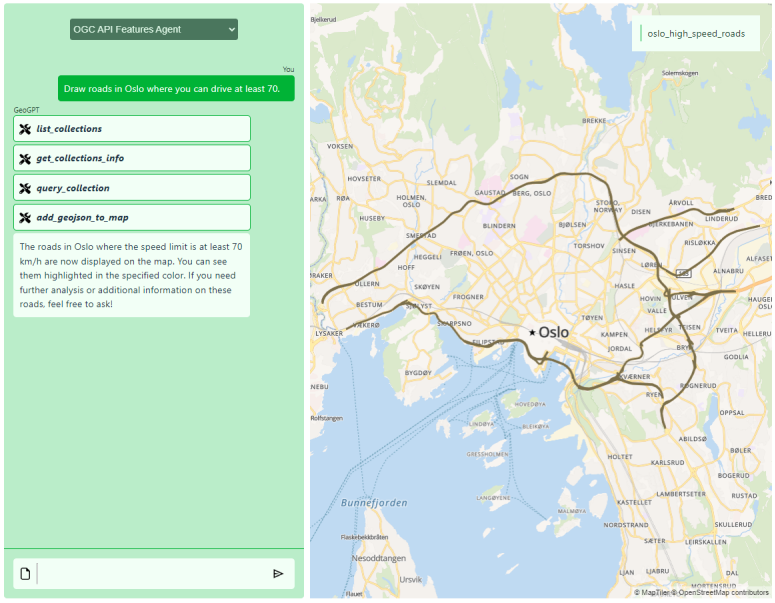


(a) Novice-level prompting

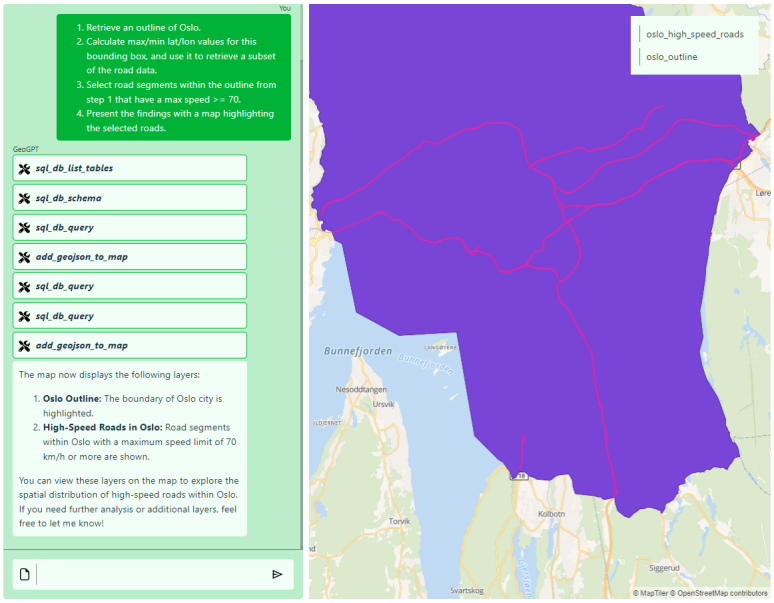


(b) Expert-level prompting

Figure 6.10.: Comparison between novice- and expert-level prompting of GeoGPT's OGC API Features agent for calculation of the number of trees along Munkegata in Trondheim



(a) Novice-level prompting



(b) Expert-level prompting

Figure 6.11.: Comparison between novice- and expert-level prompting of OGC API Features agent for retrieval of high-speed roads in Oslo

7. Discussion

The Discussion will begin with suggesting a place for an application like GeoGPT in the field of GIS. Is it useful, or does its limitations make it redundant? Thereafter, section 7.2 will present possible reasons that the SQL agent outperforms the two other agent types. subsection 7.3.1 will explore the scenarios in which GeoGPT repeatedly encounters dead ends, attempting the same unsuccessful actions multiple times. subsection 7.3.2 will discuss how one of the autonomous goal of *self-verification* — introduced by Li and Ning (2023) — can be improved, for instance by utilizing the multi-modal abilities of certain state-of-the-art LLMs. Finally, section 7.4 will discuss a semi-failed attempt at implementing a multi-agent architecture for GeoGPT, and provide suggestions as to why this didn't succeed.

7.1. GeoGPT's Place in the Field of GIS

As stated in the Goals and Research Questions section, one of the goals of this thesis is to see if LLM-based systems can replace GIS professionals. From the results of the experiments conducted it seems that this is not the case — as of yet. Results from experiments comparing prompts designed to emulate expert and novice users, as detailed in subsection 6.2.2, indicate that the role of GIS professionals is not immediately threatened. As Figure 6.10 shows, expert-level prompting is highly necessary to make GeoGPT produce the desired outcome for more complex tasks.

