LLM-Enhanced Processing of Complex Spatial Queries

Ruiyi Hao, Guanli Liu(✉), and Renata Borovica-Gajic

The University of Melbourne, Parkville VIC 3010, Australia
ruiyi.hao@student.unimelb.edu.au
{guanli.liu1,renata.borovica}@unimelb.edu.au

Abstract. Traditional spatial databases and map services handle deterministic queries well, but struggle to handle intent-rich and complex spatial queries. Even a simple request such as "In Melbourne, find a convenient petrol stop along the Tullamarine Freeway to the airport, avoiding tolls and minimizing detour" often breaks current systems, revealing a gap between natural-language intent and executable spatial operations. To fill this gap, we introduce an LLM-driven framework that parses user intent, decomposes complex spatial requests into executable sub-queries, schedules the necessary Google Maps API calls, and synthesizes the final answer. To enable systematic study, we curate a dataset of complex spatial queries with selected answers and provide a concrete implementation that compiles intents into routes and places calls while coordinating routing and POI services. We evaluate on the generated dataset with Google Maps and find that our system *substantially outperforms* a zero-shot LLM: it retrieves more feasible, on-route candidates and produces schema-valid, verifiable answers, while trading modest extra latency.

Keywords: Spatial data management · Large Language Models · Geospatial services · Query intent understanding.

1 Introduction

Spatial queries are foundational in location-based services (LBS), with well-studied operators such as point lookup, range selection, kNN, spatial join, and shortest-path routing, which modern engines answer efficiently [5]. Yet user expectations increasingly mix spatial feasibility with semantic preferences and intent. For example, "In Melbourne, find a convenient petrol stop along the Tullamarine Freeway to the airport, avoiding tolls and minimizing detour." Such intent-aware queries need reasoning over routes, traffic, and opening hours, and grounding underspecified notions like "convenient" in observable signals. While strong on deterministic operators, current spatial databases and map services lack mechanisms to parse composite intents, decompose them into executable steps across routing and POI services, and verify feasibility under a live context.

To address these challenges, we investigate using Large Language Models (LLMs), which can capture nuanced intent but must be decomposed and grounded

to operate in geospatial QA. Two obstacles arise: (i) intent needs structuring so that spatial operators and external services can execute it. However, prior question decomposition work [3, 9, 11] does not jointly ensure route feasibility and semantic preferences; and (ii) major geospatial knowledge bases are API-only, which limits retrieval-augmented generation [4]. Tool-augmented LLMs [8, 10] help, but require domain-specific grounding to avoid invalid calls and to reason over corridor-bounded candidates. We therefore study to answer *complex spatial query*: given a natural-language request, an origin–destination pair, and live context, return POIs that are feasible under route and policy constraints and aligned with user semantics.

This paper presents a practical instantiation of that workflow. We curate a realistic, human-annotated dataset of complex geospatial queries that captures both spatial and semantic intent and includes gold decompositions and expected tool-call signatures for reproducibility. Building on this dataset, we design an LLM-based planner that converts intent into tool-invocable subgoals and orchestrates routing and POI services end-to-end. The planner performs intent schema extraction, corridor-constrained candidate generation, feasibility verification through API calls and reranking with explicit checks that reject inconsistent or hallucinated results. We instantiate the system with Google Maps APIs and evaluate it on real Melbourne routes. Our created dataset, implementation, and additional demos with sample query results are open-sourced.[1]

Our contributions are as follows:

1. We formalize *intent-aware complex spatial queries* in the on-the-way setting and specify an executable evaluation protocol.
2. We generate a template-driven pipeline and release a Melbourne-scale, human-annotated dataset.
3. We build a two-phase system that compiles intents into Google Maps API calls and returns verifiable answers, comprising an LLM query planner and a sub-query execution agent.
4. The proposed method attains up to Recall@5 46% with a success rate of 89.22%, average distance deviation 2.87 km and average token cost 0.128 USD per query.

