**Karl Oskar Magnus Holm**

# LLMs - The Death of GIS Analysis?

## An Investigation into Using Large Language Models for GIS Data Analysis

Master Thesis in Computer Science and Geomatics, June 2024
Supervisor at NTNU: Hongchao Fan
External supervisors from Norkart: Alexander Salveson Nossum and Arild Nomeland

Department of Geomatics
Faculty of Engineering
Norwegian University of Science and Technology

fix double page thing if necessary

NTNU
Faculty of Engineering Science and Technology
Department of Civil and Environmental Engineering



**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

## Sammendrag

# Preface

Karl Oskar Magnus Holm
Trondheim, 6th May 2024

# Contents

# Acronyms

**AI** Artificial Intelligence.

**API** Application Programming Interface.

**AWS** Amazon Web Services.

**BERT** Bidirectional Encoder Representation from Transformers.

**BLEU** BiLingual Evaluation Understudy.

**CLI** Command Line Interface.

**CQL** Common Query Language.

**CRS** Coordinate Reference System.

**DAG** Directed Acyclic Graph.

**DBMS** DataBase Management System.

**EU** European Union.

**FAISS** Facebook AI Similarity Search.

**FKB** Felles KartdataBase.

**GAN** Generative Adverserial Network.

**GDAL** Geospatial Data Abstraction Library.

**GDPR** General Data Protection Regulation.

**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.

**ISO** International Organization for Standardization.

**JSON** JavaScript Object Notation.

**LATS** Language Agent Tree Search.

**LLaMA** Large Language Model Meta AI.

**LLM** Large Language Model.

**LSH** Locality-Sensitive Hashing.

**LSTM** Long Short-Term Memory.

**MBPP** Mostly Basic Python Programming.

**MLM** Masked Language Modelling.

**MMI** Maximum Mutual Information.

**MMLU** Multitask Language Understanding.

**NLG** Natural Language Generation.

**NLI** Natural Language Inference.

**NLIDB** Natural Language Interfaces for Database.

**NLP** Natural Language Processing.

**NLU** Natural Language Understanding.

**NSDI** National Spatial Data Infrastructure.

**NSP** Next Sentence Prediction.

**OGC** Open Geospatial Consortium.

**OSM** OpenStreetMap.

**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.

**ROUGE** Recall-Oriented Understudy for Gisting Evaluation.

**SDK** Software Development Kit.

**SMoE** Sparse Mixture-of-Experts.

**SOSI** Samordnet Opplegg for Stedfestet Informasjon.

**SQL** Structured Query Language.

**SSD** Solid-State-Disk.

**STAC** SpatioTemporal Asset Catalog.

**SWA** Sliding Window Attention.

**TC** Technical committee.

**UI** User Interface.

**WFS** Web Feature Service.

**WMS** Web Map Service.

**XML** Extensible Markup Language.

# List of Figures

# List of Tables

# List of Code Snippets

# 1. Introduction

The introductory chapter will explain the motivation behind the thesis, as well as its goals and the research questions it will attempt to answer. Section 8.1 will list the main contributions of the thesis, and section 1.5 will give a high-level overview over the thesis.

## 1.1. Background and Motivation

The release of OpenAI's ChatGPT in November, 2023 generated a hype within the general population and chat-based systems are flourishing. ChatGPT — or rather the underlying GPT-3 and GPT-4 models — is an example of how modern Large Language Models (LLMs) can provide a natural language interface between human and machine. Furthermore, significant advancements have been made within code generation, which essentially allows the LLM to execute computational tasks that can be defined within a snippet of code. Paired with the LLM's ability to often correctly interpret the user's intent regardless of the preciseness of their problem formulation, even individuals with no prior programming experience can carry out computational tasks that require the execution of code.

ref

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

## 1.2. Goals and Research Questions

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

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

1. Can an LLM-based system perform common GIS tasks?
2. What are core challenges when developing larger systems that rely on Large Language Models to control its logic flow?
3. What are state-of-the-art methods of creating autonomous LLM-based agents?

## 1.3. Research Method

Prior to this master's thesis, a specialization project on the same topic was conducted. The specialization project — as detailed in (Holm, 2023) — was of a theoretical character predominantly served as a literature study leading up to the master's thesis. The research questions listed in the above section, however, call for a different and more practical approach. Specifically, RQ1 and RQ2 can only be addressed by drawing on insights gained from the attempt to develop such an LLM-based GIS system. Therefore, this master's thesis will revolve around a "proof of concept" and evaluating the usefulness of the system, as well as lessons learned from the development process.

RQ3, on the other hand, will be a continuation of the theoretical work done in the specialization process, and will serve as a foundation for the "proof of concept" development process.

## 1.4. Contributions

## 1.5. Thesis Structure

# 2. Background Theory

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

- *Subsection 2.1.2: Reused with minor adjustments.*
- *Subsection* **??***: GPT part reused without modification.*

Chapter 2 will lay a theoretical basis for the work done in this master thesis, providing the user with the required understanding in order to understand the contributions of the work. Section **??** will explain the theoretical basis of the component which most modern Large Language Models (LLMs) are based upon — namely the Transformer — and the attention mechanism within it. The section will also touch upon a new approach to language modelling called *selective state space modelling*, which has yielded very promising results for small LLMs.

## 2.1. Large Language Models

LLMs are a type of neural networks that excel at processing language. They can be developed for different Natural Language Processing (NLP) tasks, such a text classification, masked language modelling, and text generation. While they all have their use cases, only text generation will be relevant for this thesis.

Generative LLMs are designed to take some input sequence and generate some output sequence.

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

### 2.1.1. Core Concepts

The **context window** of an LLM is the range of tokens that an LLM is able to process.

While humans understand sentences as sequences of words, LLMs perceive them as sequences of **tokens**. An LLM possesses a fixed set of unique tokens in its vocabulary, from which it constructs words and sentences. Tokens can be entire words, short character

sequences, or single characters. Figure 2.1 illustrates how the latest GPT models *tokenize* an English sentence. Notice how some words are deconstructed into more than one token. For example, the word "revolutionizing" is split into its root ("revolution") and its suffix ("izing"). Figure 2.2 shows how this process applies to other suffixes as well. Likely, the model has learned the meaning of "revolution" and elects to use different suffixes to modify its function within a sentence, as opposed to having an entirely new token for each version of the word.

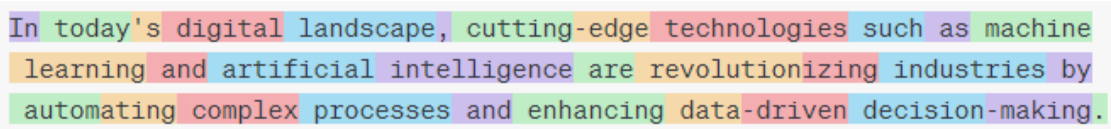



Figure 2.1.: Tokenization example for a sentence

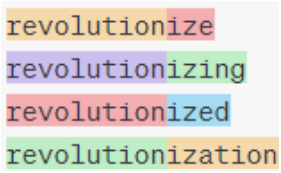



Figure 2.2.: Tokenization of "revolution" with different suffixes

When an LLM generates text, it does so by generating a new token based on what

### 2.1.2. Attention and the Transformer Architecture

Vaswani et al. (2017) managed to achieve new state-of-the-art results for machine translation tasks with their introduction of the Transformer architecture. The Transformer has later been proved effective for numerous downstream tasks, and for a variety of modalities. Titling their paper *Attention Is All You Need*, Vaswani et al. suggest that their attention-based architecture renders network architectures like Recurrent Neural Networks (RNNs) redundant, due to its superior parallelization abilities and the shorter path between combinations of position input and output sequences, making it easier to learn long-range dependencies (Vaswani et al., 2017, p. 6).

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

all individual tokens have some context from all other tokens. We improve the attention mechanism by multiplying queries, keys, and values with weight matrices that are learned through backpropagation. Self-attention is a special kind of attention in which queries, keys, and values are all the same sequence.

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

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

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

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

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

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

#### The GPT Family

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

multiple attention heads in parallel), and is restricted to only see the last $k$ tokens — with $k$ being the size of the context window — and tasked to predict the next one.