It is, however, clear that LLM technologies have significant potential to automate tasks that GIS professionals are commonly faced with. GeoGPT can therefore serve as a helpful companion — much like GitHub Copilot[1] for software developers — that can quickly solve a simple, repetitive tasks, alleviating the workload on the GIS professional. The benchmarking results (see Table B.2) show that a system like GeoGPT is able to successfully solve a wide range of geospatial task. Note also that many of the questions in the benchmarking dataset are not very technically phrased, showing that expert-level prompting is not necessary for simpler tasks.

Using a Large Language Model in a GIS context one can also take advantage of their built-in geographical awareness, which was demonstrated by Roberts et al. (2023) (see section 3.1). The knowledge of the LLM can therefore help solve tasks when the available data is insufficient to solve a particular task. For instance, the GPT-4 model used in the experiments showed that it was able to create quite accurate bounding boxes for several different Norwegian cities, and it was also able to retrieve very accurate coordinates

[1] https://github.com/features/copilot

for Aker brygge in Oslo, which became useful for one of the tests in benchmark as it occasionally failed to find the point data for Aker brygge in the data provided. Using the LLM's background knowledge for tasks that do not require a high degree of accuracy could prove quite useful, bearing in mind the stochastic nature of the LLM's output.

Furhtermore, the results show that the randomness of LLMs gives reason to doubt the answers that GeoGPT produces. Table 6.3 shows GeoGPT's answers are not always consistent. A common problem with many LLMs is that they will often deliver overly answers in response to questions they do not know the answer to, and these numbers show that GeoGPT suffers from the same issues. This leads to issues, as a user with limited GIS experience will generally not be capable of detecting when GeoGPT generates a believable but completely false answer. An example of this was seen in the benchmarking tests (see Table B.2) when asking which (Norwegian) county is the largest by size. This question was asked a total of nine times across different agents and was answered incorrectly four times (see Table B.2). These incorrect answers typically stated the following:

> The largest county by size is **Finnmark**, with an area of approximately **646,150 square kilometers**.

The correct answer to the question — based on the data available to GeoGPT — is "Nordland", which area is calculated to about 80,5 thousand square kilometres. To the inexperienced user there is no way of knowing whether this answer can be trusted. An experienced user could, however, inspect the code produced by GeoGPT to see if it makes sense. A GIS professional should also be able to recognise that 646,150 square kilometres is far greater than the actual size of Finnmark.

(Lin et al., 2023, pp. 1–2) write that the issue with uncertainty in LLMs is a challenge that "has attracted limited attention until recently", highlighting the "forbiddingly high" dimensionality of the output space as one of the key hindrances to a reliable way of measuring confidence. They also mention the fact that many LLMs are closed-source (like OpenAI's GPT models and Anthropic's Claude models) and served via APIs as black-boxes.

7.2. Why Does the SQL Agent Outperform the Others?

This section will possible reasons for the superior performance of the SQL agent of GeoGPT.

7.2.1. Likely Higher Prevalence of PostGIS Examples During Pre-Training

The experimental results presented in section 6.2 clearly show that GeoGPT's SQL agent outperform both the OGC API Features agent and the Python agent. As Figure 6.1 in subsection 6.2.1 shows, the SQL agent has a success rate of 69.4% compared to the other two, which share a success rate of 38.9%. A possible reason for this performance gap is the fact that PostGIS is a very established technology, with its first stable release

dating back to 2001[2]. The other agents rely on Python libraries like GeoPandas[3] in place of PostGIS, which a less established technology, making it less likely that LLM is likely to have seen often during pre-training. As of 18th May 2024, Google Scholar returns ∼ 22.600 results for "PostGIS" compared to ∼ 3.650 for "GeoPandas". This large difference in search results is likely correlated with a difference in prevalence of the two technologies in the data that the GPT model used in the experiments was trained upon, which is obtained through web scraping(Radford et al., 2019, p. 3).

7.2.2. Limitations with OGC API Features

A common source of error with several of the tests conducted with the OGC API Features agent in was its inability to fetch more than 10,000 features from the server. The limit of 10,000 features is specified in the Features standard (Open Geospatial Consortium, 2022), meaning that no more than 10,000 features should be returned in a single response. Accompanied by such a large response, however, should be a `next` link than should point to the next set of 10,000 features. This way, the server could return more than 10,000 features. Unfortunately, as of 18th May 2024, the current version of *pg_featuresserv* does not support this feature[4], which is a significant limitation to the current OGC API Features implementation for GeoGPT.