The rest of this paper is structured as follows: Section 2 reviews related work. Section 3 presents preliminaries, including the problem formulation, the Google Maps APIs used, and the LLM interface. Section 4 details the overall framework, covering how we construct the dataset and how the system processes complex queries end-to-end. Section 5 reports the experimental setup and results on Melbourne routes. Section 6 concludes with key takeaways and future directions.

[1] `https://github.com/haoyeye5/COMP90055_Research_Project`

2 Related Works

2.1 Classical Spatial Queries

Classical spatial queries form the backbone of LBS/GIS, with mature support for *point*, *range*, *kNN*, and *spatial join* operators. Point and range queries exploit hierarchical pruning of search regions; kNN queries leverage distance-ordered access; spatial joins enumerate object pairs that satisfy predicates such as `intersects`, `contains`, or `distance` [1,5]. These deterministic tasks are well understood and efficiently executed at scale, offering high throughput and low latency on large point datasets. Despite their strength on deterministic predicates, classical operators do not interpret *intent-rich, complex spatial queries*.

2.2 Geospatial QA Systems

Existing works on LLM-enhanced geospatial QA systems mainly follow two directions. The first focuses on fine-tuning on specific geospatial datasets, as in GeoLLM [7], which acquires geographic knowledge through fine-tuning on OpenStreetMap data, and ChatGeoAI [6], which features domain-specific code generation through a fine-tuned LLM. The second relies on few-shot prompting within a dedicated reasoning system design. For example, GeoAgent [2] combines knowledge retrieval, extensive code generation, and simulation-based search. SpatialRAG [12] adapts the RAG paradigm to spatial tasks by balancing spatial query results with semantic similarity.

2.3 Question Decomposition (QD)

Decomposed Prompting (DECOMP) [3] adopts a modular architecture for QD. The system breaks down original questions based on a dedicated prompting program, which comprises multiple adaptable functions. One of the early works [11], for example, performed table-based reasoning by leveraging few-shot prompting to decompose user queries into sub-questions.

2.4 Tool-augmented Language Models

Such a framework was first introduced through ToolFormer [8], which fine-tunes LLMs on API-augmented dataset generated through an API sampling-executing-filtering pipeline. The ReAct prompting style [10] further allows interactive tool-calling by teaching LLMs to build the reasoning trace as a sequence where three primary components interleave each other.

3 Preliminaries

This section formalizes the task and the execution setting. We first define *complex spatial queries* (CSQs) and specify inputs, constraints, and outputs. We then summarize the Google Maps services used for grounding. Finally, we outline the LLM interaction model that maps natural-language intent to API calls.

3.1 Problem Definitions

Complex spatial queries. We use the term *complex spatial queries* to denote queries that go beyond atomic operators such as point lookup, range selection, or kNN search. A complex query typically involves (i) multiple spatial constraints (e.g., route feasibility, detour budget), (ii) additional policy conditions (e.g., avoid tolls, respect opening hours), and (iii) semantic preferences expressed in natural language (e.g., "quiet", "convenient"). For example, *"Find a petrol stop on my way to the airport that avoids tolls and adds at most three minutes"* is a complex query: it integrates routing constraints with user intent, and cannot be directly answered by a single deterministic spatial operator.

Definition 1 (Complex Spatial Query (CSQ)). *Let the road network be* $\mathcal{G} = (V, E, w)$ *and the POI set be* $\mathcal{P}$*. A CSQ is a tuple*

$$Q = (o, d, \mathbf{C}, \mathbf{S}),$$

where $o, d \in V$ *are the origin and destination.* $\mathbf{C}$ *collects* spatial/policy constraints *(e.g., a detour budget* τ*, a corridor bound, avoid-tolls/avoid-tunnels);* $\mathbf{S}$ *encodes* semantic preferences *expressed in natural language (e.g., quiet atmosphere, convenient stop), grounded to observable proxies such as ratings, opening hours, and price levels.*