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

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

### The Claude Family

Developed at Anthropic, Claude is the third major, commercial LLM. Anthropic is one of the actors in the LLM market that has helped push in the direction of long-context LLMs, which their Claude 2 model being able to support up to 200k tokens (Anthropic, 2023, p. 9), being the best at the time of its release in November 2023. Latest in line is Claude 3, a family of LLMs of different sizes, largest of which is the Opus model which outperforms GPT-4 and Gemini 1.0 Ultra on most benchmarks (Anthropic, 2024, p. 6).

Claude 3 is a family of language models, much like the Gemini family, featuring three main models: Opus, Sonnet, and Haiku. Again, these are listed in descending order in terms of number of parameters. Users can choose between Opus for the most advanced capabilities, Haiku for a fast and economical option, or Sonnet for a balance of both.

#### Open-Source Alternatives

OpenAI's GPT models, Google's Gemini models, and Anthropic's Claude models are all commercial and closed-source. This prevents developers from downloading these models and making custom improvements to them through fine-tuning (see subsection 2.1.4). For these reasons, a number of open-source LLMs have joined the scene.

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

**Mistral AI** is one of the most prominent actors in the world of open-source LLMs. Their debut model, Mistral 7B, outperformed Llama 13B (which was the best open-source LLM at the time) across all the benchmarks they evaluated (Jiang et al., 2023). Mistral AI has also gained fame for their Sparse Mixture-of-Experts (SMoE) architecture, which was introduced with the Mixtral 8x7B model (Jiang et al., 2024). It shares architecture with Mistral 7B, but is each layer of the model is composed of 8 feed-forward blocks. Using a router at each layer, it is able to use only 13B out of a total of 47B parameters during inference, keeping cost and latency low.

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

### 2.1.4. Fine-Tuning

### 2.1.5. Prompt Engineering

Prompt engineering refers to the process of constructing a query that will be used as input to an LLM. In a chat-based context like that of ChatGPT, the prompt consists of a series of messages that are defined as an array of JSON objects. These messages can hold one of three different roles[1], 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.

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

- **user**: A message from the user for the assistant to respond to.
- **assistant**: A message from the AI/LLM.

### 2.1.6. Function Calling LLMs

*Function calling* — first introduced by OpenAI (Eleti et al., 2023) — allows developers to provide function definitions to an LLM and have said LLM output a JSON object containing the name of one or more of the functions provided, as well as suitable arguments to these (see **??**). Made possible through fine-tuning models to detect when functions should be calling, 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.

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

## 2.2. LangChain

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

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 2.1.6)
- Creating code executing agents for Python, SQL, or other programming languages

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

## 2.3. Geospatial Databases and Data Catalogues

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

### 2.3.1. PostGIS

PostGIS (*PostGIS* 2001) is an open-source extension for the PostgreSQL DBMS. By adding the PostGIS extension one adds support for storing, indexing, and querying geospatial data. Data can be stored in both two and three dimensions, and they can have types like points, lines, polygons. These types can be stored along with a spatial index which can significantly reduce search time for these geometries. GiST (Generalized Search Tree)[2] is commonly used in PostgreSQL/PostGIS to take advantage of various tree-based search algorithms that are developed to retrieve spatial features quickly.

PostGIS also comes with a plethora of spatial database functions that to analyse and process geospatial data. These are prefixed with `ST_`, and some examples are `ST_DWITHIN`, `ST_BUFFER`, and `ST_TRANSFORM`. Code 2.1 displays a typical query for retrieving building outlines within a bounding box specified in WGS 84 latitudes and longitudes.

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

Code 2.1: PostGIS example code for retrieving building outlines within a specified bounding box

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

### 2.3.2. OGC API Features

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

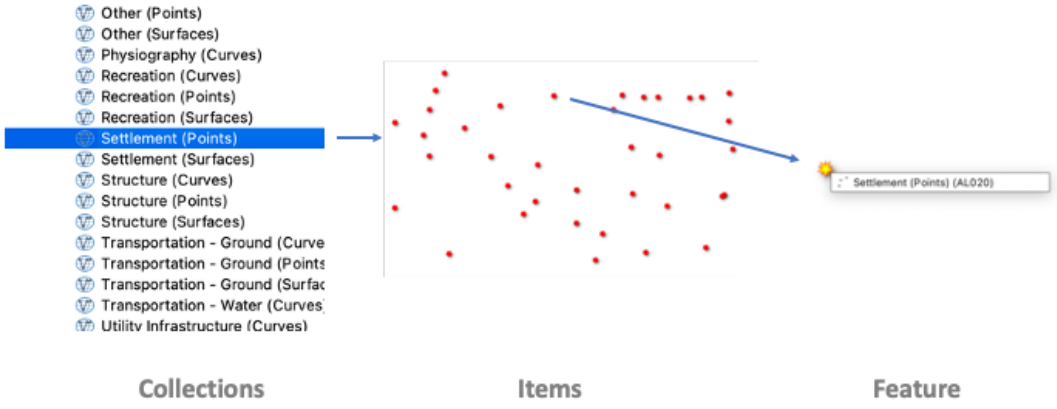



Figure 2.3.: Collections, items, and features in OGC API Features specification. Retrieved from https://features.developer.ogc.org/ on April 29, 2024.

The development of the Features standard is divided into several parts that are meant to build on top of one another. Here are the four parts that are listed on OGC's site about the standard [3]:

- Features - Part 1: Core[4]
- Features - Part 2: Coordinate Reference Systems by Reference [5]
- Features - Part 3: Filtering[6]
- Features - Part 4: Create, Replace, Update and Delete[7]
- Features - Part 5: Schemas[8]

[3] https://ogcapi.ogc.org/features/
[4] https://docs.opengeospatial.org/is/17-069r4/17-069r4.html
[5] https://docs.ogc.org/is/18-058r1/18-058r1.html
[6] https://docs.ogc.org/DRAFTS/19-079r1.html
[7] https://docs.ogc.org/DRAFTS/20-002.html
[8] https://docs.ogc.org/DRAFTS/23-058r1.html

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

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

Code 2.2: CQL examples

Part 4 defines how an API following the specification handle addition, replacement, modification, and/or removal for a collection. Part 5 describes how features can be described by a logical schema and how these are published. Furthermore, several proposed extensions, such as the Search extension which could allow for multi-collection queries, or the Geometry Simplification extension which proposes the use of simplification algorithms for retrieving simplified versions of a collection, have been created.[9]

### 2.3.3. SpatioTemporal Asset Catalog

The SpatioTemporal Asset Catalog (STAC) specification[10] is closely related to OGC API Features, and as Holmes — former board member of Open Geospatial Consortium — stated in a blog post that "STAC API implements and extends the OGC API — Features standard, and our shared goal is for STAC API to become a full OGC standard" (Holmes, 2021). The main difference to the OGC API Features specification is its requirement that all items/features should have a temporal component, thus making it *spatiotemporal*. The vision for the STAC specification is to minimize the need for developing new code with the release of each new dataset or API, by using a standard that does not change.

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

# 3. Related Work

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

- *Subsection 3.1: Reused without modification.*

## 3.1. Using LLMs for Geospatial Purposes

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

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

Zhang et al. (2023) use the LangChain framework in order to combine different GIS tools in a sequence to solve various sub-goals, focusing on using the semantic understanding and reasoning abilities of LLMs to call externally defined tools, employing the LLM as an agent or controller. The authors take great inspiration from the AutoGPT framework (Richard, 2023). The externally defined tools are described (manually) by their names

and descriptions. These descriptions contain information about the input parameters and output types of the tools/functions. Tools are defined for geospatial data collection, data processing and analysis, and data visualization. The effectiveness of the system is showcased in four case studies.

