Karl Oskar Magnus Holm

GeoGPT: An LLM-Based Geographic Information System

Using Large Language Models to Create a Natural Language Interface for GIS

Master of Science in Engineering and ICT, June 2024
Supervisor: Hongchao Fan, NTNU
Co-supervisors: Alexander Salveson Nossum and Arild Nomeland, Norkart AS

Department of Civil and Environmental Engineering
Norwegian University of Science and Technology

Oppgåve med omfang som kan tilpassast både prosjekt og masteroppgåve

LLMs - GIS-analysens død

(kan justerast seinare)

BAKGRUNN

Nyere modeller for kunstig intelligens har demonstrert spesielt gode evner til å kunne lære av store mengder ustrukturert og semi-strukturert informasjon. ChatGPT fra OpenAi tok verden med storm – og chat-baserte systemer florerer. Kan chat-baserte modeller skapes for å hente ut GIS-data effektivt? Norkart har en stor dataplattform hvor brukere utvikler mot API'er som i stor grad har GIS/Geografiske data i bunn. GeoNorge er en stor datakatalog hvor brukere slår opp, eller søker kategorisert for å finne data. QGIS, Python, PostGIS, FME og andre verktøy brukes ofte til å gjennomføre GIS-analyser – hvor en GIS-analytiker/data-scientist gjennomfører dette.

«*Finn alle bygninger innenfor 100-meters-belte som er over 100 kvm og har brygger*»

Er dette mulig å få til med dagens tilgjengelige chat-modeller?

OPPGAVEBESKRIVELSE

Oppgaven har som hovedmål å undersøke hvordan nyere språkmodeller kan benyttes for å gjennomføre klassiske GIS-analyser ved å bruke standard GIS-teknologi som PostGIS/SQL og datakataloger (OGC API Records fks). Hva finnes av tilgjengelig chat-løsninger? Hvordan spesialtilpasse til GIS-anvendelser? Hvor presise kan en GIS-Chat bli?

Relevante delmål for oppgaven:

1. Kartlegge state-of-the-art
2. Utvikle proof-of-concepts
3. Analysere begrensninger og kvalitet

Oppgaven vil med fordel deles i prosjektoppgave og masteroppgave

- Prosjektoppgave
    - State-of-the-art: Ai-modeller og multi-modal maskinlæring
    - Innhente og utvikle datagrunnlag og API-tilgjengelighet
- Masteroppgave
    - Utvikle proof-of-concepts med tilgjengelige åpne modeller/teknologi
    - Gjennomføre eksperimenter for analyse av kvalitet

Detaljert oppgavebeskrivelse utvikles i samarbeid med studenten.

ADMINISTRATIVT/VEILEDNING

Ekstern veileder: (en eller flere)
Mathilde Ørstavik, Norkart
Rune Aasgaard, Norkart
Alexander Nossum, Norkart

Aktuelle vegleiarar og ansvarleg professor ve NTNU (den som har fagansvar nærast oppgåva):

Terje Midtbø (GIS, kartografi, visualisering)
Hongchao Fan (3D modellering, fotogrammetri, laser)

Abstract

Recent technological developments in Natural Language Processing have led to the emergence of powerful Large Language Models (LLMs) like those powering ChatGPT — an AI-based chat interface created by OpenAI. Such *generative* LLMs have shown to be versatile, and the overarching goal for this master's thesis is to utilize the logical reasoning and code generation abilities of modern LLMs to develop a chat-based GIS that can solve GIS tasks based only on the user's natural language problem formulation. The application, named *GeoGPT*, features three different agent types which, as demonstrated through experiments, can perform common GIS analyses on OpenStreetMap (OSM) data with little to no help from the user. GeoGPT relies on the LLM's *function calling* abilities, which effectively give its agents the ability to use external tools. The agents have different sets of tools and also differ in the way they access the OSM data. One agent accesses the data through a PostGIS database, another through an OGC API Features server to download GeoJSON over HTTP, and the third has direct access to shapefiles stored locally in GeoGPT's environment. Experimental results obtained using a new GIS benchmark show that the SQL/PostGIS agent performs best out of the three, getting the most tasks correct while also being the fastest and cheapest. Results from an experiment conducted to evaluate the importance of the quality of the initial prompt from the user show that a more detailed step-by-step prompt significantly improves GeoGPT's chances of producing a successful answer. Overall, the work done in this thesis shows that an LLM-based GIS like GeoGPT can solve a variety of common GIS tasks based on simple natural language prompts, but that the user's GIS expertise becomes increasingly important as tasks become more difficult.

Sammendrag

Nyvinninger innenfor språkprosessering har ført til fremveksten av meget kraftige og store språkmodeller, for eksempel modellene som ligger bak ChatGPT — en AI-basert chat-applikasjon utviklet av OpenAI. Slike *generative* språkmodeller har vist seg å være allsidige, og denne masteroppgaven vil utnytte språkmodellenes evner til logisk resonnering og kodegenerering, til å utvikle en GIS-applikasjon som kan løse GIS-oppgaver basert kun på en brukers tekstlige problemformulering. Applikasjonen har fått navnet *GeoGPT*, og tilbyr tre forskjellige agenttyper som gjennom eksperimenter har vist at de kan utføre en rekke ulike GIS-analyser på OpenStreetMap (OSM)-data, uten nevneverdig hjelp fra brukeren. GeoGPT utnytter *function calling*, som i prinsippet gir agentene muligheten til å ta i bruk eksterne verktøy. Agentene har blitt utdelt forskjellige verktøy, og har også forskjellige måter å aksessere OSM-dataene på. Den ene agenten bruker en PostGIS-database, en annen bruker en OGC API Features-server for å laste ned GeoJSON over HTTP, og den tredje har direkte tilgang til shapefiler lagret lokalt i miljøet som GeoGPT kjører i. Eksperimentelle resultater, basert på en ny GIS-benchmark, viser at SQL/PostGIS-agenten gir korrekt svar på flest oppgaver, i tillegg til å være raskest og billigst. Resultater fra et eksperiment utført for å evaluere betydningen av kvaliteten til problemformuleringen til brukeren, viser at det å gi GeoGPT sekvensielle trinn for å løse problemet, forbedrer sjansen betraktelig for at den produserer riktig svar. Samlet sett viser arbeidet i denne masteroppgaven at et språkmodellbasert GIS, slik som GeoGPT, kan løse mange vanlige GIS-oppgaver basert kun på spørringer formulert ved naturlig språk, men også at GIS-ekspertise blir viktigere og viktigere etter hvert som oppgavene blir mer utfordrende.

forslag til alternative formuleringer tas imot med åpne armer

Preface

This master's thesis is written at the Norwegian University of Science and Technology, for the Division of Geomatics at the Department of Civil and Environmental Engineering.

I would like to thank Hongchao Fan, my supervisor at the Division of Geomatics, and my two supervisors from Norkart: Alexander Salveson Nossum and Arild Nomeland. I would also like to thank GeoForum and Digin for inviting me to present my thesis at Geomatikkdagene and GeoAI:Konferansen, respectively. Finally, I would like to thank my ever-supporting family for their constant encouragement and support throughout my studies.

Karl Oskar Magnus Holm
Trondheim, 31st May 2024

Contents

Acronyms

AI Artificial Intelligence.

API Application Programming Interface.

BFCL Berkeley Function-Calling Leaderboard.

CQL Common Query Language.

CRS Coordinate Reference System.

DAG Directed Acyclic Graph.

DBMS DataBase Management System.

GIS Geographic Information System.

GML Geography Markup Language.

GPT Generative Pre-trained Transformer.

GQA Grouped-Query Attention.

HTML HyperText Markup Language.

HTTP Hypertext Transfer Protocol.

JSON JavaScript Object Notation.

Llama Large Language Model Meta AI.

LLM Large Language Model.

MBPP Mostly Basic Python Programming.

NLP Natural Language Processing.

OGC Open Geospatial Consortium.

OSM OpenStreetMap.

PoSE Positional Skip-wisE.

PPO Proximal Policy Optimization.

RAG Retrieval Augmented Generation.

RAM Random Access Memory.

REPL Read-Eval-Print Loop.

RLHF Reinforcement Learning from Human Feedback.

RNN Recurrent Neural Network.

RoPE Rotary Position Embedding.

SMoE Sparse Mixture-of-Experts.

SQL Structured Query Language.

SSD Solid-State-Disk.

STAC SpatioTemporal Asset Catalog.

SWA Sliding Window Attention.

UI User Interface.

List of Figures

List of Tables

List of Code Snippets

1. Introduction

The introductory chapter open with explaining the motivation behind the thesis, its goals, and the research questions it will attempt to answer in sections section 1.1 through section 1.2. Following this, section 1.3 will explain the research method of this master's thesis, which is to be considered as a technical thesis. Section 1.4 will list the main contributions of the thesis, before section 1.5 gives the reader a high-level overview of the thesis to conclude the Introduction chapter.

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 now flourishing. Furthermore, significant advancements have been made within code generation, which makes LLMs useful even for technical tasks, enabling individuals with little to no prior programming experience to carry out computational tasks that require the code execution.

Geographic Information System (GIS) analysis has traditionally been reserved for GIS *experts*. Furthermore, GIS professional are commonly required to know their way around one or more GISs, and to be 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 find 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 Large Language Models (LLMs).

1.2. Goals and Research Questions

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

Based on the 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 in developing LLM-based GISs?
3. Can an autonomous LLM-based GIS agent replace GIS professionals?

1.3. Research Method

This master's thesis will be of a technical character, and will revolve around the development of a "proof of concept". The usefulness of the "proof of concept" will be evaluated through a series of tests, that will help answer the research questions listed in the previous section. This report will emphasize the contributions that the "proof of concept" and the experiments bring, seeing as these occupied a significant majority of the time allocated to the master's thesis.

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 tasks commonly solved using GIS software.
2. A new benchmark that will give insigth into the ability of a system like GeoGPT to solve common GIS tasks when provided only with a natural language problem formulation.
3. An investigation into the open question on the extent to which GeoGPT can replace GIS professionals.

1.5. Thesis Structure

Below is an outline of the thesis' structure:

- Chapter 2 introduces the theory and tools necessary for the reader to be familiar with in order to understand the rest of the work. It also gives insight into the work that has been done in regard to autonomous, LLM-based systems, in particular those within the field of geomatics.
- Chapter 3 will lay out the GeoGPT's architecture, providing both a high-level overview and details on important parts of the system.
- Chapter 4 presents the experimental setup, the datasets utilized, and the results and evaluations obtained from these experiments.
- Chapter 5 will elaborate upon points of discussion that arise from the experimental results.

- Chapter 6 will suggest areas of improvement that are suitable for future research.
- Chapter 7 will conclude the master's thesis, reiterating the main contributions of the work.

2. Background Knowledge and Related Work

NB! Parts of the Background Knowledge and Related Work chapter is reused material from the specialization project (Holm, 2023) that preceded this master's thesis. Below are the sections in question:

- *Subsection 2.1.2: Reused, with minor adjustments.*
- *Subsection 2.1.3: GPT part with minor adjustments.*
- *Subsection 2.4.1: Reused, with minor adjustments.*

Chapter 2 will lay a theoretical basis for the work done in this master thesis, providing the user with the necessary knowledge to understand the contributions of the work. It will also provide an overview of previous research that shares objectives similar to those of this thesis. Section 2.1 will focus on Large Language Model (LLM). First, subsection 2.1.1 will introduce the terms "tokens" and "context window", which are terms used frequently throughout this thesis. Thereafter, subsection 2.1.2 will present the component that most modern LLMs are based upon — namely the Transformer — and the attention mechanism that drives it. Continuing, subsection 2.1.3 section will present some of the leading LLMs as of 31st May 2024, both proprietary and open-source ones. Subsection 2.1.4 will present the concept of *prompt engineering*, which is a structured way of composing an input to the LLM, before subsection 2.1.5 concludes section 2.1 by presenting *function calling*, a way of allowing LLMs to use external tools. Section 2.2 will give a brief intro to LangChain, a Python/JavaScript library that is used extensively throughout the code base of GeoGPT. Section 2.3 will introduce the reader to PostGIS, OGC API Features, and STAC APIs. The first two are geospatial technologies that are used within GeoGPT. Section 2.4 will present that relates in some way to this thesis. Subsection 2.4.1 present work where LLM are applied for geospatial purposes, while subsection 2.4.2 describes different agent patterns that can be applied to LLM-based agents to improve their performance.

2.1. Large Language Models

This section will lay the theoretical groundwork required to gain an overview of the inner workings of LLMs. LLMs are a type of neural networks that excel at language processing.

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 generate some output sequence based upon some input sequence, and are the types of models behind technologies like OpenAI's *ChatGPT* (OpenAI, 2022). This input sequence is also known as the "context window", and all sequences are made up from "tokens", as subsection 2.1.1 will explain.

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 OpenAI's GPT-3.5 and GPT-4 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. This way, the model only has to learn the meaning of "revolution", and can append different suffixes to modify its function within a sentence, as opposed to learning an entirely new token for each version of the word.

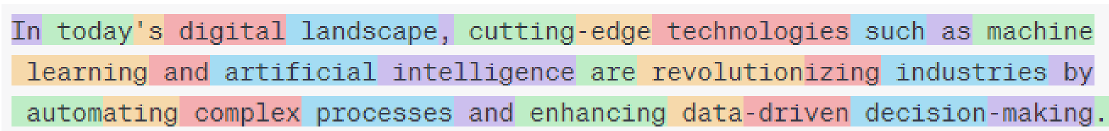


Figure 2.1.: Tokenization example for an English sentence

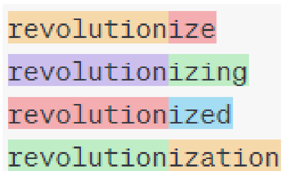


Figure 2.2.: Tokenization of the word "revolution", with different suffixes

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 the tokens 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 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, for instance, Meta AI's Llama models.

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 Transformer 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 for the model 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. The attention mechanism is improved by multiplying queries, keys, and values with *learned* weight matrices that are obtained 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). Below is an example using translation from Norwegian to German, showing how the attention blocks are used:

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 most modern LLMs.

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 exact encoder-decoder architecture. The evolution following the Transformer has favoured generative decoder-only models, focusing entirely on the generative component of the Transformer. The goal is to create models that can produce coherent and context-aware text.

The GPT Family

Generative Pre-trained Transformer (GPT) is an 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, simply attempting to elaborate upon the input sequence it is given, to the best of its abilities. 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 also how other tasks require the conversion of structured inputs into ordered sequences, as is the case when fine-tuning for tasks like multiple choice (see Figure 2.3). This *inlining* is required 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) (Christiano et al., 2023) 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) (Schulman et al., 2017), 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, which are all proprietary|: the GPT-3.5 Turbo model, which is fast and inexpensive; the GPT-4 Turbo model, which as of 31st May 2024 is described by OpenAI as their "previous high-intelligence model"; and

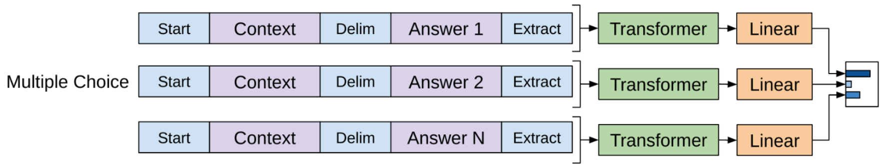


Figure 2.3.: A subsection of a figure illustrating the pre-training process for GPT models, showing how multiple choice tasks need to be *inlined* to work with fine-tuning of GPT models, which are trained on ordered sequences (Radford et al., 2018, p. 4)

.

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, includes three different models: Ultra, Pro, and Nano. These are listed in descending order in terms of size (number of model parameters). The models of the Gemini 1.0 suite are multi-modal, supporting text, image, audio, and video. Gemini 1.0 Ultra 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 latest 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 Claude 2.1's context window of 200k (see the next paragraph for more on the Claude models) 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 limited to 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 suite of proprietary LLM. Anthropic is one of the actors in the LLM market that has helped push in the direction of long-context LLMs, with their Claude 2 model (released in November 2023) being the first to support up to 200k tokens (Anthropic, 2023, p. 9). Latest in line is Claude 3, a family of language models that —- much like the Gemini family — features three models of different sizes:

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

Opus, Sonnet, and Haiku. Again, these are listed in descending order in terms of number of parameters. Opus offers the most advanced capabilities, and outperforms GPT-4 and Gemini 1.0 Ultra on most benchmarks (Anthropic, 2024, p. 6). Haiku is Anthropic's fast and economical option, while Sonnet serves as a balance between Opus' complexity and Haiku's speed.

Open-Source Alternatives

As mentioned, OpenAI's GPT models, Google's Gemini models, and Anthropic's Claude models are all proprietary. This prevents developers from downloading these models and making improvements and customizations to them through fine-tuning. This, and several other reasons, have led to the emergence of a number of *open-source* LLMs.

The **Llama** family of LLMs, developed at Meta AI, is perhaps the most famous open-source option to the proprietary LLMs. At the time of writing, the last in line is the Llama 3 model (Meta AI, 2024), which comes in two sizes: 8B and 70B parameters. Both models display state-of-the-art performance on most major benchmarks compared to comparable open-source alternatives, and the 70B model even surpasses proprietary 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 2 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 the same architecture as Mistral 7B, except that each layer of the model is composed of 8 feed-forward blocks. Using a router at each layer, Mixtral 8x7B 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 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 also 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 generally consists of a series of messages. 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.

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

- **User**: A message from the human user that the assistant should respond to.
- **Assistant**: A message generated by 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.

Patterns that turned out useful to GeoGPT — the main contribution of this thesis — 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 for the LLM.