Objective. *Given* Q *(and optional live context* $\mathcal{T}$*), return a ranked set* $\mathcal{P}^\star \subseteq \mathcal{P}$ *such that every* $p \in \mathcal{P}^\star$ *is* feasible *under* $\mathbf{C}$ *and* $\mathcal{T}$ *and is ordered by*

$$\text{score}(p \mid Q, \mathcal{T}) \;=\; f\big(\Delta t(p),\, \text{Attr}(p),\, \mathbf{S}\big),$$

where $\Delta t(p)$ *is the extra travel time to visit* p*,* $\text{Attr}(p)$ *are POI attributes from external services, and* $f(\cdot)$ *aggregates detour cost with alignment to* $\mathbf{S}$*. Unlike atomic range or kNN queries, a CSQ jointly reasons over routing feasibility, policy compliance, and semantic preferences and thus cannot be answered by a single deterministic operator.*

Example 1 (Melbourne on-the-way selection). Query: "Find a petrol stop on my way from Doncaster East to Melbourne Airport, avoiding tolls and minimizing detour." Here o and d are the trip endpoints, $\mathbf{C} = \{$*avoid-tolls*, τ=3 min$\}$, and $\mathbf{S} = \{$*convenient* $\Rightarrow$ *open-now*, *rating* $\geq 4.0\}$. Feasibility enforces the corridor and policy; ranking trades off $\Delta t(p)$ with the proxies implied by $\mathbf{S}$.

3.2 Google Maps APIs

We use two Google Maps services to ground intents in real data:

- **Routes API (v2: `computeRoutes`)**: computes feasible routes under live traffic and policy settings (e.g., `avoidTolls`). It returns travel times, distances, and polylines, which we use to establish a baseline route and to compute detours for candidate stops.
- **Places API (v1: `places:searchNearby`)**: retrieves nearby POIs by type (e.g., `gas_station`) and returns attributes such as coordinates, rating, price level, and opening hours. We use these attributes to filter and rank candidates according to semantic preferences.

3.3 LLM Interaction Model

The LLM serves as the planner that connects natural language input to API-level execution. Its role is threefold:

1. **Intent Extraction**: parse the CSQ into route and semantic components.
2. **API Call Generation**: transform the extracted intent into specific location or routes API queries.
3. **Response Interpretation**: validate API results against constraints (e.g., detour budget, toll avoidance, opening status) and resolve semantic preferences using available metadata (e.g., ratings).

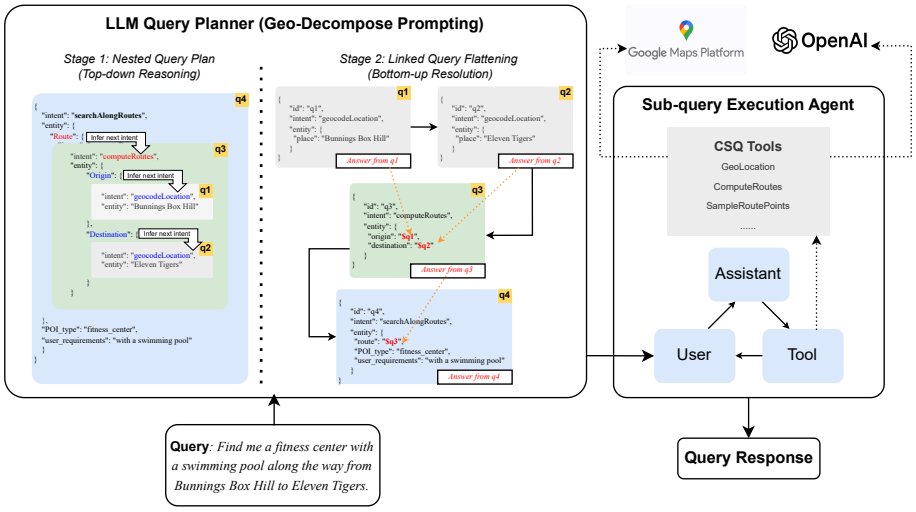

Fig. 1: System Architecture

4 Methodology

The core of our system is the LLM Query Planner, featuring a Geo-Decompose (Geo-DECOMP) prompting style, where a CSQ is broken down into a list of logically ordered and reliable sub-queries through few-shot prompting. As illustrated in Figure 1, the input query is directly fed to an LLM with the Geo-DECOMP prompting style, resulting in a linked list of sub-queries $Q = (Q_1, \ldots, Q_i)$, each in JSON format. The JSON sub-tasks in Q are then resolved by a sub-query execution agent, which interprets the sub-queries iteratively and delegates API calls. We employ GPT 4o-mini as the LLM model throughout the system, selected for its lightweight operations and cost-effectiveness.