## 3.2. Agent Patterns

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

### 3.2.1. The Multi-Agent Pattern

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

### 3.2.2. Patterns for Retrieval Augmented Generation

### 3.2.3. Patterns for Self-Reflection

# 4. Datasets

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

## 4.1. Data Sources

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

Table 4.1.: Datasets used in experiments

| **Dataset** | **Data Type** | **Description** |
|---|---|---|
| Buildings | Polygon | Contains building outlines. Its *type* attribute can have values like *house*, *university*, and *restaurant*. |
| Land Use | Polygon | Represents areas designated to different purposes and activities. Its *fclass* attribute can have values like *forest*, *farmland*, and *residential*. |

Continued on next page

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

**Table 4.1 continued from previous page**

| Dataset | Data Type | Description |
|---|---|---|
| Natural | Point | Contains outlines of various objects found in nature. Its *fclass* attribute can have values like *beach*, *glacier*, and *cave_entrance*. |
| Natural | Polygon | Similar to the point data equivalent. |
| Places of Worship | Point | Common values for *fclass* attribute: *christian*, *buddhist*, and *muslim*. |
| Places of Worship | Polygon | Similar to the point data equivalent. |
| Places | Point | Common values for *fclass* attribute: *farm*, *village*, and *island*. Repeated entries trimmed for brevity. |
| Places | Polygon | Similar to the point data equivalent. |
| Points of Interest | Point | Common values for *fclass* attribute: *tourist_info*, *bench*, and *kindergarten*. |
| Points of Interest | Polygon | Similar to the point data equivalent. |
| 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. |
| Roads | Lines | Common values for *fclass* attribute: *rail*, *subway*, and *tram*. Has additional attributes *oneway*, *maxspeed*, *bridge*, and *tunnel*. |
| Traffic | Point | Common values for *fclass* attribute: *crossing*, *street_lamp*, and *parking*. |
| Traffic | Polygon | Common values for *fclass* attribute: *parking*, *pier*, and *dam*. |
| Transport | Point | Common values for *fclass* attribute: *bus_stop*, *ferry_terminal*, and *railway_station*. |
| Transport | Polygon | Similar to the point data equivalent. |
| Water | Polygon | Common values for *fclass* attribute: *water*, *wetland*, and *river_bank*. |

Continued on next page

**Table 4.1 continued from previous page**

| Dataset | Data Type | Description |
|---|---|---|
| Waterways | Lines | Common values for *fclass* attribute: *stream*, *river*, and *canal*. |

## 4.2. Data Access

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

Insert Python benchmark examples here

The upcoming experiments therefore seek to benchmark model performance on three different data access methods. The datasets from section 4.1 are presented to the model in three different ways, as subsection 4.2.1 through subsection 4.2.2 elaborate upon.

### 4.2.1. Files

The first method of presentation is to have the files from section 4.1 remain untouched. The datasets were stored such that each dataset has its own folder. This is because some of the file types used require multiple files in order to correctly store the data—for instance, the shapefile format, which has three mandatory files: .shp, which contains the actual feature geometry; .shx, which provides a positional index of the feature geometry; and .dbf, which holds attributes for each shape.

reference

### 4.2.2. SQL Database

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

Some of the datasets come in the GML data format, which can include multiple layers with potentially different geometries. For this reason, they cannot be loaded directly into a PostGIS database such that they are stored in the same database table. Furthermore, several of the layers in the multi-layer GML files are irrelevant for most analysis situations. For instance, the flood zone data were downloaded as a multi-layer GML file and includes a total of eight layers: polygon and multi-line border for the analysis area, polygon layers

for rivers, ocean surfaces, and lakes, polygon and multi-line border for the flood zones, and a layer containing cross-sectional profile lines for the rivers. The quick clay dataset was similar. For brevity, a decision was made not to load all these layers into the PostGIS database. Only the polygon for the flood zones and two polygon layers from the quick clay dataset were loaded into the database.

### 4.2.3. OGC API Features

The third method for data access is to use the OGC API Features standard.

# 5. Architecture

## 5.1. High-Level Application Architecture

A microservice architecture was employed in order to simplify development and separate concerns between the different microservices. The services are deployed as Docker Containers, and they are orchestrated using Docker Compose. Figure 5.1 shows how the application is divided into five distinct services, and the direction of information flow between these.

quickly explain docker and why it is useful here

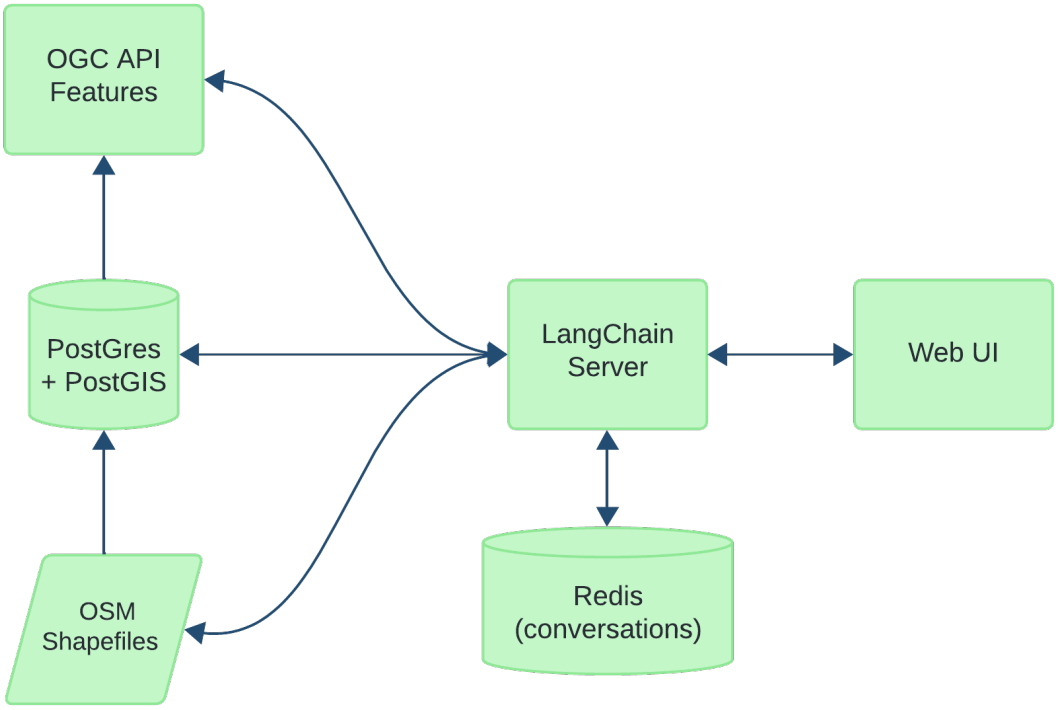



Figure 5.1.: Architecture overview

### 5.1.1. LangChain Server

The *LangChain Server* service is the heart of the application, and is where the Large Language Model (LLM)-related logic is situated. It is responsible for taking requests from

the *Web UI* and returning suitable responses in what becomes a client-server architecture between the two services. Table 5.1 show the endpoints exposed by the server and how they can be used by a client.

Table 5.1.: Summary of Server Endpoints

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

### 5.1.2. Redis for Conversations

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

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

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

### 5.1.3. PostGIS OGC API Features

A PostgreSQL database with the PostGIS extension containing OpenStreetMap (OSM) data was deployed using Docker. On top of this database is a RESTful geospatial feature server called *pg_featureserv* (CrunchyData, 2024). The web server is realized through the `pramsey/pg_featureserv` Docker image[1], which is simply passed a database connection string to the database one wishes to expose on the API. Any tables in that database which have a geometry column and a specified Coordinate Reference System (CRS) will be exposed through the web server. pg_featuresserv allows for features like bounding box filtering, feature limiting, and CQL filtering. These are added as query parameters to the URL of the collection items (for instance: `.../collections/{collection_id}/items.json?limit=1000&filter=name IS NOT NULL`). In the internals of pg_featuresserv, this URL will be converted to an SQL query that will be run against the database. Results of the query will be returned as GeoJSON. Code 5.1 shows an example of how CQL code is converted into SQL code:

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

Code 5.1: Conversion from CQL to SQL

### 5.1.4. Web UI

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

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

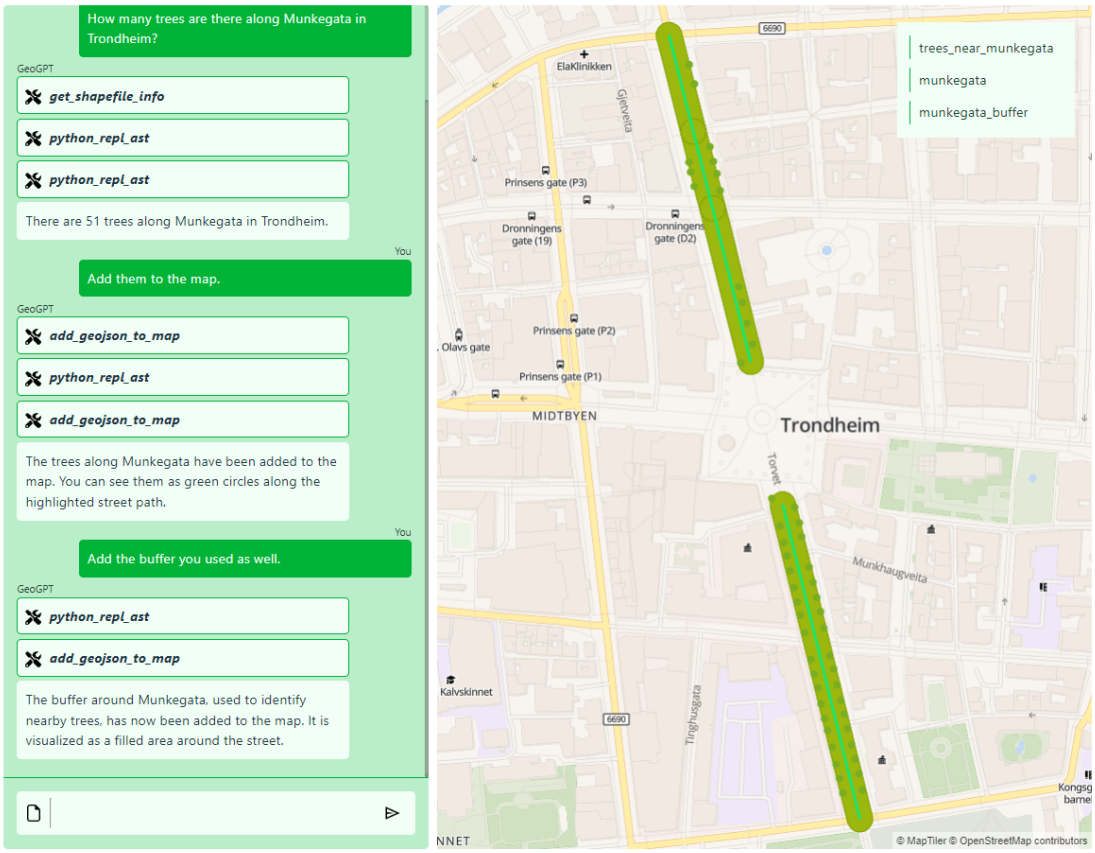



Figure 5.2.: Web UI

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

The map is created using MapLibre[3], an open-source fork of Mapbox which in December 2020 changed to a non-open-source software licence. A base map from OSM is used, fetched through a website called MapTiler. GeoJSON files of any kind that are fetched from the server will be added to the map automatically with a random color. On the

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

top-right of the map is an overlay listing all layers that are currently present in the map. Using the arrow keys on this list will change the z-index in the map of the selected layer.

## 5.2. Agent Architecture

Three different agent were implemented for GeoGPT: one agent utilizing OGC API Features, one agent which can access shapefiles using Python code, and one agent that can interact with data in an SQL database. Common to these agents is their agentic architecture, which is described in subsection 5.2.1. They way that they differ is through their assigned *tools*. These differences will be made apparent in Figure 5.3. Other slight differences are seen in the way that they are prompted. The prompting strategy used in GeoGPT and these minor differences will be discussed in subsection 5.2.3.

### 5.2.1. LangGraph Agent Implementation

The agentic behaviour of GeoGPT is implemented using LangGraph, an extension to LangChain intended to make implementation of cyclic behaviour easier. Figure 5.3 illustrates the flow between the various nodes that make up the agent. A state dictionary is passed between and updated by these nodes. Included in this state is the chat history (previous messages), the path to GeoGPT's working directory, a list of the current files in this directory, and other, less important state. The implementation is based upon a prebuilt implementation from LangGraph.[4]

The *___start___* node serves as the entry point of the agent. At this point, only one message is present in the state, namely the initial message from the user. The *___start___* node points to the *agent* node, where a response to the user is generated by an LLM. This response could contain a textual response, or it could contain instructions to execute some tool. For this reason, the current state containing both the user message and the AI message, is sent to the *conditional* node called *agent_should_continue*. This node simply checks if the agent outcome (the last AI message) includes the "tool_calls" keyword argument. If this is the case then some tool should be executed, and we should "continue". If not, the agent has generated a final textual response to the user.

In the *action* node the values inside the "tool_calls" object are converted into `ToolInvocation` objects that are used to invoke predefined tools. "tool_calls" is a list of "tool_call" objects, each of which has attributes called *name* (the name of the tool) and *arguments* (arguments that should be passed to the tool). These tools are then executed, possibly having side effects on the server, and the resulting output of the tools are appended as messages to the chat history. Figure 5.4 illustrates this behavour. This way the agent can inspect the results of the code it is "executing", as well as being notified about possible errors in the input parameters it provided. Through the cyclic behaviour of the agent graph the agent can repeatedly call tools to try and answer the request from the user. When the agent finds to reason to call any more tools it generates

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

a textual response, making *agent_should_continue* return `False` so that the termination node (*__end__*) is reached.

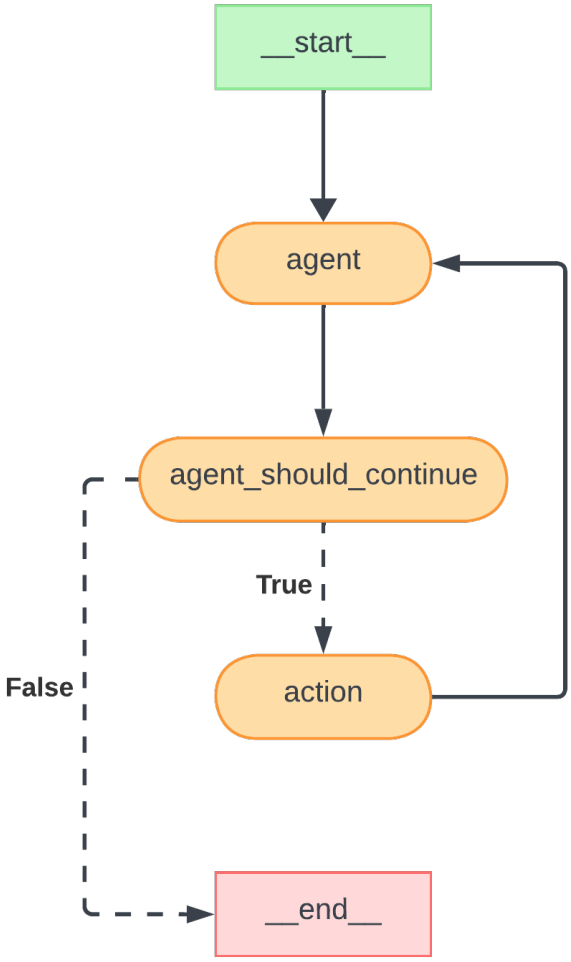



Figure 5.3.: Generic tool agent graph

### 5.2.2. Tools

Table 5.2 shows an overview of the tools that are available to each of the agents. As described in subsection 2.1.6, these are defined by a name, an overall description of the tool's functionality, and a description of the parameters the tool expects.

The OGC API Features agent has access to a total of five tools. **`list_collections`**, which takes no parameters, sends a `GET` request to the `/collections` endpoint and uses the response to construct a string listing the names of available collections along with

their descriptions. The tool is designed to give the agent an overview of what kinds of data are available. Using the response from `list_collections` the agent can now invoke the **`get_collections_info`**. This tool takes a list of collection names and returns relevant information for each of these collections. This includes the JSON response from the "landing page" of the collection, presenting details such as the collection description, spatial extent, and available attributes. In addition to this information, there is a list of common values for certain high-cardinality attributes, along with the prevalence of each value in percentages. Code 5.2 shows an example of this. This is achieved by querying a large number of features from `/collections/{collection_id}` and calculating the prevalences between them.