2.1.5. Function Calling LLMs

Function calling — also known as *tool calling* — was first introduced by OpenAI in April 2023 (Eleti et al., 2023). Function calling allows developers to provide function definitions to an LLM and have the LLM output a JSON object containing the name of one or more of the functions provided, as well as suitable parameters to these functions. Code Snippet 2.1 shows a description of a function that performs a mathematical division. It is specified by its name, description, and its two parameters: the dividend and the divisor, both of which are required. Function calling LLMs are able to detect when a certain function should be called based on the name, description, and input 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": "divide",
    "description": "Performs a mathematical division.",
    "parameters": {
      "type": "object",
      "properties": {
        "dividend": {
          "description": "The number to be divided (numerator).",
          "type": "number"
        },
        "divisor": {
          "description": "The number by which the dividend is divided (denominator).",
          "type": "number"
        }
      },
      "required": ["dividend", "divisor"]
    }
  }
}
```

Code Snippet 2.1: Example of a function definition for a tool that performs divison. It is specified by its name, description, and its two parameters: the dividend and the divisor, both of which are required.

Function calling can be used to 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 hard to do with LLMs. Another use case is code execution. Take this rather simple function signature:

```
execute_python_code(code: string) -> string
```

Such a function could take some Python code as a string parameter and return the standard output that results from executing that code. This is the principle behind what was previously known as ChatGPT's "Code Interpreter" mode, where ChatGPT serves 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. Furthermore, as Eleti et al. (2023) describes, function calling can also be used to extract structured data from text.

An initiative amongst researchers at Berkeley (Yan, Fanjia et al., 2024) 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 to help accommodate function calling abilities. At the time of writing (31st 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. As the name suggests, LangChain revolves around *chains*. Chains are Directed Acyclic Graphs (DAGs), which can be constructed from so-called *runnables*. The "Runnable" protocol is a standard interface, which makes enforces chains to implement `stream`, `invoke`, and `batch` methods, as well as their asynchronous counterparts. LangChain is supported by a large community of developers, and has components that simplify interaction with LLMs, like prompt templates, output parsers, toolkits for function calling purposes, and off-the-shelf agents.

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 framework called LangGraph, which is built on top of the LangChain ecosystem. While *chains* are well-suited for DAG workflows, they are not easily adapted to cyclic graphs. LangGraph, however, is specifically designed to simplify development of cyclic graphs for LLM applications, which are commonly used to create agent-like behaviours (LangChain AI, 2024). A LangGraph *graph* is a set of nodes that pass around and modify a state dictionary. The nodes are connected by edges that define what node can succeed another node. Edges can be conditional, meaning the state produced from one node decides which edge is followed. This allows for complex logic and simplifies implementation of advanced agent patterns, some of which are discussed in subsection 2.4.2.

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

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. The PostGIS extension 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. Geospatial types can be stored along with a spatial index which can significantly reduce search time for their 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 are used to analyse and process geospatial data. These function's names are prefixed with `ST_` (short for "spatio-temporal"), and some examples are `ST_DWITHIN`, `ST_BUFFER`, and `ST_TRANSFORM`. Code Snippet 2.2 shows a typical query for retrieving building outlines within a bounding box.

```sql
SELECT *
FROM osm_buildings_polygons
WHERE type = 'house'
  AND ST_Intersects(geom, ST_MakeEnvelope(min_lon, min_lat, max_lon,
   max_lat, 4326));
```

Code Snippet 2.2: PostGIS example code for retrieving outlines for buildings of type 'house' within a bounding box specified by coordinates in the WGS 84 map projection

2.3.2. OGC API Features

OGC API Features is an API specification that defines modular API building blocks for interacting with features, which are real-world objects (Open Geospatial Consortium, 2022). This includes 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.

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

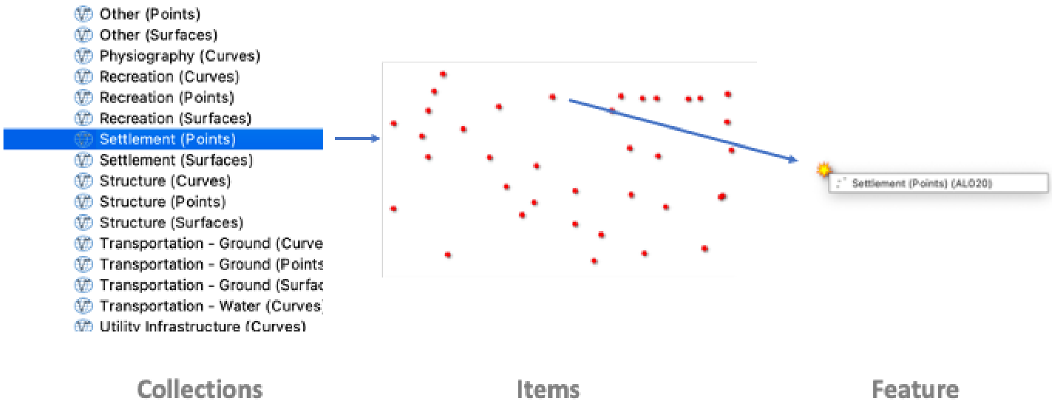


Figure 2.4.: An illustration of the structure of a typical OGC API Features server. The server would have a number of collections, which in turn have a number of items/features with a geographical component. The consumer of the API can request a collection of features (items), or individual features. The figure was 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 five parts that are listed on OGC's websites:[6]

- Features - Part 1: Core[7]
- Features - Part 2: Coordinate Reference Systems by Reference [8]
- 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 for 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 subsets of a given collection. Below are two examples of CQL queries:

[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
[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

```
\\ Example 1
county in ('Akershus', 'Buskerud', 'Ostfold')

\\ Example 2
DWITHIN(the\_geom, Point(63.4265, 10.3960), 1, kilometers)
```

Code Snippet 2.3: Examples of two simple CQL filters: Example 1 filters in all features whose `county` attribute is one of 'Akershus', 'Buskerud', 'Ostfold'; Example 2 checks whether a feature's geometry is within 1 kilometer of the point specified by the latitude and longitude 63.4265, 10.3960.

Part 4 defines how an API that implements the specification should handle addition, replacement, modification, and removal of a collection's features. Part 5 describes how features can be described by a logical schema and how these schemas are published. In addition to parts 1 through 5, several other *proposed* extensions have emerged, such as the *Search* extension, which would allow for multi-collection queries, or the *Geometry Simplification* extension which proposes the use of simplification algorithms for retrieving simplified versions of a collection.[12]

2.3.3. SpatioTemporal Asset Catalog

The SpatioTemporal Asset Catalog (STAC) specification[13] is closely related to OGC API Features, and Chris 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 between OGC API Features and STAC is the latter's requirement that all items/features should have a temporal component, thus making it *spatio-temporal*.

2.4. Related Work

The related work is divided into two main parts: subsection 2.4.1, which presents research investigating potential use cases of LLMs in the field of geospatial information technology; and subsection 2.4.2, which presents examples of three different types of patterns commonly used in LLM-based agents.

2.4.1. 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.

[12] https://github.com/opengeospatial/ogcapi-features/tree/master/proposals
[13] https://stacspec.org/en

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 (note that multi-modal models were not particularly widespread when they wrote their paper). 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 GPT-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, GPT-4 outperformed GPT-3.5 in all tasks. While clearly powerful, the authors highlight a range of challenges, among which the multi-modal nature of GIS.

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 frameworks like 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 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 through case studies.

2.4.2. Agent Patterns

LLM-based agents can be implemented in many different ways, and researchers have developed a plethora of *agent patterns* that seek to improve upon areas where LLMs tend to be less effective.This section will shortly explain three different types of patterns that were considered during the development of GeoGPT.

The **multi-agent** pattern takes inspiration from human collaboration in that it is made up from multiple specialized agents that work together to achieve some objective. 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. AutoGen (Wu et al., 2023) and crewAI (Moura, 2024) are other examples of frameworks geared towards the multi-agent pattern.

Patterns that employ **self-reflection** are commonly used with autonomous LLM-based agents. *Reflextion* (Shinn et al., 2023) is a framework that reinforces agents through linguistic feedback, essentially allowing the agent to reflect upon the outcomes of its actions. The framework utilizes three distinct models: an *Actor* model responsible for generating text and actions; an *Evaluator* model which assessses 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 tasks *similar* to that which it is trying to solve. By helping the LLM with decomposing multi-step problems into intermediate steps, Wei et al. managed to achieve state-of-the-art accuracy on the GSM8K benchmark of math word problems.

3. GeoGPT

The GeoGPT chapter will give a detailed description of the inner workings of GeoGPT, the thesis' proposed solution to an LLM-based GIS. GeoGPT is a webpage that features a chat interface and a web map, where users can ask questions and receive answers as text and/or map geometries. GeoGPT features *three* different agent types: an *OGC API Features* agent, a *Python* agent, and an SQL agent. Their differences will be made apparent in this chapter.

Section 3.1 will present a high-level overview of GeoGPT's architecture. GeoGPT employs a client-server architecture, and is supported by auxiliary services such as databases and API endpoints. The client is a chat-based web GIS, and the server, which hosts the different agents, is responsible for streaming responses to the client in the form of chat messages and GeoJSON objects that can be added to the map on the client. Section 3.2 will delve into the architecture of the generic tool agent that is implemented using LangChain and LangGraph (see section 2.2 for detailed information), which is implemented by each of the three above-mentioned GeoGPT agents.

3.1. High-Level Application Architecture

The two main parts of GeoGPT is its web client and its server, the latter of which is written in Python. They are highlighted in green and blue in Figure 3.1, respectively, and is where all of GeoGPT's source code is located. The server is supported by auxiliary services that are not directly consider parts of GeoGPT, but that are essential to making it work. This includes data sources, such as the *OSM Shapefile* service, *PostGIS*, and *OGC API Features* services, all of which are deployed as Docker Containers; the Redis database, which is responsible for storing conversations between the user and GeoGPT; and OpenAI's proprietary inference API, which is hosts LLM models that GeoGPT uses for text generation. Sections 3.1.1 through 3.1.4 will go into further details on the parts that compose GeoGPT. The sequence diagram in Figure 3.2 shows how the different GeoGPT's different parts interact with each other when the user first loads GeoGPT and enters a question into its chat interface.

3.1.1. LangChain Server

The *LangChain Server* service is the heart of the GeoGPT, and is where the Large Language Model (LLM)-related logic is situated. It is responsible for taking user-written messages passed from the web client to the server's `/streaming-chat` GET endpoint, and returning suitable responses as a stream consisting of tokens and tool messages

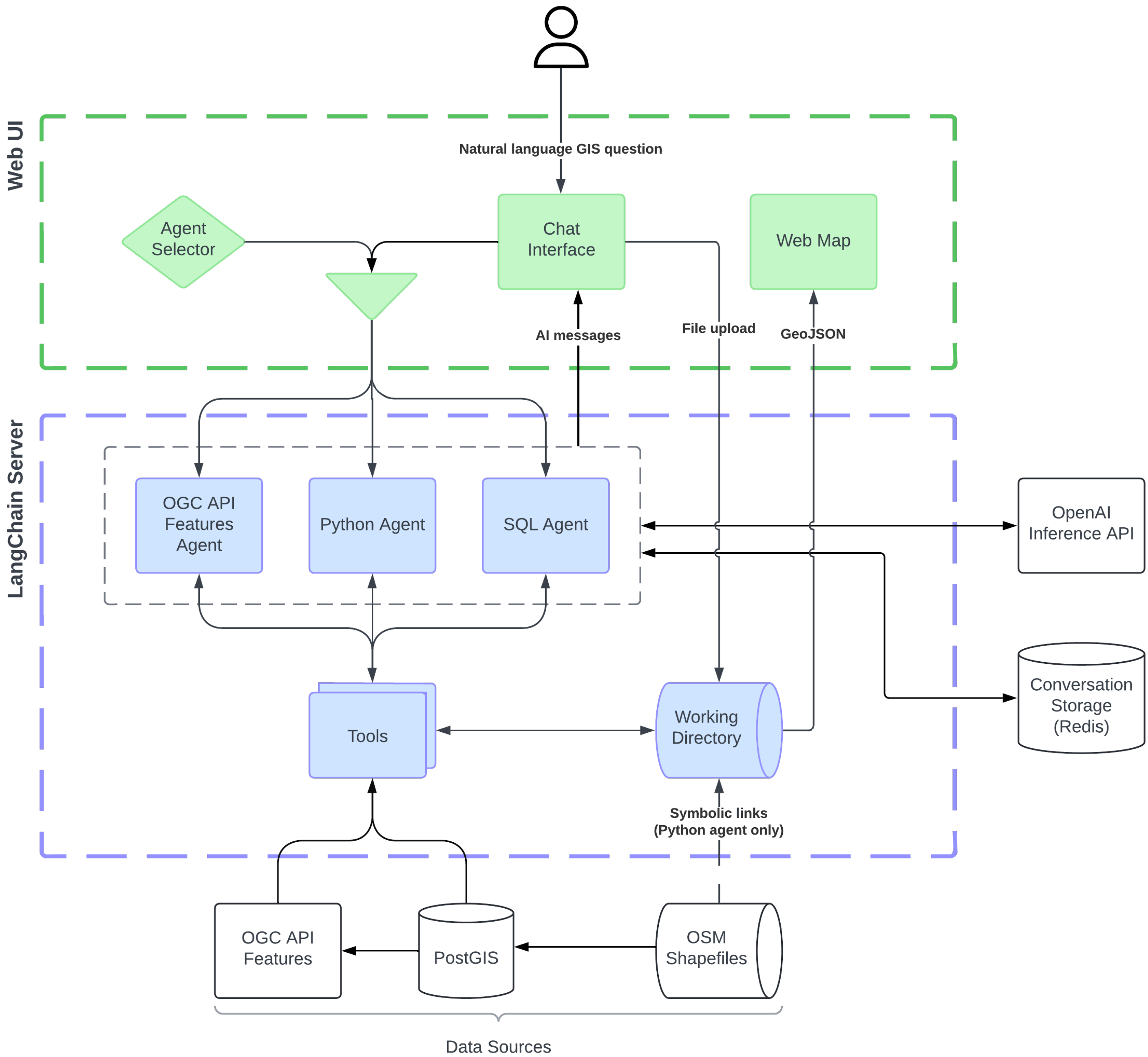


Figure 3.1.: Architecture overview highlighting the client-server architecture of GeoGPT, but also showing the major components that make up the Web UI (in green) and the LangChain Server (in blue). Auxiliary services like databases and external APIs are shows in white.

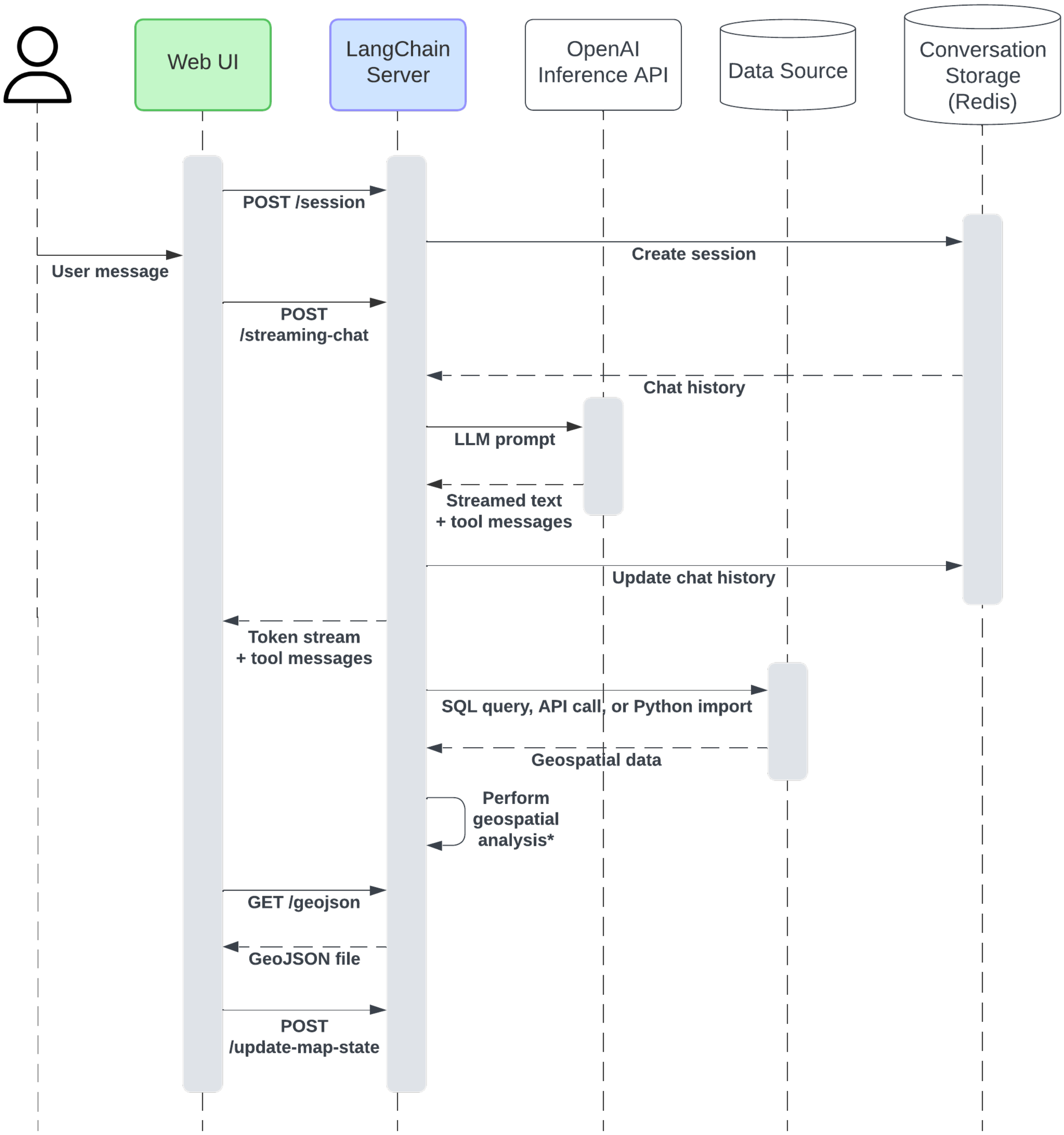


Figure 3.2.: Sequence diagram showing the information flow as the user loads and sends a message to GeoGPT. The `POST /session` event only happens on the initial page load, and not all events are required to occur every time the users enters a message into GeoGPT's chat interface.

(tools are discussed in detail in subsection 3.2.2). Table 3.1 shows a complete list of the endpoints exposed by the server and how they can be used by a client.

Table 3.1.: Summary of the endpoints exposed by the LangChain server

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.

The LangChain server features GeoGPT's three different agent types. These are listed below:

- **OGC API Features Agent** – Utilizes an OGC API Features endpoint.
- **Python Agent** – Accesses and manipulates shapefiles using Python code.
- **SQL Agent** – Interacts with data stored in an SQL database.

The agents are responsible for taking messages from the users, and utilizing the *tools* available to them to find answers to the geospatial questions from the user. Tools are Python functions that the user has access to through *function calling*, which was described in subsection 2.1.5. The tools available to each of the agents are presented in subsection 3.2.2.

The agents use OpenAI's inference API to generate text and function calls. Prompt engineering techniques are used within the agents to produce prompts that can be used as input to the LLMs that are hosted on the inference API. These techniques are described in further detail in subsection 2.1.4.

The agents have access to a working directory where they can store files on-disk. The agents can access the files in the working directory through some of the tools that are presented in subsection 3.2.2, one of which allows the agent to send GeoJSON files stored in the working directory to the map on the web client. The working directory is reset to its default state each time the web client is reloaded.

3.1.2. Redis for Conversation Storage

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, or as a simple NoSQL database. The latter option is the way it is used in GeoGPT's architecture, and its sole purpose is to store conversations between GeoGPT and its users. Whenever a user starts a conversation with GeoGPT, a session object is stored in the Redis database, as can be seen in Figure 3.2. This object holds an array that represents the conversation. This array is written to every time 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 an LLM to act as a conversational agent, a chat history needs to be included in the prompt. In the case of GeoGPT, the entire chat history is included. 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. Therefore, as the chat becomes longer each new token will be both more expensive and take longer to get generated. A long chat history could also make the resulting prompt exceed the token limit of the LLM. These issues were not considered much for this project and are left for future work.

3.1.3. Shapefiles, PostGIS, and OGC API Features

Shapefiles to be used in the experiment (see section 4.2) were downloaded to a directory on the machine from which GeoGPT is launched. These are the geospatial data that GeoGPT's will have at their disposal when the user asks a GIS-related question. These files are also "copied" to the working directory of GeoGPT's Python directory using symbolic links.[1] This way, the Python agent can access the files in the "arbitrary" directory where the *actual* file is stored, but using the equivalent file in the working directory which *points* to the real file. This makes the startup of GeoGPT much faster, since it does not copy the actual contents of the files.

A PostgreSQL database with the PostGIS extension was deployed using Docker, and populated with the shapefiles desribed above. On top of this database is an OGC API Features web server. It is deployed using the `pramsey/pg_featureserv` Docker image (CrunchyData, 2024), which is requires a connection string to the PostGIS database that

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

is to be exposed as an OGC API Features server. Any table in the database which has a geometry column and a specified Coordinate Reference System (CRS), will be exposed on the server. *pg_featuresserv* includes functionality like bounding box filtering, result limiting, and CQL filtering. These are added as query parameters in the URL used to fetch a collection's 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 3.1 shows an example of how CQL code is converted into SQL code:

```
\\ CQL code passed through the `filter` query parameter
within(geom, POINT(0 0))

\\ SQL code that will be run agains the database
ST_Within("geom",'SRID=4326;POINT(0 0)'::geometry)
```

Code Snippet 3.1: Example of how a CQL filter checking if a point at coordinates (0, 0) resides within a feature's geometry, is converted by *pg_featuresserv* to the corresponding SQL code

The shapefiles, PostGIS database, and OGC API Features endpoints all contain the same data. The rationale behind this becomes apparent in the Experiments chapter under section 4.1.

3.1.4. Web UI

GeoGPT's user interface is made using SolidJS, a JavaScript framework for creating websites. As Figure 3.3 shows, it consists of a chat interface, a web map, and an agent selector located above the chat interface. The chat interface was designed to imitate the interface of OpenAI's ChatGPT. Here, the user can ask geospatially related questions for GeoGPT to answer. Tokens and tool messages are streamed from the LangChain server and added to the chat like the Figure 3.3 shows, which helps the user follow GeoGPT's thought process as it is solving a problem. The generated text is streamed in the Markdown format.[2] A library called *showdown*[3] is used to convert Markdown into HTML, ensuring that tables, code blocks, lists, and other elements are properly rendered. Left of 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, as Figure 3.1 shows.

[2] https://en.wikipedia.org/wiki/Markdown
[3] https://github.com/showdownjs/showdown

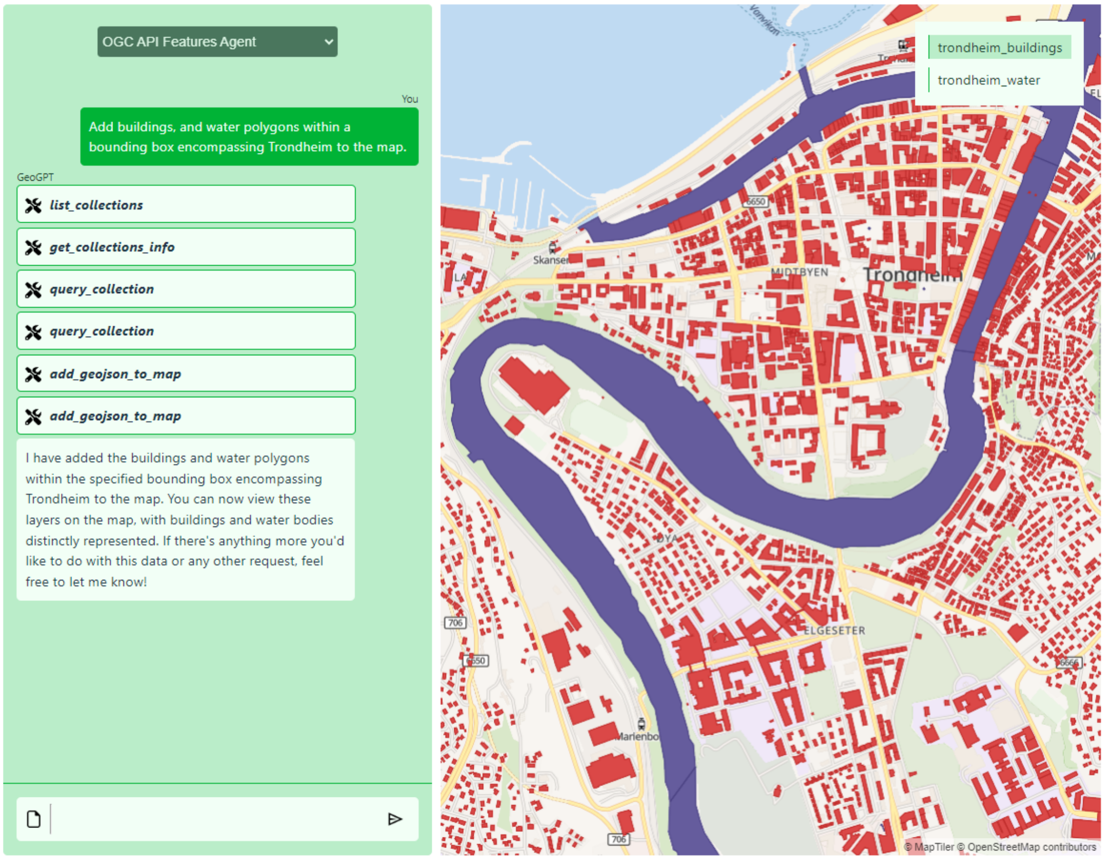


Figure 3.3.: GeoGPT's web UI, which features a chat interface and a web map that displays results from analyses. In this example, the user has asked GeoGPT to add water and building geometries for Trondheim to the map. This was successful, as layers named `trondheim_water` and `trondheim_buildings` were added by GeoGPT, after a series of tool calls.

The map is created using MapLibre,[4] an open-source fork of Mapbox. A base map from OpenStreetMap (OSM) is used, fetched through a mapping platform called MapTiler.[5] GeoJSON files that are fetched from the server will be added to the map automatically with a random colour. 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.

[4] https://github.com/maplibre/maplibre-gl-js
[5] https://www.maptiler.com/

3.2. Agent Architecture

Common to GeoGPT's agents is their agentic architecture, which is described in subsection 3.2.1. They differ, however, through their assigned *tools*, which are described in subsection 3.2.2. Other slight differences are seen in the way that they are prompted. The prompting strategy used for GeoGPT will be discussed in subsection 3.2.3.

3.2.1. LangGraph Agent Implementation

The agentic behaviour of GeoGPT is implemented using LangGraph, which was described in detail in section 2.2. Figure 3.4 illustrates the flow between the various nodes that make up the agent, and also shows the prompt engineering approach used in GeoGPT, which will be discussed in detail in subsection 3.2.3. A state dictionary is passed between the nodes, and is updated at each of these. Included in this state is a list of `Message` objects (the chat history), the path to GeoGPT's working directory, a list of the current files in this directory, as well as other less important state. The implementation is based upon a prebuilt `chat_agent_executor` implementation from LangGraph.[6]

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 "Prompt Engineering" section of the graph, where a prompt is constructed and passed on to the `agent` node. In the `agent` node, the prompt is passed to an LLM, so that a response to the user can be generated. This response could contain text and/or instructions to invoke some tool. Therefore, the current state, containing both the user message and the AI message, is sent to the conditional node called `agent_should_continue`. This node simply checks if the last `AI Message` includes the "tool_calls" keyword argument. If this is the case, then some tool should be invoked, and we should proceed to the `action` node.

In the `action` node, the values attached to the "tool_calls" keyword argument, which are LLM-generated, are used to invoke predefined tools. The LLM specifies tool *names* and suitable tool *parameters* that will be passed to the predefined tool (see subsection 2.1.5 for more details on how *function/tool calling* works). These tools are then executed, possibly having side effects on the server, and the return values from the tools are appended as tool messages to the chat history, as Figure 3.4 shows. Figure 3.5 illustrates this behaviour, which also includes some mid-conversation system messages to give the agent updates about the system's state (further discussed in subsection 3.2.3). We can now loop back to the `Prompt Engineering` and `agent` nodes, so that the agent can react to the results from the tool invocation. This cyclic behaviour allows the agent to repeatedly call tools to try and answer the request from the user. When the agent finds no reason to call any more tools, `agent_should_continue` will return `False` so that the termination node, `__end__` is reached.

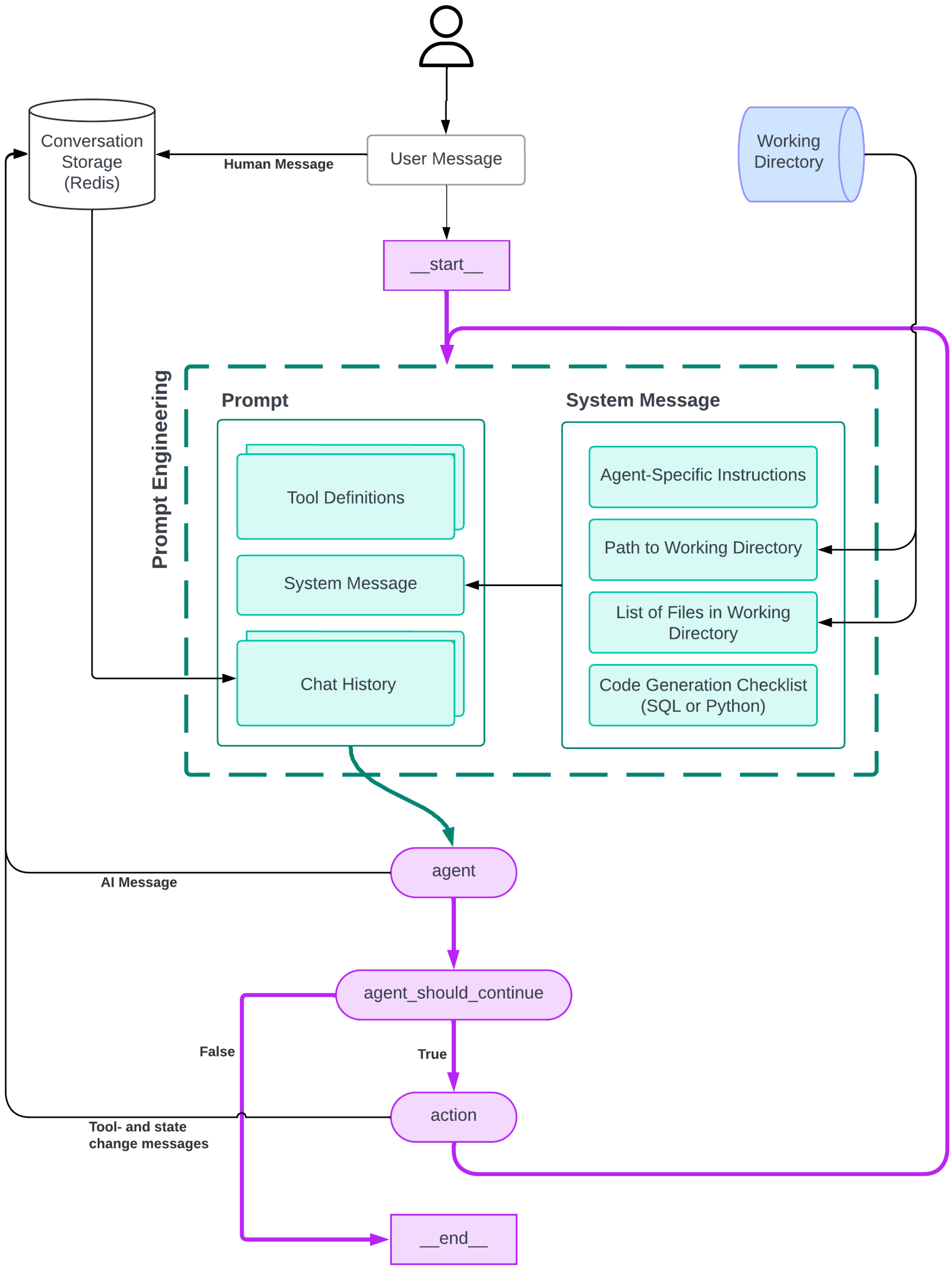


Figure 3.4.: Generic tool agent graph with prompt engineering approach. Purple nodes and edges are what makes up the *LangGraph* graph, while teal-coloured parts are related to prompt engineering.

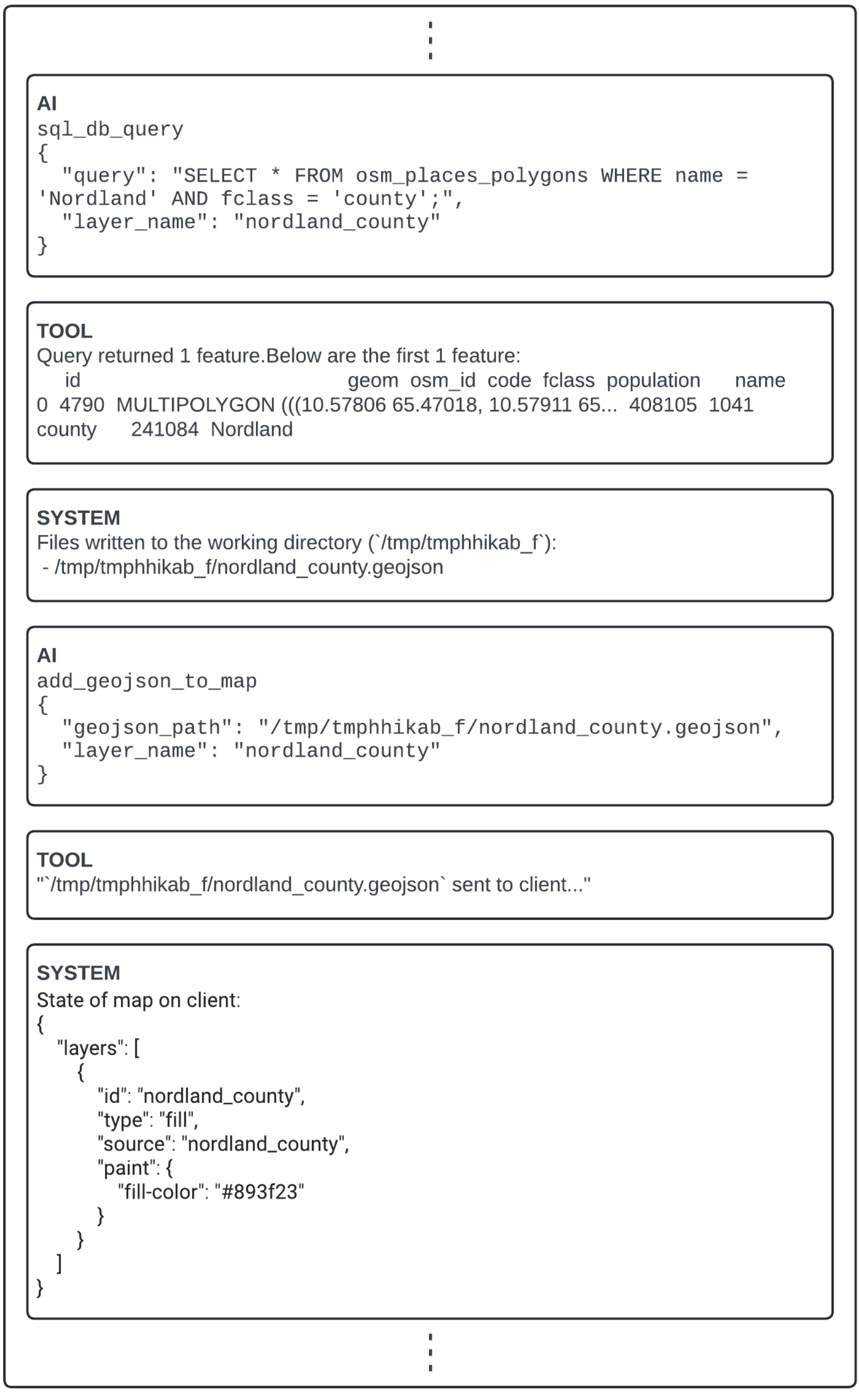


Figure 3.5.: Example of a mid-conversation chat history. GeoGPT elects to invoke the `sql_db_tool` to get a polygon for Nordland county, before adding it to the map. Tool- and system messages help ensure GeoGPT that the actions it takes are successful.

Table 3.2.: Overview of GeoGPT's 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`

3.2.2. Tools

Table 3.2 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, a 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 list of the available collections. This tool gives 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 relevant information for each of these collections. This includes the JSON response from the "landing page" of the collection, which includes details such as the collection description, spatial extent, and available attributes. Furthermore, a list of common values for certain high-cardinality attributes is included in the tool's response, as exemplified in Code Snippet 3.2. The percentages are obtained by querying a large number of features from `/collections/{collection_id}` and calculating the prevalences between them.