4.1 Dataset Generation

We release a dataset of 1,930 complex spatial queries over the Melbourne region. Each query specifies origin/destination, an optional waypoint, buffer ranges, and semantic requirements (e.g., "24-hour service", "vegan menu", "wheelchair accessible"). Ground-truth answers are POIs along the specified routes, stored in JSON. The dataset enables standardized evaluation of hybrid spatial–semantic retrieval and intent-aware query processing under realistic urban settings.

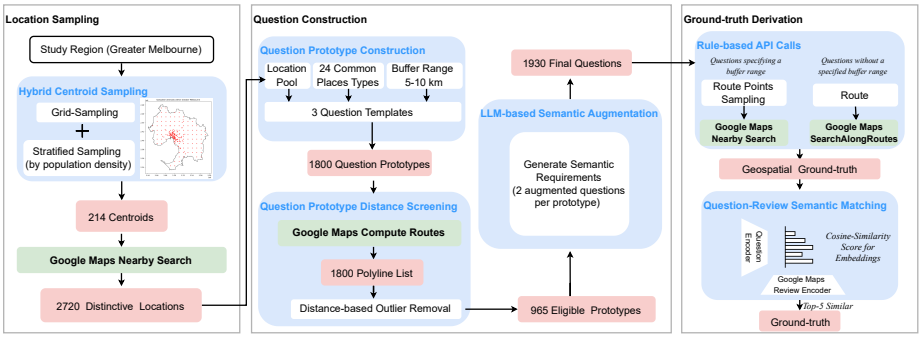

Fig. 2: Dataset construction workflow with three modules: Location Sampling, Question Construction, and Ground-Truth Derivation.

We proposed an API-based, easy-to-scale dataset construction workflow that comprises three main modules, as shown in Figure 2.

Location Sampling. The first module is designed to extract POIs within the study area (Melbourne). We implement a two-step sampling strategy: first generating a series of representative centroids, then querying nearby places around each centroid. To improve real-world representativeness of the POIs while ensuring spatial coverage of Melbourne, we employ a two-layer sampling framework that combines grid and stratified sampling (based on population density).

Question Construction. We adopted three query templates:

1. Find me a `[POI]` along the way from `[Origin]` to `[Destination]`.
2. Find me a `[POI]` along the way from `[Origin]` to `[Destination]`, passing through `[Waypoint]`.
3. Find me a `[POI]` along the way from `[Origin]` to `[Destination]`, within `[Buffer_range]`km of the route.

The placeholders `[Origin]`, `[Destination]` and `[Waypoint]` are populated with sampled places from the previous module. For `[POI]`, we randomly fill in 24 common Google Maps places types (e.g., restaurant, hotel). Outliers are removed at this stage by filtering origin–destination pairs that are excessively long or short for driving mode. In addition, to further introduce semantic preferences **S** to the queries, we leverage LLMs for augmenting user requirements.

Ground-truth Derivation. Ground-truth derivation for CSQ integrates geospatial validation with semantic matching. First, candidate locations satisfying spatial constraints **C** are retrieved. For each candidate, the Google Maps user reviews are concatenated and embedded, forming its dense semantic representation. This embedding is compared with the dense representation of the query in the same space using cosine similarity. The top-5 similar candidates serve as the ground-truth set. This proxy approach for semantic validation may introduce bias due to the subjective nature and variability of Google Maps user reviews.