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

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

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

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

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

Common for both the OGC API Features agent and the Python agent is the **`python_repl_ast`** tool. This tool accepts a string of Python code, executes said Python code, and returns whatever the code prints to the standard output in a so-called Read-Eval-Print Loop (REPL). In case the code errors, the error message is returned instead. This Python tool is the main way for these two agents perform geospatial analyses. An advantage of using REPLs is that the code can be executed in blocks, with variables from one block being shared with other blocks. This means that the first block may load large files into memory — an often time-consuming operation — while subsequent blocks can

Table 5.2.: Overview of each agent's 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` |

reuse this in-memory variable, even if that block should error. This allows the LLM to quickly retry whenever the code errors or if the outcome of the initial code wasn't as expected. The OGC API Features agent will have available the files it downloaded using the `query_collection` tool for analysis using Python, whereas the Python agent will have files corresponding to all collections available for analysis. These files are stored locally on the machine and they are added to the agent's working directory at system launch using symbolic links[5]. This is because a new working directory is created every time a new agent is created in order to give each agent a "clean canvas" to work from.

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

The SQL agent has tools very similar to the OGC API Features agent. The SQL agent is connected directly to the same database that the Features API is served on top of. **`sql_db_list_tables`** is a tool that will list all database tables along with their description. **`sql_db_schema`** takes a list of table names and will return information about

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

attributes, prevalence of different values in high-cardinality columns, etc., about these tables, much like `get_collections_info`. **`sql_db_query`** takes as parameter arbitrary SQL code executes this code against the database. The tool will make sure that any query result that has a geospatial component will be stored as GeoJSON on the server. This is required to make it possible to add the geometries to the client map.

```
{
  "type": "function",
  "function": {
    "name": "get_collections_info",
    "description": "Get the schema and sample rows for the specified collections.",
    "parameters": {
      "type": "object",
      "properties": {
        "collection_names": {
          "description": "A list of the collection_names names for which to return the schema.",
          "type": "array",
          "items": {
            "type": "string"
          }
        }
      },
      "required": [
        "collection_names"
      ]
    }
  }
}
```

Code 5.3: Example of a tool definition

### 5.2.3. Prompt Templating

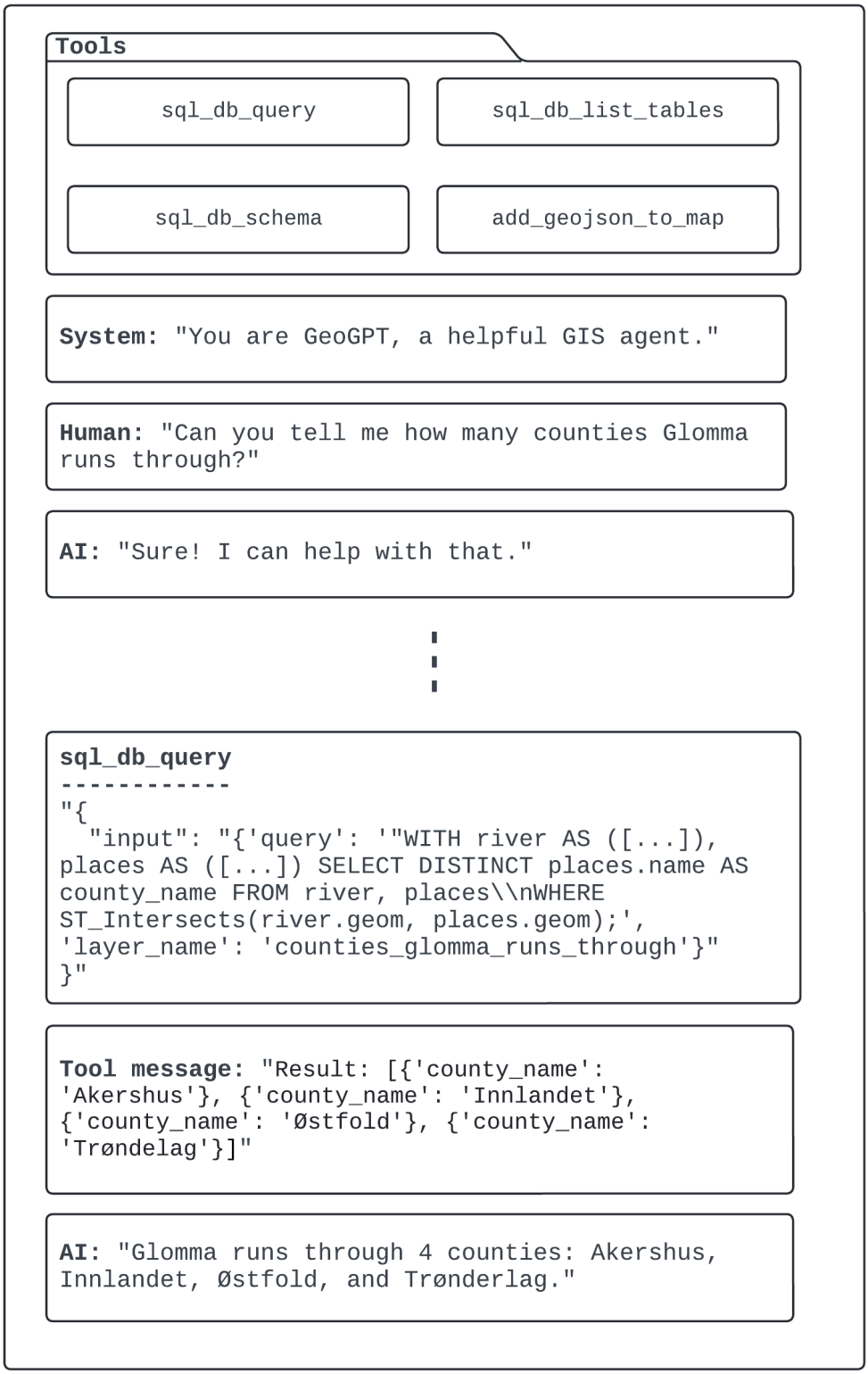



Figure 5.4.: Example of a chat trace

# 6. Experiments and Results

## 6.1. Experimental Setup

The experiments conducted to evaluate the performance of GeoGPT on geospatial tasks are divided into two approaches. These are presented in subsection 6.1.1 and subsection 6.1.2.

### 6.1.1. Quantitative Approach: Benchmarking

The first approach seeks to evaluate its ability to successfully answer questions that have a concrete answer. To do this, a Q&A dataset was constructed. This dataset consists of set of 12 GIS-related questions with corresponding correct answers. For each record in the dataset, a description of how a human would find it natural to approach the problem. This description is provided as a step-by-step path towards the solution, and is only included to guide any reader as to how the system would be expected to solve the system. The full Q&A dataset can be found in Table 6.3. This set of experiments will allow for quantitative assessment of GeoGPT's GIS-abilities, and is a feasible way of benchmarking the system.

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

With the implementation of three different agent types, the total number of test runs becomes the following:

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

#### Outcome Evaluation

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

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

Table 6.1.: Description of Success

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

Table 6.2.: Encoding for Test Outcome

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

### Other Metrics

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

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

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

Table 6.3.: Questions for quantitative tests

| **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. |

Continued on next page

Table 6.3.: Questions for quantitative tests

| Query ID | Query | Correct Response |
|---|---|---|
| num_trees_munkegata | How many trees are there along Munkegata in Trondheim? | Giving the correct number of trees (about 70-80). |
| 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. |

### 6.1.2. Evaluating Importance of Prompt Quality

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

### 6.1.3. Configuration and Hardware

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

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

## 6.2. Experimental Results

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

### 6.2.1. Quantitative Results

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

#### Test Outcomes

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

ref some table with agent types

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

#### Other Metrics

This section features three different box plots: Figure 6.2, Figure 6.3a, and Figure 6.3b. The Matplotlib implementation of the box plot follows the description found on

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

[2] https://matplotlib.org/


Figure 6.1.: Outcome distribution between different agent types

Wikipedia[3,4]. Box plots allow us to easily visualize where the 0th ($Q_0$), 25th ($Q_1$), 50th ($Q_2$), 75th ($Q_3$), and 100th ($Q_4$) percentiles of the datasets lie, as well as the dataset's outliers. Outliers are those data points fall outside 1.5 times interquartile range, that is, the distance between $Q_3$ and $Q_1$ in each direction.