```
1 Property: fclass
```

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

```
    tree: 71.1%
    peak: 27.2%
    beach: 0.9%
    cave_entrance: 0.5%
    spring: 0.2%
    cliff: 0.1%
    volcano: 0.0%
```

Code Snippet 3.2: Prevalences of common values for the *fclass* property of 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. Based on these 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 in the working directory on the LangChain server as "{layer_name}.geojson". The message returned from the tool reads something like this: "Query returned 5627 features." If the GeoJSON itself was returned as a tool message, this would quickly bloat the context window of the LLM, and therefore it is avoided.

Common for the OGC API Features agent and the Python agent is the **python_repl_-ast** tool. This tool takes a string of Python code, executes it, and returns whatever the code prints to the standard output in a so-called Read-Eval-Print Loop (REPL). If the code errors, the error message is returned instead. This Python tool is the main way for these two agents to perform geospatial analyses. An advantage of using REPLs is that code can be executed in blocks, with variables from one block being shared with other blocks. This means that if the first block loads a large file into memory — an often time-consuming operation — the subsequent blocks can reuse this in-memory variable without reloading the data. This allows the LLM to quickly retry if the code should error, or if the outcome of the initial code wasn't as expected.

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 or path of a GeoJSON file stored in GeoGPT's working directory, as well as a layer name. Invoking the tool will send a message to the client, which includes 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 (see Figure 3.2).

The SQL agent has tools very similar to the OGC API Features agent. It has a direct connected to the same database that the OGC API Features endpoint 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 returns information about attributes, prevalence of different values in high-cardinality columns, and other details about these tables, much like `get_collections_info`. `sql_db_query` accepts arbitrary SQL code that will be executed against the database. The tool will make sure that query results that has a geospatial component will be stored as GeoJSON in GeoGPT's working directory, so that it can add the geometries to the map on the client.

3.2.3. Prompt Engineering

Prompt templating is a way of producing prompts with a predefined structure for use with LLM. Figure 3.7 shows the prompt passed to GeoGPT's SQL agent 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, as Figure 3.4 shows. It is constructed before every invocation of the *agent* node, where it is passed to an LLM so that it can generate an answer. LLMs have no inherent memory, so in order to have a chat conversation, the entire chat history needs to be passed with the prompt.[7]

Figure 3.4 shows that there are four elements to the main system message:

1. Agent-specific instructions
2. A path to the working directory
3. A list of the current files in the working directory
4. A checklist that the LLM should use when generating code

The agent-specific instructions includes information intended to give the LLM contextual awareness and hints on how it should work towards solving tasks. We start by telling LLM that it's a "helpful GIS agent/consultant that has access to an SQL database containing OpenStreetMap data". This is a common way of giving LLMs a persona, and results in responses like the one in Figure 3.6. Continuing, we give it some information on how to string tool calls together to solve tasks. We tell it to first list available tables, then look up schemas of relevant tables, and then, using the information gathered, to construct an SQL query to answer the user's request. We also remind it that it needs to add the result of the analyses to the map, using the `add_geojson_to_map` tool.

[7] 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.

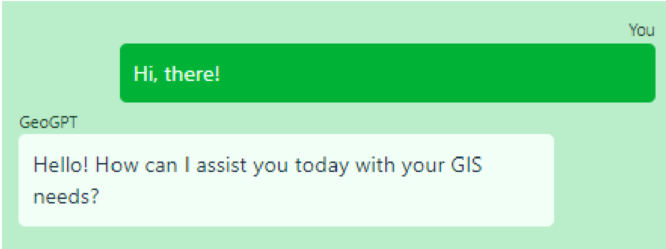


Figure 3.6.: Conversation showcasing the effect of giving an LLM a persona through the system message

The next two parts of the system message concern GeoGPT's working directory. Having a working directory is important to control which files the agent has available, and also to make sure it doesn't save files to the folder on the LangChain server where GeoGPT's source code is located, as this would bloat the actual source code. First, we tell the LLM where the working directory is located. 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 using the Python code they generate. Second, we list the files that currently reside in the working directory. In the prompt in Figure 3.7, there "are currently no files written to the working directory", but if there were any, they would be listed in a bullet point list in place of this message.

The final part of the system message is a checklist to inform the LLM about common pitfalls it could run into when generating code. The checklist in Figure 3.7 is tailored for GeoGPT's SQL agent. A similar checklist for Python is provided for the other two agents, with reminders about using metric CRSs when doing area calculations, etc.

Normally, prompts for LLM contain only a single system message. GeoGPT, however, features a — to the author's knowledge — novel usage of system messages. As can be seen in Figure 3.5, system messages are appended mid-conversation, providing updates about state changes in the system. As Figure 3.4 shows, these are generated in the `action` node. 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 in Figure 3.5, it uses this information immediately to add the only file available to the map using `add_geojson_to_map`. Another 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.

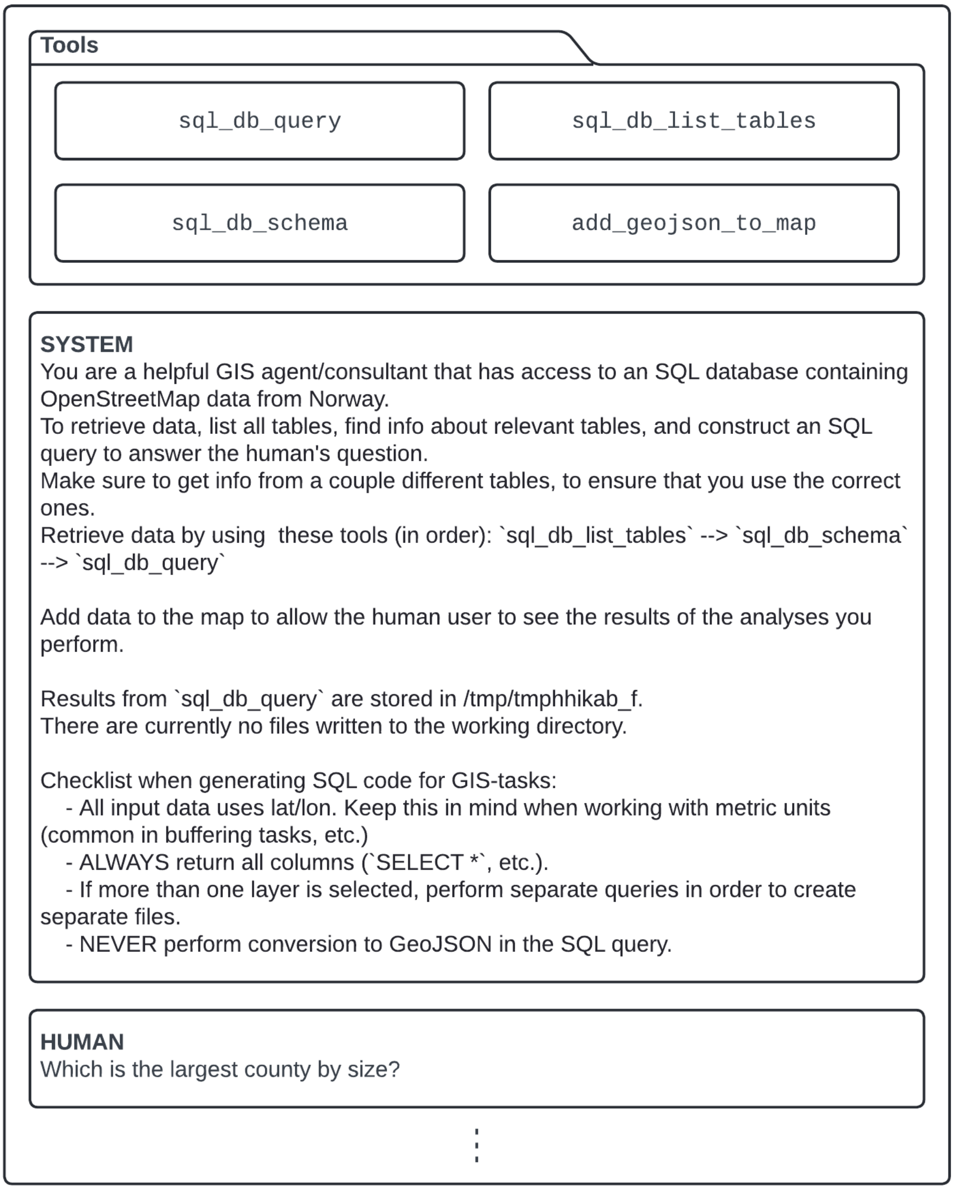


Figure 3.7.: Initial prompt to GeoGPT as the user asks a question. The prompt includes function/tool definitions, the main system message, and the chat history (which currently only includes the initial message from the user)

4. Experiments

Section 4.1 of the Experiments chapter will describe the methodology and rationale behind the two experiments conducted in this thesis. The first experiment, presented in subsection 4.1.1, assesses GeoGPT's abilities of solving common GIS tasks through a new "GIS benchmarking" dataset. The other experiment, presented in subsection 4.1.2, will assess the importance of the quality of the prompt passed from the user to GeoGPT. Section 4.2 lists the datasets used in the experiments. Section 4.3 presents the experimental results and offers interpretations of the findings from the experiments, and will also present several example results, including screenshots of the application and code that generated by GeoGPT.

4.1. Experimental Setup

This section will explain the concept and setup of the two experiments mentioned in the introduction of this chapter. Section 4.1.1 will describe the setup for the GIS benchmarking experiments, while subsection 4.1.2 will explain the setup for the prompt quality experiment. Results for the two experiments are presented in Table A.2 and Table A.4, respectively. Subsection 4.1.3 describes the hardware on which the experiments were executed, and the specific LLMs that were used.

4.1.1. GIS Benchmark

The first experiment will measure the abilities of GeoGPT three agent types to produce correct answers, and will also measure cost and time taken to generate answers, and the agent's ability to produce similar answers consistently (repeatability). To do this, a benchmarking dataset was constructed. This is a set of 12 GIS-related questions with corresponding correct answers, and can be viewed in its entirety in Table A.1. Furthermore, as described in section 3.1, GeoGPT features *three* different agent types that access data in *three* different ways. The GIS benchmarking experiment will therefore also seek to compare the performance of these three agents on the same questions, having access to the same data. Each of the 12 questions will be asked three times per agent type. As there are three agent types (see section 3.2), the total number of test runs becomes the following:

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

The following sections, named "Outcome Evaluation", "Cost and Duration", "Repeatability", will go into detail on the different ways that the results are evaluated.

Outcome Evaluation

Each test run's GeoGPT-generated answer will be manually evaluated based on how well the question was answered, and this outcome will be annotated as one of the following: *success*, *partial success*, and *failure*. Table 4.1 shows the guidelines used when assigning outcome scores.

Table 4.1.: Guidelines for evaluating outcome of tests. Test runs are evaluated as a *success*, a *partial success*, or a *failure*.

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.

Cost and Duration

The application is hooked up to LangChain AI's tracing system, *LangSmith*.[1] Apart from being a useful tool for debugging purposes, LangSmith provides a simple way to obtain detailed data on token and time usage for a particular run, as well as the total cost of the run, in American dollars.

Box plots will be used to compare these three metrics for the different agents. These box plots make it easy to compare the three models in terms of token usage and time taken to complete tasks. They follow the description for box plots found on Wikipedia.[2,3] Box plots allow us to easily see 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.

[1] https://www.langchain.com/langsmith
[2] https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.boxplot.html
[3] https://en.wikipedia.org/wiki/Box_plot

Repeatability

Another aspect that the GIS benchmarking experiment will try to evaluate is the consistency of the system, its ability to repeatedly provide an acceptable answer to the same user question. The annotated outcomes are encoded using the ordinal encoding presented Table 4.2. A higher value indicates a better outcome. Using these encoded outcomes, standard deviations will calculated for each triplet of identical test samples (samples where the question and the agent type remained the same), which serve as a suitable measure for assessing repeatability.

Table 4.2.: Encodings for test outcomes. A higher value is synonymous with a better outcome.

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

4.1.2. Prompt Quality Experiment

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

For these experiments, the three hardest questions from the GIS benchmarking experiment were picked. Then, for each of these three questions, a *novice*-level and an *expert*-level prompt was constructed. The novice-level prompt is as simple as possible, while the expert-level prompt is more elaborate and written as a step-by-step recipe for solving the problem.

4.1.3. Hardware and Model Version

All experiments were conducted 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 Inference API.

It is worth noting that two slightly different models were used during testing. This is due 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`,[4] which is the model that was for the first test runs. At `gpt-4-turbo-2024-04-09`'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 confidence that these slight model upgrades would not significantly change the outcome of the experiments.

4.2. Datasets

A total of eighteen datasets were used in the experiments, all downloaded from Geofabrik's website.[5] Geofabrik, which is 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 4.3 lists all datasets, along with short descriptions of their contents, and the number of features for each dataset. Common for all datasets is 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. This plot was created by GeoGPT.

Table 4.3.: Datasets used in experiments

Dataset	**Type**	**Description**	**#Features**
Buildings	Polygon	Contains building outlines. Its *type* attribute can have values like *house*, *university*, and *restaurant*.	4,147,645
Land Use	Polygon	Represents areas designated to different purposes and activities. Its *fclass* attribute can have values like *forest*, *farmland*, and *residential*.	541,452

Continued on next page

[4] 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.

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

Table 4.3 continued from previous page

Dataset	**Type**	**Description**	**#Features**
Natural	Point	Contains outlines of various objects found in nature. Its *fclass* attribute can have values like *beach*, *glacier*, and *cave_entrance*.	119,725
Natural	Polygon	Similar to the point data equivalent.	6,665
Places of Worship	Point	Common values for *fclass* attribute: *christian*, *buddhist*, and *muslim*.	311
Places of Worship	Polygon	Similar to the point data equivalent.	2,520
Places	Point	Common values for *fclass* attribute: *farm*, *village*, and *island*. Repeated entries trimmed for brevity.	178,997
Places	Polygon	Similar to the point data equivalent.	5,689
Points of Interest	Point	Common values for *fclass* attribute: *tourist_info*, *bench*, and *kindergarten*.	117,677
Points of Interest	Polygon	Similar to the point data equivalent.	45,825
Railways	Lines	Common values for *fclass* attribute: *rail*, *subway*, and *tram*. Also has True/False attributes saying if a given line segment is a bridge or a tunnel.	14,008
Roads	Lines	Common values for *fclass* attribute: *rail*, *subway*, and *tram*. Has additional attributes *oneway*, *maxspeed*, *bridge*, and *tunnel*.	1,741,929
Traffic	Point	Common values for *fclass* attribute: *crossing*, *street_lamp*, and *parking*.	98,860

Continued on next page

Table 4.3 continued from previous page

Dataset	Type	Description	#Features
Traffic	Polygon	Common values for *fclass* attribute: *parking*, *pier*, and *dam*.	45,623
Transport	Point	Common values for *fclass* attribute: *bus_stop*, *ferry_terminal*, and *railway_station*.	91,627
Transport	Polygon	Similar to the point data equivalent.	935
Water	Polygon	Common values for *fclass* attribute: *water*, *wetland*, and *river_bank*.	1,861,199
Waterways	Lines	Common values for *fclass* attribute: *stream*, *river*, and *canal*.	833,253

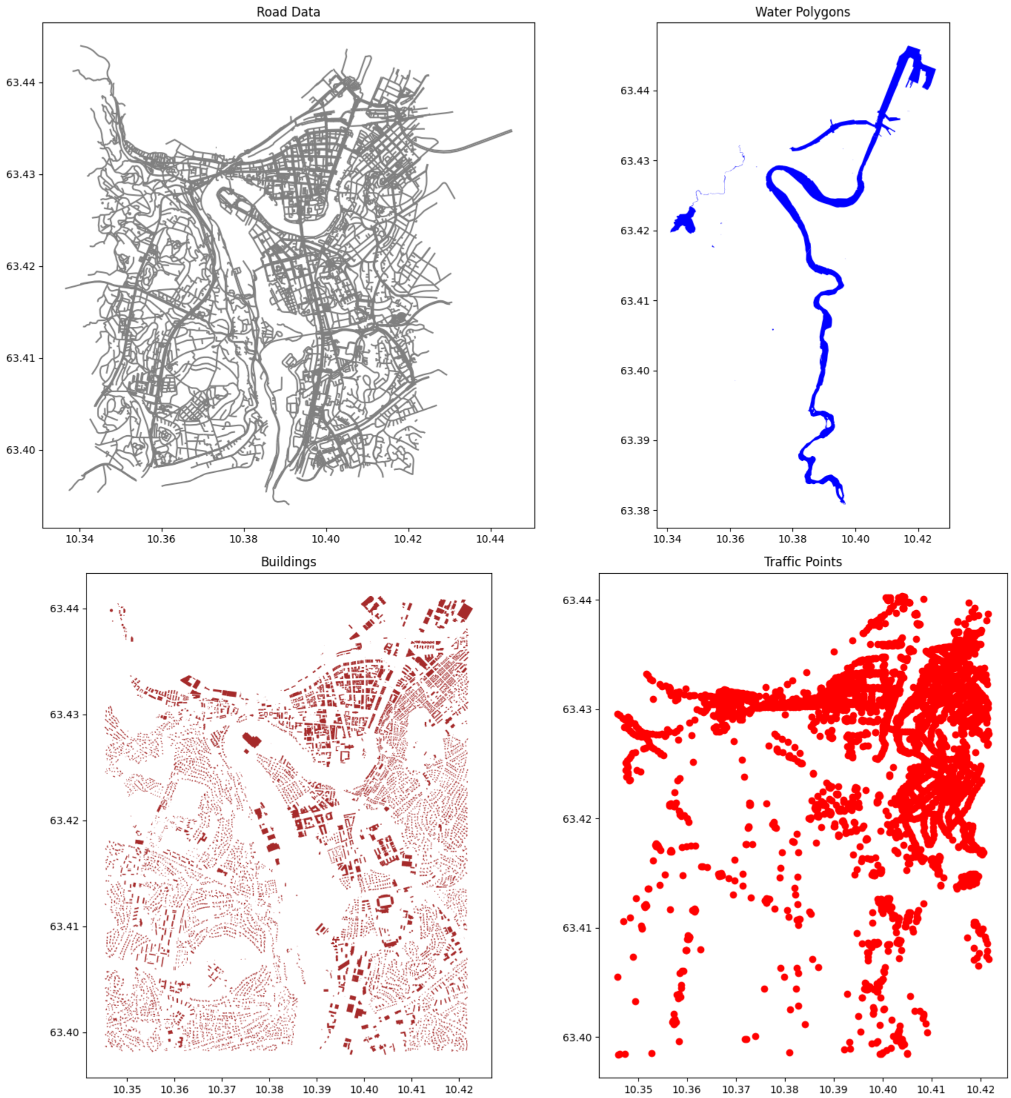


Figure 4.1.: A plot of four selected datasets constrained to a polygon of Trondheim. From top-left, clockwise: road data, water polygons (large rivers, lakes, etc.), traffic points (motorway junctions, crossings, traffic signals, etc.), and building polygons. The plots were generated by GeoGPT.

4.3. Experimental Results

Subsection 4.3.1 and subsection 4.3.2 will present the outcome of the experiments presented in subsection 4.1.1 and subsection 4.1.2, respectively. Graphs created for this chapter are created using Matplotlib,[6] a Python library suitable for creating visualizations like bar charts, box plots, etc.

4.3.1. GIS Benchmark — Results

Outcome Evaluation

Figure 4.2 displays a bar chart for the outcome distribution per agent. Figure 4.2 shows that the OGC API Features and Python agent have comparable results, but that the SQL-based agent performs significantly better compared to the other two in terms of producing the desired outcome. The SQL agent has a success rate of 69.4% compared to the other two, which share a success rate of 38.9%. Possible reasons for this are discussed in section 5.2 in the Discussion.

Figure 4.2.: Outcome distribution between GeoGPT's different agent types. The OGC API Features and Python agents perform similarly, while the SQL agent outperforms them by a significant margin.

[6] https://matplotlib.org/

Cost and Duration

Figure 4.3 displays a box plot with a logarithmic y-axis showing task durations across the different agent types. Here we can see that the SQL agent spends the least amount of time per task. The OGC API Features agent has a slightly higher median and 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 $Q4 \sim 984$ seconds. The large gap to the other two agents it largely due to the Python agent's tendency to load large datasets into memory. 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 4.1. 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 $\sim$ 1.4GB, 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.

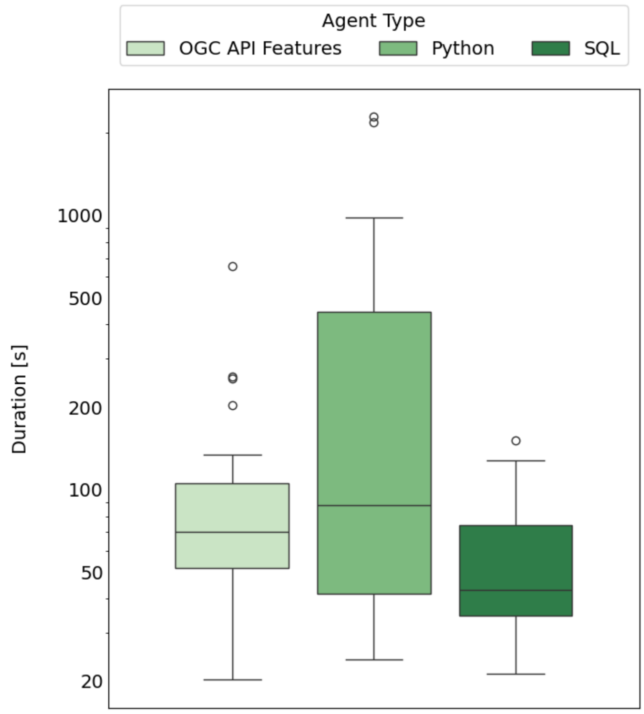
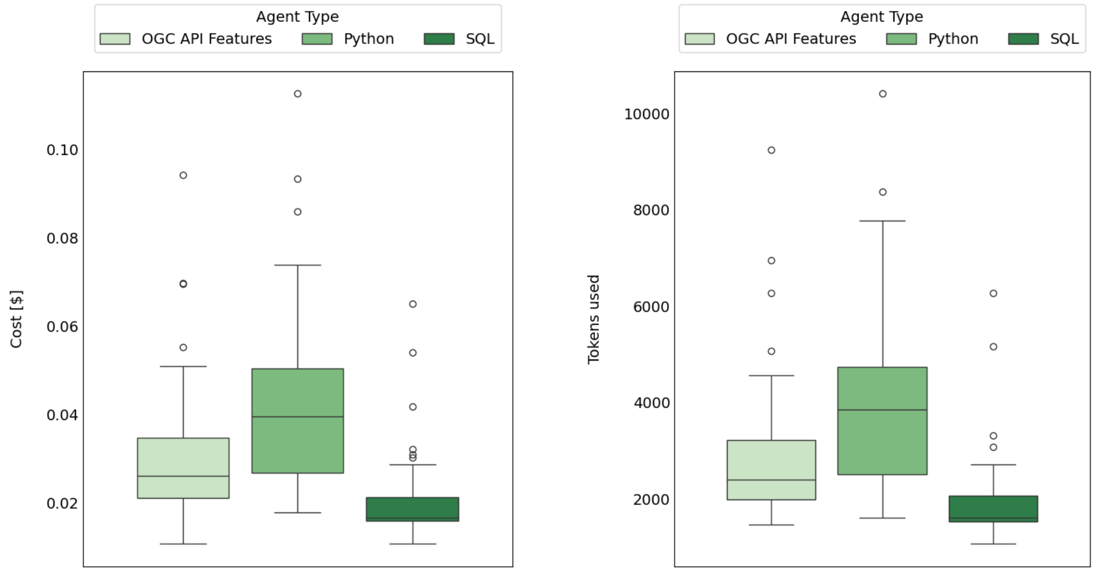


Figure 4.3.: Box plots comparing the time taken to generate answers across GeoGPT's agent types. Again, the SQL agent performs best, while the Python agent is the odd one out with very slow times compared to the two others.