4.2 LLM Query Planner

Existing LLM-based decomposition strategies focus on enumerating only the questiones and their corresponding sub-questions via a few-shot prompting, but overlook the reasoning process of how those sub-questions are derived. As a result, when generalizing to complex queries, such systems tend to underperform. Based on these limitations, Geo-DECOMP prompting aims to:

- Teach LLMs to decompose CSQ by executing Geospatial Intent Recognition.
- Leverage the systematic, hierarchical structure of JSON to help LLMs identify and preserve the correct dependencies between sub-queries.
- Convert nested sub-queries into a linked list that could be resolved sequentially by the downstream modules.

Stage 1: Nested (Hierarchical) Query Plan In the first stage of Geo-DECOMP prompting, we demonstrate a top-down reasoning process. This is carried out by a specifically designed type of intent recognition, called Geospatial Intent Recognition.

Geospatial Intent Recognition (GIR): As conventional intent recognition tasks, GIR requires the identification of: 1) **intent:** the purpose or action the user wants to perform. 2) **entity:** the parameters required to accomplish the intent. The only difference is that "intent" objects for GIR are defined to be restricted to a list of common methods for resolving geospatial queries. For the purpose of this research, these methods include:

1. *geocodeLocation* – convert a place name to coordinates
2. *searchNearby* – find something near a location
3. *computeRoutes* – calculate a route between locations
4. *sampleRoutePoints* - extract sample points along a route
5. *searchAlongRoutes* - find places or features located along a specified route

JSON as Protocol: The intent-entity JSON representation, together with the restriction rules, can be viewed as a "programming language" tailored for GIR tasks. For the geospatial query shown in Figure 1, a typical GIR-guided reasoning prompt is shown in Fig **??** (left).

The prompt recursively decomposes the query through conducting GIR, until no sub-queries can be further divided. Whenever a placeholder (e.g., <Route>)

I need to find a fitness center from Bunnings Box Hill to Eleven Tigers. I can intend to searchAlongRoutes. Conducting Geospatial Intent Recognition on the query:
– Intent: searchAlongRoutes (sub-query to find places located along a specified route between two points)
– Entity: This would include the POI type (i.e., fitness_center), the reference route (<Route>), and the user semantic requirements (i.e., with a swimming pool)

The corresponding JSON object will be:

```
{
  "intent": "searchAlongRoutes"
  "entity": {<Route>,
             "POI_type": "fitness_center",
             "user_requirements":
             "with a swimming pool"
            }
}
```

Inspecting the previous JSON object, the intent now is to derive the <Route> (i.e., the route from the origin to the destination). Conducting Geospatial Intent Recognition to the query: .....

You are a geospatial assistant that executes sub-queries sequentially using provided tools. Always follow the order of the sub-query list. For each sub-query, call the tool whose name matches the 'intent' value and fill in all required arguments correctly.

User: Now execute sub-query 1 with intent 'geocodeLocation' and entity:
{"latitude": -37.6858788, "longitude": 144.9246674}.
Assistant: Determine calling the 'gecodeLocation' tool
Tool: Call Google Maps Places API
.....

User: Now execute sub-query 3 with intent 'computeRoutes' and entity: {
"origin": {"latitude": -37.6858788, "longitude": 144.9246674},
"destination": {"latitude": -37.6782264, "longitude": 145.0143994}
}
.....

Fig. 3: (Left): Geo-DECOMP style prompt for LLM query planner. (Right): Iterative prompting for sub-query execution agent.

appears in the entity, it triggers a GIR reasoning, leading to the next sub-query. The intermediate JSON objects are asked to be generated every time after reasoning to avoid the catastrophic forgetting issue of LLMs.

Stage 2: Linked (Sequential) Query Flattening The hierarchical structures resulted from Stage 1 are perfect for capturing sub-query dependencies, but are less intuitive to be resolved sequentially. Therefore, we need to linearize the nested sub-queries back to a sequential, non-repeated list. Prompt in this stage involves a set of fixed instructions that asks the LLMs to:

– Assign each sub-query with a unique "id", and refer to previous sub-query outputs with "$id" instead of nesting.
– List the most deeply nested sub-queries first, then proceed outward so only later items can reference earlier ones.
– Remove any duplicated steps within the nested structure.