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

Figure 6.3 shows box plots for the amount of tokens used per run and the total cost of these. Naturally, these figures appear very similar, as OpenAI sets fixed prices for the input tokens and generated output tokens for their GPT models. According to their websites[5], they charge $10 per million input tokens and $30 per million output tokens for their GPT-4 Turbo models. From the results of the experiments, a ratio of approximately 10.7 per million tokens — either input *or* output — was calculated, which is very close to the input token price. This aligns with the observation that the number of input/prompt tokens is significantly greater than the number of output/generated tokens for the experiments conducted. The correlation matrix in Figure 6.4 confirms that this ratio is consistent.

Another observation that can be made from Figure 6.4 is that there is only a *slight* positive correlation between duration and token/usage. This supports the observation that code execution time is likely the most significant factor in determining the duration required to complete a task for the datasets and tasks used in this thesis' experiments.

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

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

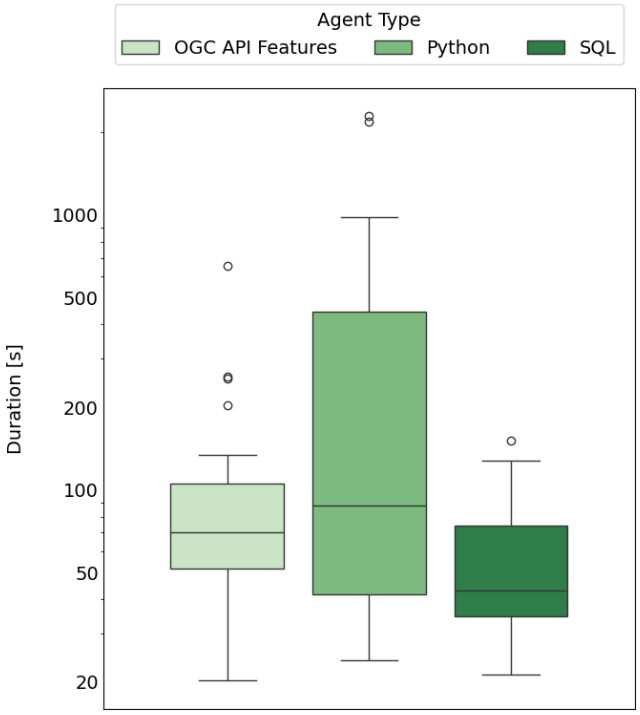
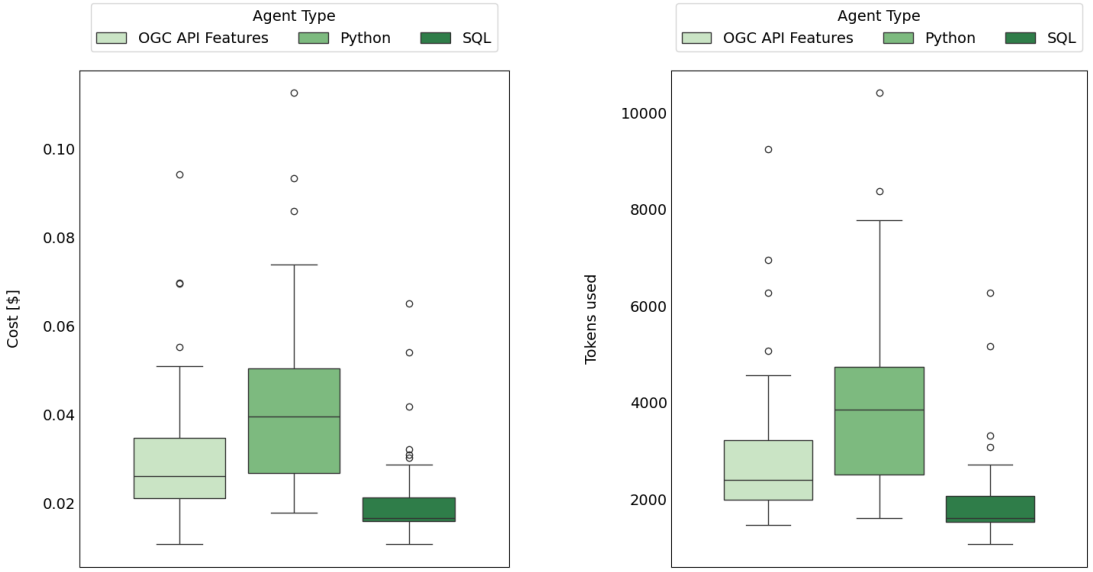