```
1 import geopandas as gpd
2
3 # Paths to the shapefiles
4 landuse_path = '/tmp/tmpsutdy6it/osm_landuse_polygons.shp'
5 places_path = '/tmp/tmpsutdy6it/osm_places_polygons.shp'
6
7 # Load the data from shapefiles
8 landuse_gdf = gpd.read_file(landuse_path)
9 places_gdf = gpd.read_file(places_path)
10
11 # Filter out 'residential' areas from the landuse data
12 residential_gdf = landuse_gdf[landuse_gdf['fclass'] == 'residential']
13
14 # Compute the spatial difference to exclude residential areas from the places data
15 oslo_outline = gpd.overlay(places_gdf, residential_gdf, how='difference')
16
17 # Path for the output GeoJSON file
18 output_path = '/tmp/tmpsutdy6it/oslo_outline_no_residential.geojson'
19
20 # Save the resulting GeoDataFrame to a GeoJSON file
21 oslo_outline.to_file(output_path, driver='GeoJSON')
22
23 # Output the path to the saved file
24 print(output_path)
```

Code Snippet 4.1: GeoGPT-generated Python code aimed at computing the difference between the Oslo outline and residental features within it. The execution time of this code block was $\sim$ 40 seconds, mostly due to lines 8-9.

Figure 4.4 shows box plots for token usage and cost per run. Naturally, these figures appear very similar, since prices for the input tokens and generated output tokens are fixed for the models used. According to OpenAI's webpages,[7] they charge $10 per million input tokens and $30 per million output tokens for their GPT-4 Turbo model. From the results of the experiments, a ratio of approximately 10.7 per million tokens — either input *or* output — was calculated, which is closest to the input token price. This shows that a lot more input tokens were used than output tokens.



(a) Average cost per call for each agent type

(b) Average token usage per agent type

Figure 4.4.: Cost and token usage across GeoGPT's different agent types. These plots will be highly correlated, but will differ somewhat because of the ratio between input and output tokens.

An observation that can be made from Figure 4.5 is that the 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 main contributor towards the total run time.

Another observation that can be made from the Figure 4.5 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 is likely to be more expensive in terms of token usage, and more likely to produce an undesirable outcome. Possible reasons as to why this is the case will be explored in subsection 5.3.1

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

the Discussion.

Repeatability

Table 4.4 shows the average standard deviation of the outcomes for each agent type, as well as the mean of these three standard deviations. These numbers indicate that there is a notable amount of inconsistency in GeoGPT's answers when task and agent type stays the same, on average deviating with more than a third of an outcome category (0.376 for the encoded values).

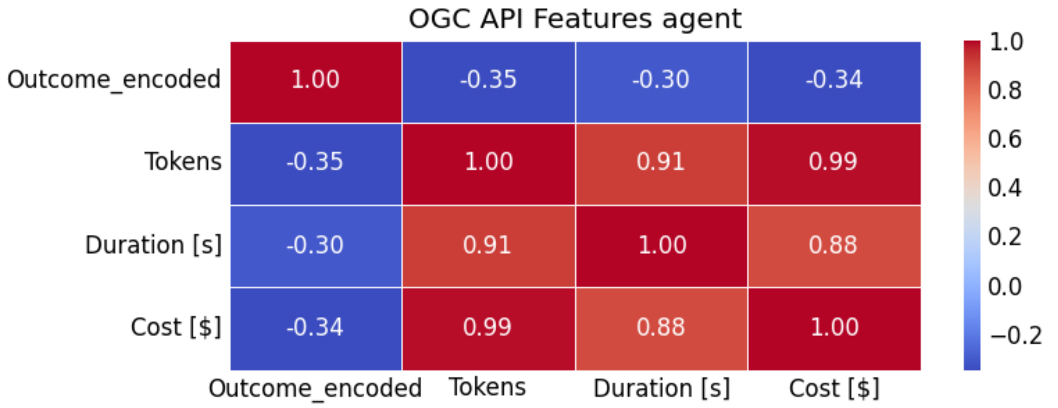
Table 4.4.: Standard deviation of encoded outcomes for GeoGPT's agent types. These numbers are obtained by using the encoded values for outcomes (see Table 4.2) for identical test runs in the GIS benchmarking experiment.

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

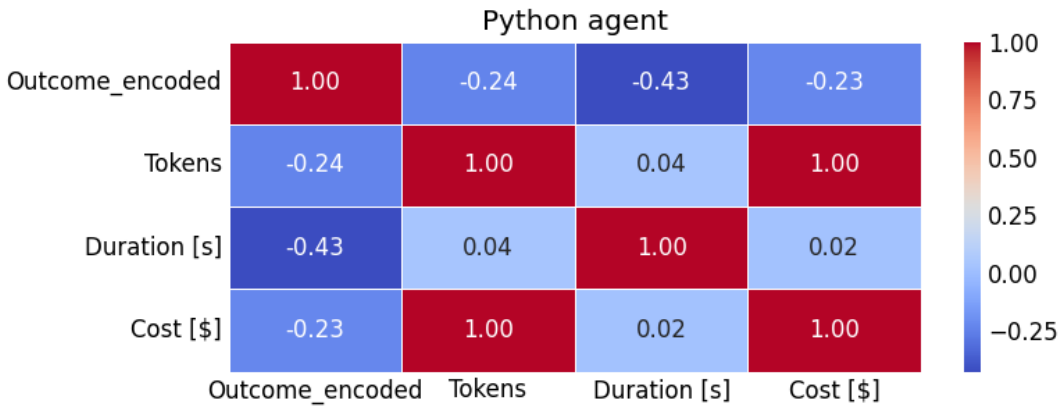
Successful Responses

Figure 4.6 shows a successful response from GeoGPT's SQL agent when asked how many counties the Glomma river runs through.[8] As a final answer to the initial prompt it lists all four correct counties in a numbered list. Code Snippet 4.2 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 was not included in the response (which the tool informs GeoGPT about), meaning GeoGPT was unable to add the results to the map. It 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 answer it gave was correct.

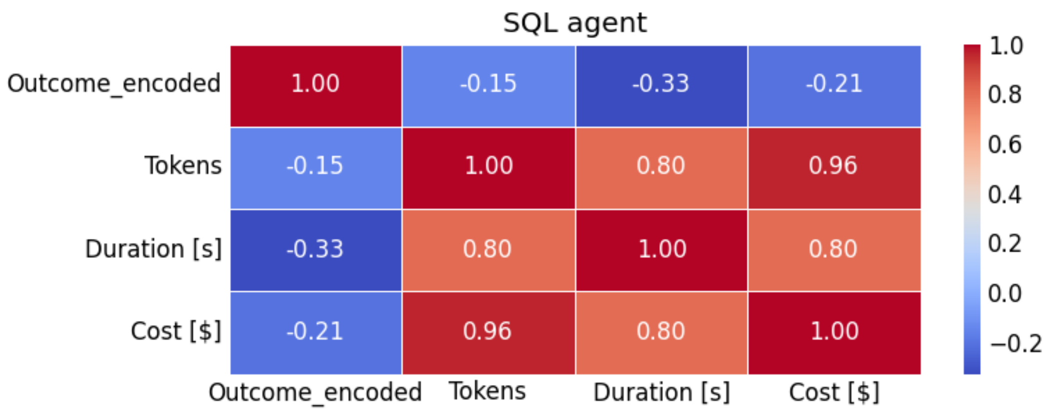
[8] This run was not part of the results used for evaluation due to a bug in the `sql_db_query` tool that caused it to unnecessarily execute queries twice. GeoGPT's response would not be different had the bug not been present for the run, but it would have taken longer to complete.



(a) Correlation matrix for the OGC API Features agent



(b) Correlation matrix for the Python agent



(c) Correlation matrix for the SQL agent

Figure 4.5.: Correlation matrices for the metrics recorded during the GIS benchmarking experiment for the three agent types

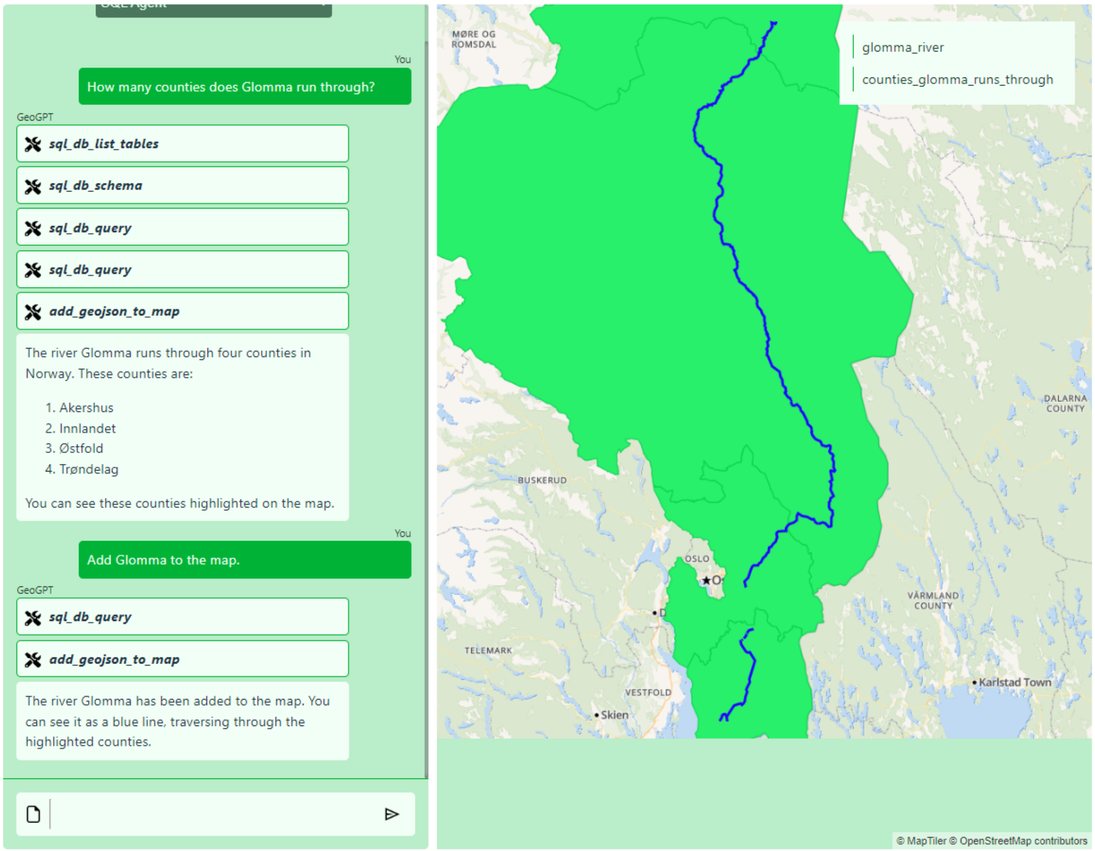


Figure 4.6.: Successful response from GeoGPT's SQL agent when asked how many counties the Glomma river runs through. Using the tools available to it, GeoGPT's SQL agent was able to to find that the river Glomma runs through the following counties: Akershus, Innlandet, Østfold, and Trøndelag. A follow-up message was added to make it add the river geometries to the map, for verification of its results.

```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 4.2: GeoGPT-generated SQL code generated to retrieve the counties that the Glomma river runs through. The `ST_Intersects` function from PostGIS was used to find the intersection with the river data and the county polygons, and the names and geometries of the counties was included in the result.

Figure 4.7 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 to the question is difficult, the correct answer was set to about 100, assuming a 20-meter buffer around the line segments that make up the road, is used. In Code Snippet 4.3 we can see how GeoGPT loads the road data around Munkegata in Trondheim, using an approximate bounding box to avoid getting roads named "Munkegata" in other cities. It then filters and saves the loaded data to only include features with name `== 'Munkegata'`. In the following code block (see Code Snippet 4.4), GeoGPT gets the tree data by filtering the `osm_natural_points` dataset on the `fclass` attribute, before loading the road data for Munkegata from the subsequent code block, converting both datasets to a metric CRS, creating a buffer around the road data, performing an intersection between the buffer, and finally printing the `tree_count` to the standard output.