4.3 Sub-query Execution Agent

The LLM query planner outputs a JSON list containing all sub-queries, and this will then be passed into a tool-augmented LLM to be validated by external APIs. We predefined a set of Google Maps tools for CSQ, including APIs and helper functions for sampling and semantic comparison.

As demonstrated in Figure 1, the sub-query execution phase comprises three primary prompting roles: User, Assistant, and Tool, interacting with each other iteratively. An instance of a prompting pipeline is demonstrated in Figure **??** (right). User role receives a sequence of sub-queries and, for each sub-query, provides the corresponding GIR intent and entities as prompts. The Assistant

then interprets the user prompts to extract necessary arguments if a tool call is required. When a tool call is initiated, the Tool role is responsible for executing the correct external functions.

5 Experiments

5.1 Experimental Setup

Environment. All experiments were performed on a 64-bit macOS machine (Apple Intel Pro i7-core CPU), with 32 GB RAM and a 512 GB SSD. We used Python 3.11 and online geospatial services via Google Maps Routes v2 (`computeRoutes`[2]) and Places v1 (`places:searchNearby`[3]). We choose `GPT-o4-mini`[4] from OpenAI as the LLM model.

Baselines. Since complex spatial queries have no established method under our API-only setting, we adopt an LLM baseline with two-shot Least-to-most [13] prompting for query decomposition, with OpenAI built-in web search as a tool. The design aims to mitigate hallucination by guiding the LLM through rough reasoning steps while equipping it with knowledge from the web.

Query Workloads. We use these three query types that are discussed in Section 4.1 as our query workloads. They span common on-the-way intents with increasing specificity: along-route (Type 1), via-waypoint (Type 2), and corridor-bounded (Type 3). Each template is instantiated with diverse POI types, producing a balanced set that stresses route feasibility and semantic alignment.

Metrics. We report four metrics:

- *Success rate (%)* — fraction of queries for which the system returns a valid JSON format and all required API calls succeed. This captures that an LLM may not always produce a valid, executable output.
- *Average runtime (sec)* — end-to-end latency per query, including LLM parsing and Google Maps API calls.
- *Average Top-5 Recall (%)* — proportion of correct answers that appear among the top-5 returned candidates.
- *Average distance deviation (km)* — mean absolute detour difference, in kilometers, between the returned answer and the correct reference along.

All metrics are computed per query and averaged over the benchmark; boldface in the table marks the better value.

5.2 Main Results

We conducted experiments on all 1930 CSQs in our benchmark dataset. As shown in Table 1, our method retrieves the exact-match with *up to* **46%** Recall@5, while the baseline returns few or no exact match. Recall is lowest on Query Type 3

[2] `https://routes.googleapis.com/directions/v2:computeRoutes`
[3] `https://places.googleapis.com/v1/places:searchNearby`
[4] `https://platform.openai.com/docs/models/o4-mini`

Table 1: System Performance: Proposed vs. Baseline

System	Query Type	Success Rate(%)	Runtime (s)	Recall@5 [1]	Deviation Distance(km)	SQD Accuracy [2]
Proposed	Type 1	89.30	42.21	0.53	1.98	96
	Type 2	85.59	57.94	0.46	1.96	91
	Type 3	90.53	51.86	0.34	5.05	90
	Overall	**89.22**	46.77	**0.46**	**2.87**	89.22
Baseline	Type 1	78.41	12.48	0.00	9.77	– [4]
	Type 2	79.73	13.58	0.00	7.50	–
	Type 3	83.64	13.57	0.00	9.80	–
	Overall	80.05	**12.92**	0.00	9.20 [3]	–

[1] Top-5 Recall [2] Accuracy of sub-query count [3] Before removing outliers, the baseline average deviation distance was 639.29 km. [4] For the baseline model, sub-queries are not in structured formats and hence are difficult to be collected.

because it imposes corridor/buffer constraints and requires sampling along the computed route, followed by nearby search; this stochastic candidate generation can vary the returned POIs and reduce exact matches. Since the ground-truth was derived using Google Maps API, the Recall@5 metric is inherently biased due to its name-based matching mechanism. The baseline, which relies on web-search, might retrieve conceptually correct answers that do not match the exact ground-truth.