Figure 6.2.: Duration per Agent Type




(a) Average cost per call for each agent type

(b) Average token usage per agent type

Figure 6.3.: Cost and token usage

```
import geopandas as gpd

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

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

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

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

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

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

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

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

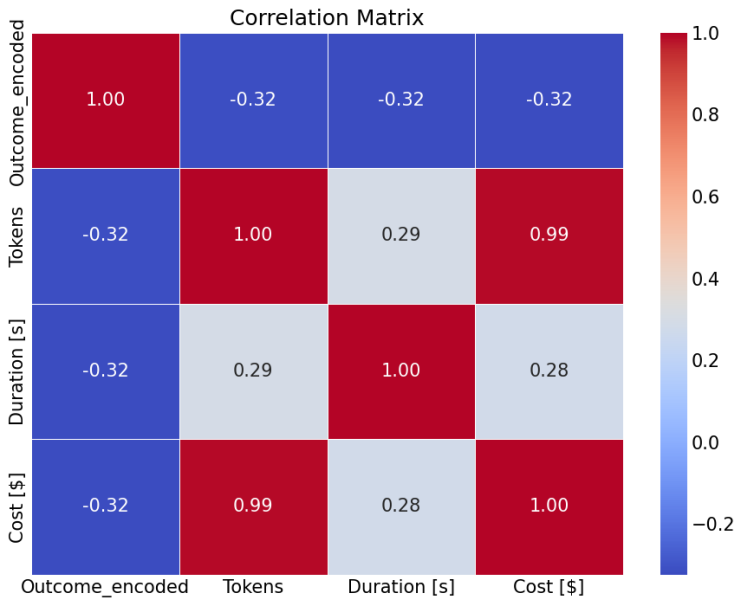



Figure 6.4.: Correlation matrix for test result metrics

### Repeatability

Table 6.4 shows the average standard deviation for each agent type, as well as the mean of these three standard deviations. The latter serves as an overall measure of GeoGPT's ability to repeatedly produce the desired outcome from a given query. Standard deviations were calculated for each triplet of identical test samples, in which both the question and the agent type remained the same. To produce a numerical value for the standard deviations, the encoded outcomes (see Table 6.2) were used. Taking the average of the standard deviations for all 12 triplets for each agent type produced the numbers found in Table 6.4.These numbers indicate that there is a notable amount of inconsistency in GeoGPT's ability to produce correct answers, on an average deviating with almost "half an outcome category" (0.408 for the encoded values) for the same task and agent type.

Table 6.4.: Standard Deviation by Agent Type

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

### 6.2.2. Prompt Quality Test Results

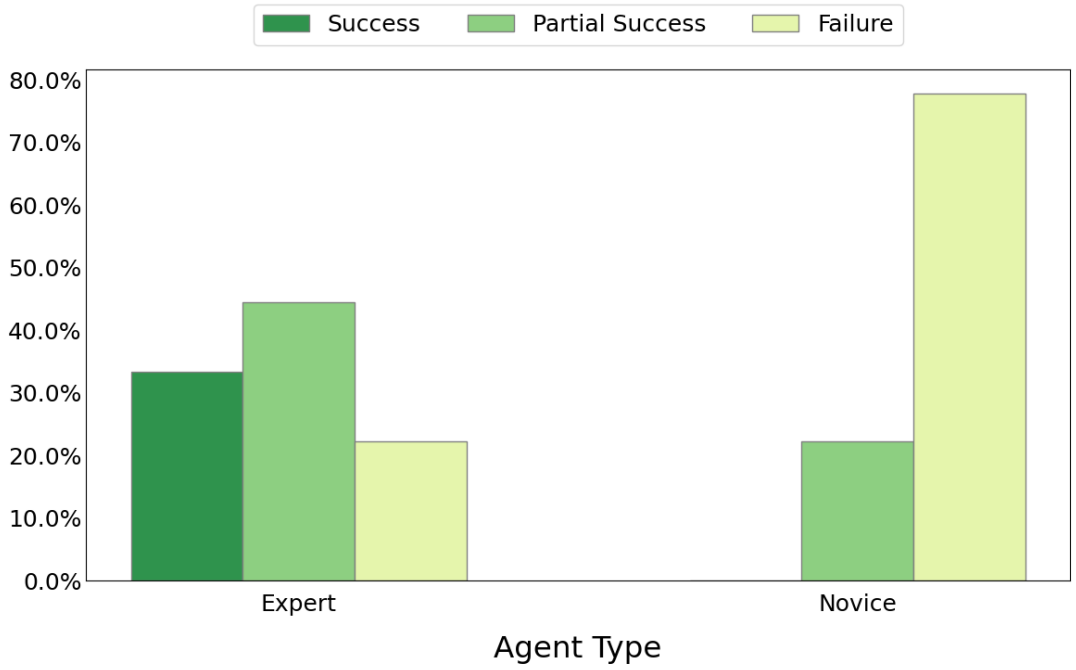



Figure 6.5.: Outcome distribution for different levels of GIS experience

Figure 6.6 shows a comparison between the results GeoGPT managed to produce for two different prompts that one would expect to produce identical outcomes. The *novice*-level prompt was as follows:

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

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

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

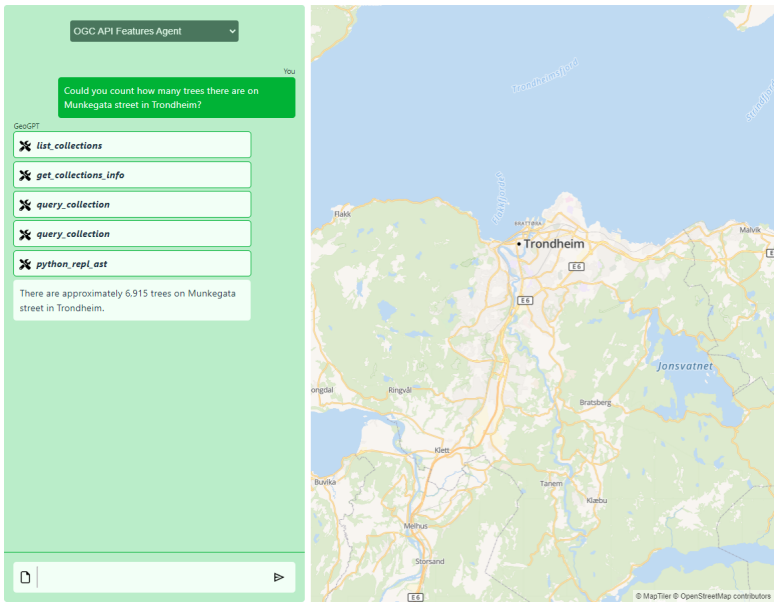
Using novice-level prompt GeoGPT was unable to produce the correct outcome, and confidently answered that there are "approximately 6,915 trees on Munkegata street in

Trondheim." (see Figure 6.6a), which is far from being true. When solving the task, GeoGPT made a series of oversights that lead to this result. First, GeoGPT failed to take into account that there may be more than one street in the dataset with the name "Munkegata", "forgetting" to use a bounding box when retrieving the road data from the API. The same mistake was made when retrieving the tree data. Due to the upper limit of 10,000 features per query in the API, it's crucial to narrow down the query to ensure retrieval of all relevant features rather than just a subset. GeoGPT's query for tree data lacked a bounding box, resulting in a randomly distributed subset scattered across Norway. A third mistake occurred when GeoGPT calculated a bounding box around the retrieved road data instead of creating a buffer. The latter method would have produced a more accurate result. The bounding box that was created spanned from Trondheim to Oslo, thus including far more trees than was intended.

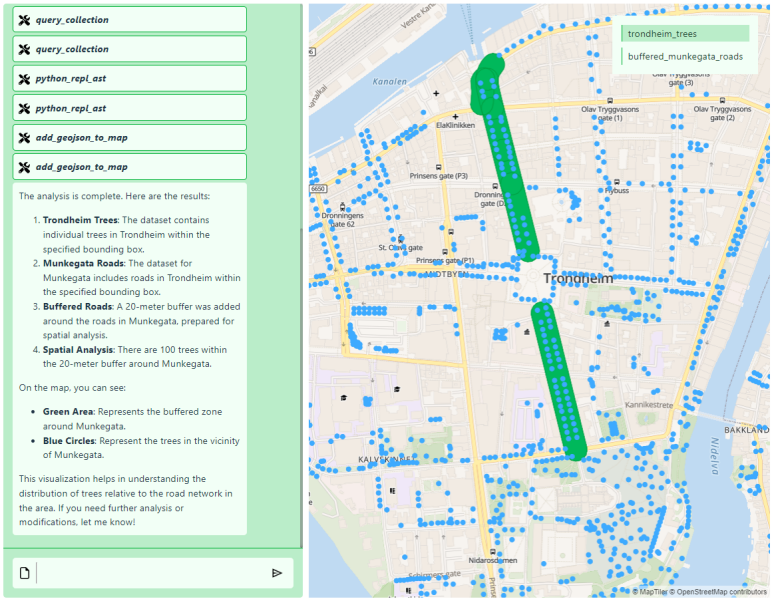
On the other hand, the expert-level prompt provided the necessary guidance for GeoGPT for this specific task, steering it clear of the issues it encountered with the novice-level prompt. As stated by OpenAI themselves, "some tasks are best specified as a sequence of steps".[6] Furthermore, they say that writing explicit steps required to solve a tasks "makes it easier for the model to follow them".

move to discussion?

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




(a) Novice-level prompting




(b) Expert-level prompting

Figure 6.6.: Comparison between novice and expert level prompting for making GeoGPT's OGC API Features agent calculate the number of trees along Munkegata in Trondheim

# 7. Evaluation and Discussion

## 7.1. Evaluation

## 7.2. Discussion

### 7.2.1. Difficulties with OGC API Features

A common source of error with several of the tests conducted with the OGC API Features agent in was its inability to fetch more than 10,000 features from the server. The limit of 10,000 features is specified in the Features standard (Open Geospatial Consortium, 2022), meaning that no more than 10,000 features should be returned in a single response. Accompanied by such a large response, however, should be a `next` link than should point to the next set of 10,000 features. This way, the server could return more than 10,000 features. Unfortunately, as of 6th May 2024, the current version of *pg_featuresserv* does not support this features[1], which is a significant limitation to the current OGC API Features implementation for GeoGPT.

Furthermore, the lack of multi-collection queries is in the author's opinion a big limitation to the current Features specification. A proposal draft[2] to such features have been created, but it is unclear whether this will accepted into the specification. This extension, called *Search*, would allow for more complex CQL queries than are not easily specified using query parameters. Code 7.1 shows one of the multi-collection query examples included in the proposal draft. Being able to construct such queries could make retrieval of features much more efficient, possibly making the Python tool in the current GeoGPT implementation redundant. This query would not be possible using the current implementation, and one would have to download the two collections, load them into memory using Python, and perform the `contains` operation there. Being able to construct such CQL queries would offload this work to the Features server which (if the data source is a PostGIS database) would run very efficient SQL code instead of the less than optimal Python code that would otherwise be necessary.

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

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