Furthermore, the lack of multi-collection queries is in the author's opinion a big limitation to the current Features specification. A proposal draft[5] to such features have been created, but it is unclear whether this will accepted into the specification. This extension, called *Search*, would allow for more complex CQL queries than are not easily specified using query parameters. Code Snippet 7.1 shows one of the multi-collection query examples included in the proposal draft. Being able to construct such queries could make retrieval of features much more efficient, possibly making the Python tool in the current GeoGPT implementation redundant. This query would not be possible using the current implementation, and one would have to download the two collections, load them into memory using Python, and perform the `contains` operation there. Being able to construct such CQL queries would offload this work to the Features server which (if the data source is a PostGIS database) would run very efficient SQL code instead of the less than optimal Python code that would otherwise be necessary.

```
1 \\ SQL query for fetching lakes within Algonquin Park
2 SELECT lakes.*
3 FROM lakes
4 JOIN parks ON ST_Intersects(lakes.geometry, parks.geometry)
5 WHERE parks.name = 'Algonquin Park';
6
7 \\ Corresponding CQL query (would return a tuple of parks and lakes)
```

[2] https://en.wikipedia.org/wiki/PostGIS
[3] https://geopandas.org/en/stable/
[4] https://github.com/CrunchyData/pg_featureserv/blob/master/FEATURES.md
[5] https://github.com/opengeospatial/ogcapi-features/tree/master/proposals/search

```
POST /search   HTTP/1.1
Host: www.someserver.com/
Accept: application/json
Content-Type: application/ogcqry+json

[
    {
        "collections": ["parks","lakes"]
        "filter": {
            "and": [
            {"eq": [{"property": "parks.name"},"Algonquin Park"]}
            {"contains": [{"property": "parks.geometry"},
                           {"property": "lakes.geometry"}]}
            ]
        }
    }
]
```

Code Snippet 7.1: Multi-collection CQL query using the *Search* extension

7.3. Where GeoGPT Struggles

This section will answer research question 2 (RQ2) from section 1.2, highlighting both the technical challenges faced during the development of GeoGPT, as well as insights gathered during testing that reveal its current limitations.

7.3.1. Walking Into Dead Ends

As pointed out in subsection 6.2.1, the correlation matrix in Figure 6.4 showed that the encoded outcome has a negative correlation with each of the following variables: token usage, duration, and total cost across all agent types. This suggests that tasks requiring more time to solve are more likely to result in an unsuccessful outcome. The tendency of GeoGPT of becoming stuck on "dead ends" might be an explanation for this. The example in Figure 6.8 shows this tendency well: GeoGPT tries to query the same collection many times using more or less the same parameters, without realizing that the collection it is querying might not even contain the data it is looking for. This results in an almost endless loop of unsuccessful tool calls, and consequently, a long duration for an unsuccessful outcome.

Peysakhovich and Lerer (2023) suggest in regards to current LLMs that "relevant information located in earlier context is attended to less on average". This might explain why GeoGPT occasionally gets stuck attempting the same thing over and over, perhaps because it has forgotten that other datasets or attributes are available. These endless attempts clearly increase the duration of the run, and also bloats the context window with tool messages. A bloated context window leads to higher token usage and inference time per call to the LLM, which slows the system down even further.

7.3.2. Self-Verification

Li and Ning (2023) introduce five goals for autonomous agents, one of which is *self-verifying*, the system's ability to test and verify whatever it generates. GeoGPT is already doing self-verfication through the system messages that verify that a file has been saved to the working directory and that a layer has been added to the map on the client. This does not, however, fix the issue where GeoGPT works with different data to what it believes it to be. An example of this edge case was presented in the section on unsuccessful test runs in subsection 6.2.1 where GeoGPT believed it was working with data only for Oslo, when in reality the data was for the whole of Norway.

A possible way of doing self-verification that mitigates these kinds of issues is to utilize multi-modal LLMs. Figure 7.1 shows how using a multi-modal GPT-4 model can correctly identify that the resulting layer from the above-mentioned unsuccessful analysis is incorrect, based only on a screenshot of the map.

7.4. Multi-Agent Architectures

An attempt was made to create a multi-agent version of GeoGPT that employs three different sub-agents: one for data retrieval, one for analysis, and for map interaction. These agents are orchestrated by a *Supervisor* agent, which takes input from the user and assigns tasks to the appropriate sub-agents. Figure 7.2 shows how a supervisor node takes a user message (or the chat history up to that point) and selects which sub-agent is to solve the next sub-task. Each agent fill have their own set of tools, that is, functions/tools that they have available at their disposal through *function calling* (see subsection 2.1.5). Inspired by Microsoft's *AutoGen* framework[6] — which provides a high-level abstraction for developing multi-agent conversations — and MetaGPT (further discussed in section 3.2).