Detour. Even when the ground-truth is not in the top-5, our candidates remain close to the reference, with average deviation at 2.87 kilometers, a reasonable driving distance. This reflects corridor-constrained generation during query decomposition and API verification in POI-dense corridors. In contrast, the baseline results are more likely to deviate from the route.

Runtime. Our pipeline is slower than the baseline ($\approx$47 s vs. $\approx$13 s). The added latency comes from LLM-driven query decomposition, tool-calling workflow, real Google Maps API calls and sematic comparisons, which are typical for travel-planning tasks. In return, the system produces verifiable, on-route answers under live constraints. An effective solution to reduce runtime is to design an algorithm that maintains sub-queries in a tree-like structure and executes independent sub-queries on leaves (e.g., deriving geographical locations for origin and destination) in parallel.

Error Analysis. We conducted quantitative analysis on error cases, and found out that all failure was caused by incomplete query decomposition during the planning stage. Specifically, the planner terminated before resolving the coordinates of origin and destination, leaving them as plain-text strings that cause invalid inputs for the route computation API. Moreover, the Sub-Query Decomposition (SQD) accuracy is lower in Type 2 and 3 queries, indicating a more frequent premature stop. These queries involves more complex decompo-

sition logic (one more reasoning hop) and hence challenge LLMs' reasoning and instruction-following ability.

5.3 System Cost

Table 2: API Usage and Cost (per CSQ)

Module	Provider	API Type	Call Count	Cost (USD)
LLM Query Planner	**OpenAI**	Chat Completion	2580 [1] 867 [2]	9.072e−4
Sub-query Execution Agent	**OpenAI**	Chat Completion	5120 2548	0.002
		Text Embedding	8831 [3]	1.77e−4
	Google Maps	Places API	2 1	0.080 0.040
		Routes API	1	0.005
			Total:	**0.128**

**Note:* The cost in this table is estimated using OpenAI GPT 4o-mini model.
[1] Input tokens [2] Output tokens [3] Input tokens

To further examine the feasibility and scalability of our system on cost, we calculate its per-query API consumption. The per-query cost with OpenAI GPT 4o-mini model is summarized in Table 2. Both the LLM query planner and the sub-query execution agent involve API usage, but the cost is mainly incurred during the sub-query execution stage, where Google Maps APIs are called and the intermediate sub-query results are iteratively passed into the LLM prompt for further tool-calling. A standard Type 1 query can consume 0.003 USD for LLM-based query decomposition and execution, followed by triggering 3 Google Maps Places API and a Route API, costing around 0.125 USD (US Dollars). In total, each CSQ costs approximately 0.128 USD. Although expensive for individual usage, the cost could be significantly reduced by caching sub-query results, such as geographical locations and route polylines.

5.4 Ablation Study

Both the LLM query planner and the sub-query execution agent play crucial roles in our system. When removing the former, the agent ends up with an immature tool-calling process, usually stops immediately after finding out the geographical locations. On the other hand, keeping only the question decomposition stage, without external tools for geospatial and semantic validation, would significantly degrade system performance due to LLM hallucination.

Table 3: Ablation Study

Model	Success Rate(%)	Runtime (s)	Recall@5	Deviation Distance(km)	Longest Sub-query
Original System	89.22	46.77	**0.46**	**2.87**	5
w/GPT 3.5-Turbo	63.32	**29.01**	0.37	3.33	5
w/GPT 5-mini	**94.94**	71.70	0.45	3.76	**6**

In this case, we conducted an ablation study by focusing on how LLM models influence the system performance. We replaced GPT 4o-mini with two other models, namely GPT 3.5-Turbo and GPT 5-mini, and denoted these two ablation versions as w/GPT 3.5-Turbo and w/GPT 5-mini, respectively.