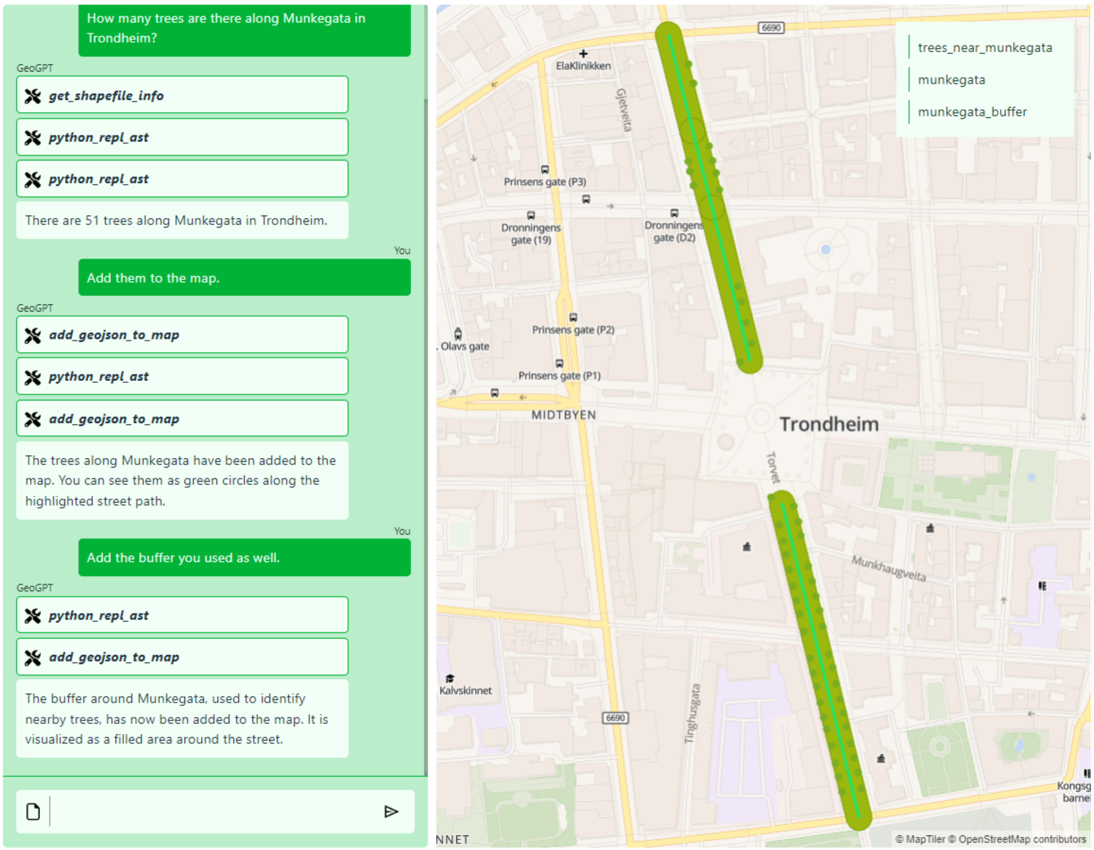


Figure 4.7.: Partially successful response from GeoGPT's Python agent when asked how many trees there are along Munkegata in Trondheim. It managed to find the road segments corresponding to Munkegata in Trondheim, but create a slightly too small buffer around it, which did not cover all the relevant trees.

```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 4.3: Python code to exclude roads named Munkegata *outside* of Trondheim. The bounding box, which is generated from background knowledge, covers a suitable area in the center of Trondheim, where Munkegata is located. This step is important, as a road named “Munkegata” can also be found in Oslo.

```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)

# --> 51
```

Code Snippet 4.4: Python code to calculate the number of trees along Munkegata in Trondheim. The datasets were converted to a metric CRS (WGS 84 / UTM zone 33N), before a buffer around the road data from Code Snippet 4.3 was created and used to find the intersection with the tree data. 51 trees were found, but ideally ~ 100 should be returned.

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 4.8 exemplifies a common mistake with GeoGPT, which will be 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, not just Oslo. As Code Snippet 4.5 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.

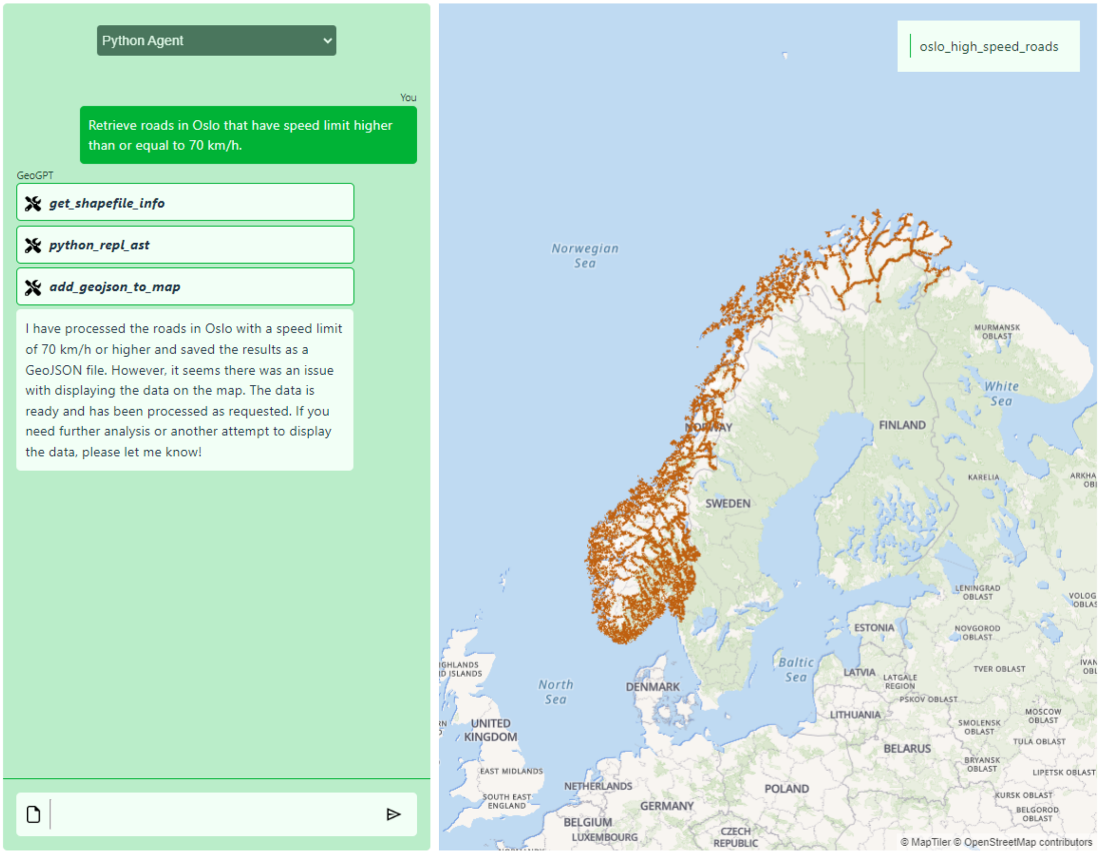


Figure 4.8.: Unsuccessful attempt by GeoGPT's Python agent to retrieve high-speed roads in Oslo. The figure shows roads throughout Norway with speed limits of 70 km/h or higher, but it should only have shown those in Oslo, the Norwegian capital.

```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 4.5: GeoGPT-generated Python code that saves high-speed roads with a misleading filename. This was the source of error that lead to the result in Figure 4.8. GeoGPT assumed that the `osm_roads_lines.shp` included road data only for *Oslo*, when in reality it includes data for the *whole of Norway*.

Figure 4.9 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 4.6 shows 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 many 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 4.7, where it picks the first point feature in each collection and assumes that their coordinates are those of Oslo and Bergen. These two features actually correspond to are Rakkestad Airport and a bus stop on a road along Ofotfjorden in Nordland. In addition to this mistake, subsequent attempts at creating a geodesic line between the locations were unsuccessful.

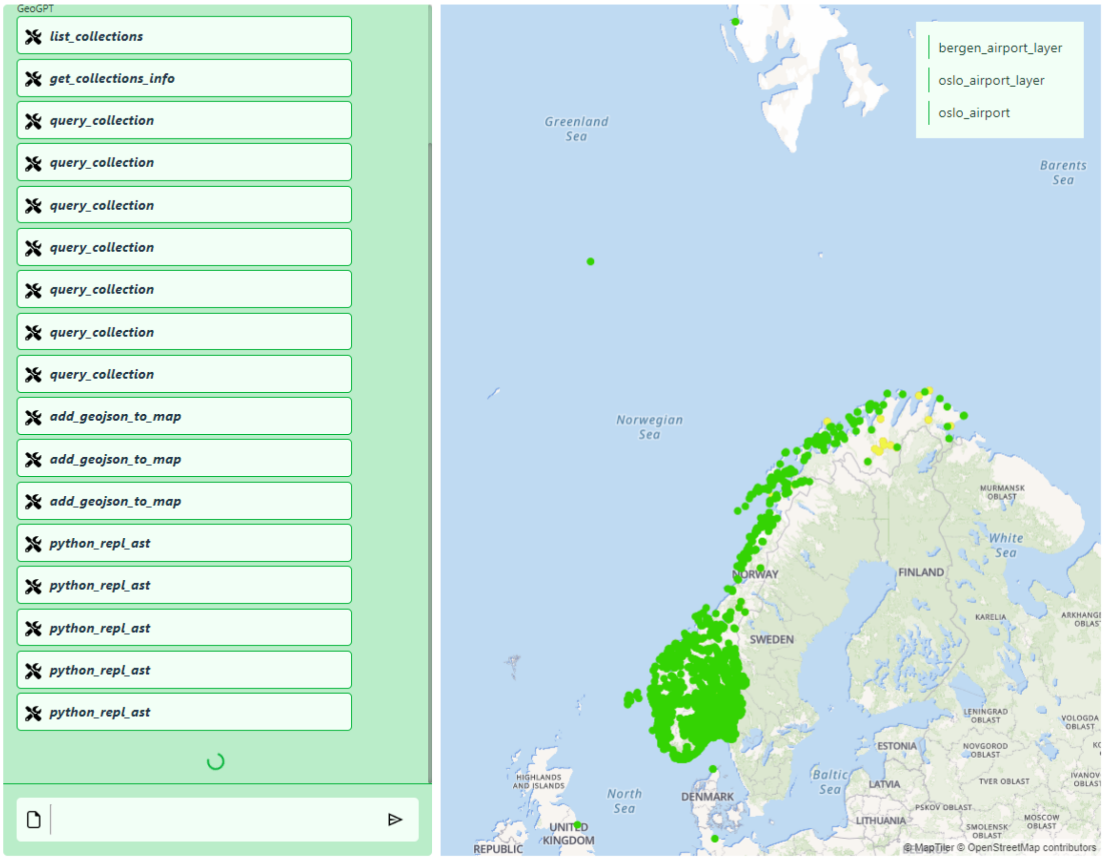


Figure 4.9.: Unsuccessful attempt by GeoGPT's OGC API Features agent to create a geodesic line between Oslo Airport Gardermoen and Bergen Airport Flesland. GeoGPT is has added datasets, assuming they contain airport point data, when in reality the contained completely different data for the whole of Norway. The result should have been a geodesic line between Gardermoen (in the south-east of Norway) and Flesland (in the south-west of Norway).

```
{
  "collection_name": "public.osm_transport_points",
  "cql_filter": "fclass='airport' AND name='Oslo Airport'",
  "layer_name": "oslo_airport"
}

--> No features were found at http://localhost:9001/collections/public.osm_transport_points/items?limit=10000&filter=fclass='airport' AND name='Oslo Airport'.
Try to change the parameters, or make them less restrictive.
```

Code Snippet 4.6: Tool invocation of `query_collection` with no results. This is because there are no features named "Oslo Airport" in the `public.osm_transport_points` collection. The JSON contains the parameters sent to the tool, and the text after the "–>" is the string returned from the tool.

```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 4.7: GeoGPT-generated Python code that picks the first features two collections including various point data in the *hope* that they correcspond to Gardermoen and Flesland. This attempt was of course insuccessful

4.3.2. Prompt Quality Experiment — Results

Figure 4.10 shows the outcomes of the experiments where the importance of the quality of the user's initial prompt was assessed. It is clear to see that *expert*-level prompting significantly outperforms *novice*-level prompting, the latter of which produced *no* fully successful responses. It is worth noting, however, that the questions picked were the three hardest ones from the GIS benchmarking experiment.

Figure 4.11 compares the responses that GeoGPT's OGC API Features agent managed to produce for the different prompt levels on one of the tasks. The *novice*-level prompt was as follows:

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

The *expert*-level prompt 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."

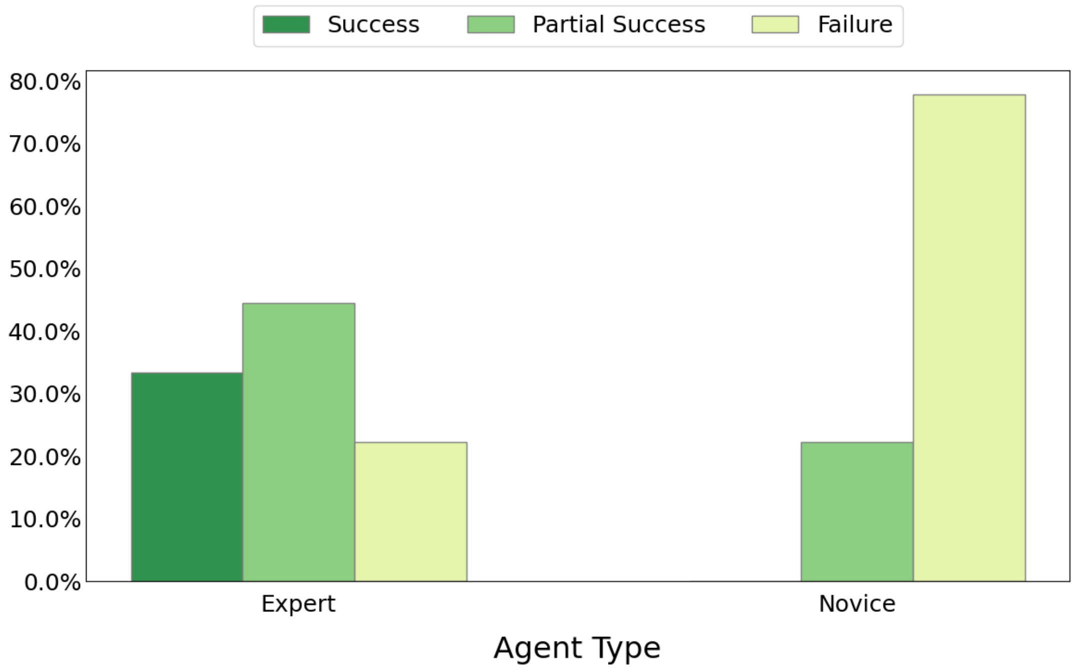
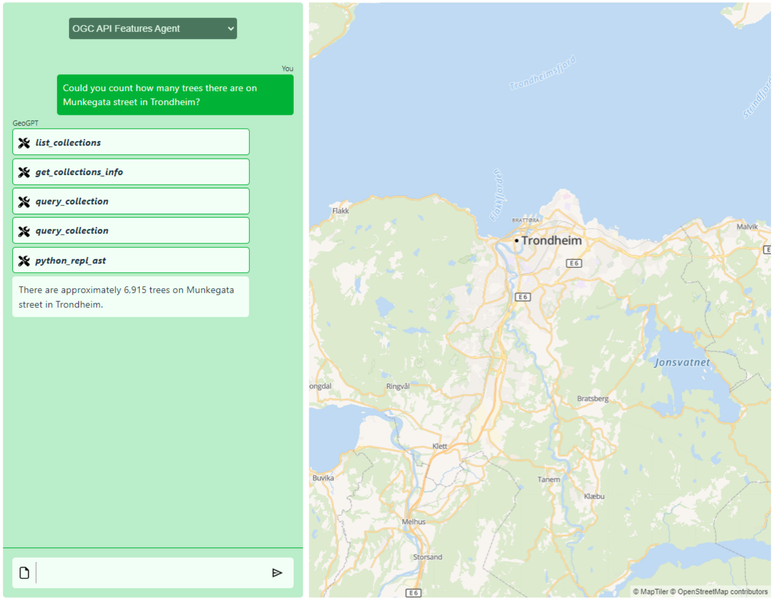


Figure 4.10.: Outcome distribution for different levels of GIS experience. Using novice-level prompting, GeoGPT failed to complete the tasks consistently, but using the more elaborate expert-level prompt, GeoGPT performed much better.

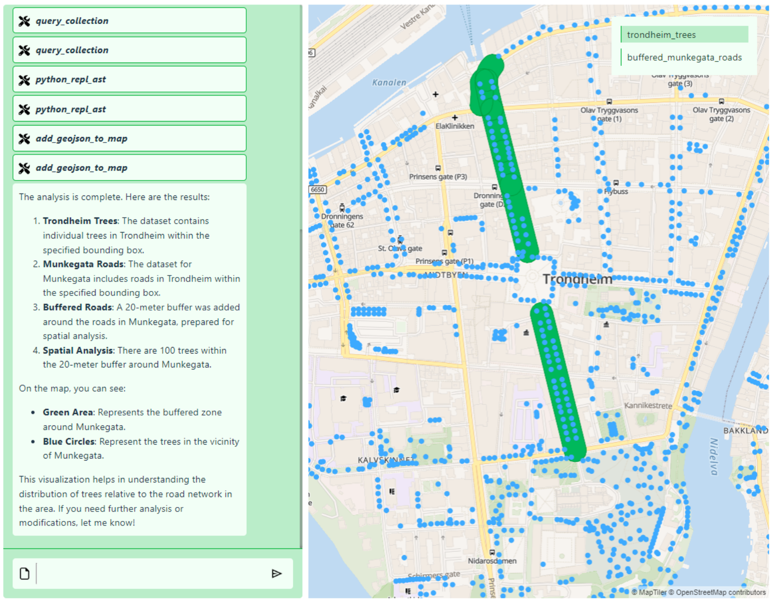
Using the novice-level prompt GeoGPT was unable to produce the correct outcome, and confidently answered that there are “approximately 6,915 trees on Munkegata street in Trondheim”, which is far from being true. When solving the task, GeoGPT made a series of oversights that lead to this result. First, it failed to take into account that there may be more than one street in the dataset named “Munkegata”, missing the important step of applying a bounding box filter to the query. A similar 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 lacked a bounding box, resulting in a randomly distributed subset of 10,000 trees 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.

The expert-level prompt, on the other hand, provided the necessary guidance for GeoGPT for this specific task, steering it clear of the issues it encountered with the novice-level prompt. As OpenAI themselves stated, “some tasks are best specified as a sequence of steps”.[9] Furthermore, they say that writing explicit steps required to solve a task “makes it easier for the model to follow them”.

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



(a) Novice-level prompting



(b) Expert-level prompting

Figure 4.11.: 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. Using the novice-level prompt resulted, GeoGPT found 6,915 trees (too many), but using the expert-level prompt, it was able to solve task correctly and add appropriate layers to the map.

5. Discussion

Section 5.1 will open the Discussion chapter by suggesting a place for an application like GeoGPT in the field of GIS. Thereafter, section 5.2 will present possible reasons as to why the SQL agent outperforms the other two agent types. Section 5.3 will discuss problem encountered during the development of GeoGPT. Finally, section 5.4 will discuss a semi-failed attempt at implementing a multi-agent architecture for GeoGPT, and provide suggestions as to why this didn't succeed.

5.1. GeoGPT's Place in the Field of GIS

As stated in the section on Goals and Research Questions, one of the goals of this thesis is to see if LLM-based systems can replace GIS professionals. Results from experiments comparing prompts designed to emulate expert and novice users, as detailed in subsection 4.3.2, indicate that the role of GIS professionals is not immediately threatened. As Figure 4.11 shows, expert-level prompting is highly necessary to make GeoGPT consistently produce the desired outcomes 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 GIS professionals. The benchmarking results 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 (see section A.1) are not very technically phrased, showing that expert-level prompting is not necessary for simpler tasks.

By using a Large Language Model in a GIS context one can also take advantage of their built-in geographical awareness. This geospatial awareness was demonstrated by Roberts et al. (2023) (see subsection 2.4.1 for more details). The knowledge within 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 was able to create quite accurate bounding boxes for several different Norwegian cities, and it was also able to retrieve very accurate coordinates for places like Aker brygge in Oslo. This became useful in 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.

On the other hand, the results show that the randomness of LLMs gives reason to

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

doubt the answers that GeoGPT produces. Table 4.4 shows GeoGPT's answers are not always consistent. A common problem with many LLMs is that they will often deliver overly confident answers in response to questions they do not know the answer to. This is problematic, 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 as GeoGPT was asked which (Norwegian) county is the largest by size (see section A.1). This question was asked a total of nine times across different agents and was answered incorrectly four times (see Table A.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 GIS 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.

5.2. Why Does the SQL Agent Outperform the Others?

This section will possible reasons for the superior performance of the SQL agent of GeoGPT. The experimental results presented in section 4.3 clearly show that GeoGPT's SQL agent outperform both the OGC API Features agent and the Python agent. As Figure 4.2 in subsection 4.3.1 shows.

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

A possible reason for the performance gap between the SQL and the other two, 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 is a less established technology. As of 31st May 2024, Google Scholar returns $\sim$ 22.600 results for "PostGIS" compared to $\sim$ 3.650 for "GeoPandas". This large difference in search results is likely correlated with the respective prevalences 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).

[2] https://en.wikipedia.org/wiki/PostGIS
[3] https://geopandas.org/en/stable/

5.2.2. Limitations with OGC API Features

A common source of error with several of the tests conducted with the OGC API Features agent was its inability to fetch more than 10,000 features from the server. The limit of 10,000 features is specified in the OGC API Features standard (Open Geospatial Consortium, 2022), which states 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 31st 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 agent in GeoGPT.

Furthermore, the lacking support for multi-collection queries is, in the author's opinion, a big limitation to the current Features specification. A proposal draft to such features have been created,[5] but it is unclear whether this will be 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 5.1 shows one of the multi-collection query examples included in the proposal draft. The ability to construct such queries could make retrieval of features much more efficient, possibly making the Python tool in redundant to GeoGPT's OGC API Features agent. The query in Code Snippet 5.1 would not be possible using the current specification, and one would have to download the two collections, load them into memory using Python, and perform the `contains` operation there. Using multi-collection queries that are converted into PostGIS queries, which are generally more efficient than their Python equivalents, could also speed up analysis.