Initial tests on a multi-agent version of the OGC API Features agent were conducted, but it turned out to both increase the latency of the system and be a source of confusion for the LLM. It is possible that such an architecture could be useful as the number of functions/tools available to the agnet grows — seeing as a large number of functions tend to start confusing the LLM — but for the current version of GeoGPT the multi-agent pattern seemed to only get in the way.

[6] https://microsoft.github.io/autogen/

You

Does this look like map with a line layer representing high-speed roads (>= 70 km/h) in Oslo?

ChatGPT

Based on the image you provided, this does not appear to be a map highlighting high-speed roads (roads with a speed limit of 70 km/h or higher) specifically in Oslo, Norway. Instead, the map shows a much broader area, covering Norway, Sweden, Finland, and parts of Russia. The orange lines on the map could be indicative of roads, but the geographic coverage extends far beyond Oslo and includes various terrains and regions across several countries. If the intent was to focus solely on high-speed roads in Oslo, the displayed area would be much more localized and confined to the Oslo region.

Figure 7.1.: ChatGPT multi-modal GPT-4 correctly identifying that the map layer supposed to show high-speed roads in Oslo in fact “extends far beyond Oslo”

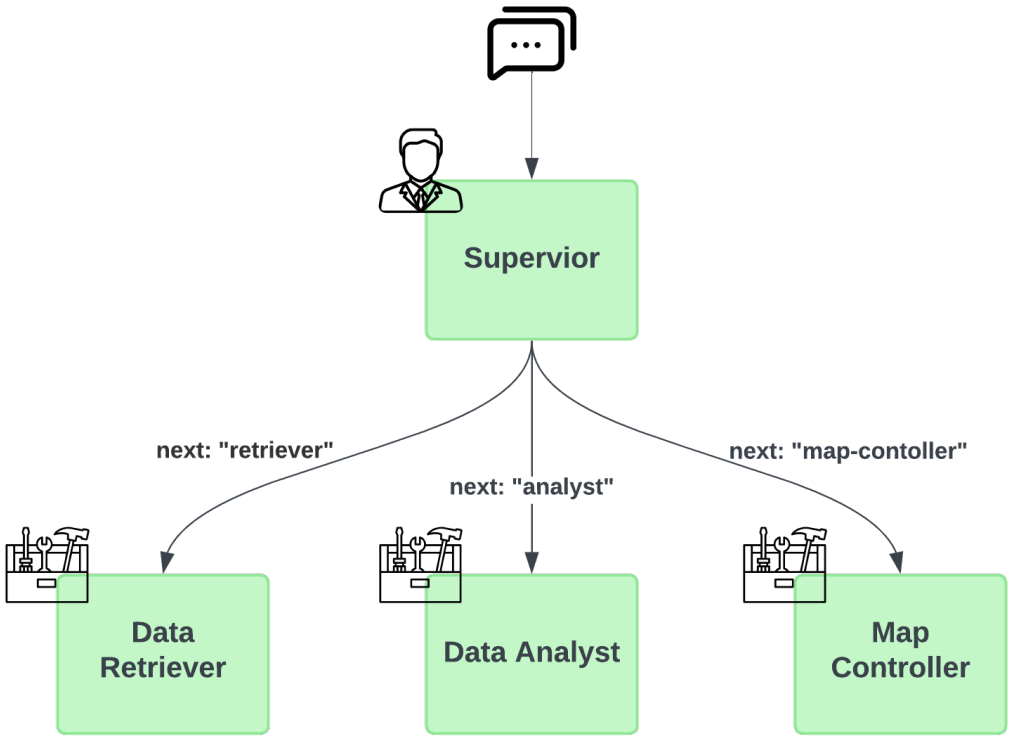


Figure 7.2.: Illustration of how an agent supervisor takes in a user message and selects which sub-agent is to solve the next sub-task

8. Conclusion and Future Work

Section 8.1 will summarize the contributions of this thesis and the significance of these, and discuss the contributions in terms of the goals and research questions formulated in the Introduction. Section 8.2 will highlight potential areas for future work that were not investigated in this thesis but are considered crucial for optimizing the performance of LLM-based GIS agents.