```
 6
 7 \\ Corresponding CQL query (would return a tuple of parks and lakes)
 8 POST /search   HTTP/1.1
 9 Host: www.someserver.com/
10 Accept: application/json
11 Content-Type: application/ogcqry+json
12
13 [
14     {
15         "collections": ["parks","lakes"]
16         "filter": {
17             "and": [
18             {"eq": [{"property": "parks.name"},"Algonquin Park"]}
19             {"contains": [{"property": "parks.geometry"},
20                           {"property": "lakes.geometry"}]}
21             ]
22         }
23     }
24 ]
```

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

### 7.2.2. Multi-Agent Architectures

# 8. Conclusion and Future Work

## 8.1. Contributions

## 8.2. Future Work

### 8.2.1. Automated Data Access

The experiments in chapter 6 were based upon a pre-existing database. A fully autonomous GIS agent should, however, be able to search the web for suitable datasets, based on the user's query. In a Norwegian context, one could imagine asking for a noise analysis for a particular building. The agent would then search Geonorge for datasets related to noise (firing ranges, roads, etc.), downloading these, and then performing analysis. Simple experiments were conducted in this thesis to see if this was possible, but results were somewhat poor. Methods like semantic search based upon the documentations of datasets should be explored in future research.

# 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

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

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/

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

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.

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/

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

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

# Appendices

# A. Test Results

Table A.1.: Test results for quantitative tests

| **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 | 129.03 | 4498 |
| aker_brygge_national | sql | partial success | 114.39 | 5269 |
| aker_brygge_national | sql | success | 95.83 | 5584 |
| cliff_clusters | oaf | partial success | 93.21 | 3047 |
| cliff_clusters | oaf | partial success | 134.62 | 3713 |
| cliff_clusters | oaf | failure | 253.77 | 6956 |

Continued on next page

Table A.1.: Test results for quantitative tests

| **Query ID** | **Agent Type** | **Outcome** | **Duration [s]** | **Tokens** |
|---|---|---|---|---|
| cliff_clusters | python | failure | 66.42 | 5421 |
| cliff_clusters | python | failure | 125.96 | 4650 |
| cliff_clusters | python | failure | 109.29 | 4768 |
| cliff_clusters | sql | success | 65.47 | 1909 |
| cliff_clusters | sql | success | 61.22 | 2837 |
| cliff_clusters | sql | success | 156.00 | 4485 |
| 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 | 18.89 | 1149 |
| county_names | sql | success | 19.74 | 1169 |
| county_names | sql | partial success | 47.36 | 2489 |
| 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 | 36.54 | 1231 |
| glomma_counties | sql | success | 63.65 | 1810 |

Continued on next page

Table A.1.: Test results for quantitative tests

| **Query ID** | **Agent Type** | **Outcome** | **Duration [s]** | **Tokens** |
|---|---|---|---|---|
| glomma_counties | sql | success | 22.81 | 1079 |
| 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 |
| largest_county | sql | success | 37.52 | 1654 |
| largest_county | sql | success | 27.55 | 1483 |
| largest_county | sql | success | 41.22 | 1765 |
| 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 | 34.60 | 1495 |
| nidarosdomen_polygon | sql | success | 63.43 | 1519 |
| nidarosdomen_polygon | sql | success | 34.77 | 1518 |
| 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 |

Continued on next page

Table A.1.: Test results for quantitative tests

| Query ID | Agent Type | Outcome | Duration [s] | Tokens |
|---|---|---|---|---|
| num_trees_munkegata | python | failure | 916.13 | 2362 |
| num_trees_munkegata | python | failure | 911.18 | 8385 |
| num_trees_munkegata | sql | failure | 63.93 | 1831 |
| num_trees_munkegata | sql | failure | 77.45 | 3086 |
| num_trees_munkegata | sql | failure | 64.47 | 1640 |
| oslo_bergen_geodesic | oaf | failure | 652.88 | 9253 |
| oslo_bergen_geodesic | oaf | failure | 102.38 | 3360 |
| 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 | 80.49 | 3253 |
| oslo_bergen_geodesic | sql | failure | 83.78 | 2466 |
| oslo_bergen_geodesic | sql | failure | 88.63 | 2825 |
| 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 | 168.13 | 1944 |
| oslo_residental_diff | sql | failure | 94.40 | 2536 |

Continued on next page

Table A.1.: Test results for quantitative tests

| Query ID | Agent Type | Outcome | Duration [s] | Tokens |
|---|---|---|---|---|
| oslo_residental_diff | sql | failure | 136.12 | 2511 |
| 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 |
| 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 | partial success | 59.47 | 1571 |
| oslo_roads_gte_70_kmh | sql | partial success | 44.79 | 1565 |
| oslo_roads_gte_70_kmh | sql | partial success | 41.67 | 1548 |
| 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 | 25.81 | 1170 |
| vestfold_bbox | sql | failure | 32.59 | 1561 |
| vestfold_bbox | sql | success | 20.00 | 1192 |
| viken_dissolve | oaf | success | 52.24 | 2336 |

Continued on next page

Table A.1.: Test results for quantitative tests

| **Query ID** | **Agent Type** | **Outcome** | **Duration [s]** | **Tokens** |
|---|---|---|---|---|
| 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 | 26.49 | 1447 |
| viken_dissolve | sql | success | 47.59 | 1963 |
| viken_dissolve | sql | partial success | 36.20 | 1555 |

# B. Code

## B.1. Python Example

```
1         import numpy as np
2
3         def incmatrix(genl1,genl2):
4         m = len(genl1)
5         n = len(genl2)
6         M = None #to become the incidence matrix
7         VT = np.zeros((n*m,1), int)  #dummy variable
8
9         #compute the bitwise xor matrix
10        M1 = bitxormatrix(genl1)
11        M2 = np.triu(bitxormatrix(genl2),1)
12
13        for i in range(m-1):
14        for j in range(i+1, m):
15        [r,c] = np.where(M2 == M1[i,j])
16        for k in range(len(r)):
17        VT[(i)*n + r[k]] = 1;
18        VT[(i)*n + c[k]] = 1;
19        VT[(j)*n + r[k]] = 1;
20        VT[(j)*n + c[k]] = 1;
21
22        if M is None:
23        M = np.copy(VT)
24        else:
25        M = np.concatenate((M, VT), 1)
26
27        VT = np.zeros((n*m,1), int)
28
29        return M
30
31        import numpy as np
32
33        def incmatrix(genl1,genl2):
34        m = len(genl1)
35        n = len(genl2)
36        M = None #to become the incidence matrix
37        VT = np.zeros((n*m,1), int)  #dummy variable
38
39        #compute the bitwise xor matrix
40        M1 = bitxormatrix(genl1)
```

```
41          M2 = np.triu(bitxormatrix(genl2),1)
42
43          for i in range(m-1):
44          for j in range(i+1, m):
45          [r,c] = np.where(M2 == M1[i,j])
46          for k in range(len(r)):
47          VT[(i)*n + r[k]] = 1;
48          VT[(i)*n + c[k]] = 1;
49          VT[(j)*n + r[k]] = 1;
50          VT[(j)*n + c[k]] = 1;
51
52          if M is None:
53          M = np.copy(VT)
54          else:
55          M = np.concatenate((M, VT), 1)
56
57          VT = np.zeros((n*m,1), int)
58
59          return M
60
61
62          import numpy as np
63
64          def incmatrix(genl1,genl2):
65          m = len(genl1)
66          n = len(genl2)
67          M = None #to become the incidence matrix
68          VT = np.zeros((n*m,1), int)  #dummy variable
69
70          #compute the bitwise xor matrix
71          M1 = bitxormatrix(genl1)
72          M2 = np.triu(bitxormatrix(genl2),1)
73
74          for i in range(m-1):
75          for j in range(i+1, m):
76          [r,c] = np.where(M2 == M1[i,j])
77          for k in range(len(r)):
78          VT[(i)*n + r[k]] = 1;
79          VT[(i)*n + c[k]] = 1;
80          VT[(j)*n + r[k]] = 1;
81          VT[(j)*n + c[k]] = 1;
82
83          if M is None:
84          M = np.copy(VT)
85          else:
86          M = np.concatenate((M, VT), 1)
87
88          VT = np.zeros((n*m,1), int)
89
90          return M
91
92          import numpy as np
93
```

```
94          ya = "hei"
95
96          def incmatrix(genl1,genl2):
97          m = len(genl1)
98          n = len(genl2)
99          M = None #to become the incidence matrix
100         VT = np.zeros((n*m,1), int)  #dummy variable
101
102         #compute the bitwise xor matrix
103         M1 = bitxormatrix(genl1)
104         M2 = np.triu(bitxormatrix(genl2),1)
105
106         for i in range(m-1):
107             for j in range(i+1, m):
108                 [r,c] = np.where(M2 == M1[i,j])
109         for k in range(len(r)):
110             VT[(i)*n + r[k]] = 1;
111             VT[(i)*n + c[k]] = 1;
112             VT[(j)*n + r[k]] = 1;
113             VT[(j)*n + c[k]] = 1;
114
115         if M is None:
116         M = np.copy(VT)
117         else:
118         M = np.concatenate((M, VT), 1)
119
120         VT = np.zeros((n*m,1), int)
121
122         return M
123
```

Code B.1: Python example