```
\\ SQL query for fetching lakes within Algonquin Park
SELECT lakes.*
FROM lakes
JOIN parks ON ST_Intersects(lakes.geometry, parks.geometry)
WHERE parks.name = 'Algonquin Park';

\\ Corresponding CQL query (would return a tuple of parks and lakes)
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"},
```

[4]https://github.com/CrunchyData/pg_featureserv/blob/master/FEATURES.md
[5]https://github.com/opengeospatial/ogcapi-features/tree/master/proposals/search

```
                                {"property": "lakes.geometry"}]}
                ]
            }
        }
]
```

Code Snippet 5.1: Multi-collection CQL query using the *Search* extension proposed for the OGC API Features specification

5.3. Where GeoGPT Struggles

This section will present two issues emerged from the Experiments. Subsection 5.3.1 will address an issue that occurs when GeoGPT gets stuck, trying to solve a task by repeatedly making the same, unsuccessful attempts. Subsection 5.3.2 will discuss GeoGPT's *self-verification* abilities.

5.3.1. Walking Into Dead Ends

As pointed out in subsection 4.3.1 (GIS Benchmark — Results), the correlation matrices in Figure 4.5 show that the encoded outcome has a negative correlation with each of the following variables: token usage, duration, and total cost. This suggests that tasks requiring more time to solve are more likely to result in unsuccessful outcomes. The tendency of GeoGPT of getting stuck in "dead ends" might be an explanation for this. The example in Figure 4.9 shows this tendency well: GeoGPT tries to query the same collection many times in more or less the same way, 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, regarding 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 options from earlier in the context window. GeoGPT's endless responses 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.

5.3.2. 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 mid-conversation 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 data that is different to what it believes it to be. An example of this issue was presented in the section on unsuccessful outcomes in subsection 4.3.1, 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 5.1 shows how 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.

5.4. 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 data analysis, and one for map interaction. Each agent will a collection of tools, which are relevant for the sort of tasks they will be asked to solve. These agents are orchestrated by a *Supervisor* agent, which takes input from the user and assigns tasks to the appropriate sub-agents. Figure 5.2 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. Inspired by Microsoft's *AutoGen* framework[6] — which provides a high-level abstraction for developing multi-agent conversations — and MetaGPT (further discussed in subsection 2.4.2).

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 tools available to the agent 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/

Figure 5.1.: ChatGPT's multi-modal GPT-4 correctly identifying that the map layer intended to show high-speed roads in Oslo in fact "extends far beyond Oslo"

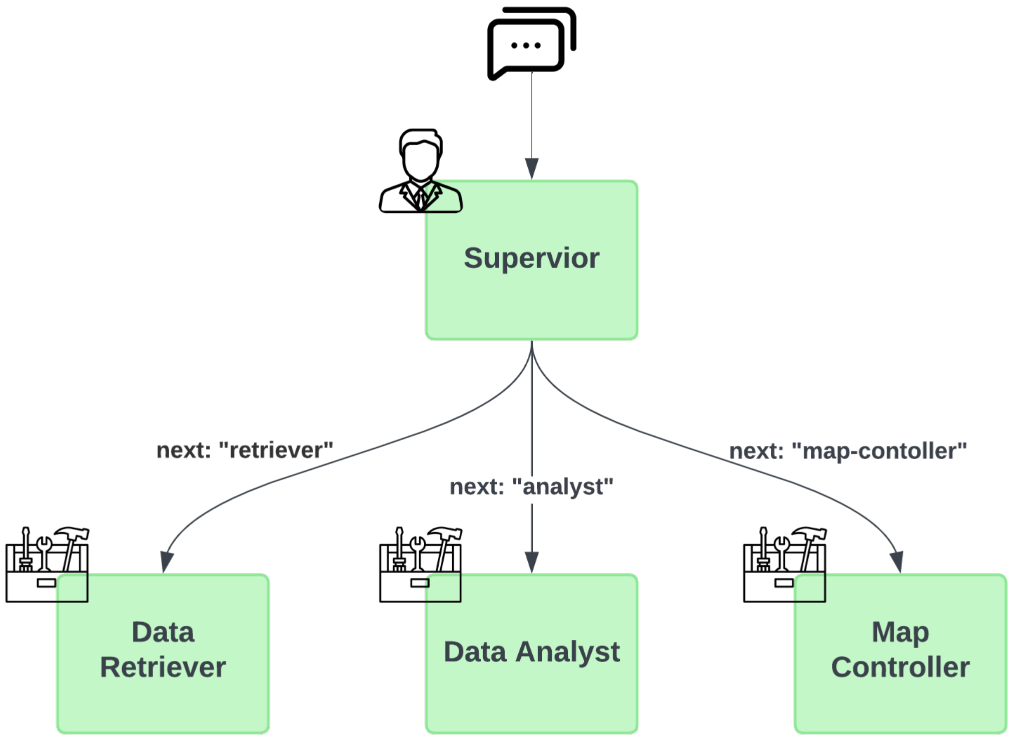


Figure 5.2.: Illustration for multi-agent implementation for GeoGPT's OGC API Features agent, where an agent supervisor takes in a user message and selects which sub-agent is to solve the next sub-task

6. Future Work

The Future Work chapter will present challenges that are reserved for future research. Section 6.1 addresses limitations with the experiments conducted in this thesis, and emphasizes the need for experiments that investigate the ability of a system like GeoGPT to answer questions where there is no concrete answers, and where ethical considerations and assumptions must be made. Section 6.2 highlight the need for testing different models for the purpose of creating an LLM-based GIS. Section 6.3 discusses the issue with data discovery. Currently, to use GeoGPT, the user has to provide it with data through, for instance, a geospatial database, but not all users will have the technical know-how required to set up such a database themselves.

6.1. Ability to Answer Questions with no Clear Answer

The experiments conducted in this thesis focused on the technical GIS abilities of the system, and 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 decide to incorporate weather forecasts into the analysis? Future research should find methods of measuring the ability of LLM-based GIS agents to provide suitable, and safe, answers to such questions.

6.2. 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 be 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 for open-source models going into 2024 (Wang, Sarah and Xu, Shangda, 2024). *Control* and *customizability* 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.

6.3. Automated Data Discovery

The experiments in this thesis were based upon a pre-existing collection of geospatial datasets that were made available to GeoGPT through different channels. GeoGPT is currently reliant on having the user provide data either as a geospatial database, through an OGC API Features endpoint, or as a collection of geospatial files that can be manipulated through Python code. 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 location. 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. Initial experiments were conducted towards Geonorge in this thesis to see if this was possible, but results were inconsistent.

7. Conclusion

This thesis has shown the viability of using Large Language Model (LLM) technology to create autonomous agents aimed at 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 can type geospatial questions into the chat interface, that GeoGPT will attempt answer to through text and/or by adding geometries to the map.

GeoGPT relies heavily on *function calling*, a way of giving function/tool definitions to an LLM, enabling it to essentially *invoke* these tools through specifying the name of the tool and suitable parameters that will be passed to it. The tools run code created by the developer.

Featuring three different agent types, GeoGPT shows that it can manipulate geospatial data that are discovered in different ways. One agent accesses data directly from a PostGIS database, another agent through an OGC API Features endpoint that lives on top of this PostGIS database, and the final agent by having access to shapefiles in its local environment. All agents have access to the exact same data. The agents have different sets of tools that allow them to work with their data. The SQL agent has (amongst other) a tool that takes a string of SQL code that will be run against the database, enabling 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 to this, the former has access to functions/tools work against the OGC API Features endpoint.

Two sets of experiments were conducted. The first compares the three agent types to see which is best at solving common GIS-related tasks. Restuls from this experiment 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 formulated two in different ways: one simple problem formulation, resembling a user with little GIS experience, and another more accurate and detailed formulation, resembling a user with great GIS experience. The results from the experiment show that providing GeoGPT with a better (more accurate and detailed) initial problem greatly increases the likelyhood 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.

Bibliography

Anthropic. (2023). Model Card and Evaluations for Claude Models. https://www-cdn.anthropic.com/bd2a28d2535bfb0494cc8e2a3bf135d2e7523226/Model-Card-Claude-2.pdf

Anthropic. (2024). *The Claude 3 Model Family: Opus, Sonnet, Haiku* (tech. rep.). https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf

Christiano, P., Leike, J., Brown, T. B., Martic, M., Legg, S., & Amodei, D. (2023). Deep reinforcement learning from human preferences. https://doi.org/10.48550/arXiv.1706.03741

CrunchyData. (2024). CrunchyData/pg_featureserv. Retrieved April 19, 2024, from https://github.com/CrunchyData/pg_featureserv

Eleti, A., Harris, J., & Kilpatrick, L. (2023). Function calling and other API updates. Retrieved March 10, 2024, from https://openai.com/blog/function-calling-and-other-api-updates

Gemini Team, Anil, R., Borgeaud, S., Alayrac, J.-B., Yu, J., Soricut, R., Schalkwyk, J., Dai, A. M., Hauth, A., Millican, K., Silver, D., Johnson, M., Antonoglou, I., Schrittwieser, J., Glaese, A., Chen, J., Pitler, E., Lillicrap, T., Lazaridou, A., Firat, O., Molloy, J., Isard, M., Barham, P. R., Hennigan, T., Lee, B., Viola, F., Reynolds, M., Xu, Y., Doherty, R., Collins, E., Meyer, C., Rutherford, E., Moreira, E., Ayoub, K., Goel, M., Krawczyk, J., Du, C., Chi, E., Cheng, H.-T., Ni, E., Shah, P., Kane, P., Chan, B., Faruqui, M., Severyn, A., Lin, H., Li, Y., Cheng, Y., Ittycheriah, A., Mahdieh, M., Chen, M., Sun, P., Tran, D., Bagri, S., Lakshminarayanan, B., Liu, J., Orban, A., Güra, F., Zhou, H., Song, X., Boffy, A., Ganapathy, H., Zheng, S., Choe, H., Weisz, Á., Zhu, T., Lu, Y., Gopal, S., Kahn, J., Kula, M., Pitman, J., Shah, R., Taropa, E., Merey, M. A., Baeuml, M., Chen, Z., Shafey, L. E., Zhang, Y., Sercinoglu, O., Tucker, G., Piqueras, E., Krikun, M., Barr, I., Savinov, N., Danihelka, I., Roelofs, B., White, A., Andreassen, A., von Glehn, T., Yagati, L., Kazemi, M., Gonzalez, L., Khalman, M., Sygnowski, J., Frechette, A., Smith, C., Culp, L., Proleev, L., . . . Vinyals, O. (2024a). Gemini: A Family of Highly Capable Multimodal Models. https://doi.org/10.48550/arXiv.2312.11805

Gemini Team, Reid, M., Savinov, N., Teplyashin, D., Dmitry, Lepikhin, Lillicrap, T., Alayrac, J.-b., Soricut, R., Lazaridou, A., Firat, O., Schrittwieser, J., Antonoglou, I., Anil, R., Borgeaud, S., Dai, A., Millican, K., Dyer, E., Glaese, M., Sottiaux, T., Lee, B., Viola, F., Reynolds, M., Xu, Y., Molloy, J., Chen, J., Isard, M., Barham, P., Hennigan, T., McIlroy, R., Johnson, M., Schalkwyk, J., Collins, E., Rutherford, E., Moreira, E., Ayoub, K., Goel, M., Meyer, C., Thornton, G., Yang, Z., Michalewski, H., Abbas, Z., Schucher, N., Anand, A., Ives, R., Keeling, J., Lenc, K., Haykal, S., Shakeri, S., Shyam, P., Chowdhery, A., Ring, R., Spencer, S., Sezener, E., Vilnis, L., Chang, O., Morioka, N., Tucker, G., Zheng, C., Woodman, O., Attaluri, N., Kocisky, T., Eltyshev, E., Chen, X., Chung, T., Selo, V., Brahma, S., Georgiev, P., Slone, A., Zhu, Z., Lottes, J., Qiao, S., Caine, B., Riedel, S., Tomala, A., Chadwick, M., Love, J., Choy, P., Mittal, S., Houlsby, N., Tang, Y., Lamm, M., Bai, L., Zhang, Q., He, L., Cheng, Y., Humphreys, P., Li, Y., Brin, S., Cassirer, A., Miao, Y., Zilka, L., Tobin, T., Xu, K., Proleev, L., Sohn, D., Magni, A., Hendricks, L. A., . . . Vinyals, O. (2024b). Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. https://doi.org/10.48550/arXiv.2403.05530

Gemma Team, Mesnard, T., Hardin, C., Dadashi, R., Bhupatiraju, S., Pathak, S., Sifre, L., Rivière, M., Kale, M. S., Love, J., Tafti, P., Hussenot, L., Sessa, P. G., Chowdhery, A., Roberts, A., Barua, A., Botev, A., Castro-Ros, A., Slone, A., Héliou, A., Tacchetti, A., Bulanova, A., Paterson, A., Tsai, B., Shahriari, B., Lan, C. L., Choquette-Choo, C. A., Crepy, C., Cer, D., Ippolito, D., Reid, D., Buchatskaya, E., Ni, E., Noland, E., Yan, G., Tucker, G., Muraru, G.-C., Rozhdestvenskiy, G., Michalewski, H., Tenney, I., Grishchenko, I., Austin, J., Keeling, J., Labanowski, J., Lespiau, J.-B., Stanway, J., Brennan, J., Chen, J., Ferret, J., Chiu, J., Mao-Jones, J., Lee, K., Yu, K., Millican, K., Sjoesund, L. L., Lee, L., Dixon, L., Reid, M., Mikuła, M., Wirth, M., Sharman, M., Chinaev, N., Thain, N., Bachem, O., Chang, O., Wahltinez, O., Bailey, P., Michel, P., Yotov, P., Chaabouni, R., Comanescu, R., Jana, R., Anil, R., McIlroy, R., Liu, R., Mullins, R., Smith, S. L., Borgeaud, S., Girgin, S., Douglas, S., Pandya, S., Shakeri, S., De, S., Klimenko, T., Hennigan, T., Feinberg, V., Stokowiec, W., Chen, Y.-h., Ahmed, Z., Gong, Z., Warkentin, T., Peran, L., Giang, M., Farabet, C., Vinyals, O., Dean, J., Kavukcuoglu, K., Hassabis, D., . . . Kenealy, K. (2024). Gemma: Open Models Based on Gemini Research and Technology. https://doi.org/10.48550/arXiv.2403.08295

Holm, O. (2023). *LLMs - The Death of GIS Analysis?* (Specialization Project). NTNU. Trondheim. https://kartai.no/wp-content/uploads/2024/01/Holm-2023-LLMs-The-Death-of-GIS-Analysis.pdf

Holmes, C. (2021). SpatioTemporal Asset Catalogs and the Open Geospatial Consortium. Retrieved May 2, 2024, from https://medium.com/radiant-earth-insights/spatiotemporal-asset-catalogs-and-the-open-geospatial-consortium-659538dce5c7

Hong, S., Zhuge, M., Chen, J., Zheng, X., Cheng, Y., Zhang, C., Wang, J., Wang, Z., Yau, S. K. S., Lin, Z., Zhou, L., Ran, C., Xiao, L., Wu, C., & Schmidhuber, J. (2023). MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. https://doi.org/10.48550/arXiv.2308.00352

Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., de las Casas, D., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., Lavaud, L. R., Lachaux, M.-A., Stock, P., Scao, T. L., Lavril, T., Wang, T., Lacroix, T., & Sayed, W. E. (2023). Mistral 7B. Retrieved May 3, 2024, from http://arxiv.org/abs/2310.06825

Jiang, A. Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., Chaplot, D. S., de las Casas, D., Hanna, E. B., Bressand, F., Lengyel, G., Bour, G., Lample, G., Lavaud, L. R., Saulnier, L., Lachaux, M.-A., Stock, P., Subramanian, S., Yang, S., Antoniak, S., Scao, T. L., Gervet, T., Lavril, T., Wang, T., Lacroix, T., & Sayed, W. E. (2024). Mixtral of Experts. Retrieved May 3, 2024, from http://arxiv.org/abs/2401.04088

LangChain AI. (2022). Langchain-ai/langchain. Retrieved October 5, 2023, from https://github.com/langchain-ai/langchain

LangChain AI. (2024). Langchain-ai/langgraph. Retrieved March 21, 2024, from https://github.com/langchain-ai/langgraph

Li, Z., & Ning, H. (2023). Autonomous GIS: The next-generation AI-powered GIS. https://doi.org/10.48550/arXiv.2305.06453