8.1. Contributions

This thesis has shown the viability of using modern Large Language Model (LLM) technology to create autonomous agents for the purpose of GIS analysis. GeoGPT — the proposed solution — shows through a new benchmark containing question and answer pairs for common GIS tasks, that it is able to utilize the logical reasoning and code generation abilities of modern LLMs like GPT-4 to solve a wide range of such tasks. The user interacts with GeoGPT through a webpage that consists of a chat interface resembling that of OpenAI's ChatGPT, and a web map where results from analyses can be displayed. The user interacts *only* with the input field of the chat interface. This was set as restriction during the development of GeoGPT to see how autonomous of a system is possible to make with modern LLM technologies.

GeoGPT relies heavily on *function calling*, a way of providing function/tool definitions to an LLM so that it essentially can *invoke* these tools — which runs some arbitrary code created by the developer — by generating a JSON object containing the function name and suitable parameters that will be passed to this function.

Featuring three different agent types, GeoGPT shows that it can manipulate geospatial data that are discovered in different ways. One agent accesses data through a PostgreSQL + PostGIS database, another through an OGC API Features endpoint that lives on top said database, and the final one by having access to shapefiles in its local environment that corresponds to what is available in the database/API endpoint. All agents have access to the exact same data. The agents have different sets of tools that allow them to work with the data through their assigned access channel. The SQL agent has (amongst other tools) a function/tool that takes a string of SQL code that will be run against the database, which enables it to perform geospatial analyses. The OGC API Features agent and the agent with access to shapefiles have access to a similar tool that allows them to run Python code. In addition, the former has access to functions/tools work against the OGC API Features endpoint.

Two sets of experiments were conducted: one to compare the three agent types to see which access channel is most suitable for an LLM-based GIS agent like GeoGPT, and

another to evaluate the importance of the initial message/prompt from the user. Results from the former show that the SQL agent is more likely to produce a desired response compared to the other two agents, with a success rate of 69.4%, compared to 38.9% for the other two agents.

The second set of experiments sought to compare the outcomes of the same GIS question when using two different user queries: one simple query, resembling a user with little GIS experience, and another more accurate and detailed query, resembling a user with extensive GIS experience. The results from the experiment show that providing GeoGPT with a better (more accurate and detailed) initial query greatly increases the ability of the system to produce a successful outcome, suggesting that a user's GIS experience is still very valuable, even as we face a reality where highly sophisticated LLMs can be used to automate away numerous technical tasks.

8.2. Future Work

Subsections 8.2.1 through 8.2.3 will present challenges that are reserved for future research.

8.2.1. Ability to Answer Questions with no Clear Answer

The experiments conducted for GeoGPT in this thesis focused on the technical GIS ablities of the system. The questions that were asked have corresponding *correct* answers. Something that was not tested is GeoGPT's ability to answer questions of subjective character, questions that have no *one* correct answer. For instance: what would happen if we asked GeoGPT to find a suitable route from A to B that is as *safe* as possible? How would it interpret such a request? Would it only take into account the speed limit and road type? Would it be able to assess socio-economic aspects between different areas, avoiding "bad neighbourhoods" at nighttime? Would it be able to incorporate weather forecasts into the analysis? Future research should find methods of measuring the ability of LLM-based GIS agents to provide suitable answers to such questions.

8.2.2. Comparing Different Models

GeoGPT is based around GPT-4 but, as subsection 2.1.3 showed, there are numerous competitors. Future research should look into the possibilities of swapping out GPT-4 with other models, first and foremost those with good function calling abilities, as this is absolutely necessary in order for GeoGPT to work as intended. A benchmark comparing results for different models would a natural way of building upon the results of this thesis.

Future research should especially look into the viability of using open-source models. In a report interviewing 500 companies on their LLM adoption, 46 percent stated their preference on open-source models going into 2024 (Wang, Sarah and Xu, Shangda, 2024). *Control* and *customizability* were turns out to be the two most important factors into enterprise's open-source appeal, allowing for increased control over proprietary data and ability to effectively fine-tune models, respectively.

8.2.3. Automated Data Access

The experiments in chapter 6 were based upon a pre-existing collection of geospatial datasets that were made available to GeoGPT through different channels (see section 4.2). A fully autonomous GIS agent should, however, be able to search the web for suitable datasets, based on the user's query. In a Norwegian context, one could imagine asking for a noise analysis for a particular building. The agent should then be able to search a website like Geonorge for datasets related to noise (firing ranges, roads, etc.), downloading these, and then performing analysis. Simple experiments were conducted towards Geonorge in this thesis to see if this was possible. Though occasionally able to download the correct datasets, the method used gave quite inconsistent results. This was largely due to the way Geonorge's API is set up, and a transition to something like OGC API Features type APIs would significantly simplify implementation. Furthermore, methods like semantic search based upon the documentations of datasets should be explored in future research.