Table 3 compares the two substituted versions with the original system on 5 metrics. Switching to GPT 3.5-Turbo resulted in shorter response time ($\approx$29 s) but a lower success rate. The model frequently generates malformed sub-query JSON outputs and wrongly-identified API arguments, such as encoded route polylines, leading to a broken pipeline. In contrast, the w/GPT 5-mini system achieves the highest success rate (94.94%), benefiting from its longer reasoning time and stronger instruction-following capability. Interestingly, the GPT-5-mini ablation version also generates additional sub-queries for semantic comparison, with the longest sub-query chain reaching a length of six, indicating potential for generalization to semantically richer queries.

6 Conclusion

We study intent-aware spatial queries that couple route feasibility with semantic preferences, release a human-annotated dataset, propose an LLM planner that compiles natural-language intents into verifiable Google Maps API calls, and present a tool-augmented sub-query execution agent that automatically resolve spatial queries. On Melbourne routes, our system attains a Top-5 Recall of 53% on Type 1 queries. As query constraints increase (Types 2–3), recall decreases, but the average distance deviation remains small relative to the ground truth. This indicates that our proposed complex spatial query answering strategy is effective in retrieving feasible, preference-aligned candidates. Future work includes richer semantics beyond POI metadata, multi-stop planning, and learned policies that trade off quality against API budgets.

7 Acknowledgments

We gratefully acknowledge support from the Australian Research Council Discovery Early Career Researcher Award DE230100366.

References

1. Brinkhoff, T., Kriegel, H.P., Seeger, B.: Efficient processing of spatial joins using r-trees. In: SIGMOD. p. 237–246 (1993)
2. Chen, Y., Wang, W., Lobry, S., Kurtz, C.: An LLM Agent for Automatic Geospatial Data Analysis. arXiv preprint arXiv:2410.18792 (2024)
3. Khot, T., Trivedi, H., Finlayson, M., Fu, Y., Richardson, K., Clark, P., Sabharwal, A.: Decomposed Prompting: A Modular Approach for Solving Complex Tasks. arXiv preprint arXiv:2210.02406 (2022)
4. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.t., Rocktäschel, T., et al.: Retrieval-augmented Generation for Knowledge-intensive NLP Tasks. NeurIPS **33**, 9459–9474 (2020)
5. Manolopoulos, Y., Nanopoulos, A., Papadopoulos, A.N., Theodoridis, Y.: R-Trees: Theory and Applications. Springer Publishing Company, Incorporated (2005)
6. Mansourian, A., Oucheikh, R.: ChatGeoAI: Enabling Geospatial Analysis for Public through Natural Language, with Large Language Models. International Journal of Geo-Information **13**(10), 348 (2024)
7. Manvi, R., Khanna, S., Mai, G., Burke, M., Lobell, D., Ermon, S.: GeoLLM: Extracting Geospatial Knowledge from Large Language Models. arXiv preprint arXiv:2310.06213 (2023)
8. Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Hambro, E., Zettlemoyer, L., Cancedda, N., Scialom, T.: Toolformer: Language Models can Teach Themselves to Use Tools. NeurIPS **36**, 68539–68551 (2023)
9. Wu, J., Yang, L., Ji, Y., Huang, W., Karlsson, B.F., Okumura, M.: Gendec: A Robust Generative Question-decomposition Method for Multi-hop Reasoning. arXiv preprint arXiv:2402.11166 (2024)
10. Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., Cao, Y.: React: Synergizing Reasoning and Acting in Language Models. In: ICLR (2023)
11. Ye, Y., Hui, B., Yang, M., Li, B., Huang, F., Li, Y.: Large Language Models are Versatile Decomposers: Decomposing Evidence and Questions for Table-based Reasoning. In: SIGIR. pp. 174–184 (2023)
12. Yu, D., Bao, R., Mai, G., Zhao, L.: Spatial-RAG: Spatial Retrieval Augmented Generation for Real-world Spatial Reasoning Questions. arXiv preprint arXiv:2502.18470 (2025)
13. Zhou, D., Schärli, N., Hou, L., Wei, J., Scales, N., Wang, X., Schuurmans, D., Cui, C., Bousquet, O., Le, Q., et al.: Least-to-most prompting enables complex reasoning in large language models. arXiv preprint arXiv:2205.10625 (2022)