Lin, Z., Trivedi, S., & Sun, J. (2023). Generating with Confidence: Uncertainty Quantification for Black-box Large Language Models. https://doi.org/10.48550/arXiv.2305.19187

Meta AI. (2024). Introducing Meta Llama 3: The most capable openly available LLM to date. Retrieved May 3, 2024, from https://ai.meta.com/blog/meta-llama-3/

Mooney, P., Cui, W., Guan, B., & Juhász, L. (2023). *Towards Understanding the Geospatial Skills of ChatGPT: Taking a Geographic Information Systems (GIS) Exam.* https://doi.org/10.1145/3615886.3627745

Moura, J. (2024). Joaomdmoura/crewAI. Retrieved May 21, 2024, from https://github.com/joaomdmoura/crewAI

Open Geospatial Consortium. (2022). OGC API - Features - Part 1: Core corrigendum. Retrieved April 29, 2024, from https://docs.ogc.org/is/17-069r4/17-069r4.html

OpenAI. (2022). Introducing ChatGPT. Retrieved October 26, 2023, from https://openai.com/blog/chatgpt

Peysakhovich, A., & Lerer, A. (2023). Attention Sorting Combats Recency Bias In Long Context Language Models. Retrieved May 14, 2024, from http://arxiv.org/abs/2310.01427

Pichai, S., & Hassabis, D. (2024). Our next-generation model: Gemini 1.5. Retrieved May 3, 2024, from https://blog.google/technology/ai/google-gemini-next-generation-model-february-2024/

PostGIS. (2001). Retrieved April 29, 2024, from https://postgis.net/

Radford, A., Narasimhan, K., Salimans, T., & Sutskever, I. (2018). Improving Language Understanding by Generative Pre-Training.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). Language Models are Unsupervised Multitask Learners.

Richard, T. B. (2023). AutoGPT: The heart of the open-source agent ecosystem. Retrieved October 5, 2023, from https://github.com/Significant-Gravitas/AutoGPT

Roberts, J., Lüddecke, T., Das, S., Han, K., & Albanie, S. (2023). GPT4GEO: How a Language Model Sees the World's Geography. https://doi.org/10.48550/arXiv.2306.00020

Sanfilippo, S. (2009). Redis - The Real-time Data Platform. Retrieved April 19, 2024, from https://redis.io/

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal Policy Optimization Algorithms. https://doi.org/10.48550/arXiv.1707.06347

Shinn, N., Cassano, F., Berman, E., Gopinath, A., Narasimhan, K., & Yao, S. (2023). Reflexion: Language Agents with Verbal Reinforcement Learning. Retrieved May 13, 2024, from http://arxiv.org/abs/2303.11366

Su, J., Ahmed, M., Lu, Y., Pan, S., Bo, W., & Liu, Y. (2024). RoFormer: Enhanced transformer with Rotary Position Embedding. *Neurocomputing*, *568*, 127063. https://doi.org/10.1016/j.neucom.2023.127063

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention Is All You Need. Retrieved October 10, 2023, from https://arxiv.org/abs/1706.03762v7

Wang, Sarah & Xu, Shangda. (2024). 16 Changes to the Way Enterprises Are Building and Buying Generative AI. Retrieved May 9, 2024, from https://a16z.com/generative-ai-enterprise-2024/

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E. H., Le, Q. V., & Zhou, D. (2023). Chain-of-Thought Prompting Elicits Reasoning in Large Language Models.

White, J., Fu, Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J., & Schmidt, D. C. (2023). A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT. Retrieved May 6, 2024, from http://arxiv.org/abs/2302.11382

Wu, Q., Bansal, G., Zhang, J., Wu, Y., Li, B., Zhu, E., Jiang, L., Zhang, X., Zhang, S., Liu, J., Awadallah, A. H., White, R. W., Burger, D., & Wang, C. (2023). AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. https://doi.org/10.48550/arXiv.2308.08155

Yan, Fanjia, Mao, Huanzhi, Ji, Charlie Cheng-Jie, Stoica, Ion, Gonzalez, Joseph E., Zhang, Tianjun & Patil, Shishir G. (2024). Berkeley Function Calling Leaderboard. Retrieved May 9, 2024, from https://gorilla.cs.berkeley.edu/blogs/8_berkeley_function_calling_leaderboard.html#metrics

Zhang, Y., Wei, C., Wu, S., He, Z., & Yu, W. (2023). GeoGPT: Understanding and Processing Geospatial Tasks through An Autonomous GPT. https://doi.org/10.48550/arXiv.2307.07930

Zhu, D., Yang, N., Wang, L., Song, Y., Wu, W., Wei, F., & Li, S. (2024). PoSE: Efficient Context Window Extension of LLMs via Positional Skip-wise Training. https://doi.org/10.48550/arXiv.2309.10400

Appendices

A. Experiments

A.1. GIS Benchmark

Table A.1.: Questions for GIS benchmarking experiment

Query ID	Query	Correct Response
aker_brygge_national	Which is the closest railway station to Aker brygge?	Nationalteateret
cliff_clusters	Locate clusters of cliffs, each containing more than 10 cliffs, with cliffs within each cluster no more than 0.1 degrees apart.	Should be about 8-9 clusters.
county_names	What are the names of the counties in found in the data?	Nordland, Telemark, Troms, Rogaland, Vestland, Trøndelag, Vestfold, Buskerud, Akershus, Østfold, Innlandet, Møre og Romsdal, Agder Finnmark
glomma_counties	How many counties does Glomma run through?	4. Trøndelag, Innlandet, Akershus, and Østfold.
largest_county	Which is the largest county by size?	Nordland
nidarosdomen_polygon	Retrieve a polygon of Nidarosdomen.	Adding a polygon of Nidarosdomen to the map.
num_trees_munkegata	How many trees are there along Munkegata in Trondheim?	Giving the correct number of trees (about 70-80).

Continued on next page

Table A.1.: Questions for GIS benchmarking experiment

Query ID	Query	Correct Response
oslo_bergen_geodesic	Create a geodesic curve between the airports of Oslo and Bergen.	A geodesic, slightly curved line between Gardermoen and Flesland.
oslo_residental_diff	Provide an outline of Oslo but exclude residential areas by computing their difference.	The polygonal outline of Oslo with cutouts where there area areas classified as residental.
oslo_roads_gte_70_-kmh	Retrieve roads in Oslo that have speed limit higher than or equal to 70 km/h.	Adding corresponding line segments to the map.
vestfold_bbox	Find the coordinates that define a the extent of a rectangular bounding box that encompasses the county of Vestfold.	(9.7553357, 58.720455, 10.6750198, 59.674011)
viken_dissolve	Combine the countries that correspond to the historical country of Viken into a single feature.	Display the outline of Viken, based on the outlines of Buskerud, Akershus, and Østfold, that are found in the data. Should be dissolved, i.e., no borders.

Table A.2.: Results from GIS benchmakring experiment

Query ID	**Agent Type**	**Outcome**	**Duration [s]**	**Tokens**
aker_brygge_national	oaf	partial success	87.16	3347
aker_brygge_national	oaf	partial success	97.85	3042
aker_brygge_national	oaf	partial success	70.36	4563
aker_brygge_national	python	partial success	68.10	4736
aker_brygge_national	python	partial success	49.48	2192
aker_brygge_national	python	partial success	76.84	4270
aker_brygge_national	sql	success	63.51	2569
aker_brygge_national	sql	success	151.46	6273
aker_brygge_national	sql	success	103.98	5169
cliff_clusters	oaf	partial success	93.21	3047
cliff_clusters	oaf	partial success	134.62	3713
cliff_clusters	oaf	failure	253.77	6956
cliff_clusters	python	failure	66.42	5421
cliff_clusters	python	failure	125.96	4650
cliff_clusters	python	failure	109.29	4768
cliff_clusters	sql	failure	39.36	1633
cliff_clusters	sql	failure	75.27	3085
cliff_clusters	sql	success	31.64	1632

Continued on next page

Table A.2.: Results from GIS benchmakring experiment

Query ID	Agent Type	Outcome	Duration [s]	Tokens
county_names	oaf	success	78.08	2389
county_names	oaf	success	67.45	2399
county_names	oaf	failure	20.23	1471
county_names	python	success	44.33	2532
county_names	python	success	30.50	2355
county_names	python	success	31.09	2355
county_names	sql	success	64.21	1874
county_names	sql	success	52.28	1880
county_names	sql	success	47.62	1886
glomma_counties	oaf	success	69.73	2253
glomma_counties	oaf	success	65.57	1945
glomma_counties	oaf	success	67.37	2397
glomma_counties	python	success	663.22	3658
glomma_counties	python	failure	295.93	4388
glomma_counties	python	success	285.23	2281
glomma_counties	sql	success	102.48	1828
glomma_counties	sql	success	25.01	1266
glomma_counties	sql	success	21.36	1064
largest_county	oaf	success	41.82	1713
largest_county	oaf	success	71.82	2223
largest_county	oaf	failure	47.00	1797
largest_county	python	failure	29.37	2415
largest_county	python	failure	41.95	2810
largest_county	python	failure	40.55	2407

Continued on next page

Table A.2.: Results from GIS benchmakring experiment

Query ID	Agent Type	Outcome	Duration [s]	Tokens
largest_county	sql	success	40.08	1771
largest_county	sql	success	28.51	1474
largest_county	sql	success	38.30	1381
nidarosdomen_polygon	oaf	success	29.82	1923
nidarosdomen_polygon	oaf	success	31.73	1909
nidarosdomen_polygon	oaf	success	31.61	1922
nidarosdomen_polygon	python	success	35.10	4510
nidarosdomen_polygon	python	success	25.36	3649
nidarosdomen_polygon	python	success	33.35	3663
nidarosdomen_polygon	sql	success	29.64	1546
nidarosdomen_polygon	sql	success	90.61	1529
nidarosdomen_polygon	sql	success	39.39	1513
num_trees_munkegata	oaf	failure	75.60	2977
num_trees_munkegata	oaf	failure	80.13	2984
num_trees_munkegata	oaf	failure	57.66	2407
num_trees_munkegata	python	partial success	122.99	3930
num_trees_munkegata	python	failure	916.13	2362
num_trees_munkegata	python	failure	911.18	8385
num_trees_munkegata	sql	failure	80.41	1359
num_trees_munkegata	sql	failure	67.41	2626
num_trees_munkegata	sql	failure	31.86	1359
oslo_bergen_geodesic	oaf	failure	652.88	9253
oslo_bergen_geodesic	oaf	failure	102.38	3360

Continued on next page

Table A.2.: Results from GIS benchmakring experiment

Query ID	**Agent Type**	**Outcome**	**Duration [s]**	**Tokens**
oslo_bergen_geodesic	oaf	failure	258.52	6271
oslo_bergen_geodesic	python	partial success	118.54	6683
oslo_bergen_geodesic	python	partial success	129.40	7782
oslo_bergen_geodesic	python	failure	373.37	10412
oslo_bergen_geodesic	sql	failure	97.79	2710
oslo_bergen_geodesic	sql	failure	73.78	2575
oslo_bergen_geodesic	sql	failure	127.43	3318
oslo_residental_diff	oaf	failure	119.39	2373
oslo_residental_diff	oaf	partial success	114.92	3033
oslo_residental_diff	oaf	success	121.17	2621
oslo_residental_diff	python	success	673.96	4013
oslo_residental_diff	python	failure	2190.08	3867
oslo_residental_diff	python	failure	2290.09	3650
oslo_residental_diff	sql	success	42.49	1537
oslo_residental_diff	sql	failure	94.35	2485
oslo_residental_diff	sql	success	52.14	1790
oslo_roads_gte_70_kmh	oaf	partial success	44.44	1959
oslo_roads_gte_70_kmh	oaf	failure	58.61	1989
oslo_roads_gte_70_kmh	oaf	partial success	35.55	1961
oslo_roads_gte_70_kmh	python	failure	983.76	3912

Continued on next page

Table A.2.: Results from GIS benchmakring experiment

Query ID	**Agent Type**	**Outcome**	**Duration [s]**	**Tokens**
oslo_roads_gte_70_kmh	python	failure	757.70	3819
oslo_roads_gte_70_kmh	python	failure	849.06	3827
oslo_roads_gte_70_kmh	sql	success	43.88	1542
oslo_roads_gte_70_kmh	sql	success	37.26	1577
oslo_roads_gte_70_kmh	sql	success	38.38	1566
vestfold_bbox	oaf	failure	56.33	2001
vestfold_bbox	oaf	partial success	50.38	3177
vestfold_bbox	oaf	success	55.82	2480
vestfold_bbox	python	success	24.08	1612
vestfold_bbox	python	success	33.63	2366
vestfold_bbox	python	success	42.36	2808
vestfold_bbox	sql	success	23.76	1188
vestfold_bbox	sql	success	47.35	1932
vestfold_bbox	sql	success	23.30	1182
viken_dissolve	oaf	success	52.24	2336
viken_dissolve	oaf	success	204.07	5078
viken_dissolve	oaf	failure	133.40	3985
viken_dissolve	python	success	68.23	6099
viken_dissolve	python	success	86.32	6007
viken_dissolve	python	partial success	89.98	6461
viken_dissolve	sql	partial success	34.58	1553
viken_dissolve	sql	success	34.72	1540

Continued on next page

Table A.2.: Results from GIS benchmakring experiment

Query ID	**Agent Type**	**Outcome**	**Duration [s]**	**Tokens**
viken_dissolve	sql	partial success	37.36	1543

A.2. Prompt Quality Experiment

Table A.3.: Questions for prompt quality experiment

Query ID	Level	Formulation
oslo_bergen_geodesic	novice	Please plot the shortest flight path on a map between Oslo and Bergen's airports.
oslo_bergen_geodesic	expert	1. Get info on all potentially relevant datasets. The airports are possibly stored as polygons in the available data. 2. Filter those datasets for airports and list the names of the airports. 3. Retrieve the geographic coordinates for Oslo Gardermoen Airport (OSL) and Bergen Flesland Airport (BGO) by filtering on the names you found. 4. If no name was found, try a different dataset and go back to step 2. 5. Utilize available tools to draw a geodesic curve that represents the shortest path on the earth's surface between these two points. 6. Present the findings with a map highlighting the largest county.
oslo_roads_gte_70_kmh	novice	Draw roads in Oslo where you can drive at least 70.
oslo_roads_gte_70_kmh	expert	1. Retrieve an outline of Oslo. 2. Calculate max/min lat/lon values for this bounding box, and use it to retrieve a subset of the road data. 3. Select road segments within the outline from step 1 that have a max speed >= 70. 4. Present the findings with a map highlighting the selected roads.
num_trees_munkegata	novice	Could you count how many trees there are on Munkegata street in Trondheim?

Continued on next page

Table A.3.: Questions for prompt quality experiment

Query ID	Level	Formulation
num_trees_munkegata	expert	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.

Table A.4.: Results from prompt quality experiment

Query ID	**Agent Type**	**Level**	**Outcome**
oslo_bergen_geodesic	sql	novice	failure
oslo_bergen_geodesic	oaf	novice	failure
oslo_bergen_geodesic	python	novice	failure
oslo_bergen_geodesic	sql	expert	failure
oslo_bergen_geodesic	oaf	expert	failure
oslo_bergen_geodesic	python	expert	partial success
oslo_roads_gte_70_kmh	sql	novice	partial success
oslo_roads_gte_70_kmh	oaf	novice	partial success
oslo_roads_gte_70_kmh	python	novice	failure
oslo_roads_gte_70_kmh	sql	expert	success
oslo_roads_gte_70_kmh	oaf	expert	partial success
oslo_roads_gte_70_kmh	python	expert	success
num_trees_munkegata	sql	novice	failure
num_trees_munkegata	oaf	novice	failure
num_trees_munkegata	python	novice	failure
num_trees_munkegata	sql	expert	partial success
num_trees_munkegata	oaf	expert	success
num_trees_munkegata	python	expert	partial success